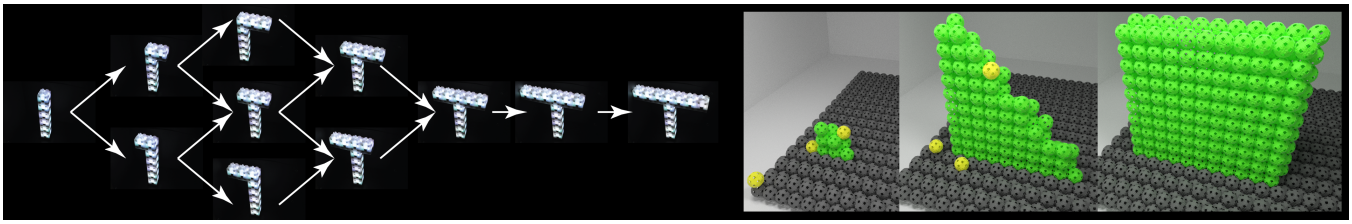
GAPCoD: A Generic Assembly Planner by Constrained Disassembly

Florian Pescher
FEMTO-ST Institute, UBFC, CNRS
Montbéliard, France
florian.pescher@femto-st.fr

Nils Napp
Cornell University
Ithaca, NY, USA
nnapp@cornell.edu

Benoît Piranda
FEMTO-ST Institute, UBFC, CNRS
Montbéliard, France
benoit.piranda@femto-st.fr

Julien Bourgeois
FEMTO-ST Institute, UBFC, CNRS
Montbéliard, France
julien.bourgeois@femto-st.fr



ABSTRACT

In the literature we can find many kinds of modular robot that can build a wide variety of structures. In general, finding an assembly order to reach the final configuration, while respecting the insertion constraints of each kind of modular robot is a difficult process that requires system-specific tuning. In this article, we introduce a generic assembly planner by constrained disassembly (GAPCoD) which works with all kinds of modular robots. It outputs a directed acyclic graph where vertices are modules needing to be placed before his child nodes. This graph is obtained through the disassembly of the desired structure submitted to user chosen constraints. We detail the compiler as well as the way to choose constraints and their influence on performance. The robots embed simple path planning algorithm to reach the destination and act as decentralized agents. Examples are provided to show the possibilities that the compiler offers with two very different robot systems and constraints.

KEYWORDS

Networked systems and distributed robotics; Multi-robot systems

1 INTRODUCTION

Analyzing assemblies and encoding assembly sequences has been explored since the 1980s, and an algorithm for generating the complete set of partial assemblies and assembly sequences, represented as and-or graphs, exists. However, the scaling of the algorithm depends on several problem-specific factors, but is exponential in most cases [8]. In robotic construction and modular robot reconfiguration, instead of assembling larger sub-assemblies, components are typically added one by one and many difficulties arise from the motion of active robots, which is not represented in and-or graphs. As such, robotic reconfiguration systems mostly develop their own planning algorithms and plan representations. In robotic assembly, plans for reconfiguration are often co-developed with a particular set of hardware, and it is difficult to compare between systems [4, 7, 13, 16]. There are a few abstract models for robotic reconfiguration, but while they are typically based on an original robotic system [1, 9], it is often not easy to build robots that fit them. However, they allow for the development of generic reconfiguration algorithms, e.g. [6] and the references therein describe recent developments. Instead of solving a motion problem, these strategies build an intermediate (porous lattice) structure to facilitate motion [14]. In robotic construction [4, 7, 17], i.e. active robots manipulating passive components, navigating intermediate structures is paramount, and this is often achieved by structuring the travel paths/directions of robots so that they do not collide. Here, we adapt ideas from planners for the *TERMES* [17] systems to lattice-based robotic systems, such as quasi-spherical *3D Catoms* [12] in the context of the polymer molding application [10] and cubic *Blinky Blocks* [5] with balance constraints. The goal of the *TERMES* compiler is to make the runtime execution distributed and requires essentially no information exchange between robots at runtime (beyond locally avoiding collision). Assembly checks can be fully determined by the local state of the structure. The planning process is termed "compilation" since it prepares a high-level target for execution by the runtime system (i.e. the robots and raw materials) which do not change. Recent work has significantly increased the speed and scalability of the *TERMES* compiler [2]. The key idea in this compiler is to use a partial order reduction, a technique in formal verification to combat state-space explosion [3]. Some

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 2020, Auckland, New Zealand

https://doi.org/doi

states are not explicitly enumerated, since they can be reached by a combination of independent actions, instead of checking the 2^N possible combinations of N actions, only N independent actions need to be checked. This idea is exploited implicitly in many assembly planning algorithms in the literature, e.g. [15], by asserting that some actions can happen parallel, i.e. their order does not matter. Instead of focusing on a particular hardware platform or type of motion, connection, or stability constraint, we focus on the way the states are enumerated and order dependencies captured, and show that these ideas can be applied to a variety of systems. The resulting assembly orders are directed acyclic graphs (DAGs), which can be interpreted as a partial order on the assembly steps. This representation can only capture a subset of the full states and the compiler commits to specific order choices when it encounters them. Within these restrictions it attempts greedily find orders that allow for as many parallel steps as possible.

2 THE *GAPCOD* PLANNER

We are now going to present *GAPCoD*, a Generic Assembly Planner by Constrained Disassembly. This planner produces an assembly order schedule according to user-defined constraints by disassembling the desired target shape. First, we show the general workflow of *GAPCoD*, then we explain how *GAPCoD* Python compiler part works, then we detail how to implement different constraints for different physical modules or rules of assembly, and finally we discuss the complexity and scalability of *GAPCoD* and its dependency on the physical constraints. The key idea is that the logic for exploring and ordering states can be largely separated from the details of why constraints exist and how they are checked for a particular robot system.

2.1 *GAPCoD* workflow

GAPCoD is composed of two main parts: a physical part, whose goal is to check the constraints for each modules, and a logical part in charge of producing the output DAG. In our implementation, the physical part is handled by the *VisibleSim* [11] simulator allowing us to simulate constraints for different kinds of modules with ease, and the logical part is handled by a Python compiler which is an adaptation of the *TERMES* compiler [2, 17]. The general workflow of the compiler as well as the interactions between *VisibleSim* and the Python compiler can be seen in Figure 1. The detailed physical state is only maintained in the simulator, while the state-traversal logic keeps track of candidate locations in the assembly to check next.

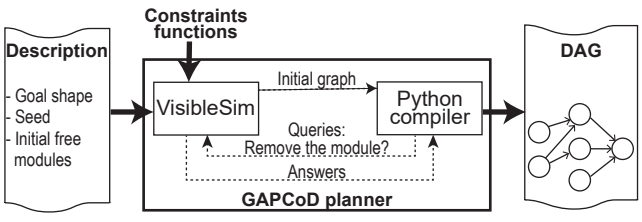


Figure 1: Overview of *GAPCoD* workflow

In order to plan an assembly order with *GAPCoD*, we must first determine and implement the constraints that will be applied in our scenario. More details about how to implement constraints can be found in Section 2.3. That part is the specific tuning one must perform in order to use *GAPCoD* with different kinds of systems. Once the constraints are implemented, we can start the planning for our structures by giving *GAPCoD* a shape description and seed modules, i.e. the last and first (free) modules to be disassembled. In our current implementation, we limited ourselves to a single initially free module.

2.2 Python compiler explanation

The compiler computes a DAG in which the roots are the seed modules and every vertex represent a module that can only be assembled if all of its parents modules have already been assembled. This graph is created and updated by disassembling the complete structure that we are trying to find an assembly order for. All sets and variables used by the compiler are described in Table 1.

Symbol	Description
C	Set of connected modules
$G(V, E)$	Assembly graph where V is the set of vertices and E is the set of edges
S	Set of seed modules
R, R'	Sets of removable modules
BL	Set of modules blocking the removal of another module when removes
BL_i	Set of modules blocked by the removal of module $i \in BL$
D	Set of disassembled modules
Dep_i	Boolean: True if there is an edge pointing towards the node representing module $i \in D$ in the graph False otherwise
n, n', m, k	Lower case single characters represents a module

Table 1: Table of sets and variables used by the compiler

Algorithm 1 shows the global process of the graph creation until the structure has been fully disassembled. In this section, whenever we refer to a removable or free module, we are talking about a module that can be disassembled without breaking any of the constraints the compiler has been given. Let *isRemovable(k,C)* be the function that returns *true* if the module k can be removed from C without breaking any constraints (and *false* otherwise).

Algorithm 1: Creation of assembly order DAG

while $C \neq \emptyset$ **do**
  $\{R, BL\} \longleftarrow$ CheckBlockages(C,R);
  $\{C, R, D, E, Dep\} \longleftarrow$ UpdateDependencies(R,C,E,S);
  $\{C, R, E\} \longleftarrow$ CreateEdges(C,R,D,E,BL,Dep);
  $R \longleftarrow R \cup BL$;
return $G(V, E)$

The compiler is based on two main functions called all along the disassembly of the structure: Disassemble(n,C,R) that disassemble

a module n, updating both the list of removable modules R and the structure C (detailed in Algorithm 2), and Reassemble(n,C,R) that reassemble a disassembled module n, updating both the list of removable modules R and the structure C (detailed in Algorithm 3).

Algorithm 2: Disassembly process of a module

Function *Disassemble(n,C,R)*:
  $C \longleftarrow (C - n)$;
  $R \longleftarrow (R - n)$;
  forall $k \in C$; k *is an influence neighbor of* n **do**
    if $k \in R$ **then**
      if *not isRemovable(k,C)* **then**
        $R \longleftarrow (R - k)$;
    if *isRemovable(k,C)* **then**
      $R \longleftarrow (R \cup k)$;
  return $\{C, R\}$;

Algorithm 3: Reassembly process of a module

Function *Reassemble(n,C,R)*:
  $C \longleftarrow (C \cup n)$;
  $R \longleftarrow (R \cup n)$;
  forall $k \in C$; k *is an influence neighbor of* n **do**
    if $k \in R$ **then**
      if *not isRemovable(k,C)* **then**
        $R \longleftarrow (R - k)$;
    if *isRemovable(k,C)* **then**
      $R \longleftarrow (R \cup k)$;
  return $\{C, R\}$;

The disassembly of the currently free modules can be divided into three main steps: Checking blockages (Algorithm 4), finding and updating the dependencies (Algorithm 5) and creating required additional edges (Algorithm 6). One execution of each of those three steps will be called a "level" in the rest of the paper. The algorithms are detailed in the following paragraphs.

2.2.1 Step 1: Checking blockages. Before actually removing a module from the structure, we must ensure that the removal of this module will not prevent the removal of another module trying to disassemble itself this level. The reasons for those blockages depend on the particular constraints. Some examples would be the removal of a module used as a pivot for the movement of another one (as it can be seen in Figure 2), or making another module the only module responsible for maintaining the connectivity in the structure. Detecting and preventing those blockages according to the constraints given to the compiler is the goal of Algorithm 4.

When such a blockage is detected the blocker module is added to a list called BL. The modules in BL will be considered as not free for the rest of the current level. Although considering the blocker modules as not free until the next level solve most problems, we must still be careful about the case of two modules blocking each other, causing a deadlock. When a deadlock happens, the first module tested for blockage is considered free again, thus giving it priority for disassembly. An example showing how this interblocking situation is handled is given in Figure 6.

Algorithm 4: CheckBlockages(C,R): check blockages and remove interlocks

Function *CheckBlockages(C,R)*:
  $BL \longleftarrow \emptyset$;
  $\forall i \in C; BL_i \longleftarrow \emptyset$;
  forall $n \in R$ **do**
    $\{C, R'\} \longleftarrow$ Disassemble(n,C,R);
    if $(\exists m \neq n; m \in R \wedge m \notin R')$ **then**
      $BL \longleftarrow BL \cup n$;
      $BL_n \longleftarrow BL_n \cup m$;
      if $(BL_m = \{n\})$ **then**
        $BL \longleftarrow BL - m$;
    $\{C, R\} \longleftarrow$ Reassemble(n,C,R');
  $R \longleftarrow R - BL$;
  return $\{R, BL\}$;

2.2.2 Step 2: Finding and updating the dependencies. Considering that the list of removable modules has been updated to prevent blockages, we can now start actually disassembling this level of the structure and updating the output graph (cf. Algorithm 5). During this step, we disassemble all the free modules of this level and whenever a new module is freed we create the corresponding edge in the output graph. Some modules might need the removal of several other modules in order to be freed i.e. when we find a newly freed module, we check if the previously removed modules during this level impact the removability of this freed module and create the corresponding dependencies. Figure 4 shows an example of how this combo check is performed on a simple configuration.

This algorithm flags every free module either as *"with dependencies"* or *"without dependencies"* with the exception of seed modules that will not free any other module. An example of a module with no dependencies is given in Figure 5.

2.2.3 Step 3: Creating required additional edges. Algorithm 6 is the last step of the disassembly process during which it constructs the graph G.

During this step, we create additional edges in G between the modules that will still be present at the next level and the ones that were just removed (flagged as *"with dependencies"*). Those additional edges ensure that when we will try to assemble the structure, all the modules that must be present for this assembly actually will be.

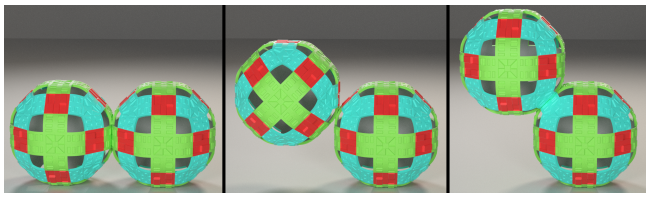


Figure 2: *3D Catom* in rotation around a pivot.

Algorithm 5: Find dependencies between next free modules and current ones

Function *UpdateDependencies(R,C,E,S)*:

$D \longleftarrow \emptyset$;
$\forall i \in C; Dep_i \longleftarrow \emptyset$;
$R' \longleftarrow R$;
forall $(n \in R)$ **do**
  $\{C, R'\} \longleftarrow$ Disassemble(n,C,R');
  if $(\exists m; m \in R' \wedge m \notin R)$ **then**
    $E \longleftarrow E \cup (m \rightarrow n)$;
    $Dep_n \longleftarrow True$;
    forall $(n' \in D)$ **do**
      $\{C, R'\} \longleftarrow$ Reassemble(n',C,R') ;
      if $(m \notin R')$ **then**
        $E \longleftarrow E \cup (m \rightarrow n')$;
        $Dep_{n'} \longleftarrow True$;
      $\{C, R'\} \longleftarrow$ Disassemble(n',C,R');
  else if $(n \notin S)$ **then**
    $Dep_n \longleftarrow False$;
  $D \longleftarrow D \cup n$;
  $R \longleftarrow R'$;
return $\{C, R', D, E, Dep\}$;

Those additional edges ensure the presence of all needed modules during the assembly process.

Algorithm 6: Pushes blocker and no dependencies modules to next level + creates required edges in graph

Function *CreateEdges(C,R,D,E,BL,Dep)*:

forall $(n \in BL)$ **do**
  forall $(m \in D; Dep_m = True)$ **do**
    $E \longleftarrow E \cup (n \rightarrow m)$;
forall $(n \in D; Dep_n = False)$ **do**
  $\{C, R\} \longleftarrow$ Reassemble(n,C,R);
  forall $(m; Dep_m = True)$ **do**
    $E \longleftarrow E \cup (n \rightarrow m)$;
return $\{C, R, E\}$;

Let us show that those edges are necessary to obtain a correct order. For this example we suppose that we have square modules which can only be disassembled/assembled if there is a module or a wall directly connected on their left side and no module above them. We consider the configuration shown in Figure 3 and perform the compiler process on this configuration with (case *A* in grey) and without (case *B*) the call to the *CreateEdges* function described in Algorithm 6.

Considering that all seed modules are placed, if we try to assemble the structure following the graphs, we get 6 order possibilities for case *B*, but only one is correct. Case *A* is more efficient as it gives the only correct solution.

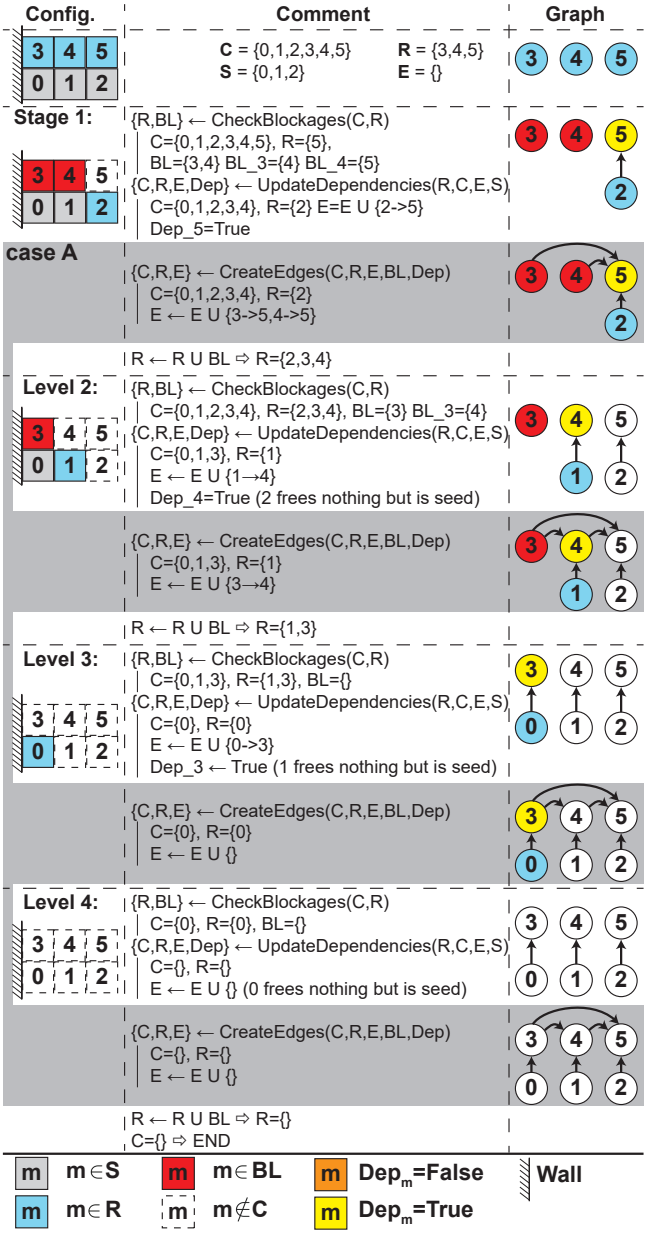


Figure 3: Creating graphs with (Case A in grey) and without Algorithm 6. The middle column shows the operations performed by the algorithm and their influence of the different sets. Left column is a visual representation of those sets showing the configuration at each step of the compiler and right column shows the graph created at each of those steps.

2.3 Implementing new constraints

This section details which kind of constraints may be used with our compiler.

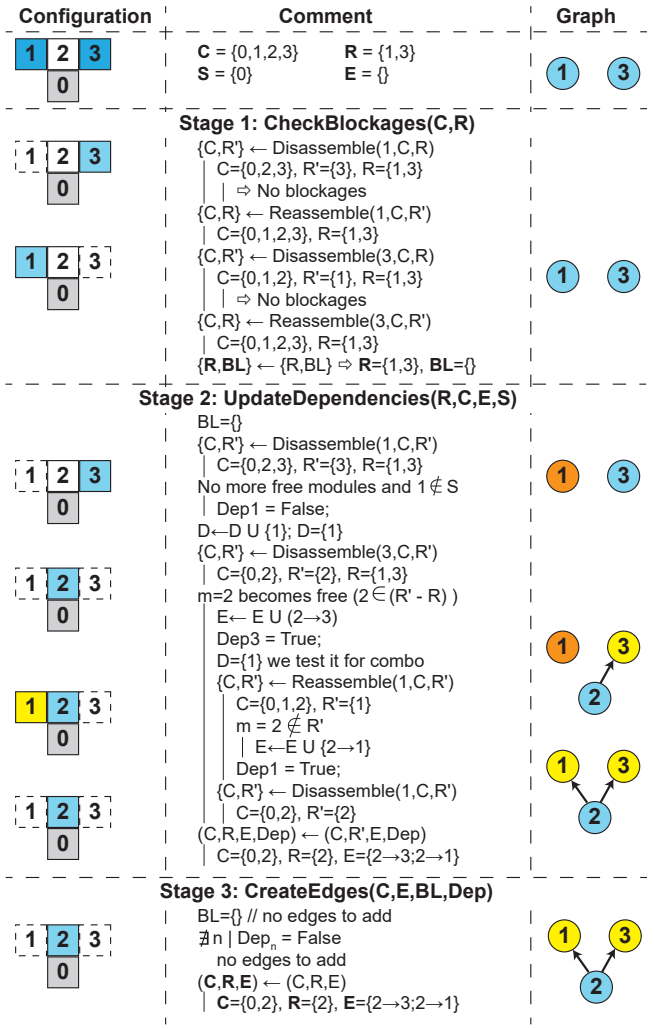


Figure 4: Example 1 illustrating the algorithm. In this example a module can be removed if and only if the structure connectivity is maintained. (Same color scheme as Figure 3.)

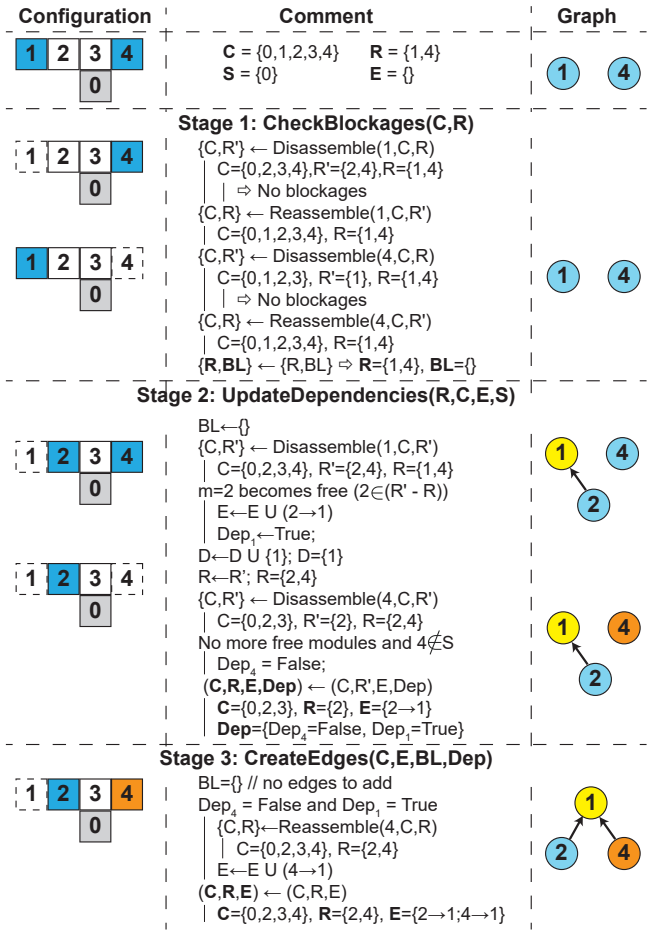


Figure 5: Example 2 illustrating the algorithm. In this example a module can be removed if and only if the structure connectivity is maintained. (Same color scheme as Figure 3.)

A constraint is compatible with *GAPCoD* if it is associated to a function that returns *true* if the module can be removed according to the constraint and *false* otherwise.

All constraints have a few pieces of information that will have an influence on the compiler efficiency:

- The number of influence neighbors. Influence neighbors of a module are the modules that can be affected (meaning modules that can change their removability state) when assembling/disassembling this module according to the considered constraint.
- The complexity of the boolean function checking if the constraint (mobility constraint, balance constraint…) is satisfied.

We are now going to detail some examples of constraints: the connectivity constraint as most modular robotic systems needs their modules to be connected to one another at all times, the *3D Catoms* mobility constraints and the balance constraint of a set of *Blinky Blocks*.

2.3.1 Connectivity. When removing or adding a module according to the connectivity constraint only the physical neighbors are affected. This leads to 6 influence neighbors for modules in a cubic lattice such as the *Blinky Blocks* or 12 influence neighbors for the *3D Catom* that are organized in Face-Centered Cubic (FCC) lattice. Despite having only a few influence neighbors the connectivity constraint is still a global constraint. In order to check if a module m can be removed we need to verify that all modules can be reached from any module $s \in S$ even after the removal of m. This lead to a complexity of $O(n)$ where n is the number of modules in C.

2.3.2 3D Catom mobility constraints. The movement of the *3D Catoms* are subject to a lot of constraints. Indeed to perform a single move we must make sure that the required pivot is present and that a bunch of other positions are free. Theoretical motions of *3D Catoms* [12] give 120 different moves for a *3D Catom*. However in

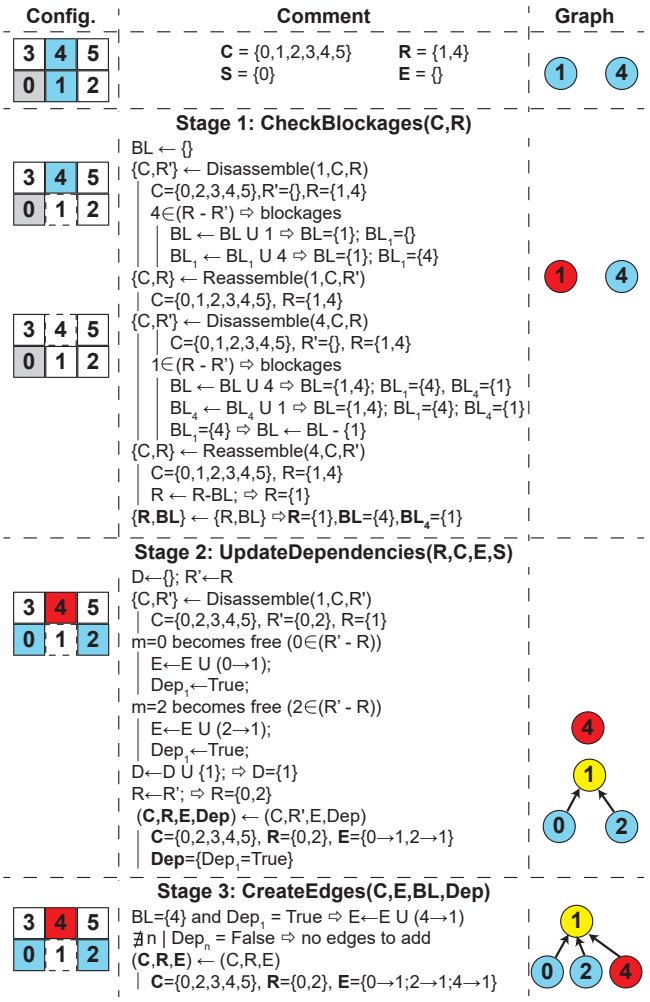


Figure 6: Example 3 illustrating the algorithm. In this example a module can be removed if and only if the structure connectivity is maintained. (Same color scheme as Figure 3.)

the context of the polymer molding application presented in [10] we can reduce this number to 84.

Those moves can all be checked by knowing if 48 specific neighboring cells of the lattice are filled or empty. We can notice that the 12 actual physical neighbors of the *3D Catom* are included in those 48 influence neighbors.

The occupancy requirements (limited to the 12 physical neighbors minus the one above) for all possibles move can be summarized in a 84×11 matrix which when multiplied by an occupancy state of those 11 cells where the value is 0 if the cell is empty and 1 if the cell is filled gives us a vector V_1 of all possible moves. V_1 is a vector of presence of the neighbors (component equal to 1 if the pivot is present and 0 otherwise).

In the same way we can verify the empty cells requirements by multiplying a 84×48 matrix with the opposite of the occupancy state of all 48 influence neighbors (vector V_2).

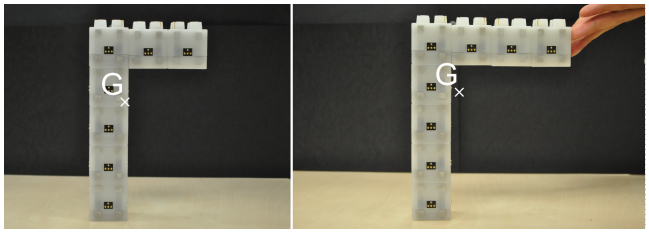


Figure 7: Example for a stable and an unstable configuration for the T-shape stability test

If both occupancy an empty cells requirement are met for a move m ($V_1(m) = V_2(m) = 1$) then the move is possible.

In that way we designed a way to check with a constant complexity if a *3D Catom* is able to move.

2.3.3 Blinky Blocks balance of a T-shapes. The last constraint is the stability of T-shapes like structures limited in the (x, z) plane for the *Blinky Blocks*. In this case, the influence neighbors for this global constraint are all the n modules of C.

In order to check if the structure is stable we first compute the gravity center G of the shape. We consider that the structure is stable if the vertical line passing by G cross the basis of the structure as shown in Figure 7.

The complexity of this check is then the complexity to compute the gravity center of the shape and as such can be done in $O(n)$.

2.4 Complexity

We detailed how the compiler works in Section 2.2 and how constraints for the compiler can be implemented in Section 2.3, allowing us to express the complexity of the compiler depending on the number of modules in the final assembly, n. Let k be the number of influence neighbor and define the highest complexity to check a constraint as $O(M)$.

In the worst case scenario all the n modules are free at a given level but only one is definitely removed during this level, leading to n levels. However, such a system is both difficult to imagine, and would not be a good candidate for this planner, which exploits the fact that parallel actions reduce the number of states that need to enumerated, and that any any given time only a limited number of actions are potentially parallel.

By definition we have $isRemovable(k, C)$ with a complexity of $O(M)$. Knowing that we can now compute that Algorithm 2 and 3 are in $O(kM)$. This allows us to find the complexity for Algorithms 4,5 and 6: Algorithm 4 is in $O(nkM)$, Algorithm 5 in $O(n^2kM)$

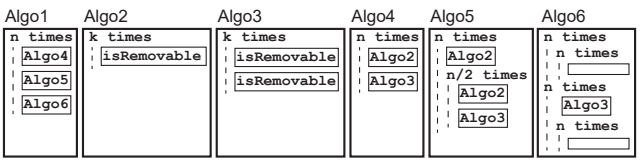


Figure 8: Complexity computation for the algorithms

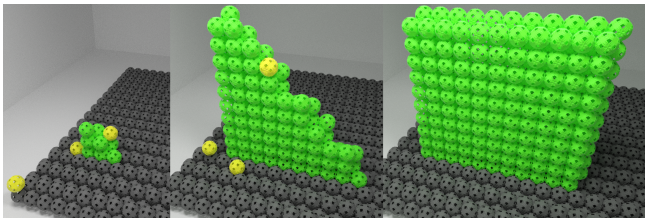


Figure 9: Assembly of a wall of *3D Catoms*.

and Algorithm 6 in $O(n^2)$ or $O(nkM)$ whichever is the highest. Finally this leads to a worst case complexity of $O(n^3kM)$ for the complete compiler described in Algorithm 1.

This complexity seems important but let us not forget than in most cases not all modules will be free during a level and that the compiler will usually not perform as many levels as modules in the structure. Plus the main bottleneck in our compiler is the combo check performed in Algorithm 5, however it is this combo check that yields one of the most important feature of our compiler: the ability to find possible parallel operations that will result in a time gain when assembling the structure. In other words we choose to increase compile time in order to reduce assembly time.

3 EXPERIMENTATION

Finally we are going to use this compiler to find an assembly order in different scenarios with different kind of modules associated to different set of constraints. We study the scalability of those different scenarios and we talk about the complexity of the compiler (using the notations of Section 2.4).

3.1 *3D Catom* assembly

The first scenario we will study is an assembly order with the *3D Catoms* module associated to a mobility constraint (can the *3D Catom* be inserted at a given position of the grid) and a connectivity constraint (the structure must always be connected).

Figure 9 shows some steps of the assembly of a wall shape made of 220 *3D Catoms*. The linked video[1] presents the full construction of the wall.

Let us now compute the complexity of the compiler in this particular scenario. The number of influence neighbors is $k = 48$: We have 12 physical neighbors for the connectivity constraint and 48 influence neighbors, including the 12 physical ones. Then the constraint that is checked with the highest complexity is the connectivity which is done in $O(n)$ against the mobility constraint which is done in $O(1)$, giving us $M = O(n)$. Given that the complexity of the compiler in any scenario is $O(n^3kM)$, in this case we have a worst case complexity of $O(n^4)$.

Let us now compute an approximate experimental complexity to compare with this worst case theoretical one. In order to do so we are going to run the compiler on different size of cubes and check both the time it takes the compiler to produce an order as well as the dept of the produced graph to see the efficiency of our method. The results can be found in Table 2.

[1]YouTube video of the assembly of the wall: https://youtu.be/s_-LyYyVnCU

Nb of modules	Graph Depth	Disassembly time (s)
1	1	0.002279
12	6	0.366118
36	11	5.386122
96	19	67.193171
175	24	263.753336
288	29	990.736958
490	37	3724.907382
704	41	9992.689347
1053	48	23334.396296
1400	55	47485.460037

Table 2: Graph depth and disassembly time for different size cubes made of *3D Catoms*.

If we take the disassembly time for the six smaller cubes and compute the polynomial tendencies as a function of the number of module and compare it to the actual disassembly times we get the graph in Figure 10.

Let us now analyze those results. We can see that the experimental disassembly time is comprised between n^2 and n^3, leading to an experimental complexity of $O(n^3)$ which is much better than the theoretical $O(n^4)$. This means that the compiler performs better than the worst case scenario in this case. This difference is explained by the fact that in the worst case we supposed that all modules were free every level and that we had as many levels (the graph depth) as modules in the structure when in practice that is not the case as shown by the graph depth column in Table 2. The other interesting to notice is that the depth of the output graph does not increase drastically as the number of modules increase, meaning that for bigger structure the compiler finds more possible parallel operations which implies possible time gains when we will try to assemble the structure. In other words we are performing a time expensive computation to produce the assembly graph for a given shape in order to gain time when assembling the structure by maximizing the number of parallel operations.

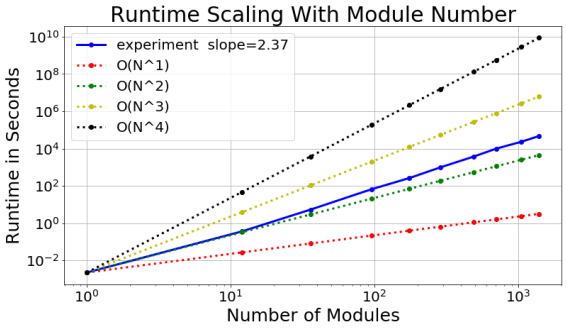
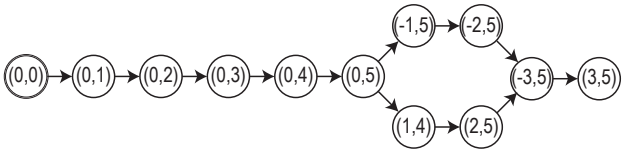
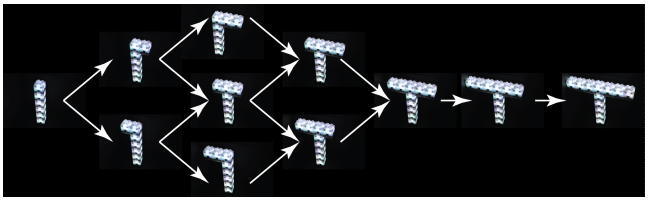


Figure 10: Experimental disassembly times starting from a varying number of modules, see Table 2. The best fit scaling exponent is $O(n^{2.37})$, which is significantly better than the worst case analysis $O(n^4)$.



(a) Graph output for the assembly of the T-shape with *Blinky Blocks*. Basis of the T has coordinate $(0, 0)$ in the (x, z) plan. This graph does not represent all possibilities, only the ones found by the compiler according to the constraints and initial free modules are drawn.



(b) Different possible orders according to the graph.

Figure 11: Compiler output and hardware assembly with *Blinky Blocks* for a T-shape.

3.2 *Blinky Blocks* T-shape example

We will now look into a T-shape assembly order for the *Blinky Blocks* submitted to the mechanical stability constraint described in Section 2.3.3 as well as the connectivity constraint.

We can see the graph generated by the compiler as well as all the possible assembly orders we can find by following this graph for a small example in Figure 11. Although it should be possible to assemble the last module in any branch first, the way the compiler is currently implemented, the structure is disassembled starting by an initial free module chosen by the user. In this example we chose the last module of the right branch as the only initially free module which is the reason why we end up assembling the last module of the left branch before the last module of the right branch.

Let us now compute the complexity of the compiler in this scenario. The number of influence neighbors is $k = n$: We have the 6 physical neighbors for the connectivity constraint, and n for the stability constraint which affects the whole structure. Then both constraints are checked with a complexity of $O(n)$ leading us to $M = O(n)$. Given that the complexity of the compiler in any scenario is $O(n^3kM)$, in this case we have a complexity of $O(n^5)$.

3.3 Discussion

The experiments show that the compiler can produce assembly orders for different kinds of constraints. However, even though the compiler is adaptable there is a cost in scalability, especially for constraints that affect the whole structure and/or need a global check at each step. We decided to prioritize the correctness of the order and the number of possible parallel operation of the assembly plan over the runtime of the planner. As a result the scaling of the planner depends on the hardest constraint check times the number of influence neighbors. In situations where the influence of a module is limited, the worst-case complexity analysis is very conservative and *GAPCoD* is ≈ 1.5 orders lower than the worst case polynomial power. Examples for the *3D Catoms* and the *Blinky Blocks* illustrate that global constraints incur an additional n scaling factor suggesting some loss of efficient scalability. However global constraints can take into account constraints that can not be checked locally, such as the mechanical constraint of stability for T-shapes. This means that when giving constraints to the compiler a tradeoff between adaptability and scalability must be done.
The current implementation does not optimize the removability check in the simulator. However, the way *GAPCoD* traverses assembly states consecutive checks are usually for similar states, which means they are easy targets for optimization by caching some of the repetitive computation, similar to using a spanning tree to speed up connectivity checking in [2].

Another thing to notice in the experiments is that we only used the graph to find a correct assembly order and then sequentially assemble modules. Even though it shows the compiler gives a correct assembly order, the actual assembly does not make use of all the interesting properties of the output graph. Since the graph indicates which modules should be placed before assembling a specific module this allows for some operations to be parallelized once the requirements are met. If we take the graph from Figure 11a as an example, after assembling $(0, 5)$ we can assemble either $(-1, 5)$, $(1, 5)$ or both at the same time in parallel. By maximizing the number of parallel operations we can reduce the time it takes for the assembly to be performed. If we call a time step the maximum time it takes a single module to be placed for a given structure, by maximizing the number of parallel operations we reduce the number of time steps to the graph depth thus reducing the assembly time. In the example from Figure 11 by maximizing the number of parallel operations we can go from 12 to 10 time steps. This speedup is even more impressive in the example of cubes made of *3D Catoms* as can be seen in Table 2 where for a cube made of 1400 modules we go from 1400 time steps to 55 if we perform all possible parallel operations at the same time. This theoretical speedup by maximizing the number of parallel operation performed at the same time seems very promising and will be the object of future works.

4 CONCLUSION

In this paper we introduced *GAPCoD* a Generic Assembly Planner by Constrained Disassembly. We explained how this planner produces a DAG representing possible orders of assembly with as much parallel operations as possible. We then tested this method in two different scenarios showing the adaptability of our method to different constraints and physical modules. Future works will focus on choosing between different possible DAG orders to optimize assembly time as well as testing *GAPCoD* on many other scenarios than the one presented here.

ACKNOWLEDGMENTS

This work was partially supported by the ANR (ANR-16-CE33-0022-02), the French Investissements d'Avenir program, ISITE-BFC project (ANR-15-IDEX-03), Labex ACTION program (ANR-11-LABX-01-01),the Mobilitech project, and the National Science Foundation (NSF #1846340).

REFERENCES

[1] Z. Butler, S. Byrnes, and D. Rus. 2001. Distributed motion planning for modular robots with unit-compressible modules. In *Proceedings 2001 IEEE/RSJ International*

Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180), Vol. 2. 790–796 vol.2. https://doi.org/10.1109/IROS.2001.976265

[2] Yawen Deng, Yiwen Hua, Nils Napp, and Kirstin Petersen. 2019. A Compiler for Scalable Construction by the TERMES Robot Collective. *Robotics and Autonomous Systems* 121 (2019), 103240. https://doi.org/10.1016/j.robot.2019.07.010

[3] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg.

[4] B. Jenett, A. Abdel-Rahman, K. Cheung, and N. Gershenfeld. 2019. Material Robot System for Assembly of Discrete Cellular Structures. *IEEE Robotics and Automation Letters* 4, 4 (Oct 2019), 4019–4026. https://doi.org/10.1109/LRA.2019.2930486

[5] Brian T. Kirby, Michael Ashley-Rollman, and Seth Copen Goldstein. 2011. Blinky blocks: a physical ensemble programming platform. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems (CHI EA '11)*. ACM, New York, NY, USA, 1111–1116.

[6] Jakub Lengiewicz and Paweł Hołobut. 2019. Efficient collective shape shifting and locomotion of massively-modular robotic structures. *Autonomous Robots* 43, 1 (01 Jan 2019), 97–122. https://doi.org/10.1007/s10514-018-9709-6

[7] V Lindsey, Q., Mellinger, D., Kumar. 2011. Construction of Cubic Structures with Quadrotor Teams. *Robotics: Science and Systems VII* (2011). https://doi.org/10.15607/RSS.2011.VII.025

[8] Luiz S. Homem dd Mello and Arthur C. Sandcrson. 1986. AND / OR REPRESENTATION OF ASSEMBLY. In *AAAI Proceedings*. 1113–1119.

[9] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. 1998. A 3-D self-reconfigurable structure. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, Vol. 1. 432–439 vol.1. https://doi.org/10.1109/ROBOT.1998.677012

[10] Florian Pescher, Benoit Piranda, Stephane Delalande, and Julien Bourgeois. 2018. Molding a Shape-Memory Polymer with Programmable Matter. In *Distributed Autonomous Robotic Systems, The 14th International Symposium*. Boulder, Colorado, USA, 65–78.

[11] Benoit Piranda. 2016. VisibleSim: Your simulator for Programmable Matter. In *Algorithmic Foundations of Programmable Matter (Dagstuhl Seminar 16271)*.

[12] Benoit Piranda and Julien Bourgeois. 2018. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robot Journal, Special Issue: 'Distributed Robotics: From Fundamentals to Applications'* 42, 8 (2018), 1619–1633. https://doi.org/10.1007/s10514-018-9710-0

[13] Jungwon Seo, Mark Yim, and Vijay Kumar. 2013. Assembly Planning for Planar Structures of a Brick Wall Pattern with Rectangular Modular Robots. (2013), 1016–1021.

[14] Pierre Thalamy, Benoît Piranda, and Julien Bourgeois. 2019. Distributed Self-Reconfiguration using a Deterministic Autonomous Scaffolding Structure. In *Proceedings of the 19th International Conference on Autonomous Agents and Multi-Agent Systems*. Montreal QC, Canada, 140–148.

[15] M T Tolley, M Kalontarov, J Neubert, D Erickson, and H Lipson. 2010. Stochastic Modular Robotic Systems: A Study of Fluidic Assembly Strategies. *IEEE Transactions on Robotics* 26, 3 (jun 2010), 518–530. https://doi.org/10.1109/TRO.2010.2047299

[16] Thadeu Tucci, Benoît Piranda, and Julien Bourgeois. 2018. A Distributed Self-Assembly Planning Algorithm for Modular Robots. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 550–558. http://dl.acm.org/citation.cfm?id=3237383.3237465

[17] Justin Werfel, Kirstin Petersen, and Radhika Nagpal. 2014. Designing collective behavior in a termite-inspired robot construction team. *Science* 343, 6172 (2014), 754–8.