Statically Stable Assembly Sequence Generation for Many Identical Blocks

Georgia Tech Research Report: GIT-IC-07-06

October 16, 2007

Sebastien J. Wolff
Atronix Engineering
3100 Medlock Bridge Road
Suite 110
Norcross, Georgia
Email: swolff@ieee.org

Imme Ebert-Uphoff
Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0405

Joint appointment with:
College of Computing
Robotics and Intelligent Machines Center
Interactive & Intelligent Computing Division
Georgia Institute of Technology
Atlanta, Georgia 30308
Email: ebert@me.gatech.edu

Harvey Lipkin
Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0405
Email: harvey.lipkin@me.gatech.edu

Statically Stable Assembly Sequence Generation for Many Identical Blocks

Sebastien J. Wolff
Atronix Engineering
3100 Medlock Bridge Road
Suite 110
Norcross, Georgia
Email: swolff@ieee.org

Imme Ebert-Uphoff
Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0405
Email: ebert@me.gatech.edu

Harvey Lipkin
Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0405
Email: harvey.lipkin@me.gatech.edu

This work develops optimal assembly sequences for modular building blocks. The underlying concept is that an automated device could take a virtual shape such as a CAD file, and decide how to physically build the shape using simple, identical building blocks. The primary focus of this work is the development of methods for generating assembly sequences in a time-feasible manner that ensure static stability at each step of the assembly. This is accomplished by a multi-hierarchical rule-based approach, consisting of a set of low-level, mid-level and high-level assembly rules. Both high-level and mid-level assembly rules are primarily based on static considerations. The best performing rules are presented and their behavior is analyzed.

1 Introduction

Assembly sequences are crucial to many engineering problems. Most existing work details assembly sequence enumeration and generation for assemblies with few parts, but with complicated interfaces. Some situations however require assembly sequences to be generated for a large number of parts that prevent traditional methods from being useful. One such instance, which is the focus of this work, is rapid shape display.

1.1 Shape Display

Most current methods of representing shape rely on communicating shape with visual information. A number of recent projects have sought to expand the representation of shape to the physical world. For example, *rapid prototyping* provides a means to obtain physical, touchable models in a limited amount of time. In contrast, *modular robots* [1], [2], [3], [4], consist of actuated connected modules, but are limited due to the fact that their small actuators have to carry all weight. Finally, *Digital Clay* [5], [6], [7] is an application currently being developed at the Georgia Institute of Technology which uses micro-fluidic actuation. The current implementation is a large array of pins that is actuated vertically only. While it may serve as both an input and an output device, it will not be able to represent certain shapes such as overhangs and toroids.

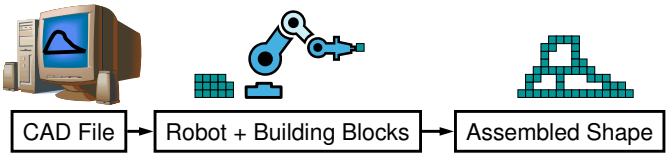


Fig. 1. Recyclable and Scalable Rapid Prototyping For Shape Display

1.2 Recyclable Rapid Prototyping

Most aforementioned concepts rely on an automated device *being* a shape. This work proposes to have an automated device *assemble* shapes instead. The main idea is for a robot to autonomously be able to build a structure to fit any prescribed shape. This paper presents the methods by which an assembly sequence is selected that will ensure that the structure can be built in a statically stable manner. One way to think of the proposed device is as scalable and recyclable rapid prototyping: "recyclable" since blocks used in one prototype can be disassembled and re-used for future prototypes, and "scalable" since the procedure can be applied to a wide range of block sizes, ranging from tiny MEMS parts to huge hollow wood blocks. Figure 1 shows the basic procedure of recyclable rapid prototyping for shape display. The user provides a CAD file. A robot then assembles the shape from a

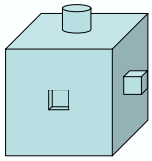


Fig. 2. A Simple 6-sided Block

large number of identical building blocks.

1.3 Existing Work: Assembly Sequences

Homem de Mello and Sanderson [8] developed the main representations of assembly sequences, in the form of graphs. Most existing work consists of first enumerating all possible assembly sequences in graph form. Bourjault [9], De Fazio and Whitney [10], Wilson [11], Ames et al. [12], Kaufman et al. [13], and Homem de Mello and Sanderson [14] all develop methods by which all possible assembly sequences for a given assembly can be enumerated. Once the full list of assembly sequences is generated, Homem de Mello and Desai [15], [16], Nilsson [17], Wilson [11], Ames et al. [12], and Kaufman et al. [13] all then use graph searching techniques which search many of the possible assembly sequences, and test all options at any given step. It should be noted that none of this work considers static testing but rather optimizes values such as assembly time. In addition, only Homem de Mello and Desai [15], [16] consider systems with a large number of parts, but statics are not involved because they work with trusses in a weightless environment.

1.4 Block Type and Modeling Environment

This work uses a fairly simple modular block type and a simplified static connection model to keep the focus of the work on the big picture. The blocks being considered are rectangular blocks that can connect to one another on all 6 sides with male and female connections as seen in Figure 2. The horizontal connections require a square connection to support a moment due to gravity, while the top connections never have to support a moment and can thus be round. Furthermore, it is easier to model the top connections as round so that a moment in any direction is equally likely to cause the connection to snap.

Once the general technique has been developed to address this type of problem, the block type can be generalized and the static connection model can be refined. The static connection model defines the conditions on the forces and moments that will cause a connection to break.

It is assumed that the blocks are always oriented in the same direction, which means that the blocks can only be translated. The workspace can thus be discretized and any structure that can be built with these units is represented by a three dimensional array, S, for the spatial case. The shape to be represented is fit into a grid, whose unit length is a block length. The blocks connect at the center of their sides and once connected, the surfaces of the connecting blocks (or neighbors) match. The blocks were modeled as PETP plastic cubes with a side length of 1 cm.

A full testing and visualization package was developed, and it allows the user to input a structure (array) and assembly sequence (list of arrays), and can output the maximum forces and moments that occur at the connections at any step of assembly. It can also animate the entire assembly. All the programming for this work was developed in Matlab Version 7.0.4. Programming for the static testing was performed with Femlab, which is a finite element modeling package that interfaces with Matlab.

1.5 Organization of this Article

The rest of this paper is organized as follows. Section 2 describes and justifies the novel rule-based approach to assembly sequence generation. Section 3 describes the best performing rules, how they were generated, and how they perform. Section 4 details some sample numerical results. Section 5 presents a synthesis of the results and a summary of when various rules should be used. Finally, Section 6 presents the conclusions.

2 Rule-Based Approach

The main contribution of this work is to make assembly sequence generation for a large number of parts time-feasible while taking static considerations into account. There are two primary challenges for time-feasibility:

1. **Exponential search space**: The search space for assembly sequences is exponential in the number of parts and thus searching *all* possibilities is impossible. For n parts, the number of assembly sequences can be up to $n!$.
2. **Static Stability Tests are Time-Consuming**: Even a search of only a polynomial number of assembly sequences may not be time-feasible if static stability has to be tested explicitly for every considered sequence, particularly with FEM testing.

The first challenge above (exponential search space) was addressed by viewing this problem as a search space reduction problem.

In the developed algorithm so-called *high-level rules* drastically reduce the search-space by identifying a portion of the structure to be assembled first. So-called *mid-level rules* decide what order to assemble those blocks in. Finally *low-level rules* are invoked at every step to see what blocks are viable options. It should be noted that mid-level rules can also be used on their own to assemble structures, as well as in conjunction with high-level rules.

The second challenge above (time-consuming static stability tests) was addressed by designing the rules specifically to address static stability as far as possible in open-loop. Namely, the rules are designed to generate an assembly sequence that is already *likely* to be statically stable, and static stability is only tested at the very end, once the final assembly sequence is generated. Thus static testing of an assembly sequence only has to be performed a *single* time if the algorithm performs well.

2.1 Types of Rules

Assembly sequences are created by applying either assembly or disassembly rules. **Assembly rules** take in a structure, and virtually assemble it by adding one block at a time. In contrast, **disassembly rules** take in a structure, and virtually disassemble it by removing one block at a time. An assembly sequence is still generated by reversing the order of the disassembly sequence. Regardless of which direction is chosen at any given time, three "levels" of rules are established:

Low-level rules enforce constraints by observing the structure and determining which blocks *can* be assembled or disassembled next. These rules enforce that the robot can reach blocks to be assembled and that no blocks "float in the air" during assembly.

Mid-level rules are rules which take in a structure, and at each step determine which block *should* be assembled (disassembled) next, after calling the low-level rules to determine which blocks *can* be assembled next. While an entire structure can be assembled by the mid-level rules, each step considers only which block is assembled next, without considering future steps.

High-level rules are rules which take a structure and at each step identify a *portion* of the structure that should be assembled next. The portion of the structure, which consists of one or more blocks, can then be assembled by applying a mid-level rule.

The two **low-level rules** implemented in this work are accessibility and connectivity. **Accessibility** refers to the robot's ability to get to a block of interest. The current implementation assumes a snake-like robot that can access a block whenever there is a path of vacant blocks from the outside of the structure to the block being added. Other robot types result in more stringent criteria which reduce the search space further. **Connectivity** refers to the constraint that there can be no "floating" blocks. Namely, a block can not be assembled unless one of its neighbors is already in place, and a block can not be removed if it will cause neighboring blocks to no longer be connected to ground.

Another important constraint is **static stability**, which means that the structure should be able to resist gravity at any moment in time during assembly. Ideally, static stability would be enforced as a low-level rule. Unfortunately, this would take far too long. The preferred solution, as discussed in Section 2.3, is to use intelligent rule design through mid and high-level rules to only have to test the steps of the final selected assembly sequence. The top performing high-level and mid-level rules are discussed in Section 3.

2.2 Combining Rules

The overall approach is to efficiently combine high-level, mid-level, and low-level assembly rules in order to develop statically stable assembly sequences rapidly. Figure 3 displays the information flow in the *development*[1] of the synthesis algorithm, which in turn generates assembly sequences. The boxes with dashed lines represent elements that can be replaced for various applications. For instance, connection models that more closely represent a particular application may be substituted. The arrows represent general information flow. For instance, the connection model is used by the static stability test and the low-level rules.

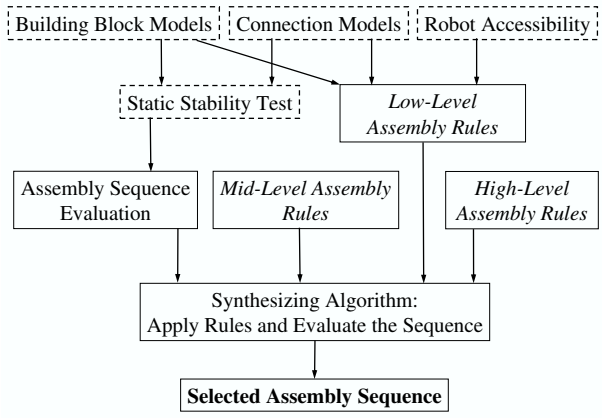


Fig. 3. Information Flowchart Representation of the Development of the Synthesis Algorithm

The blocks written in italics represent the main contributions of the work: the low-, mid-, and high-level assembly rules. What is important to note is that the low-, mid- and high-level rules are independent of the block model, thus opening the door to apply them to a much wider class of blocks than described here. In fact, one can easily imagine an implementation of these rules which would remove the requirement that blocks be identical, thus allowing to use this approach for many civil engineering applications.

A synthesizing algorithm should take an input structure, select what rules to apply, generate an assembly sequence and test its static stability at the end. For the rule selection process we expected to have to classify structures and assign a best rule for each class. However, for the block model considered here it turned out that a *single* rule is superior for *all* sample structures considered (see Section 4). It is not yet clear whether this would also be the case for all other block types, potentially requiring additional research to classify structures accordingly for other block types.

The proposed synthesis approach is to combine the selected rules in assembly and disassembly direction. Suppose that the first rule applied is a high-level *assembly* rule. As explained previously, all high-level rules are implemented with a mid-level rule, which in turn uses low-level rules to check for constraints. After this initial assembly rule, the structure that results from applying these assembly tasks will become the new initial structure, S_{init}, as seen in Figure 4. The problem has now been reduced to finding an assembly sequence to assemble from S_{init} to the final structure.

Suppose that a high-level or mid-level *disassembly* rule is selected next. The disassembly rules start with the final structure and remove blocks, resulting in a new target structure, S_{target}. The problem at this point becomes a search for appropriate sequences to disassemble S_{target} into S_{init}, or as-

[1]Not to be confused with the information flow at *run-time* of the algorithm.

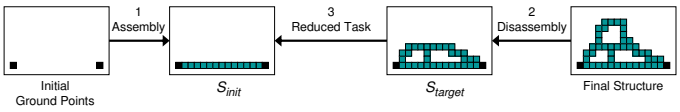


Fig. 4. Outline of the Rule Synthesis Process

semble S_{init} into S_{target}. Figure 4 illustrates the process. The entire process can then be repeated until the assembly and disassembly sides meet. So long as the partial assembly / disassembly sequences are saved, the end-result of this process will be an assembly sequence that assembles from the ground points to the final structure.

2.3 Computational Complexity

The time required by our algorithm to *generate* an assembly sequence is very small. For example, for typical mid-level rules the entire assembly sequence generation takes less time than a *single* static stability test of the structure. Thus the critical criterion for algorithm time is the number of static tests performed.

While previous methods would need to perform static stability tests many times *for every single block selection*, intelligent rule selection allows us to test the static conditions only after the entire assembly sequence has been determined, at which time each assembly step is verified once to make sure that no conditions are violated. In the unlikely event that unacceptable conditions occur during assembly, different rules are applied. This is why it is crucial to understand the behavior of all rules in various scenarios to be able to predict which ones yield the best static results. As a result this open-loop approach typically requires the *minimal* number of static tests. Namely, for a structure with N blocks, our algorithm typically only requires static stability testing N times (one test for each step of the final assembly sequence).

3 Overview of the Best Performing Rules

A large number of high- and mid-level rules were developed and are discussed in much detail in [18]. It should be emphasized that the primary consideration for all rules was to establish good static stability conditions.

We started out with several intuitive rules, such as building from the ground up (i.e. always assembling blocks with lowest z-coordinate first) or trying to establish connections between grounded blocks as quickly as possible. A description of the general behavior was then obtained for each such rule using the following procedure:

1. Apply the algorithm to many (or all) sample structures to generate the assembly sequences. Both accessibility and connectivity is verified.
2. Use the analysis tools that were developed to investigate the largest forces and moments for each assembly sequence. These are the potential failures.
3. Visually inspect the assembly sequence using the animation tools to look for any unusual or unexpected behavior.
4. Assemble a list of all observations from (2) and (3) and identify the specific properties of the algorithm that are responsible for this behavior.
5. Summarize all findings. There is certainly no guarantee that this procedure can identify all problems or issues that can arise for any of the algorithms, simply because the number and nature of sample structures being considered is limited. However, tremendous insight was gained in the general behavior of the algorithms that helps select the best candidates.

Insight gained from the above approach was used to better understand these rules and to develop even better ones. This process generated a total of 27 mid- and high-level rules. Due to space limitations only the top performing rules are presented here. Section 3.1 briefly presents the best mid-level rule, Section 3.2 presents the best high-level rule, and finally Section 3.3 introduces the random rules that were created as a baseline.

3.1 Best Mid-Level Rule: Assemble_D_Solid_Local

The best mid-level rule is a rule called "Assemble_D_Solid_Local". This rule was created by combining two rules, Assemble_Solid_Local and Assemble_D_Local, which are presented briefly below in Sections 3.1.1 and 3.1.2. This rule was the best mid-level rule because it combines the advantages of both of those rules.

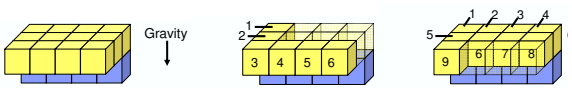
Essentially, Assemble_D_Solid_Local is an *assembly* rule that uses a primary and a secondary criterion. The primary criterion (indicated by "D") is *smallest distance of a block to ground*, where distance is defined as the shortest neighbor-to-neighbor path through blocks of the current structure. Among blocks of equal distance to ground the secondary criterion (indicated by "Solid") selects the block with the *largest number of neighbors in the intermediate structure*.

The equivalent *disassembly* rule, Disassemble_D_Solid_Local, is also presented below. It is the disassembly counterpart of the above rule and is equally successful.

3.1.1 Assemble_Solid_Local

At any time, this algorithm assembles the block that - out of all possible choices - would have the most current neighbors if it was assembled. A block's neighbor is a block that shares a face with it. A block can have up to 4 neighbors. At any iteration, the rule considers all possible blocks that *can* be assembled (ensured by the low-level rules), and determines how many neighbors each potential block would have.

The motivation for this algorithm is to assemble blocks that will have the most neighbors. The motivating concept for this algorithm is that by having more neighbors, the blocks being assembled will have more connections to the existing structure. In general, more connections to the structure allow for better distribution of the forces and moments due to the addition of this block. In particular, **Assemble_Solid_Local is very effective at preventing unnecessary**



(a) (b) (c)

Fig. 5. Example of (a) an Overhang Being Assembled by (b) Most Neighbors Last and (c) Most Neighbors First.

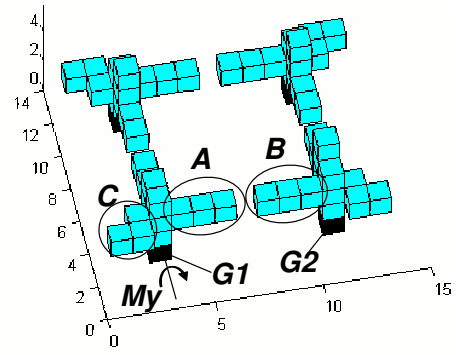


Fig. 6. Branches Being Assembled By Assemble_D.

cantilever structures, and at building solid bases which can better support the loads created by necessary cantilever structures. This can be seen in Figure 5 where assembling blocks with more neighbors first avoids the large moments that would be generated by the cantilever in Figure 5(b).

Unfortunately, **Assemble_Solid_Local is inefficient when cantilever structures are inevitable during assembly AND there are supplementary blocks on the cantilever which do not help support or connect the beam.** This is because these blocks will generally have more neighbors and thus be assembled, yet they only worsen the static conditions.

3.1.2 Assemble_D_Local

Assemble_D_Local uses a distance measure to assemble blocks, namely measuring *the number of blocks on the shortest path to ground*. For instance, any block that neighbors a ground block has a distance of 1, and its neighbors have a distance of 2.

This rule performed well. Consider Figure 6, in which a sample structure called "Branches" is being assembled by this rule. One can see that ground points $G1$ and $G2$ are being connected by the shortest path between them. This connection occurs equally from each side, which minimizes the intermediary cantilevers, and thus the loads. In general, **Assemble_D_Local first connects the shortest connections between ground blocks by assembling equally from each ground point, and does so very efficiently for thin, isolated connections. It is not as efficient when these connections have many extra blocks neighboring them in some areas that do not help support the cantilever.**

Additionally, this rule assembles equally in all directions, as seen in Figure 6. Consider group A, which extends in the $+x$ direction, and group C which extends in the $-x$ direction. Group C creates a moment that opposes the moment My, created by the weight of A at the connection above $G1$. **By assembling in all directions, Assemble_D_Z_Local takes advantage of any symmetry about an axis above the ground block and reduces the moments that occur in the horizontal (top, bottom) connections.**

3.1.3 Assemble_D_Local Versus Disassemble_D_Local

As shown in previous work [19], for some criteria disassembly differed significantly from assembly. For example, if the primary criterion is height (z-value) of a block over ground, there are situations where blocks with low z values could only be assembled after blocks with higher z-values were assembled. In essence, the strategies differ when there are less desirable blocks that have to be assembled before some more desirable blocks can be assembled, because of connectivity.

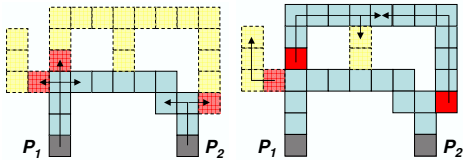
This scenario, however, generally does not occur with the distance from ground measure used in Assemble_D_Local. The basic reason is that there won't be any small-distance blocks that can only be assembled if high-distance blocks are in place first. By definition, it is the close ones that are required to be in place before the farther ones. Additionally, accessibility is unlikely to be a problem, because it is rare for blocks that are closer from ground to be in the way of blocks that are further from ground, when a robot attempts to access them. Because accessibility and connectivity are highly unlikely to prevent blocks from being (dis)assembled, either direction will result in blocks being assembled directly according to distance, with no change of direction. **Therefore proceeding by assembly or disassembly makes little to no difference when the primary criterion is distance to ground.**

3.1.4 Disassemble_D_Solid_Local

This function is the disassembly equivalent function to Assemble_D_Solid_Local. At any time, the blocks with the furthest distance from ground are removed first, and the blocks with the least neighbors are removed first among equal blocks. The motivation is identical to that of Assemble_D_Solid_Local, with an added benefit. Indeed, it has been shown that no dead-ends due to accessibility or connectivity can occur when sequences are being generated by disassembly [16]. The behavior is also virtually identical to that of its assembly counterpart, because the impact of choosing assembly or disassembly is minimal when distance to ground is the primary criterion.

3.2 Best High-Level Rule: CGL_RUC

This section details the best high-level rule that was developed and tested on sample structures. High-level rules consider the entire structure and determine a *set* of blocks to be removed or added next. Once the high-level rules select a portion of the structure that is to be assembled or disassembled, mid-level rules are typically selected to assemble these portions. Additionally, if high level rules can only assemble



(a) (b)

Fig. 7. (a) A Structure's Start Points, and (b) the Selected Path.

part of the structure, the mid-level rule assembles the rest.

The best rule is named Connect_Ground_Local_Recursive_Unique_and_Close, abbreviated below as CGL_RUC. This rule was created by making iterative improvements on the Connect_Ground rule, presented below in Section 3.2.1. The improvements indicated by the suffixes 'Local', 'Recursive, 'Unique', and 'Close' are respectively detailed in Sections 3.2.2, 3.2.3, 3.2.4, and 3.2.5. This rule was successful because it combines the advantages of all of these improvements. The results are synthesized in Section 5.

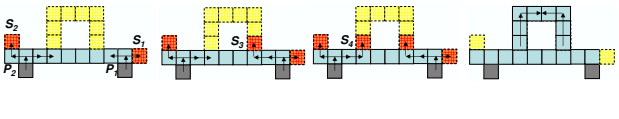
3.2.1 Connect_Ground

The overall goal of this rule is to begin with the ground points of S and to *connect* all of the ground points first with minimal thickness connections. The motivation for this rule is to begin assembly by building a solid base on top of which the rest of the structure can be built. Certain portions of the final structure can be such that they will generate large forces or moments in the structure. The goal of connecting ground points is to assemble the structure in such a fashion that the load will be distributed as evenly as possible, and that the various ground points will provide parallel support. The impact of this function is limited, because it stops assembling as soon as all ground points are connected, and does not force the chosen paths to be new.

3.2.2 Connect_Ground_Local

The basic idea of this algorithm is to find each ground point's closest unassembled neighbors and connect them. This function can be called at any point in the assembly process. At each iteration, each ground point's closest unassembled neighbors are found. These blocks are called start points. The shortest entirely new path that connects two of these points is then assembled, as long as the two start points are associated with different ground points.

An example of this process is shown in Figure 7(a), where solid blocks represent blocks that are assembled, and the blocks with dashed lines are unassembled. The arrows show the ink-flow-like progression searching from neighbor to neighbor until start points are found. The start points are shown in red (darker gray), and there are two start points for the ground point $P1$, and one for the ground point $P2$. It should be noted that these blocks are not yet assembled. Figure 7(b) shows the ink-flow-like search from block to block that finds the shortest new path between start points. Once



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

Fig. 8. An Example of Assembly Using Connect_Ground_Local_Recursive.

the two flows meet, the minimum length new path is established and assembled by a mid-level rule.

The main motivation for this improvement on Connect_Ground is two-fold. First, it aims to provide better selection of the order in which the ground points are connected to one another by focusing only on entirely new blocks that could be assembled. A connection of length 1 is always preferred if possible. Secondly, this method allows the algorithm to keep attempting to close connections after all ground points are connected. **This is desirable because the concepts of load distribution and parallel support are not unique to the blocks immediately around the ground blocks.**

3.2.3 Connect_Ground_Local_Recursive

This rule (CGL_R) is almost identical to Connect_Ground_Local. The difference occurs when no new path can be found that connects the start points, and the previous rule would stop. When no path can be found, this rule seeks to find some new additional start points by taking the ground point(s) whose furthest start point is the closest to ground, and finding its next closest unassembled neighbors. This process is repeated until a path is found or it becomes evident that there is no such path because no new start points can be added.

Figure 8 shows a simple example of this process where there are only two ground points, $P1$ and $P2$. Figure 8(a) shows (in red - or darker gray) the start points corresponding to those ground points, $S1$ and $S2$, that are respectively at a distance of 2 and 3. Blocks with dotted lines are unassembled. There is no entirely new path connecting $S1$ and $S2$, so a search for new start points begin. Eventually, start points $S3$ and $S4$ corresponding to $P1$ and $P2$ are found, and the entirely new path connecting $S2$ and $S3$ is assembled, as shown in Figure 8(d).

The main motivation for this function was to improve on the shortcomings of Connect_Ground_Local by looking for new start points until a new connection can be found. This is desirable because, as mentioned in Section 3.2.2, it is advantageous to attempt to connect the various parts of the structure to one another due to parallel support and load distribution.

Unexpectedly, the algorithm also built some branches that seemingly do not connect any ground points. The basic reason is that while these points connect two start points, this connection is not part of a non-redundant path.

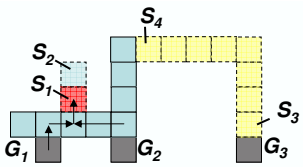


Fig. 9. Illustration of the Motivation for Path Uniqueness.

3.2.4 Connect_Ground_Local_Recursive_Unique

This rule (CGL_RU) is virtually identical to CGL_R, except that a redundancy check was introduced. Consider a potential path from one start point, $S1$, corresponding to ground point $G1$, to another start point, $S2$, corresponding to ground point $G2$, being considered for assembly. **It then checks to see whether there is a minimal-length path from P1 to G1, and a minimal-length path from P2 to G2, such that no block is in both paths.** If so, the path is non-redundant and is acceptable. If all minimal paths from $P1$ to $G1$ and from $P2$ to $G2$ must go through a common point, then connecting $P1$ and $P2$ is not considered to be connecting $G1$ and $G2$, and the connection is unacceptable.

In Figure 9, CGL_R selects $S1$ as the shortest path that connects start points $G1$ and $G2$. In reality however, adding $S1$ does not do much in terms of actually connecting the two ground points. Specifically, there is no path that goes from $G1$, to $S1$, and then to $G2$ without having to go through the same point multiple times. That is why this rule ensures that there is in fact a non-redundant path, that is likely to improve the static conditions. In this example, once it is established that adding $S1$ alone is not truly connecting two ground points, the program adds $S4$ as a new start point for $G2$, at which point the yellow blocks are assembled to connect $S3$ and $S4$. Therefore, **by ensuring that redundant paths are not accepted, CGL_RU decides not to assemble** $S1$ **or** $S2$**, but rather the yellow portion of the structure which actually connects ground points** $G2$ **and** $G3$**.** This algorithm does not assemble "superfluous" blocks such as blocks $S1$ and $S2$ (in Figure 9). These blocks will typically only worsen static conditions, and it is preferable to only add them at the end, rather than in previous steps when the structure might be weaker. The price to pay for this feature is that the redundancy check routine can be time consuming during the assembly sequence generation sequence, depending on how long it is allowed to run in bad situations. The situations in which it will be bad will be those in which there is a large number of paths between the ground points and the start points.

3.2.5 CGL_RUC

The novelty of Connect_Ground_Local_Recursive_Unique_and_Close is that instead of finding the new path between any two acceptable start points that is the shortest, this function finds the new path that is closest to ground. This algorithm combines the advantages of CGL_RU and Assemble_D_Local. Indeed, the idea of connecting the ground blocks as often as possible is very powerful. By assembling paths that are closer to ground, this rule aims to incorporate all of the benefits of assembling the closest blocks to ground first, as discussed in Section 3.1.2, while ensuring that new ground block connections are provided at every step. The behavior is thus more intuitive, and it adds the benefits of assembling according to distance. This rule was found to be the overall best performer.

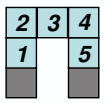


Fig. 10. A Tiny Structure Illustrating Random Assembly.

3.3 Random Assembly / Disassembly

Two random rules were also written. Assemble_Random maintains a list of all blocks that could be assembled at any step, and picks a block at random. Disassemble_Random disassembles the structure at random by selecting blocks at random amongst the feasible candidates. The disassembly sequence that is obtained is reversed to obtain an assembly sequence. The motivation for random assembly and disassembly is to establish references against which the other routines can be compared.

Surprisingly, random assembly outperformed random disassembly by a large margin. The reason for lower moments and forces can be understood by examining Figure 10. By disassembling at random, any of the five blocks that are numbered on the figure are equally likely to be removed first, meaning that each block has a 20% chance of being assembled last. In contrast, with Assemble_Random, once the percentages are carried out, there is only a 6.25% chance each that blocks 1 or 5 are assembled last, a 25% chance each that blocks 2 or 4 are assembled last, and a 37.5% chance that block 3 is assembled last.

Any block that neighbors a ground block is an equal-odds candidate for assembly every time a block is selected for assembly, therefore the overall odds of it getting picked early in the assembly process are excellent. On the other hand, if a block is at a distance of seven for instance, it is only candidate for assembly if an entire path of at least six blocks that happen to connect it to ground have been assembled. It is therefore likely that it will take much longer for this candidate to be selected. In the long run, **blocks that have the largest distance from ground are more likely to be assembled later in the sequence**, and Assemble_Random's behavior on average will resemble Assemble_D. In practice, any single run of Assemble_Random may or may not resemble Assemble_D at all. Disassemble_Random's behavior could be described as "more"' random, and it is generally suggested as the baseline against which to compare the assembly sequences.

4 Numerical Results

This section presents a sample of the numerical results from the simulations. The maximum forces and moments that occur at a connection during the entire assembly se-

quence were calculated. All 27 rules that were developed were tested on 8 test structures. Therefore for any given structure being assembled by one particular rule, the output is the largest moment and force that occur at any connection during the entire assembly. In a real system, if this force or moment is larger than a certain value, the connection will snap.

In order to facilitate the understanding of the results, they are presented as relative results (see Equation (1) below). For instance, CGL_RUC's maximum force in assembling the "Paper" structure was 0.447256 N, the lowest of all rules. It is referred to as F_{best} for that structure. Disassemble_D_Solid_Local generates a maximal connection force of 1.909921 N. Therefore, the maximum force encountered by Disassemble_D_Solid_Local relative to the best case will be as shown in Equation (1):

$$F_{rel} = \frac{F_{max} - F_{best}}{F_{best}} = \frac{1.909921 - 0.447256}{0.447256} = 327\%. \quad (1)$$

These relative results are much easier to read than raw forces, and more significant in comparing the various rules. The relative moments are also calculated in the exact same manner.

In an attempt to summarize all of these results, the average relative force and moment for each rule (over all structures) is presented in Table 1. This number is obtained by averaging the values for a given rule over all structures. Occasionally a rule failed to yield a result for some of the structures due to a dead-end sequence occurring. For these rules, the average of all successful cases is taken. This is not a major issue as this problem did not arise for any of the top performing rules. Each rule's ranking amongst the 27 rules according to this criterion is also presented. It should be noted that all rules that were tested are shown in Table 1, even though there is not enough space to describe them all. The six top ranked results for each column are shown in a bold, red font so that the best performers can be found easily among the 27 rows.

Table 1. Average and Worst-Case Moments and Forces, Relative to the Best Case For Each Structure.

	Relative Moments		Relative Forces	
	Average	(rank)	**Average**	(rank)
MID-LEVEL:				
assemble_random	367%	24	208%	24
disassemble_random	6927%	27	3026%	27
assemble_z	961%	26	780%	26
disassemble_z	387%	25	262%	25
assemble_z_new	87%	14	140%	19
assemble_z_solid	115%	16	182%	23
disassemble_z_solid	199%	20	144%	21
assemble_solid_z	74%	12	132%	16
disassemble_solid_z	338%	23	119%	15
assemble_solid_d	40%	8	62%	7
disassemble_solid_d	94%	15	**46%**	**3**
assemble_d_z	**30%**	**5**	65%	8
disassemble_d_z	**26%**	**3**	65%	9
assemble_d_solid	**30%**	**6**	**53%**	**4**
disassemble_d_solid	**27%**	**4**	**54%**	**6**
HIGH-LEVEL, WITH ASS._Z_NEW:				
connect_g_dist	76%	13	80%	12
connect_g_short	139%	18	147%	22
connect_g_low	149%	19	144%	20
Connect Ground Thick	116%	17	110%	14
C_G_L_R	210%	22	134%	18
C_G_L_R_U	73%	11	77%	11
C_G_L_R_L_C	67%	10	85%	13
C_G_L_R_U_C	62%	9	68%	10
HIGH-LEVEL, WITH ASS._D_SOLID:				
C_G_L_R	202%	21	132%	17
C_G_L_R_U	36%	7	**30%**	**2**
C_G_L_R_L_C	**24%**	**2**	**54%**	**5**
C_G_L_R_U_C	**7%**	**1**	**29%**	**1**

Two results are very clear from the results in this table. First, **intelligent rules clearly outperformed random assembly sequences.** Disassemble_Random encounters a maximum moment that is over 70 times larger than the best case assembly sequence on average, and a maximum force that is over 30 times larger than the best case sequence on average.

Secondly, **CGL_RUC performed the best out of all rules.** It had the best average moment, only 7% above the best case on average, and the best average force, only 29% above the best case. In fact, there are only two structures out of 8 where the maximum force or moment is more than 5% above the best case scenario.

As far as mid-level rules are concerned, Assemble_D_Solid_Local and Disassemble_D_Solid_Local were the best performers. Both rules were amongst the best four average relative moments for mid-level rules, where the top four ranged from 26% to just 30%. The results for both rules were close to identical, and both rules outperformed the other mid-level rules. The assembly sequences that were generated by these two rules are virtually identical because distance is the primary criterion, as explained in Section 3.1.3. The only reason there are slight differences in some of the results is because of the secondary criterion (number of neighbors), or the implicit third criterion (new neighbors last in assembly, see [18] for details). Overall, the algorithms perform virtually identically, but the disassembly algorithm would be preferred because it guarantees that no dead-ends due to accessibility or connectivity will occur. Unfortunately, the disassembly rule was only created late in the process as a check, therefore the assembly rule is the one that was implemented in conjunction with the high-level rules. Fortunately, using either algorithm should make little to no difference for static performance.

It should be noted that because of the time it takes to statically evaluate each assembly sequence, only eight main structures were tested thoroughly. Because of this, the actual significance of the numbers shown in Table 1 is somewhat diminished, and it is important to look at the individual structures in order to understand why certain rules outperformed others. Space limitations do not allow this detailed analysis to be presented here.

In particular, CGL_RUC performed excellently on every single structure except for the so-called inverted pyramid, for which it generated a maximum moment that was three times larger than the best rule for this structure. However, it should be noted that the reason that CGL_RUC performed poorly on this structure is so specific and unique to this structure (see [18]), that it does not prevent recommending it as the single rule to be used on all structures.

5 Synthesis

This section presents the overall conclusions for rule selection taking different priorities into account, e.g. speed, avoiding deadlocks, etc.

5.1 Overall Best Rule

As discussed above, the rule that performed best overall by far was the high-level rule **CGL_RUC**, described in Section 3.2.5. This function was implemented very successfully with Assemble_D_Solid_Local as a mid-level rule. Ideally, it should be implemented with **Disassemble_D_Solid_Local**, whose performance is virtually identical, except that deadlocks due to accessibility are less likely to occur.

The basic reasons that CGL_RUC implemented with Assemble or Disassemble_D_Solid_Local is so successful are:

- Connecting ground points repeatedly leads to better load distribution and parallel support. Assembling the closest connections to ground first improves this process.
- By checking for unique paths, it is ensured that only "real" connections are included, and not groups of blocks on the periphery that do not help distribute the loads. These superfluous blocks will be added last.
- While connecting these ground points, only a minimal thickness connection is established at any given time, and additional blocks that would only worsen static conditions are not included.
- By actually assembling the connections according to distance first, the largest intermediary cantilevers are generally reduced, as explained in Section 3.1.2.

Therefore, the rule that should be selected first to assemble a structure is Connect_Ground_Local_Recursive_Unique_and_Close, implemented with the mid-level rule Disassemble_D_Solid_Local.

5.2 Best Rule to Guarantee No Deadlocks

While deadlocks due to accessibility or connectivity are unlikely with CGL_RUC applied with Disassemble_D_Solid_Local, they are nevertheless possible. It is therefore necessary to have a rule for this case. It is known that any mid-level disassembly rule can always generate a full assembly sequence with no accessibility or connectivity problems. Disassemble_D_Solid_Local was by far the best performing disassembly rule. **For this reason, Disassemble_D_Solid_Local should be implemented when deadlocks are encountered and it is necessary to guarantee a full solution with no deadlocks in assembly.**

Table 2. Assembly Sequence Generation and Static Testing Time for Inv_Pyramid.

	CPU Time (s)	Actual Time (s)
Assemble_D_Solid_Local	4.99718	7.00537
Disassemble_D_Solid_Local	6.76973	8.73539
CGL_RUC with Assemble_D_Local	147.42198	152.60900
Static Beamtest on Inv_Pyramid	57.70297	101.15630
Static Evaluation of a Full Assembly Sequence for Inv_Pyramid	6981.40877	10442.76788

5.3 Best Rule when Assembly Sequence Generation Time is Problematic

It should be noted that one issue with CGL_RUC is that it takes more time to generate a sequence, mostly because of having to test for unique paths. Table 2 shows a few relevant times related to generating and testing the sequence for Inv_Pyr, which is the largest structure considered, with 200 blocks.

Generally speaking, all mid-level rules operated rapidly. The disassembly rules were slightly slower because they have to test the connectivity before removing any block. Table 2 shows the results for the best assembly rule, the best disassembly rule, and the best high-level rule. These tests were performed on a machine with a Mobile Intel Pentium 4, 2.8 GHz processor, and 512Mb of RAM. The assembly rule took 5 seconds of CPU time, the disassembly rule took nearly 7 seconds, and the high-level rule took 147 seconds, or slightly less than two and a half minutes. While the time that is taken by the high-level rule seems very large, one should consider that the static evaluation of the whole sequence took nearly 2 hours. **Therefore, even for this slowest rule, the sequence generation is performed 47 times faster than the necessary static testing of all 200 steps.** This confirms once again why methods that measure the static conditions of multiple options before selecting a block for assembly are not acceptable.

If one were to decide that implementing the higher-level rules was too time consuming and that faster routines were necessary, the suggested option is Assemble_D_Solid_Local. This function performs the best out of the mid-level assembly rules.

5.4 Unacceptable Static Conditions

The goal of this method is that by selecting smart rules to assemble the structure, it is very likely that the sequence can

be generated before being fully tested statically once. However, it can happen that the sequence that is selected fails the static testing, meaning that at least one connection will break during the assembly. In such a case, another rule should be applied to generate another assembly sequence that will pass the static test. The following are the top four recommended rules, in order:

1. **Connect_Ground_Local_Recursive_Unique_and_-Close implemented with Disassemble_D_Solid_Local as the mid-level rule**.
2. **Disassemble_D_Solid_Local**.
3. **Assemble_Solid_D_Local**: This rule was the best rule of the rules whose primary criterion was the number of neighbors.
4. **Disassemble_Solid_Z_Local**: This rule outperformed all other rules for the peculiar Inv_Pyramid structure (see [18]). It is not generally speaking a very effective rule. However, in the extremely unlikely event that the first three rules have failed to assemble a particular structure, it should be tested.

6 Conclusions

This work's approach is to intelligently generate the entire sequence based on the geometry of the structure, and test the static stability of each step once the sequence is generated. The behavior of many rules was analyzed through theoretical, observational and numerical methods. Many patterns of behavior were identified, and related to structures where such methods would succeed or fail. The anticipated result was that a variety of rules would need to be applied for a wide range of scenarios. However, CGL_RUC performed so well that the overall conclusion is that the most efficient method is to apply this high-level rule to attempt to generate all assembly sequences. Alternative rules are presented for the cases where accessibility or static stability conditions can not be met by the sequence in question.

References

[1] Wurst, K., 1986. "The conception and construction of a modular robot system". In Proceedings of the 16th Int. Sym.Industrial Robotics (ISIR), pp. 37–44.

[2] Benhabib, B., Cohen, R., Lipton, M., and Dai, M., 1992. "Conceptual design of a modular robot". *ASME Journal of Mechanical Design,* **114**, pp. 117–125.

[3] Chen, I.-M., and Burdick, J., 1998. "Enumerating non-isomorphic assembly configurations of a modular robotic system". *International Journal of Robotics Research,* **17**(7), pp. 702–719.

[4] Pamecha, A., Ebert-Uphoff, I., and Chirikjian, G., 1997. "Useful metrics for modular robot motion planning". In IEEE Transactions on Robotics and Automation, Vol. 13-4, pp. 531–545.

[5] Rossignac, J., Allen, M., Book, W., Glezer, A., Ebert-Uphoff, I., Shaw, C., Rosen, D., Askins, S., Bai, J., Bosscher, P., Gargus, J., Kim, B. M., Llamas, I., Nguyen, A., Yuan, G., and Zhu, H., 2003. "Finger sculpting with digital clay: 3d shape input and output through a computer-controlled real surface". *Shape Modeling International*, pp. 229–231.

[6] Bosscher, P., and Ebert-Uphoff, I., 2003. "Digital clay: architecture designs for shape-generating mechanisms". In Proceedings of 2003 IEEE International Conference on Robotics and Automation, Vol. 1, pp. 834–841.

[7] Zhu, H., and Book, W. J., 2003. "Control concepts for digital clay". In 2003 IFAC Symposium on Robot Control (SyRoCo).

[8] Homem de Mello, L., and Sanderson, A., 1991. "Representation of mechanical assembly sequences". *IEEE Transaction on Robotics and Automation,* **7**(2), April, pp. 211–227.

[9] Bourjault, A., 1984. "Contribution a une approche methodologique de l'assemblage automatise: Elaboration automatique des sequnces automatiques". PhD thesis, L'Universite de Franche-Comte, France, Nov.

[10] De Fazio, T. L., and Whitney, D. E., 1987. "Simplified generation of all mechanical assembly sequences". *IEEE Journal Of Robotics And Automation,* **RA-3**(6), Dec.

[11] Wilson, R., 1995. "Minimizing user queries in interactive assembly planning". *IEEE Transactions on Robotics and Automation,* **11**(2), April, pp. 308–312.

[12] Ames, A., Calton, T., Jones, R., Kaufman, S., Laguna, C., and Wilson, R., 1996. "Lessons learned from a second generation assembly planning system". In Proceedings of the 1996 IEEE International Conference on Robotics and Automation, pp. 41–47.

[13] Kaufman, S., Wilson, R., Jones, R., Calton, T., and Ames, A., 1996. "The Archimedes 2 mechanical assembly planning system". In Proceedings of the 1996 IEEE International Conference on Robotics and Automation, Vol. 4, pp. 3361–3368.

[14] Homem de Mello, L., and Sanderson, A., 1989. "A correct and complete algorithm for the generation of mechanical assembly sequences". *IEEE Transactions on Robotics and Automation,* **1**, May, pp. 56–61.

[15] Homem de Mello, L. S., 1995. "Sequence planning for robotic assembly of tetrahedral truss structures". *IEEE Transactions On Systems, Man, and Cybernetics,* **25-2**, Feb.

[16] Homem de Mello, L., and Sanderson, A., 1990. "Assembly planning for large truss structures in space". In IEEE International Conference on Systems Engineering, pp. 404–407.

[17] Nilsson, N., 1980. *Principles of Artificial Intelligence*. Tioga.

[18] Wolff, S., 2006. "Statically stable assembly sequence generation and structure optimization for a large number of identical building blocks". PhD thesis, George W. Woodruff School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, Georgia.

[19] Wolff, S., and Ebert-Uphoff, I., 2006. "Preliminary results on generating assembly sequences for shape dis-

play". In Proceedings of the ASME International 26th Computers and Information in Engineering Conference (CIE). Paper number DETC2006-99233, Sept 2006.