2023 IEEE International Conference on Robotics and Automation (ICRA 2023)
May 29 - June 2, 2023. London, UK

Tentacle-Based Shape Shifting of Metamorphic Robots Using Fast Inverse Kinematics*

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

***Abstract*— We present a new approach to tackle the problem of metamorphic robots' reconfiguration. Given the chain-type metamorphic robot's initial and target configuration, we compute a reconfiguration plan that is provably physically collision-free. Our solution employs a specific heuristic. The robot initially reconfigures to a shape that resembles an octopus with many tentacles. After that, the tentacles gradually reconnect to each other using inverse kinematics, separating one tentacle from the body and keeping the other one connected. This strategy eventually leads to a snake-like structure of the robot. For the target configuration, we compute the reconfiguration plan with the same procedure, however, we reverse the plan to reconfigure the robot from the snake-like structure to the target shape. According to our experimental evaluation, our newly introduced strategy for finding reconfiguration plans is successful. It efficiently finds collision-free plans even for robots consisting of hundreds of modules.**

I. Introduction

A metamorphic robot is made of independent but uniform mechatronic modules that can autonomously connect, disconnect, and climb over other modules. The swarm of modules may cooperate to make the robot move or change its shape to fit the robot's mission the most. However, to fully leverage such a robot's versatility, one has to be able to compute a shape-shifting (reconfiguration) plan efficiently. The plan is a sequence of actions of individual modules to morph the robot from one shape (configuration) to another.

The definition of the configuration of a robot and the elementary actions of modules depend on the type of metamorphic platform. In general, we have *Lattice* and *Chain* architectures [30]. In the Lattice architecture, modules are strictly arranged in a regular 3D grid and tightly packed together, see M-Blocks [24] and Atron [12]. In chain architectures, such as Polybots [29] or Molecubes [31], the robots may easily take a shape that resembles a body with limbs, arms, or tentacles. Naturally, many platforms are *Hybrid*, i.e., designed to allow for both arrangements of modules. See e.g. M-TRAN [18], Roombots [26], SMORES [11], HyMod [20], Omni-Pi-tent [22], and RoFI [17]. Clearly, the 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], [23], but a single module cannot efficiently move many modules in one step. On the other hand, the chain-type modules can move modules via their *tentacles* or even attach and detach subassemblies of the modules.

We present a new approach to solving the problem of computing the shape-reconfiguration plan for chain-type systems in which the modules cannot locomote on their own, and thus, the configuration of modules has to stay continuous throughout the reconfiguration. We do not rely on individual modules' locomotion as it is not present on many existing hybrid-type robots [30] and it cannot be reliably used in rough terrain or in outer space. The presented solution aims at respecting the spatial arrangement of the modules and yielding collision-free reconfiguration plans.

Our shape-reconfiguration strategy is to first let the robot unwind to an octopus-like shape and then repeatedly decrease the number of tentacles by separating a tentacle from the body of the robot and reconnecting it to another tentacle. The procedure for reconnecting two tentacles employs inverse kinematics to compute the positions of tentacles in which the tentacles' tips are connected. In this manner, one of the tentacles grows in length until all the tentacles are connected together. Once finished, the robot exhibits a snake-like structure — the basic shape for many metamorphic platforms [30]. Reconfiguring both the source and target shape into a snake gives the complete reconfiguration plan.

This approach has several benefits. The first one is efficiency; the procedure scales well as it can tackle reconfiguration of hundreds of modules in reasonable time and space, which is out of reach for the traditional approaches. Also, when shape-shifting within a given set of predefined configurations, only one reconfiguration plan for each configuration has to be precomputed instead of linearly many in the case of the direct configuration-to-configuration reconfiguration. The presented approach is, however, a heuristic. Therefore these benefits come at the cost of not finding the shortest reconfiguration plan or not finding the reconfiguration plan at all. Fortunately, the latter rarely happens according to our experiments.

II. Related Work

One of the traditional and well-explored approaches to reconfiguration is to traverse the configuration state space. Even though the number of states grows exponentially [6], there have been some successful applications, A* and RRT for ATRONs [4], heuristics to improve performance of the state-space search [2], to employ pruning [13], or use symbolic representations [3]. These approaches yield optimal plans and respect the spatial arrangement of the modules, but they do not scale. The reason for that is that the state space

*This research was supported by ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16 019/0000822).

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

of the possible reconfiguration steps grows exponentially fast, and so far, no efficient heuristic has been presented to circumvent that.

Weak configuration equality may also be used to alleviate the burden of exponential blow-up in the state-space search; however, it does not come for free as presented by [19], [2], and [27].

Some constraints may be relaxed to tackle larger instances, e.g., the spatial arrangement of modules. But such a simplification is suitable only for modules with a high degree of module locomotion, as they can disconnect, move to the target location and connect again. Authors of [10] reduce the problem to graph-based reconfiguration and show that finding optimal reconfiguration is NP-complete. Later, [14] introduced an algorithm based on dynamic programming that efficiently solves the case for tree-shaped connector graphs.

[8] presents a rather original approach for the general case – they reduce the reconfiguration problem to SAT. The possibility of reducing the problem to SMT (satisfiability modulo theory, generalization of SAT to first-order formulae within a given theory) to get collision-free plans has also been explored [16]. However, none of these approaches are efficient enough to be used in practice.

Another approach for reconfiguring chain-like robots is to leverage their tree-like structures, the divide and conquer technique [5] or distributed and online algorithms [21],[9] for some reconfiguration. Mostly, the resulting algorithms do not produce collision-free plans, work on connector graphs only, or introduce other restrictions that have to be met.

Closest to our approach is the algorithm by [25] that forms a line out of MBLOCKs. This algorithm, however, is not applicable to chain-type modules – especially when the modules cannot move on their own, as it is required that the individual modules can climb one over others.

III. Preliminaries

For the rest of the paper, we assume a metamorphic platform similar to the M-TRAN system [18], or RoFI [17] where all the 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 arranged in a *configuration*.

Usually, the modules are uniquely identifiable in order to address and control them separately. However, since the modules are indistinguishable and symmetrical, in practice, we want to only consider the robot's *shape* and not its precise configuration (see Fig. 2). Mathematically speaking, the shape is a class of all the configurations that are the same apart from the module permutation and module symmetries.

For our algorithm, 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 together provided they are close enough and facing against each other.

We consider joint movement, connection, and disconnection as atomic actions. Finding *a reconfiguration plan* from the source shape to the target shape means finding a sequence of these atomic actions such that the robot in the source shape ends up in the target shape. Since we assume no individual locomotion of the modules, we also require that no action in a reconfiguration splits the robot into two pieces.

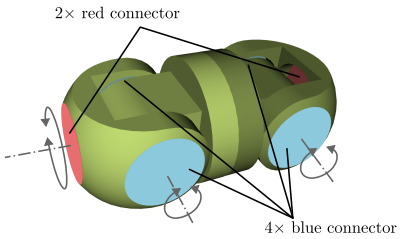


Fig. 1: The module schematics. There are six connectors and three joints. Two joints allow the connectors to rotate, the third joint allows for rotation between the light and dark half of the module.

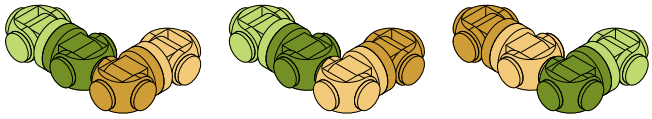


Fig. 2: All three robots are in the same shape; however, their internal configurations differ. Overall, there are 2^8 unique internal configurations for this particular shape. This is possible due to the module symmetries and permutations; we can swap dark and light half of a single module, rotate the middle join by 180°, change the orientation of the connection, and 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 Fig. 1. The module has three degrees of freedom and six connectors (depicted as blue and red, we will explain the distinction in Section IV). The connectors can connect in 4 different orientations. The side joints allow for turning the bodies with connectors by 180°, and the middle joint allows for unlimited turning of the module's halves.

IV. Reconfiguration Algorithm

Existing solutions to reconfiguration feature directly or indirectly a common challenge: identifying the distance between two shapes in the number of steps needed to turn one shape into another. This distance can be used for directing a state-space search or identifying a set of shape-changing rules that make progress in reconfiguration. Unfortunately, this cannot be done efficiently in the general case [10]. Therefore, previous approaches addressed this inefficiency by restricting the class of input shapes for which the reconfiguration can be computed. We, however, pursue a different approach. We introduce an intermediate shape between the source and target ones, towards which we can define some shape-changing rules that make progress in reconfiguration with high certainty.

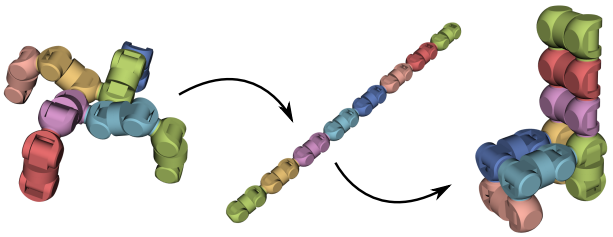


Fig. 3: Illustration of the general idea of reconfiguration via snake on reconfiguration from a spider to a chair.

There is one natural choice of the intermediate shape — a snake-like shape. The unique property of a snake-like structure is that it is composed of smaller snake-like structures. Hence, assembling a set of snake-like pieces in arbitrary order always yields a snake-like structure. In other words, the snake-like shape exhibits self-resemblance.

Since we impose no restrictions on the precise arrangement of modules, we formally define a snake shape (snake for short) as follows. Let us choose any pair of connectors of a module, for which it holds that if a module with all joints in the basic position is connected in the north orientation through either of the connectors from the selected pair, then there is no difference in the resulting shape. Let us refer to these connectors of the selected pair as *red connectors*. Then, a configuration is a *snake* if and only if the configuration is continuous, the modules are mutually interconnected via red connectors only, and there are exactly two red connectors unused in the whole configuration. One of the possible and natural choices of the red connectors for a RoFI module is depicted in Fig. 1.

We can leverage the snake self-resemblance to build a better-performing reconfiguration algorithm as follows. We let both the source and target shapes reconfigure into a snake, and then we reverse the target reconfiguration plan and append it to the source one. See Fig. 3.

Note that reconfiguration via an intermediate shape naturally yields longer reconfiguration plans than a direct reconfiguration. However, it also provides several benefits. The snake configuration allows us to trivially map modules from the source shape to modules in the target shape. Thus, we can avoid the costly computation of a graph isomorphism that other shape-reconfiguring algorithms do, e.g., [2] or [27].

Also, the transformation to the snake shape may be precomputed for each configuration in use. Thus, for the set of known configurations, we can compute reconfiguration plan from one configuration to another easily by concatenating the precomputed plans. This is an advantage compared to procedures that directly find reconfiguration between shapes. If we want to be able to reconfigure between any two configurations from a set of size n, our approach requires only finding n reconfiguration plans, whereas the direct approach requires n^2 plans.

Algorithm 1 Reconfiguration to snake

```
1:  function ToSnake(c: Configuration) → [Configuration]?
2:      sequence := MakeTree(c)
3:      s := TreeToSnake(sequence.last())
4:      if s is null then
5:          return null
6:      return sequence + s
7:  function TreeToSnake(c) → [Configuration]?
8:      for all pairs of free red connectors (i, j) do
9:          s_1 := connectAndCut(i, j)
10:         if s_1 is not null then
11:             s_2 := TreeToSnake(s_1.last())
12:             if s_2 is not null then
13:                 return s_1 + s_2
14:     if c is a snake then
15:         return []
16:     else
17:         return null
```

V. Reconfiguration To a Snake

The strategy of reconfiguration into a snake shape is based on the observation that the chain-type metamorphic robots often contain parts of the robot that can be moved with in isolation with respect to the rest of the robot. Let us refer to these parts as tentacles. More formally, a *tentacle* is a subtree of the configuration in which there is at least one module with an unconnected red connector. Let us denote one of these connectors as the *tip* of the tentacle. Since the tentacle is a tree, it has to be connected to the rest of the robot via a single connector which we refer to as a *shoulder*.

The idea of our algorithm is to identify two tentacles in the configuration and make them connect via their tips. By connecting the tips of two tentacles, we create a new red connection, as well as we form a new cycle in the configuration. If there is a non-red connection on the cycle, we may now disconnect it without breaking the property of continuity of the configuration (see Fig. 5). If there are only red connections, we release one of the shoulders. By repeated application of this step, we decrease the number of non-red connections and the number of tentacles in the configuration until there are no non-red connections. Note that if we connect the tentacle tips' connectors in the proper orientation, the procedure ends up with a configuration that is a snake. The whole process is illustrated in Fig. 4 and listed as Algorithm 1.

The algorithm proceeds as follows. We first turn the configuration into a tree (procedure MakeTree) via releasing redundant connections. Then we execute the key procedure TreeToSnake which tries to turn the robot into a snake.

TreeToSnake is a recursive procedure. In each recursive step, it chooses a pair of free red connectors. We find a common predecessor for the selected modules and declare its connectors as tentacle shoulders. Note that the predecessor choice is unambiguous as we consider the configuration as a rooted tree. The tree root is a module closest to the physical center of mass of the configuration. The tips and shoulders define two tentacles. The procedure tries to find a sequence of movements of these tentacles to connect their tips. If it

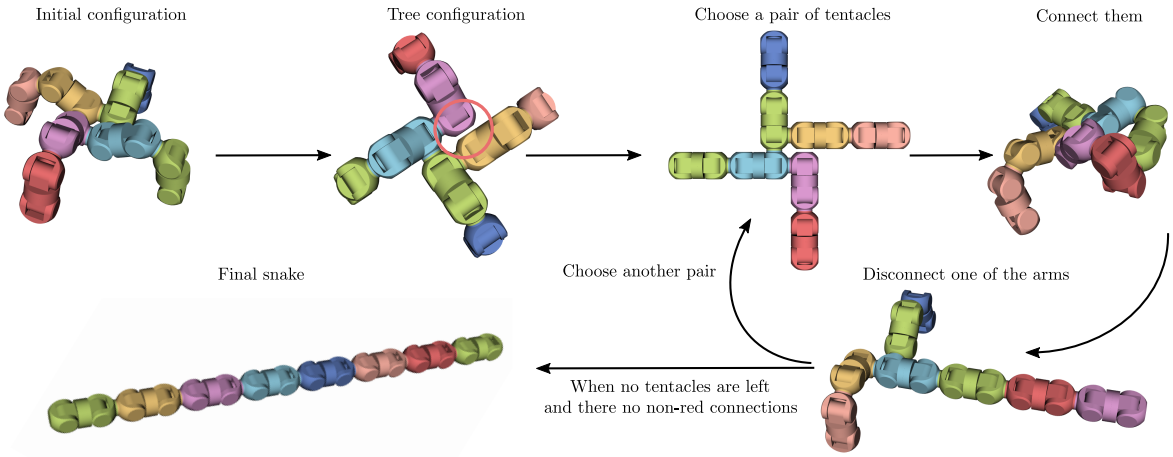
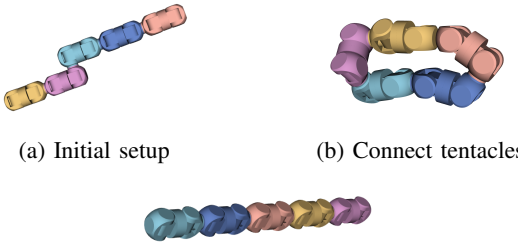


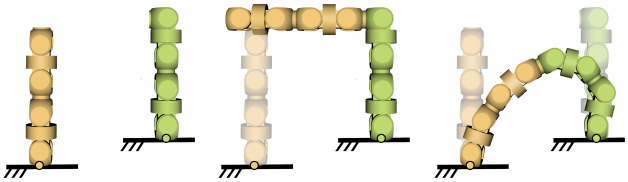
Fig. 4: Illustration of the ToSnake procedure.



(a) Initial setup (b) Connect tentacles

(c) Release non-red connection and straighten

Fig. 5: Illustration of removing non-red connection



(a) Initial setup (b) Concatenate (c) Perform IK

Fig. 6: Illustration of reduction of tentacle connection to inverse kinematics.

succeeds, it continues, otherwise, it backtracks and tries a different pair. This is necessary as not every choice of the order of the tentacle pairs leads to a solution. As we show later, due to the efficiency of the procedure for connecting the tentacles, backtracking over all possible choices of pairs of tentacles is actually computationally feasible. Once the procedure finishes, it ensures the robot forms a snake or there is no tentacle-based reconfiguration.

A. Turning a Configuration Into a Tree

There are multiple ways to choose connections to drop in order to turn an arbitrary configuration into a tree configuration. Since the shape of the tree affects the efficiency of the subsequent procedures, we want to generate trees that feature long branches that do not split very often. To achieve that, we pick a module closest to the physical center of mass of the configuration and compute the spanning tree by performing a breadth-first search.

B. Connecting Tentacles

To connect the tentacles of a tree-shaped configuration, we leverage inverse kinematics of robotic arms (*IK*). IK algorithms compute a set of positions of a robotic arm's joints in order to reach a given position in the space of an end-effector. We can reduce the problem of connecting the tips of two tentacles to the problem of computing the IK of a robotic arm using the following procedure:

Suppose there are two tentacles, yellow and green, whose tips should be connected (see Fig. 6a). Each tentacle has a tip and a shoulder with a mounting point. Without loss of generality, we can virtually unmount the yellow tentacle from its mounting point and attach it via its tip to the tip of the green tentacle (Fig. 6b). Thus, we obtain one long tentacle formed by the concatenated yellow and green tentacle. The end effector of the newly formed tentacle is the yellow shoulder mounting point. Now we use IK to compute the joint positions such that the connected tentacle reaches the yellow mounting point. Based on the IK solution, we can trivially reconstruct the movement required for the original pair of tentacles to connect.

Recently, a new algorithm has been presented to compute inverse kinematics for metamorphic robots specifically [15]. Nevertheless, FABRIK algorithm [1] suited our purposes better and, according to our experiments, significantly outperformed the approach from [15]. We, therefore, decided to adapt FABRIK algorithm to the RoFI module arrangement of joints, and use it for computation of the IK for the connecting tentacles. Nevertheless, the tentacle-based reconfiguration scheme is not limited to FABRIK IK algorithm. It is possible to use any other approach for computing the connection of two tentacles.

The original FABRIK algorithm, however, does not present a way to avoid collisions. The tentacle connections

do not happen in free space as there are other parts of the robot's body that may be in the way of the two tentacles to be connected. We, therefore, have to consider collisions in order to reach collision-free reconfiguration plans. [28] presents a way of implementing collision avoidance for FABRIK, nevertheless, the technique seems to be incompatible with the specific arrangement of the RoFI modules due to the presence of the middle rotational axis and due to the fact our tentacles may not be pure chains but may take a form of a tree. As a result, we have introduced our own modification of the FABRIK algorithm to deal with collision avoidance.

The following explanation expects that the reader is familiar with the FABRIK principle of operation [1]. Let us consider other parts of the robot besides the two active tentacles as static obstacles that do not move. During the forward propagation, FABRIK relinks the parts of the robotic arm one by one in such a way that they locally minimize the error from the target position. However, the position of the module that minimizes the error can cause a collision. Therefore, before relinking that part, we first compute new joint limits in which we are sure the relinked part cannot cause a collision, and let the FABRIK algorithm place the relinked module within those computed limits.

This method of collision avoidance is a greedy heuristic, thus it is, in theory, susceptible to be stuck in local minima. However, for the purpose of connecting the tentacles, where the obstacles are often clustered, this limitation has been shown as negligible by the experiments.

VI. Evaluation

For the purpose of the evaluation, we have tested our algorithm implemented in C++[1] on datasets:

1) 25 hand-crafted shapes, e.g. a chair, sphere or spider. The module count ranges from 6 to 10 modules.
2) 3504 shapes obtained by tangling a snake configuration by randomly applying 6000 actions. The modules count ranges from 5 to 200 modules.

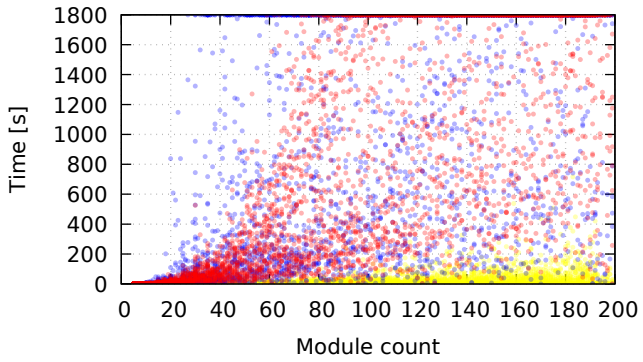
To evaluate the computing efficiency and validity of our collision avoidance approach, we experimentally evaluated the three following setups:

1) the first approach is a baseline where no collisions are detected and thus, we might produce a plan that is not collision-free.
2) The second approach uses FABRIK without any collision avoidance. Only if there is a collision during the movement, we simply claim that the pair of tentacles cannot be connected.
3) The last approach uses the proposed collision avoidance from Section V-B.

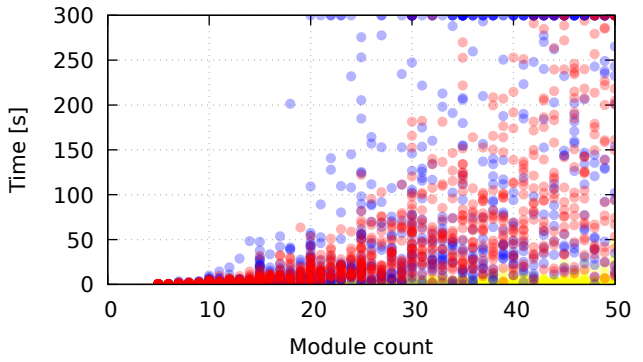
We tested our algorithm on a the AMD EPYC 7371 processor with cores running at 2.00 GHz. Each run was bounded by 30 minutes of the CPU time and 1 GB of RAM.

A summary of our experimental evaluation is provided in Table I and Fig. 7. We can make a couple of observations

[1]The implementation, source data and script to rerun experiments are publicly available at `https://github.com/paradise-fi/RoFI`.



(a) Whole dataset.



(b) Detail for 5–50 modules.

Fig. 7: Scatter plot of benchmark run time vs. module count. Yellow set ignores collisions (approach 1), blue one backtracks on collision (approach 2), red one uses collision avoidance (approach 3). Note the cluster of red points between 5 and 50 modules (see the detail) that shows that the algorithm has large success rate in short run times.

from the experiments. First of all, our approach is capable of solving some of very large instances, which is not the case for any complete solution to reconfiguration that has been published so far. A traditional state-space search algorithm can solve only instances of a few modules in the same time and space.

The algorithm found solutions to all our practically motivated examples (that is the hand-crafted set) in less than 2 seconds per benchmark. Overall, the algorithm found a collision-free reconfiguration path in 65% of cases. For practically-sized configuration of less than 50 modules it found a solution in 85% of the cases. Also, if a solution was found, it was usually found very fast (see Fig. 7) or it was not found within the time limit. This is given by the fact that often the solution can be found no matter of tentacles ordering, but sometimes there are only a few orderings that lead to a reconfiguration and, therefore, the whole state-space of tentacles' ordering has to be explored.

We can also see from Fig. 7 and Table I that without collision detections, the solution is found very quickly and it has a success rate of more than 98%. However, those plans are not collision free. When we want to achieve collision-

Dataset	Modules / Collisions	5–49 S	U	T		50–99 S	U	T		100–149 S	U	T		149–200 S	U	T		Overall S	U	T	
Hand crafted	none	25	0	0	100%	-	-	-	-	-	-	-	-	-	-	-	-	25	0	0	100%
	detection	22	3	0	88%	-	-	-	-	-	-	-	-	-	-	-	-	22	3	0	88%
	avoidance	25	0	0	100%	-	-	-	-	-	-	-	-	-	-	-	-	25	0	0	100%
Tangled	none	843	39	0	96%	900	0	0	100%	900	0	0	100%	822	0	0	100%	3465	39	0	98%
	detection	686	158	38	78%	390	150	360	43%	194	166	540	22%	128	93	601	16%	1398	567	1539	40%
	avoidance	815	67	0	92%	746	58	96	83%	421	5	474	47%	295	4	523	36%	2277	134	1093	65%

TABLE I: Table summarizing the evaluation TOSNAKE of the two datasets with various approaches to collisions. Four values are provided for each evaluation: the number of successfully solved benchmarks (S), number of benchmarks that finished but did not find reconfiguration path (U), number of benchmarks that timeouted (T), and success rate – that is the ratio of solved benchmarks vs. number of all benchmarks.

free plans, the clear winner is the collision avoidance over collision detection. The extra work put into computing collision avoidance pays off, as the algorithm often does not have to explore a large portion of the state space; unlike when the algorithm uses only collision detection. Nevertheless, Fig. 7 shows the inherent exponential trend of the solution's complexity when it falls into exploring many tentacle orderings.

Nevertheless, our solution is a fast heuristic, and with a short timeout it can serve as the first phase of finding a reconfiguration. One can quickly test if there is a tentacle-based solution and if not, they can fall back into a slower, but more successful method.

VII. Algorithm Analysis

Let us analyze the worst-time complexity of the presented solution. This is the case when there is no tentacle-based solution.

Let n be the number of modules in a configuration. The algorithm starts with a BFS treeify procedure – linear in time to the number of modules ($\mathcal{O}(n)$). The initial number of tentacles can simply be bounded by n. Then, at each step, the worst case scenario is that every tip tries to connect with every other tip before finding a suitable connection – the connect procedure is called up to n^2 times. After making a connection, one tentacle or non-red connection is removed – the steps to make a connection are called at most n times. FABRIK itself is linear to the length of a tentacle (also bounded by n), and n^2 with collision avoidance. Therefore, when following a single ordering of tentacles our algorithm runs in $\mathcal{O}(n^5)$ worst-case. With backtracking enabled, the number of connections we try to make may grow to be exponential, but with the reliability and speed of the FABRIK algorithm, we did not find the option detrimental to the overall performance, as we can see from the high success rate in the experimental evaluation.

VIII. Conclusions

We presented a novel approach for efficient computation of collision-free reconfiguration plans for chain- and hybrid-type self-reconfigurable metamorphic robots with limited self-locomotion. Our approach takes advantage of the symmetry and interchangeability of the modules and thus avoids the expensive computation of a graph isomorphism at the expense of generating larger reconfiguration plans. Therefore, it scales better than traditional approaches and it is able to tackle large configurations. The traditional A*- and RRT-based approaches were able to solve instances only of up to 10 modules in our setup, whereas our solution can handle hundreds. Though, it is a heuristic and it might not find a reconfiguration plan, it has proven to work well on all practically motivated examples. Thanks to its fast run time, it can be used before trying a more expensive method.

Compared to existing solutions based on hand-crafted strategy for reconfiguration our solution does not impose any restrictions on the input shape and can be easily adapted to various arrangements of the modules. Unlike the reconfiguration procedures that do not go through the intermediate shape, our approach allows for efficient precomputation of reconfigurations among a given set of shapes. With a direct reconfiguration, one needs to precompute and store n^2 reconfigurations plans for n shapes; with our approach, only n plans needs to be computed. Also, it is cheap to introduce new shapes into the set.

A. Possible Improvements

We could improve the length of the found reconfiguration plans by considering a different intermediate shape. This shape could be either chosen from a hand-crafted set or it could be derived from the largest common subshape of the source and target shapes.

We plan also explore some heuristics for choosing the order of tentacles to reconnect in order to improve the run time of several pathological cases. Also, we plan to employ a divide and conquer extension to our algorithm, where we can split the configuration into smaller pieces, turn them into snakes and then merge them. This would allow us leverage the fast run times on small configurations to improve run times on large instances.

Lastly, our algorithm produces collision-free plans, however, the plans might not always be schedulable on physicals robots, as some of the movements could exceed the joint's maximal torque (unless they operate in near-zero gravity conditions). Therefore, it would be worth it to explore the possibilities of limiting the set of actions only to actions that do not overload the joints' actuators and experimentally evaluate the impact on performance and success rate.

REFERENCES

[1] A. Aristidou and J. Lasenby. FABRIK: A fast, iterative solver for the inverse kinematics problem. *Graph. Model.*, 73(5):243–260, 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] D. Brandt. Comparison of A* and rrt-connect motion planning techniques for self/reconfiguration planning. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 892–897. IEEE, 2006.

[5] 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.

[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] A. A. Gorbenko and V. Y. Popov. Programming for modular reconfigurable robots. *Programming and Computer Software*, 38(1), 2012.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] C. Liu and M. Yim. A quadratic programming approach to modular robot control and motion planning. In *Fourth IEEE International Conference on Robotic Computing, IRC 2020, Taichung, Taiwan, November 9-11, 2020*, pages 1–8. IEEE, 2020.

[16] J. Mrázek, M. Jonás, and J. Barnat. Reconfiguring metamorphic robots via SMT: is it a viable way? In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - Oct. 1, 2021*, pages 6935–6940. IEEE, 2021.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] J. W. Romanishin, J. Mamish, and D. Rus. Decentralized control for 3d m-blocks for path following, line formation, and light gradient aggregation. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4862–4868. IEEE, 2019.

[26] 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.

[27] 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.

[28] S. Tao, H. Tao, and Y. Yang. Extending FABRIK with obstacle avoidance for solving the inverse kinematics problem. *J. Robotics*, 2021:5568702:1–5568702:10, 2021.

[29] 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.

[30] 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.

[31] 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.