A Tree-Based Algorithm for Construction Robots

T. K. Satish Kumar*
Computer Science Department
University of Southern California
tkskwork@gmail.com

Sangmook Jung
Computer Science Department
University of Southern California
sangmooj@usc.edu

Sven Koenig
Computer Science Department
University of Southern California
skoenig@usc.edu

Abstract

In this paper,[1] we present a tree-based algorithm for construction robots. Inspired by the TERMES project of Harvard University, robots in this domain are required to gather construction blocks from a reservoir and build user-specified structures much larger than themselves. While the robots are of roughly the same size as the blocks, they can scale greater heights by using temporarily constructed ramps in the substructures. In this paper, we consider the problem of minimizing the number of pickup and drop-off operations performed on blocks in order to build user-specified structures. Our polynomial-time algorithm heuristically solves this problem and is based on the idea of performing dynamic programming on a spanning tree in the inner loop and searching for a good tree to do so in the outer loop. Our algorithm performs very well in simulation and scales easily to large problem instances. For planning problems of this nature that are akin to construction domains, we believe that valuable lessons can be learned from comparing the success of our algorithm with the failure of off-the-shelf planning technologies.

Introduction

While many tools and equipment are used for construction tasks, humans are still directly involved in critical phases of construction that can otherwise benefit from automation. For example, automated planning and scheduling techniques can be used to increase speed and decrease costs for construction tasks. Furthermore, delegating the actual construction operations to robots can make it safer for humans in hostile situations such as constructing shelter in disaster areas or on other planets.

Towards the goal of automated construction, teams of smaller robots are often more effective than a few larger robots. Smaller robots are usually cheaper, easier to program, and easier to deploy. Despite their possibly limited sensing and computational capabilities, teams of smaller robots are more fault tolerant and provide more parallelism than a few larger robots.

Many examples of collective construction are provided in nature that are analogous to the capabilities of teams of smaller robots. For instance, among many other species of animals, termites are capable of building mounds that are much larger than themselves. Inspired by termites and their building activities, the Harvard TERMES project investigated how teams of robots can cooperate to build user-specified 3-dimensional structures much larger than themselves (Petersen, Nagpal, and Werfel 2011).

The TERMES hardware system consists of small autonomous mobile robots and a reservoir of passive “building blocks”, simply referred to as “blocks”. The robots are of roughly the same size as the blocks. Yet, they can manipulate these blocks to build tall structures by stacking the blocks on each other and building ramps to scale greater heights. Multiple robots should be able to cooperate in a decentralized fashion to build a user-specified structure.

The three basic operations of a TERMES robot are: (1) climbing up or down blocks one block-height at a time; (2) navigating with proper localization on a partially built structure without falling down; and (3) lifting, carrying, and putting down a block so as to attach it to or detach it from a partially built structure. The robustness of the TERMES hardware system ensures the high reliability of these three operations. More background information with proper references to (Jones and Mataric 2004), (Grushin and Reggia 2008), (Napp and Klavins 2010) and (Petersen, Nagpal, and Werfel 2011) can be found in the full version of the paper (Kumar, Jung, and Koenig 2014).

In this paper, we present a tree-based construction algorithm for the TERMES robots; but we only consider the problem of minimizing the number of pickup and drop-off operations performed on blocks in order to build a user-specified structure. The plan generated by our algorithm can be executed either by a single robot or a team of robots with proper coordination. However, because the coordination problem for multiple robots is not discussed completely in this paper, we assume that a single robot executes the plan generated by our algorithm. Our polynomial-time algorithm heuristically solves the problem of minimizing the number of pickup and drop-off operations on blocks and is based on the following two-fold idea: (1) we perform dynamic programming on a spanning tree in the inner loop; and (2) we search for a good tree to do so in the outer loop.

*Alias: Satish Kumar Thittamaranahalli

[1]This is a shorter version of a longer paper published in the Proceedings of the Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling

Our algorithm performs very well in simulation and scales easily to large problem instances. Besides being a useful technique for the problem of automated construction, we believe that valuable lessons can be learned from comparing the success of our algorithm with the failure of off-the-shelf planning technologies for this problem domain.

Problem Formulation

We are given an empty initial configuration and a 2D matrix of non-negative integers, referred to as the input matrix, that represents the desired goal configuration. The cells of the matrix represent physical locations, and the non-negative integers represent the heights of the towers that need to be constructed by stacking up blocks at those locations.[2] At any intermediate stage, the top of a tower is said to be *reachable* if and only if starting from the ground level, the robot can reach the top of that tower by turning and driving forward.[3] A block can be placed on the top of a tower if and only if there is a neighboring tower of equal height, the top of which is reachable.[4] A block can be removed from the top of a tower if and only if there is a neighboring tower of height 1 less, the top of which is reachable.[5] Under these restrictions, the problem is to build the final configuration using as few add and remove operations, or equivalently, as few pickup and drop-off operations on blocks as possible.

Many variants of the collective construction problem for TERMES robots are NP-hard. For example, the *euclidean* traveling salesman problem, which is NP-hard, is reducible to optimal planning, even for a single robot, with a non-empty initial configuration and costs associated with traversing distances. Although many other variants of the collective construction problem are also similarly NP-hard, the complexity class to which the problem of minimizing the number of pickup and drop-off operations belongs is unknown. We leave this complexity classification task for future work.

Without knowing the complexity class to which it belongs, we aim for a heuristic solution strategy for the problem of minimizing the number of pickup and drop-off operations. One naive approach to build a user-specified structure is to do it tower by tower starting from one of the corners. In this approach, we need to build a tower of height h in conjunction with a ramp consisting of towers of heights $h-1, h-2 \dots 1$. The extraneous towers should then be deconstructed, resulting in $O(h^2)$ total number of pickup and drop-off operations. This naive strategy is nowhere close to optimal even for simple input instances.

We can do slightly better if we avoid deconstructing the ramp completely after a tower is built. We can reuse parts of the ramp for adjacent towers and follow the strategy of completing the structure row by row or column by column.

[2]A *tower* is a vertical stack of blocks standing at any location.

[3]Because of the whegs, driving forward includes moving on level surface, climbing up a block, and climbing down a block.

[4]Two towers are said to be *neighbors* of each other when they stand on adjacent cells. However, diagonally adjacent towers are not considered neighbors since the robot cannot move diagonally atop the towers.

[5]Level ground can be considered as a tower of height 0.

(a) (b)

Figure 1: (a) shows the input matrix. (b) shows the workspace matrix. Blank cells indicate towers of height 0.

As we will show later in the paper, this strategy corresponds to performing dynamic programming on a particular kind of spanning tree. Our empirical results show that this strategy is also far from optimal and that we can do much better with other kinds of spanning trees.

Of course, "blocks world" domains are well studied in the area of automated planning and scheduling. Unfortunately, however, we could not solve even small instances of our construction problem using any of the state-of-the-art planners from the 2011 International Planning Competition. FastForward could not solve SAS formulations of our problem using any of the built-in heuristics (Richter, Westphal, and Helmert 2011). The failure to generate even feasible plans in more than a few minutes merely for 4×4 input matrices prompted us to develop the specialized techniques illustrated in this paper.

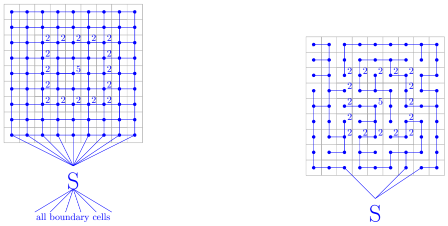
A Tree-Based Construction Algorithm

In this section, we will describe a tree-based construction algorithm for the TERMES robots. The main idea is to perform dynamic programming on a tree spanning the cells of a *workspace* matrix that represent physical locations on a grid frame of reference. The use of dynamic programming allows us to exploit common substructure and reduce the number of operations on blocks significantly. Of course, two questions need to be answered: (1) "how exactly do we perform the dynamic programming"; and (2) "how do we find the best spanning tree for this purpose?" The first question is answered in the inner loop of the algorithm, and the second question is answered in the outer loop.

We will start by describing a few preprocessing steps that construct the workspace matrix and its graphical representation for a frame of reference. In the next subsections, we will describe the inner and outer loops.

Given an input matrix, our first task is to establish a frame of reference that encompasses ramps that might be constructed at intermediate stages. We refer to this frame of reference as the *workspace* matrix. The workspace matrix is a conservative estimate of how much space is required around the final structure during the course of its construction. The workspace matrix is designed before we make decisions about the directions in which the ramps should be built to reach the towers. This means that the workspace matrix is conservative in all directions.

It is easy to observe that a tower of height h can always be manipulated by building a ramp that starts from a location that is at most a Manhattan distance $h-1$ away from the location of the tower. One conservative way to build the



(a) (b)

Figure 2: Shows the graphical representation for the example from Figure 1. (a) shows the graph G. (b) shows a spanning tree of G that constitutes a spanning forest for the cells of the workspace matrix when S is ignored. Only non-zero weights annotating the nodes are shown.

Algorithm 1: Procedure Build-All-Lists

Input: a node-weighted tree T spanning G
Output: an annotation of each node of T with a list of markers

(1) Initialize all lists to contain the single element 0
(2) Call Construct-List for T and its root node S

workspace matrix, therefore, is to include all neighborhood cells of the specified structure with an x-coordinate or y-coordinate that is at most $h-1$ away from the corresponding x-coordinate or y-coordinate of any tower of height h in the final configuration.[6] Figure 1 shows an example.

An undirected graphical representation of the workspace matrix is relatively straightforward to construct. Each cell in the matrix is represented by a node in the undirected graph. The nodes are then annotated with weights corresponding to the entries in the workspace matrix. Two nodes are joined by an undirected edge if and only if the corresponding cells in the workspace matrix are a Manhattan distance of 1 away from each other. In addition, a special node S is used to represent the reservoir of blocks assumed to be relatively far away from the site of construction. This special node is made adjacent to only those nodes that correspond to the boundary cells of the workspace matrix. This is indicative of the fact that a robot carrying a block to or from the reservoir must cross the boundary of the workspace.[7]

Figure 2 shows the graphical representation for the example from Figure 1. We note that for any workspace matrix, a spanning tree T on the graphical representation G induces a spanning forest on the cells of the workspace matrix when S is ignored.

The Inner Loop: Dynamic Programming

The inner loop of the algorithm solves the planning problem by performing dynamic programming on the spanning tree facilitated by the outer loop. Let T be this spanning tree of G with S as the root. The nodes of the tree correspond to cells in the workspace matrix and are annotated with weights equal to the heights of the towers standing at those cells in the user-specified structure.

[6]Measuring distance using the maximum of the differences in x-coordinates and y-coordinates results in a rectangular workspace matrix instead of an arbitrarily shaped boundary of cells.

[7]S need not have a weight assigned to it, but for simplicity, we assign a weight of 0 to it.

Algorithm 2: Procedure Construct-List

Input: the spanning tree T, and a node N in it
Output: an annotation of N with a list of markers

(1) If N is a leaf node in T:
(a) Add the user-specified height of the tower at that location to N's list
(b) Return
(2) Call Construct-List recursively for all of N's children
(3) Let len be the maximum length of the lists constructed for N's children
(4) For $i = 2 \dots \text{len}$, construct the i-th element $L_N(i)$ of the list for N as follows:
(a) If i is even, set $L_N(i)$ to be $\max(L_N(i-1), g_N(i))$ where $g_N(i)$ is the maximum of the i-th elements in the lists of N's children -1
(b) If i is odd, set $L_N(i)$ to be $\min(L_N(i-1), g_N(i))$ where $g_N(i)$ is the minimum of the i-th elements in the lists of N's children
(5) Construct the last element as follows:
(a) If len is even and $L_N(\text{len})$ is less than or equal to the user-specified height h at N, then set $L_N(\text{len}) = h$
(b) If len is even and $L_N(\text{len})$ is greater than h, then add h to N's list
(c) If len is odd and $L_N(\text{len})$ is greater than or equal to the user-specified height h at N, then set $L_N(\text{len}) = h$
(d) If len is odd and $L_N(\text{len})$ is less than h, then add h to N's list

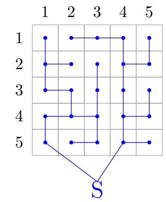
The first stage of the inner loop transforms the weights on the nodes of T to *list* annotations. Instead of a single integer, each node is now annotated with a list of non-negative integers referred to as *markers*. Intuitively, the markers indicate the variations in height that the tower standing at that node has to go through in the course of constructing the user-specified structure.

More formally, the lists of markers satisfy the following properties: (1) the first marker in any list is always 0, indicating that we start from an empty initial configuration; (2) the last marker in any list is equal to the height of the tower in the user-specified structure at that node; (3) barring the first and last marker, and when i is even, the i-th marker of the list for node N, $L_N(i)$, is equal to $\max(L_N(i-1), g_N(i))$ where $g_N(i)$ is the maximum of the i-th markers for the children of N minus 1 (only those children for which the i-th marker exists are considered); (4) barring the first and last marker, and when i is odd, the i-th marker of the list for node N, $L_N(i)$, is equal to $\min(L_N(i-1), g_N(i))$ where $g_N(i)$ is the minimum of the i-th markers for the children of N (only those children for which the i-th marker exists are considered); (5) for a node at height k in the tree, the length of the list annotating it is no larger than $k+2$; and (6) for any list, consecutive markers in it define alternating non-decreasing and non-increasing intervals of non-negative integers.

Algorithms 1 and 2 show how to construct the lists of markers for each node in a given spanning tree T. Figures 3 and 4 show an example. Property 1 is ensured by step 1 of Algorithm 1. Property 2 is ensured by step 5 of Algorithm 2. Here, all steps 5(a), 5(b), 5(c) and 5(d) ensure that for any node, the user-specified height in the final configuration gets added as the last marker. Properties 3 and 4 are ensured by steps 4(a) and 4(b) respectively of Algorithm 2. Property 5 is

1	1	1	1	1
1				1
1		3		1
1				1
1	1	1	1	1

(a)



(b)

Figure 3: Shows a running example. (a) shows the input matrix, identical to its workspace matrix. (b) shows the spanning tree T that we will use. (T is not the optimal spanning tree. It is used merely to illustrate the working of the inner loop.)

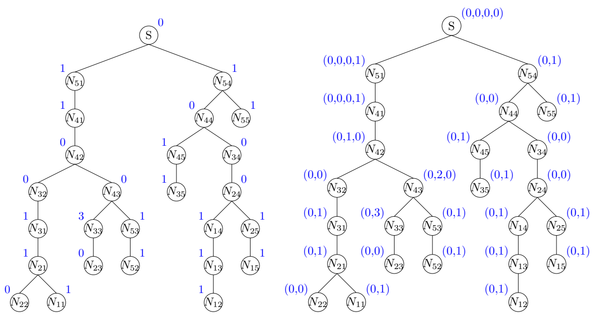
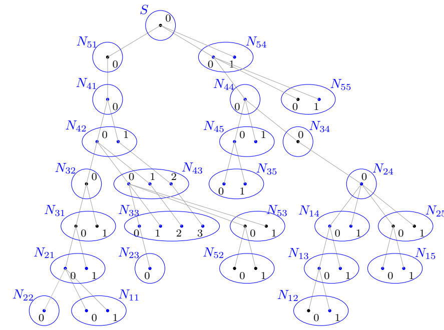


Figure 4: Shows the working of Algorithm 2. (a) shows the spanning tree from Figure 3(b) where node N_{ij} corresponds to the cell in the i^{th} row and j^{th} column. (b) shows the list of markers annotating each node.

ensured by steps 3, 4 and 5 of Algorithm 2 where the number of iterations in step 4 depend on the value of 'len' derived from step 3. Property 6 is ensured by the use of 'max' and 'min' operators in steps 4(a) and 4(b) of Algorithm 2 along with the case analysis done in step 5 of Algorithm 2 for the last marker.

Consecutive markers in a list can be viewed as specifying intervals of non-negative integers. Because of steps 4(a) and 4(b) in Algorithm 2, these intervals are alternating non-decreasing and non-increasing in any list. The first interval between the first and second markers of any list is non-decreasing; the second interval between the second and third markers of any list is non-increasing, and so on. In general, the k-th interval contains a non-decreasing sequence of non-negative integers between its end-point markers if k is odd, and a non-increasing sequence if k is even.

In the next stage of the inner loop, the required plan is generated by traversing a series of *event trees* in depth-first order. Event trees are trees constructed from the original spanning tree T and an interval for each node of T chosen from its list annotation. The first event tree corresponds to choosing the first interval for each node; the second event tree corresponds to choosing the second interval for each node; and so on. Of course, when the k-th interval is not defined for a node, it is simply ignored. From Algorithm 2,



(a)

	1	2	3	4	5
1	1	1	1	1	1
2	1				1
3	1		3		1
4		1	2		1
5		1	1	1	1

(b)

Figure 5: Shows the working of Algorithm 3. (a) shows the first (positive) event tree. (b) shows the partial structure generated by its depth-first traversal.

it is easy to see that the list generated for any node is no longer than the list generated for its parent. This means that the set of nodes for which the k-th interval is defined always forms a subtree of T.

When k is odd, we refer to the corresponding event tree as a *positive event tree*. Traversing a positive event tree in depth-first order generates actions in the plan that add blocks to the structure. When k is even, we refer to the corresponding event tree as a *negative event tree*. Traversing a negative event tree in depth-first order generates actions in the plan that remove blocks from the structure.

Any k-th event tree has a *macro-structure* and a *micro-structure*. The macro-structure has *super-nodes* corresponding to the nodes of T that have a k-th interval. The real nodes, however, correspond to the different non-negative integers occurring in the k-th intervals of the super-nodes. The edges between these nodes, constituting the micro-structure of the event tree, are constructed by finding for each node, a *supporting* node in the parent super-node.

We note that each node in the event tree corresponds to a non-negative integer, referred to as the *value* of that node. For a node with value v in a positive event tree, its supporting node in the parent super-node is the one with the lowest value $\geq v - 1$. Intuitively, this indicates that, in order to create a tower of height v at that super-node by adding a block on top of it, the height of the tower currently standing at the parent super-node must be $v - 1$. Similarly, for a node with value v in a negative event tree, its supporting node in the parent super-node is the one with the highest value $\leq v$. Intuitively, this indicates that, in order to create a tower of height v at that super-node by removing a block from top of it, the height of the tower currently standing at the parent super-node must be v.

Figures 5 and 6 show positive and negative event trees for the running example from Figures 3 and 4. Algorithms 3, 4 and 5 show the procedure for plan generation using positive and negative event trees. Rigorous arguments for correctness are presented in (Kumar, Jung, and Koenig 2014).

Algorithm 3: Procedure Generate-Plan

Input: the spanning tree T with root S, and the list annotations on T's nodes
Output: a sequence of actions for constructing the user-specified structure

(1) Let M be the maximum number of intervals in any list
(2) For $k = 1 \ldots M$:
  (a) If k is odd:
    (i) Construct a positive event tree with super-nodes corresponding to the nodes of T that have a k-th interval
    (ii) Construct nodes corresponding to the different values in the k-th interval of each super-node
    (iii) For node with value v, construct an edge joining it to the node with the lowest value $\geq v - 1$ in the parent super-node
    (iv) Call Generate-Positive-Steps on this event tree
  (b) If k is even:
    (i) Construct a negative event tree with super-nodes corresponding to the nodes of T that have a k-th interval
    (ii) Construct nodes corresponding to the different values in the k-th interval of each super-node
    (iii) For node with value v, construct an edge joining it to the node with the highest value $\leq v$ in the parent super-node
    (iv) Call Generate-Negative-Steps on this event tree

Algorithm 4: Procedure Generate-Positive-Steps

Input: a positive event tree E with root super-node S
Output: a sequence of actions that add blocks to the structure

(1) Traverse the tree E in depth-first order in such a way that a lower-valued sibling is visited before a higher-valued sibling
(2) When a node n is visited, add a block at the corresponding super-node N if the value of n is not the lowest in its interval
(3) The route between N and the reservoir is indicated by the path of super-nodes between N and S in the event tree

The Outer Loop: Search for a Good Tree

The outer loop of the algorithm searches for a good spanning tree to be used in the inner loop. Clearly, one possible spanning tree falls out of connecting all neighboring cells in each row and connecting the first cell in each row to S. This tree corresponds to the intuitive method of constructing the structure row by row starting from one end. Of course, the last cell of each row could have been connected to S instead of the first cell of each row. Similarly, the columns could have been chosen instead of the rows. All these correspond to intuitive strategies for construction.

We note, however, that these options are only a few specific trees in the space of all trees spanning G. Other trees can be used to yield much better results. Two such trees are (a) the *minimum spanning tree* and (b) the *reweighted minimum spanning tree*. Both of them are, of course, heuristic choices, but perform very well in practice. To construct the minimum spanning tree, an edge-weighted graph is constructed from G where the weight of an edge is the absolute value of the difference between the weights of its end-point nodes. All edges incident on S are set to weight 0. Intuitively, a minimum spanning tree for the edge-weighted version of G finds paths in the user-specified structure with minimum height variations.

Although the minimum spanning tree approach is much better in practice than row by row or column by column construction, the problem with it is that the edge weights measure the height variations only among neighboring cells. Two towers with a single gap between them do not influence the edge weights in any way, when clearly, the ramp constructed for one can be used by the other. To address this problem, we construct a variant of the minimum spanning tree called the *reweighted minimum spanning tree*.

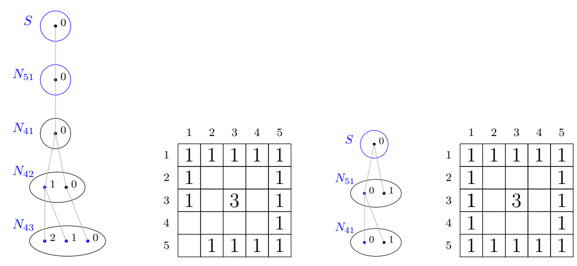
We say that a tower of height h at cell (x, y) casts a *shadow* at cell (x', y') if $h > |x - x'| + |y - y'|$. When a shadow is cast, the *size* of the shadow s is given by $h - |x - x'| - |y - y'|$. Algorithm 6 presents a procedure for reweighting the edges of G. The idea is to scale down the weight of each edge in G by an amount that is proportional to the "usefulness factors" of its end-point nodes. The usefulness factor of a cell in the workspace matrix (represented as a node in G) is the sum of the relative shadow sizes[8] cast on it by all towers in the user-specified structure.

This heuristic way of reweighting the edges transfers information about a tower to farther nodes in the graph G than just its immediate neighbors. When a user-specified structure consists of disconnected substructures, the reweighted minimum spanning tree can construct long "backbone" ramps that are useful for all disjoint substructures. The minimum spanning tree heuristic typically does not produce such backbones.

Algorithm 5: Procedure Generate-Negative-Steps

Input: a negative event tree E with root super-node S
Output: a sequence of actions that remove blocks from the structure

(1) Traverse the tree E in depth-first order in such a way that a higher-valued sibling is visited before a lower-valued sibling
(2) When a node n is visited, remove a block from the corresponding super-node N if the value of n is not the highest in its interval
(3) The route between N and the reservoir is indicated by the path of super-nodes between N and S in the event tree



(a) (b) (c) (d)

Figure 6: Shows the working of Algorithm 3. (a) shows the second (negative) event tree. (b) shows the partial structure resulting from its depth-first traversal. (c) shows the third and final (positive) event tree. (d) shows that the user-specified configuration is achieved by the end of its depth-first traversal.

[8] the fractions s/h

Algorithm 6: Procedure Reweight-Edges

Input: the graphical representation G of a workspace matrix W
Output: reweighted edges for G

(1) For each cell (i, j) in the workspace matrix W, compute the usefulness factor $u(i, j)$ as follows:
  (a) $u(i, j) = 0$
  (b) For each tower of height h at location (i', j') that casts a non-zero shadow s on (i, j):
    (i) $u(i, j) = u(i, j) + s/h$
(2) For each edge e in G:
  (a) $nmr = 1 + |$difference in weights of its end-point nodes$|$
  (b) $dnr = 1 +$ sum of the usefulness factors of its end-point nodes
  (c) weight of $e = nmr/dnr$

%Empty	TBT	RBR	MST	RMST
10	7537	4139	1799	1701
20	6977	3691	1627	1581
30	6332	3712	2226	1948
40	5712	3760	1968	1806
50	4106	3138	1852	1632
60	3452	3108	1964	1530
70	2751	2445	1691	1299
80	1583	1465	1159	0773
90	0812	0812	0770	0552

Table 1: Shows the relative performances of TBT, RBR, MST, and RMST on randomly generated structures.

Empirical Evaluation

In this section, we provide an empirical evaluation of our algorithms. As mentioned previously, none of the off-the-shelf domain-independent planners were able to solve even small instances of the construction problem. Our empirical results, therefore, are a performance comparison between the tower by tower (TBT) method, the row by row (RBR) method, the minimum spanning tree (MST) heuristic, and the reweighted minimum spanning tree (RMST) heuristic. We used three categories of problem instances.

In the first category, we used structures generated at random. Table 1 shows the comparative performances of TBT, RBR, MST and RMST on 10×10 randomly generated matrices with a maximum height of 15. The percentage parameter indicates the fraction of empty locations where no towers stand. 5 trials were used to generate the data in each row. In each row, the median number[9] of additions and removals of blocks in the plans generated by each algorithm is reported. Here, we see the superior performance of RMST in all cases. As the percentage of empty locations increases, the number of disconnected substructures also increases, and RMST begins to outperform even MST more and more.

In the second category, we designed LEGO models of world famous buildings and gave them as input instances to our algorithms. A sample of the empirical data is provided in Table 2 where the total number of additions and removals of blocks required by each algorithm is shown. Even for these instances, MST and RMST outperformed RBR and TBT by a large margin.

[9] over these 5 trials

Building Model	Matrix	Max H	TBT	RBR	MST	RMST
Eiffel Tower	7×7	15	845	845	781	509
Empire State	6×8	15	3152	932	450	476
Taj Mahal	12×12	6	896	384	352	350
Giza Pyramid	15×15	8	2752	680	680	680
Disney Hall	22×16	10	11091	2245	1493	1499

Table 2: Shows the relative performances of TBT, RBR, MST, and RMST on models of world famous buildings.

In the third category, we used handcrafted examples to provide "worst-case" scenarios. Even here, we observed the same trend. That is, MST and RMST outperformed RBR and TBT by a large margin. RMST typically generates plans that require only $45-60\%$ of pickup and drop-off operations on blocks compared to TBT and RBR.

Our tree-based algorithms solved all instances in less than 5 seconds each. The algorithms are implemented in Java. All experiments were run on a single 2.3GHz quad-core (Intel Core i7) MacBook Pro machine with 16GB 1600MHz memory. A separate visualization program is used to get insights into the working of the algorithms.

Conclusions and Future Work

In this paper, we presented a tree-based algorithm for construction robots. Inspired by the TERMES project of Harvard University, robots in this domain are required to gather construction blocks from a reservoir and build user-specified structures much larger than themselves. Common to many variants of the problem, we identified the core combinatorial task of minimizing the number of pickup and drop-off operations performed on blocks in order to build user-specified structures. Our polynomial-time algorithm heuristically solves this problem and is based on the idea of performing dynamic programming on a spanning tree in the inner loop, and the search for a good tree to do so in the outer loop. Empirical results showed that our algorithm performs very well in simulation and scales easily to large problem instances. Besides being a useful technique for the problem of automated construction, we believe that valuable lessons can be learned from comparing the success of our algorithm with the failure of off-the-shelf planning technologies for this problem domain.

There are many avenues for future work. One important direction is to conduct local search on trees in the outer loop. This could not only lead to better solutions for the problem addressed in this paper, but could also lead to a general principle for solving the many variants of the construction problem. Lessons learned in the context of local search methods for solving other combinatorial problems, like SAT, can be employed here as well. Other directions include adapting our techniques to specific construction tasks from real-world domains and comparing them with other approximate heuristic strategies like Ant Algorithms (Dorigo, Di'Caro, and Gambardella 1999). Construction of more complex structures such as with roofs, hollow enclosures and non-uniform block sizes is also an interesting avenue for future work. More discussions with references to (Yu and LaValle

2012) and (Yun and Rus 2010) can be found in the full version of the paper (Kumar, Jung, and Koenig 2014).

Acknowledgements

We thank Marcello Cirillo, Tansel Uras and Liron Cohen for their suggestions and helpful discussions, Tansel Uras and Liron Cohen also for their help by experimenting with general-purpose planners in our domain, and Radhika Nagpal for her interest in our project and her encouragement. Our research was supported by NSF under grant number IIS-1319966 and ONR under grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

References

Dorigo, M.; Di'Caro, G.; and Gambardella, L. 1999. Ant algorithms for discrete optimization. In *Artificial Life, MIT Press, 1999*.

Grushin, A., and Reggia, J. 2008. Automated design of distributed control rules for the self-assembly of prespecified artificial structures. In *Robotics and Autonomous Systems, 56(4):334-359*.

Jones, C., and Mataric, M. 2004. Automatic synthesis of communication-based coordinated multi-robot systems. In *Proceedings of the 2004 International Conference on Intelligent Robots and Systems*.

Kumar, T. K. S.; Jung, S.; and Koenig, S. 2014. A tree-based algorithm for construction robots. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling*.

Napp, N., and Klavins, E. 2010. Robust by composition: Programs for multi-robot systems. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*.

Petersen, K.; Nagpal, R.; and Werfel, J. 2011. Termes: An autonomous robotic system for three-dimensional collective construction. In *Proceedings of Robotics: Science and Systems*.

Richter, S.; Westphal, M.; and Helmert, M. 2011. Lama 2008 and 2011. In *Proceedings of the 2011 International Planning Competition*.

Yu, J., and LaValle, S. 2012. Multi-agent path planning and network flow. In *Proceedings of the 10th Workshop on Algorithmic Foundations of Robotics*.

Yun, S., and Rus, D. 2010. Adaptation to robot failures and shape change in decentralized construction. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation*.