2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Tree-Based Reconfiguration of Metamorphic Robots

Patrick Ondika[1], Jan Mrázek[1], Jiří Barnat[1]

Abstract— **Metamorphic robots have gained the attention of many researchers due to their ability to change shape and adapt to various tasks. In order to utilize the versatility of metamorphic systems, we need to be able to find a shape-shifting (reconfiguration) plan efficiently; however, finding these plans is challenging due to the high degree of freedom of modular systems. Reconfiguration algorithms proposed so far either scale poorly with a growing number of modules, impose specific restrictions on modules, or produce plans that are unrealistic outside of zero-gravity environments. This paper presents a new approach to the reconfiguration problem of chain-type metamorphic robots. Our algorithm relies on forming tentacles and using them to transport modules, which allows us to search through a reduced state space by computing many smaller planning instances. As a result, we obtain a heuristic solution that is more scalable than optimal planners, while producing realistic plans that impose no specific module requirements.**

I. Introduction

Since the start of the 21^{st} century, the greatest challenge of robotics research has been creating robotic systems that can solve various tasks, rather than just serve a single purpose. One of the approaches to the problem is creating metamorphic robots. These consist of small independent modules, which cooperate to make the robot move or change its shape in accordance with robot's current mission. In general, the modules that can autonomously connect, disconnect, and climb over other modules. Possible applications for modular robots lie in the exploration of tight spaces or building temporary structures. Due to the robots' modularity, they can be reused for various tasks and save costs or storage space as a result.

To fully leverage a metamorphic robot's versatility, one has to be able to compute a shape-shifting (reconfiguration) plan efficiently. A reconfiguration plan is a sequence of actions individual modules can take to morph the robot from one shape to another. The problem of efficiently finding a reconfiguration plan has been studied, but no universal and efficient solution has been found yet.

The definition of the configuration of a robot and the elementary actions of modules depend on the type of module architecture. In general, we have *Lattice* and *Chain* architectures [31]. In the Lattice architecture, modules are strictly arranged in a regular 3D grid and tightly packed together, see M-Blocks [23] and Atron [11]. In chain architectures, such as Polybots [30] or Molecubes [32], the robots may easily take a shape that resembles a body with limbs, arms, or tentacles. Naturally, many architectures are *Hybrid*, i.e., designed to allow for both arrangements of modules. Some examples of hybrid systems are M-TRAN [17], Roombots [25], SMORES [10], HyMod [19], Omni-Pi-tent [21], and RoFI [15].

Shape reconfiguration approaches for lattice-type and chain-type robots differ. The lattice-type system's modules can usually crawl on the system's surface [7], [22], but a single module cannot efficiently move many modules in one step, due to the weak connections between modules. On the other hand, chain-type modules commonly utilize mechanical connections, and can form *tentacles* to transport modules or even attach and detach subassemblies of the modules.

Utilizing the observation that we can form tentacles, we present a new approach to solving the problem of computing the shape-reconfiguration plan for chain or hybrid-type systems in which the modules cannot locomote on their own. By forming octopus-like shapes with many tentacles, we can decompose the reconfiguration problem to multiple smaller motion planning instances, which results in a scalable heuristic solution.

To control the tentacles, we use an extension of inverse kinematics, which lets the tentacle reach the desired position. Once we move the final module of the tentacle to the correct position, we fasten it there and disconnect it from the remainder of the tentacle. By repeating this procedure for all the modules, we arrive at the desired shape.

The presented solution allows the configuration of modules to stay connected throughout the reconfiguration, which makes the plans schedulable without individual module locomotion. The solution respects the spatial arrangement of modules and yields collision-free reconfiguration plans. The ideas are not bound to a specific module shape, and can be applied to any platform. Unlike previous approaches [16], we are able to limit the amount of consequent modules a joint needs to lift at a single time, which can account for physical limits of joints.

Experimental evaluation has validated that our new algorithm outperforms existing algorithms in terms of the resulting path quality and speed of computation.

II. Related Work

Metamorphic robot reconfiguration is a computationally challenging problem. Previous researchers have reduced the problem to graph-based reconfiguration and shown that the task of finding an optimal reconfiguration plan is NP-hard [9], even for the simplified problem without considering physical module constraints.

One of the traditional and well-explored approaches to reconfiguration is to traverse the configuration state space. Approaches which search the state space provide optimal

[1]All authors are with Faculty of Informatics, Masaryk University, Czechia
rofi@fi.muni.cz

plans and respect the spatial arrangement of the modules, but do not scale. The state space of the possible reconfiguration steps grows exponentially fast with each module [6], and so far, no efficient heuristic has been presented to circumvent that. Some attempts to reduce the size of the space have been suggested, such as heuristics to improve the performance of the search [2] or pruning the state space [12]. Some authors tried to use symbolic representations [3] or lower the exponent using weak configuration equality [18], [2], and [26]. Ultimately, for modules a high degree of freedom, such as RoFI [15], finding plans for robots consisting of over five modules is computationally infeasible, even with several heuristics.

More scalable techniques for reconfiguring chain-like robots use the divide and conquer technique [4], or distributed and online algorithms [20], [8]. However, the algorithms either do not guarantee collision-free plans, require a specific module shape, or introduce other restrictions that have to be met.

Tree-like structures have been leveraged successfully to simplify the problem. SMORES authors introduced an algorithm based on dynamic programming that efficiently solves the case for tree-shaped connector graphs [13]. However, the algorithm only works for shapes that can be assembled in a planar way, and relies on the individual locomotion of modules, which may not be present in all modules.

Snake reconfiguration [16] introduces a reconfiguration technique that uses an intermediate shape and leverages inverse kinematics for module locomotion. The computation is fast and scales well up to a hundred modules. However, it is impractical in terms of physical realization; the long tentacles used in the connection process are error prone and may be impossible to lift due to the limited torque of module joints.

In this work, we build on top of existing methods and introduce a way to generalize snake reconfiguration to arbitrary trees. As a result, we can limit the number of modules that need to be lifted at a time, which accounts for limited torque of real joints, and makes the plans schedulable in tighter spaces. In addition, we can shorten the reconfiguration plans for shapes that already share similarities.

III. Preliminaries

A. Module requirements

For the rest of the paper, we assume a metamorphic hybrid-type platform where all modules are uniform. When we refer to *a module*, we refer to one self-contained autonomous building block, by *a robot*, we mean an object composed of individual modules.

The core ideas of our algorithm work for any shape. We do not expect the modules to feature any specific number of joints or connectors, however, we expect the connectors to be genderless; hence, any pair of connectors may connect, provided they are close enough and facing each other. Clearly, some kind of mobility is required; modules are either expected to have at least one revolute joint, or need to be able to revolve around each other.

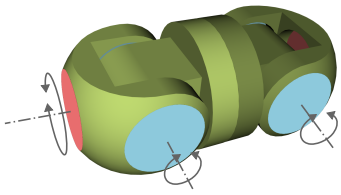
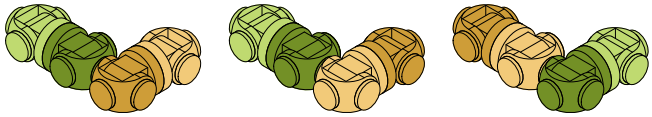


Fig. 1: RoFI module schematics. There are six connectors and three joints. Each half of the module has a rotating shoe, which can be used to adjust connectors. The middle joint allows for unlimited rotation between the two module halves.



(a) (b) (c)

Fig. 2: All three robots are in the same shape; however, their internal configurations differ. Overall, there are 2^8 unique internal configurations for this particular shape, since we can swap the halves of a single module, rotate the middle joint by 180°, change the orientation of the connection, or swap the modules.

We choose to demonstrate our approach using RoFI modules as they feature modules with a high number of degrees of freedom that the traditional reconfiguration approaches struggle with. A single RoFI module is depicted in Figure 1. The module has three degrees of freedom and six mechanical connectors. The connectors can connect in four different orientations. The side joints allow for turning the bodies with connectors by 180°, and the middle joint allows for unlimited rotation of the module's halves.

B. Definition of Reconfiguration

Given a robot comprised of multiple modules, we can distinguish between several abstractions of the robot's state:

- *configuration*, which is the arrangement of modules, including their inner state (unique identifier and joint position) and the way they are connected. Formally, a configuration is labeled graph where each vertex corresponds to a single module, vertex labels capture the inner state of a module, edges represent connections between modules and edge labels capture connectors used for connection and their orientation;
- *connector graphs* [9] capture the connections between modules. We can obtain a connector graph from a configuration by removing the inner state of the modules and keeping only the module identifier.
- *shape* only captures the overall shape of the robots, abstracting away from the inner state of the modules. Formally, a shape is a class of isomorphic configurations from which we removed modules' identifiers. Such a representation abstracts away from possible symmetries (see Fig. 2).

Since the robot's mission generally only depends on the shape, we will deal with reconfigurations of shapes and consider modules interchangeable.

For robots comprised of multiple modules, we consider joint movement of a single module, connection, and disconnection as atomic actions.

Finding *a reconfiguration plan* from the source to the target shape means finding a sequence of these atomic actions such that the source shape matches the target. Since we assume no individual locomotion of the modules, we also require that no action in a reconfiguration splits the robot into multiple pieces.

IV. Reconfiguration algorithm

A. Algorithm outline

As stated, the most common approaches to reconfiguration consider the *module state space* and traverse through it. In such a search, a single state consists of the entire configuration, and transitions between states consist of atomic actions. However, this approach suffers due to the exponential blow-up in the size of the state space with each module. SMORES [13] address this problem by only considering tree-shaped connector graphs and moving individual modules to reconfigure. Similarly, our approach relies on utilizing tree-shaped connector graphs, but since we operate without module locomotion, we will form *tentacles* that can transport modules and perform a more complex search.

Formally, a *tentacle* is a linear chain of modules that is considered moveable and does not branch further. By utilizing tree structures in the robots and forming tentacles, we can lift the problem to a higher level of abstraction, which we refer to as *tentacle space*: the space of all possible tentacles that can be formed. When searching the tentacle space, a state is represented by a connector graph. Each transition between two states is a *macro step*: a sequence of atomic actions that leads to transporting a module and reconnecting it at a new position.

Snake reconfiguration [16] first explores the possibility of traversing the *tentacle space*. The algorithm finds reconfiguration plans from the source and target to an intermediate snake shape, then reverses one of the plans to obtain a complete transformation. Transforming to a snake results in algorithmic simplicity and a clear heuristic: connect any pair of tentacles and disconnect one of them to reduce the number of remaining tentacles. Repeating this procedure converges to a linear chain of modules—the snake shape.

Traversing the tentacle space has two major advantages:

- the space of all possible tentacles is significantly smaller than the space of all possible configurations, which enables searching through all possible states for a significantly higher number of modules;
- the distance between states is easier to measure, which allows us to define reliable heuristics.

However, the intermediate snake shape is unsuitable in terms of physical realization. The long tentacles used to connect modules quickly become unschedulable due to the limited torque of module joints. In addition, some plans are unnecessarily long, since they go through the snake shape even if the original shapes are relatively similar.

To address the problems associated with the intermediate shape, we introduce more refined macro steps (described in Section V), and a way of composing them to systematically transform into the target. As a result, we can generalize the snake transformation method to arbitrary trees and perform a more direct transformation into the target.

B. Measuring the distance of configurations

To form mobile tentacles and easily measure distance between configurations, we need to identify spanning trees of the given robot shapes. The trees have to be rooted; hence the first step of any reconfiguration consists of finding the center of the configuration, which will remain in place during the procedure. A way to efficiently find the center of a metamorphic robot has been presented in [14].

Once a center is found, we traverse the configuration in breadth-first order, retaining the first found connection to a module, disconnecting cycles that may be present. Any way to find a spanning tree of the configuration can be used, but the breadth-first order is deliberate, since we benefit from relatively short and uniform tentacles.

Once the source and target configurations have been turned into trees, we can find the maximal common subtree using the method presented in [14]. Modules in the source configuration which fall outside of the common subtree are considered *defects*.

We identify 3 types of defects (see Figure 3):

- *missing branch*: target subtree is missing altogether;
- *missing leaf*: target subtree is present and can be moved, but is missing additional modules;
- *branch mismatch*: target subtree is connected incorrectly.

We will address exactly how these issues can be fixed in Section VI.

C. Tentacle state space exploration

Given two connector graphs, a naive solution to reconfiguration is to perform exhaustive search on which tentacles can transport misplaced modules to a new location, in an effort to fix the defects. A heuristic for such a search would be the number of modules that can be matched to the target.

However, the tentacle space is still exponential to the number of modules, even though the exponent is smaller compared to the configuration space. In addition, computing a single transition between states is a complex operation compared to a single atomic action. Moving individual tentacles consists of high-dimensional motion planning instances: finding a solution can be in the magnitude of seconds, depending on tentacle size. These observations highlight a need for a reliable heuristic for choosing which reconnections bring us closer to a solution.

Since we are aiming to create plans that are schedulable on physical robots, there is a clear direction to approach the search: fix the defects near the root first to create a solid core

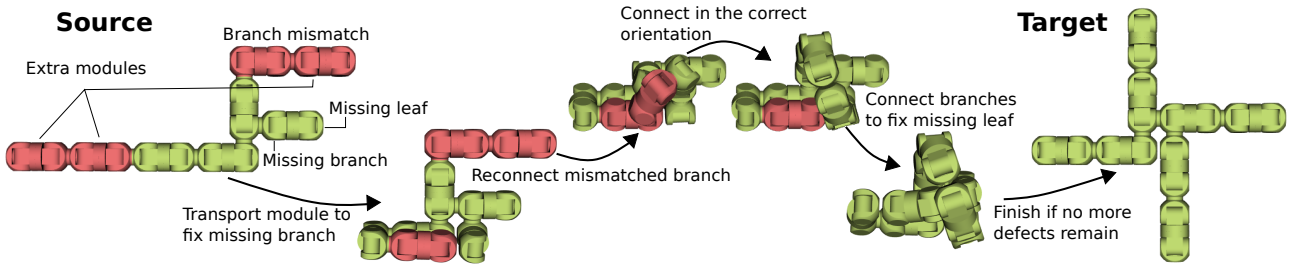


Fig. 3: An example of how the algorithm can transform a given shape into the target

and avoid forming long tentacles. This observation results in a very systematic way to perform the search: traverse the source tree in a breadth-first way starting from the root, and try to fix defects that are detected along the way.

The procedure is shown in Algorithm 1 and an example is visualized in Figure 3. The algorithm builds a BFS queue of vertices that can be matched onto the target. While traversing the graph, it attempts to fix any defects. If none of the possible fixes is successful, the algorithm backtracks to a previous decision and explores a different order of connecting tentacles. If the queue is empty, all vertices can be matched to the target and the algorithm returns the sequence of actions that led to the transformation.

Algorithm 1: Reconfiguration algorithm overview

```
input : Spanning trees Source and Target
output: Sequence of steps or None
queue = {Source.root};
while queue is not empty do
    v = queue.pop();
    foreach child u in v do
        if u cannot be matched to a node in target
          then
            call function that tries to fix the defect;
        end
        if u can be matched then
            queue.push(u);
        end
    end
    if fixing any defect failed then
        queue.push(v);
    end
    if tree was traversed without any changes then
        backtrack to the last fix;
        if no more unexplored paths remain then
            return None;
        end
    end
end
adjust joint positions to match the target;
return Transformation sequence;
```

V. Macro steps

To complete the reconfiguration algorithm, we need to define basic building blocks, which will allow tentacle movement, and result in the fix of defects. The basic building blocks of our algorithm are the following macro steps:

A. Reach

Tentacles comprised of modules can be treated as regular robotic manipulators. Hence, reaching a position that allows module connection becomes equivalent to reaching a specified end effector position and rotation, while satisfying joint constraints and avoiding collisions.

Snake reconfiguration [16] utilized simple inverse kinematics, which works for finding connections as long as we assume no other obstacles in the space where we are connecting tentacles. However, when computing connections near the center of the configuration, we need a way to actively avoid other obstacles. Although we can still use inverse kinematics to find a collision-free final position, the transition to the computed position would be prone to collisions.

To achieve collision avoidance, any planner can be used, including A* [27] or sampling methods such as RRT [28] or PRM [5]. However, since we are dealing with differently sized tentacles and the number of joints grows with each module[1], an algorithm which can find fast solutions even for a high number of joints is necessary. Similarly to the technique introduced in [29], we reduced the problem further to 3 dimensions, sampling end effector paths and computing inverse kinematics along the path via the FABRIK algorithm [1]. The core advantage of this approach is that it is barely affected by additional degrees of freedom [24].

Note that in high-dimensional spaces, any efficient planning method is inherently heuristic; given limited time, it will provide suboptimal solutions and can fail even though a solution exists. The possible impact on the overall algorithm will be discussed later.

B. Approach

When searching for a way to connect two tentacles, we use a two-step process based on the reduction in [16]. Given tentacles A and B, we virtually disconnect B from its original position, and connect it to A via the desired connector. Then, we compute REACH on the extended tentacle, with B's original position as the target. If the operation is successful, we obtain a point where the original two branches can be

[1]we considered 2 to 18 degrees of freedom, but the number will vary based on module capabilities

connected, and then individually replicate reaching that point with both tentacles.

VI. Fixing defects

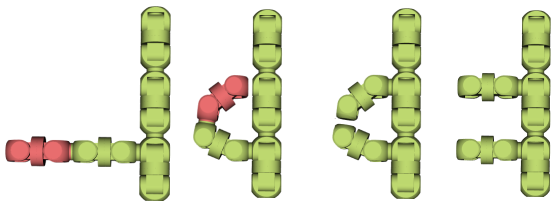
We can perform more complex operations by sequencing multiple macro steps and atomic actions. For instance, a natural operation of FETCHING a module with another tentacle and removing it from its original position is a composition of REACH – CONNECT – DISCONNECT.

With this observation, we can define sequences of steps which lead to fixing a defect.

A. Missing branch

When a branch is missing altogether, the algorithm finds all the extra leaves in the configuration, and tries to connect any that are close enough via the REACH step.

If this operation is successful, the module is docked in the new position, and disconnected from the remainder of the tentacle (see Figure 4).



(a) Source (b) Reach (c) Reconnect (d) Target

Fig. 4: Fixing a missing branch

B. Missing leaf

When a module is missing, but would be connected to a mobile tentacle, we use the APPROACH step, connect the two tentacles, and then disconnect (see Figure 5).

(a) Source (b) Approach (c) Reconnect (d) Target

Fig. 5: Fixing a missing leaf

C. Branch mismatch

If a module is already docked in a connector, but via the wrong connector or in the wrong orientation, multiple steps are required:

- identify a tentacle the module is a part of and APPROACH it with a different tentacle;
- CONNECT the two tentacles and DISCONNECT the module from the remainder of the configuration;
- REACH the new position with a module, prioritizing the one that was just disconnected;
- CONNECT the module and DISCONNECT it from the remainder of the tentacle.

The entire procedure is visualized in Figure 6.

(a) Source (b) Fetch (c) Reach (d) Reconnect (e) Target

Fig. 6: Fixing a branch mismatch

D. Unreachable position

The algorithm can get to a configuration where a missing module position is not reachable with an extra module, due to distance or physical constraints. To account for failure to connect, we allow the algorithm to take intermediate steps which do not necessarily get the configuration closer to its target, but instead get the module closer to the desired position physically.

The steps consist of performing a FETCH operation with any tentacle that is closer than the original tentacle, until a connection can be made (see Figure 7).

(a) Source (b) Fetch (c) Fix defect (d) Target

Fig. 7: Fetching a module with another tentacle to fix a defect

Since this operation can result in unnecessary movement and reconnection actions, we perform it with the lowest priority, only if the algorithm did not fix any defects between two visits of the same vertex.

Allowing this operation also accounts for cases where the position is reachable but fails due to the heuristic nature of planning algorithms. This ultimately leads to being able to recover from unexpected failure, but can result in unnecessary reconnections. If the number of reconnections needs to be minimized, we can allot more time to arm connections and decrease the chance of failure on a reachable goal at the cost of a longer computation time.

VII. Evaluation

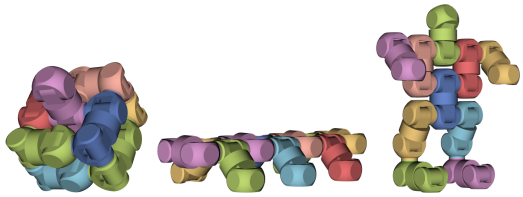
We implemented the proposed algorithm in C++[2]. We evaluated our algorithm using two scenarios:

- We compare module space exploration using several heuristics, snake reconfiguration [16], and the newly presented tree reconfiguration on 60 randomly generated shapes. The module count only ranges from 3 to 5 modules, since module space exploration for any more modules is too costly.
- We compare snake [16] and our new algorithm on 9 hand-crafted shapes, which consist of 10 modules each. The shapes are chosen to represent real-life objects and have distinct topologies, see Figure 8.

[2]The implementation and source data are publicly available at `https://github.com/paradise-fi/RoFI`.

	Ball	Board	Caterpillar	Chair	Column	Human	Puppy	Stool
Ball	—	38s, 17	100s, 20	24s, 13	25s, 18	69s, 17	71s, 24	44s, 16
Board	42s, 24	—	59s, 14	39s, 22	23s, 18	71s, 18	271s, 20	16s, 9
Caterpillar	57s, 15	156s, 23	—	30s, 15	30s, 14	18s, 12	70s, 20	62s, 15
Chair	130s, 12	116s, 19	40s, 14	—	80s, 18	30s, 15	159s, 22	33s, 17
Column	100s, 20	106s, 22	126s, 20	132s, 21	—	49s, 21	38s, 17	193s, 19
Human	109s, 20	23s, 17	22s, 13	62s, 21	73s, 20	—	157s, 18	146s, 15
Puppy	79s, 19	98s, 21	64s, 25	127s, 23	162s, 20	146s, 18	—	114s, 19
Stool	119s, 25	13s, 9	100s, 18	35s, 17	78s, 20	78s, 20	66s, 16	—
Snake	7s, 8	5s, 8	6s, 9	5s, 8	—	5s, 9	4s, 6	7s, 7

TABLE I: Tree reconfiguration on distinct shapes. Rows denote the source shape, and columns denote the target. Each cell contains computation time and the number of reconnections in the found plan. The last row shows the time and reconnections required to find a transformation into a snake shape.



(a) Ball (b) Caterpillar (c) Humanoid

Fig. 8: Illustration of hand-crafted shapes

	Module Count		
	3	4	5
Module-Space	1s, 100MB	20s, 0.5GB	30min, 1GB
Snake	0.3s, 20MB	0.5s, 30MB	1s, 40MB
Tree	0.3s, 30MB	1s, 40MB	4s, 50MB

TABLE II: Comparison of average time and memory requirements for reconfiguration of shapes with growing number of modules using module-space exploration, snake reconfiguration, and tree reconfiguration.

We tested our algorithm on the AMD EPYC 7371 processor with cores running at 2.00 GHz. Each run was bounded by 30 minutes of CPU time and 1 GB of RAM.

The results from the first test scenario are captured in Table II. This comparison clearly shows that the module-space approaches scale poorly and are unusable even for very low module count. On the contrary, the tentacle-space approaches are an order of magnitude faster, and have low memory requirements.

The results from the second test scenario are captured in Table I. By concatenating two snake transformation plans, we consistently reach around ~ 17 reconnections, since each reconnection is guaranteed to bring us closer to the desired intermediate shape. Meanwhile, tree reconfiguration shows a wide range based on the given shapes. Since the new plans are executed in a smaller space, intermediate actions may be necessary to avoid collisions or account for limited tentacle length—regardless, most plans are of comparable length even if the shapes share no similarity. Most importantly, the new approach finds plans even when we limit the maximal tentacle length to 2 or 3 modules, whereas the snake shape requires the entire shape to be movable.

Due to the increased amount of collisions and a higher complexity in choosing the next step, the new method often traverses a significantly greater portion of the tentacle space. This results in more varied and larger computation times. Still, the times are in usable range compared to approaches that search the module space, and scalability is not limited by the amount of available memory. The average memory usage for 10 modules was under 100MB for both methods.

Overall, we can see that our new reconfiguration algorithm demands more resources than snake reconfiguration. However, the quality of the resulting plan is higher: plans can be performed in a smaller space, and we can account for limited joint strength. If the shapes share any similarities, the new plans are shorter.

Furthermore, the approach still has small resource requirements compared to module space exploration:

- even the worst cases only take a few minutes to compute, which is orders of magnitude faster;
- the memory usage is negligible.

Due to the low memory requirements, there is no need to precompute and store reconfiguration plans. Instead, the plan can be computed on the modules themselves, even if they have limited memory and computational strength. This enhances the usability and robustness of metamorphic robots, as they can i.e. find plans that replace defective modules offline and continue their mission.

VIII. Conclusion

We have presented a novel algorithm for reconfiguration of metamorphic robots that yields collision-free plans and respects limited joint strength. The algorithm is not bound to a specific module type and can be easily adapted for various systems. The core idea lies in shifting the exploration from the module space to the tentacle space via leveraging inverse kinematics.

The algorithm scales well compared to optimal approaches while addressing some issues of previous heuristic approaches. Memory usage is negligible, and the computation time is generally insignificant compared to the time required for the physical reconfiguration. Although the algorithm is a heuristic and might not find a reconfiguration plan, it has performed well on tested examples.

REFERENCES

[1] A. Aristidou and J. Lasenby. Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*, 73:243–260, 09 2011.

[2] M. Asadpour, M. H. Z. Ashtiani, A. Spröwitz, and A. J. Ijspeert. Graph signature for self-reconfiguration planning of modules with symmetry. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2009.

[3] S. Baarir, L. Hillah, F. Kordon, and E. Renault. Self/reconfigurable modular robots and their symbolic configuration space. In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 2010.

[4] A. Casal and M. H. Yim. Self-reconfiguration planning for a class of modular robots. In *Sensor Fusion and Decentralized Control in Robotic Systems II*, volume 3839. International Society for Optics and Photonics, SPIE, 1999.

[5] G. Chen, N. Luo, D. Liu, Z. Zhao, and C. Liang. Path planning for manipulators based on an improved probabilistic roadmap method. *Robotics and Computer-Integrated Manufacturing*, 72:102196, 2021.

[6] G. S. Chirikjian, A. Pamecha, and I. Ebert-Uphoff. Evaluating efficiency of self/reconfiguration in a class of modular robots. *Journal of Field Robotics*, 13(5):317–338, 1996.

[7] D. J. Christensen. Experiments on fault-tolerant self/reconfiguration and emergent self/repair. In *First IEEE Symposium on Artificial Life, ALIFE 2007, Honolulu, Hawaii, USA, April 1-5, 2007*, pages 355–361. IEEE, 2007.

[8] F. Hou and W. Shen. Distributed, dynamic, and autonomous reconfiguration planning for chain-type self-reconfigurable robots. In *2008 IEEE International Conference on Robotics and Automation, ICRA 2008, May 19-23, 2008, Pasadena, California, USA*, pages 3135–3140. IEEE, 2008.

[9] F. Hou and W. Shen. Graph-based optimal reconfiguration planning for self-reconfigurable robots. *Robotics and Autonomous Systems*, 62(7), 2014.

[10] G. Jing, T. Tosun, M. Yim, and H. Kress-Gazit. An End-To-End System for Accomplishing Tasks with Modular Robots. In *Robotics: Science and Systems XII*, 2016.

[11] M. W. Jörgensen, E. H. Östergaard, and H. H. Lund. Modular ATRON: modules for a self-reconfigurable robot. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2004.

[12] H. Khodr, M. Mutlu, S. Hauser, A. Bernardino, and A. J. Ijspeert. An optimal planning framework to deploy self/reconfigurable modular robots. *IEEE Robotics Automation Letters*, 4(4):4278–4285, 2019.

[13] C. Liu, M. Whitzer, and M. Yim. A distributed reconfiguration planning algorithm for modular robots. *IEEE Robotics Automation Letters*, 4(4):4231–4238, 2019.

[14] C. Liu and M. Yim. Configuration recognition with distributed information for modular robots. In N. M. Amato, G. Hager, S. L. Thomas, and M. Torres-Torriti, editors, *Robotics Research, The 18th International Symposium, ISRR 2017, Puerto Varas, Chile, December 11-14, 2017*, volume 10 of *Springer Proceedings in Advanced Robotics*, pages 967–983. Springer, 2017.

[15] J. Mrázek and J. Barnat. Roficom – first open-hardware connector for metamorphic robots. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2720–2725, Nov. 2019.

[16] J. Mrázek, P. Ondika, I. Černá, and J. Barnat. Tentacle-based shape shifting of metamorphic robots using fast inverse kinematics. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11894–11900, 2023.

[17] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji. M-TRAN: Self-reconfigurable modular robotic system. *IEEE/ASME transactions on mechatronics*, 7(4), 2002.

[18] M. Park, S. Chitta, A. Teichman, and M. Yim. Automatic configuration recognition methods in modular robots. *Iternation Journal of Robotics Research*, 27(3-4):403–421, 2008.

[19] C. Parrott, T. J. Dodd, and R. Gross. HyMod: A 3-DOF Hybrid Mobile and Self-Reconfigurable Modular Robot and its Extensions. In *Distributed Autonomous Robotic Systems, The 13th International Symposium, DARS 2016*, volume 6 of *Springer Proceedings in Advanced Robotics*. Springer, 2016.

[20] K. Payne, B. Salemi, P. M. Will, and W. Shen. Sensor-based distributed control for chain-typed self-reconfiguration. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, Sendai, Japan, September 28 - October 2, 2004*, pages 2074–2080. IEEE, 2004.

[21] R. H. Peck, J. Timmis, and A. M. Tyrrell. Omni-pi-tent: An omnidirectional modular robot with genderless docking. In *Towards Autonomous Robotic Systems*, volume 11650 of *Lecture Notes in Computer Science*, pages 307–318. Springer, 2019.

[22] B. Piranda and J. Bourgeois. A distributed algorithm for reconfiguration of lattice-based modular self/reconfigurable robots. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,*, pages 1–9. IEEE Computer Society, 2016.

[23] J. Romanishin, K. Gilpin, and D. Rus. M-blocks: Momentum-driven, magnetic modular robots. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013.

[24] A. Shkolnik and R. Tedrake. Path planning in 1000+ dimensions using a task-space voronoi bias. In *2009 IEEE International Conference on Robotics and Automation*, pages 2061–2067, 2009.

[25] A. Spröwitz, A. Billard, P. Dillenbourg, and A. J. Ijspeert. Roombots-mechanical design of self/reconfiguring modular robots for adaptive furniture. In *2009 IEEE International Conference on Robotics and Automation*, pages 4259–4264. IEEE, 2009.

[26] K. Taheri, H. Moradi, M. Asadpour, and P. Parhami. MVGS: A new graph signature for self/reconfiguration planning of modular robots based on multiple views theory. *Robotics Autonomous Systems*, 79:72–86, 2016.

[27] P. Tavares, J. Lima, P. Costa, and A. Moreira. Multiple manipulators path planning using double a*. *Industrial Robot: An International Journal*, 43:657–664, 10 2016.

[28] K. Wei and B. Ren. A method on dynamic path planning for robotic manipulator autonomous obstacle avoidance based on an improved rrt algorithm. *Sensors*, 18(2), 2018.

[29] Y. Xie, Z. Zhang, X. Wu, Z. Shi, Y. Chen, B. Wu, and K. A. Mantey. Obstacle avoidance and path planning for multi-joint manipulator in a space robot. *IEEE Access*, 8:3511–3526, 2020.

[30] M. Yim, D. Duff, and K. Roufas. Polybot: A modular reconfigurable robot. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pages 514–520. IEEE, 2000.

[31] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. Modular self/reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics Automation Magazine*, 14(1):43–52, 2007.

[32] V. Zykov, E. Mytilinaios, M. Desnoyer, and H. Lipson. Evolved and designed self/reproducing modular robotics. *IEEE Transansactions on Robotics*, 23(2):308–319, 2007.