2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

An Obstacle-crossing Strategy Based on the Fast Self-reconfiguration for Modular Sphere Robots

Haobo Luo[1,2] , Ming Li[1,2] , Guangqi Liang[1,2] , Huihuan Qian[1,2] , and Tin Lun Lam[1,2,†]

Abstract— **This paper introduces an obstacle-crossing strategy, and the self-reconfiguration algorithm for a new class of modular robots called the rolling sphere, which can fit obstacles represented by cubes of different sizes due to the chain connection of multiple spheres. For the self-reconfiguration of the rolling spheres, a large gradient is obtained by classifying its action types and hierarchically minimizing the distance between the initial configuration and the final configuration. The most direct use of this large gradient is the fast crossing of various obstacles, by jointing multiple self-reconfigurations according to the OctoMap of the obstacles. It is verified in simulation that the self-reconfiguration takes full advantage of the parallel movement of multiple modules to reduce the total time steps, and the obstacle-crossing strategy can adapt to a variety of obstacles.**

I. INTRODUCTION

Modular Self-Reconfiguration Robots (MSRRs) perform better than traditional fixed robots in versatility and adaptability [1]. MSRRs consist of multiple modules, and the overall configuration can be changed by changing the connection relationship or mutual position. The different configurations are for different tasks and different environmental obstacles. The action sequence transforming one configuration of modular robots into the final configuration is defined as the Self-Reconfiguration (SR) process.

In scenarios such as fire rescue, reaching the destination in the shortest time, regardless of the morphological change of the modular robot in this process, is a more important issue than power consumption. The question discussed in this paper is how to reach the destination with overcoming obstacles in the environment quickly. Butler introduces the sliding cube in [2] and the algorithm called Cellular Automata to overcome obstacles using the sliding transitions and the convex transitions of the sliding cube. The Million Module March algorithm inspired by reinforcement learning is further proposed for the sliding cube in [3]. However, subject to the shape of a cube, the sliding cube can only overcome obstacles composed of module-sized cubes. To fit obstacles composed of different sizes of cubes with the help of the chain connection of multiple spheres, the $rolling\ sphere$ is proposed, as shown in Fig. 1. The Million Module March algorithm [3], as well as other agent-based approaches [4] [2], are difficult to reproduce the best results and debug when it is actually deployed on hardware [5]. The more applicable solution is to fit the shape of the obstacle and joint two self-reconfiguration processes, which is inspired by a divide-and-conquer approach [6].

As for the reconfiguration algorithms of MSRRs, they are summarized into four categories [1]: bio-inspired approaches, agent-based approaches, search-based approaches, and control-based approaches. The search-based approaches such as [7] [8] [9], suffer from the volume of configuration space [10] and the computation of the guided search [11] which grow exponentially with the number of modules. So we focus on designing non-sampling control algorithms, which simulates the feedback control loop to navigate the sequence of configuration changes. In the gradient-based [12] [13] [14] method, non-source modules follow the steepest descent way computed by the density of neighbors' attractors which are broadcast by a source module. Our self-reconfiguration algorithm also computes the steepest gradient in three steps. A global goal ordering method is introduced for the reconfiguration of the Proteo models [15], but it may fall into a local minimum when composing multilayer entities. For our obstacle-crossing strategy, the layer number is limited to be less than 3 so as to avoid the local minimum. 3D Catoms [16] use a decentralization method to assemble various shapes. This decentralization method requires each agent to have the same input and a non-random calculation process, so that each agent can choose its own actions from the same output according to its own Id. We also applied this method. 3D Catoms [17] [18] further propose the distributed reconfiguration method through messaging, but require a free position between the modules that move concurrently.

Our obstacle-crossing strategy consists of several consecutive self-reconfiguration processes and the actions to joint them. The final or initial configuration in those SR processes are designed to fit obstacles rather than arbitrary morphological changes. This obstacle-crossing strategy simulates the adaptability of water to different surfaces, thus also called the flow process. The contributions of this paper are concluded as:

(1) A fast parallel self-reconfiguration algorithm based on the $rolling\ sphere$ using three-step minimization for a large gradient.
(2) An obstacle-crossing strategy: jointing multiple self-reconfiguration processes to fit and cross various obstacles.

The content of this article is organized as follows. Section II details the problem. Section III and IV introduce the self-reconfiguration algorithm and the obstacle-crossing strategy. Simulation results are shown in Section V.

* This paper is partially supported by funding 2019-INT008 from the Shenzhen Institute of Artificial Intelligence and Robotics for Society.

[1] The Chinese University of Hong Kong, Shenzhen, China.
[2] Shenzhen Institute of Artificial Intelligence and Robotics for Society.
† Corresponding author: Tin Lun Lam, email: tllam@cuhk.edu.cn

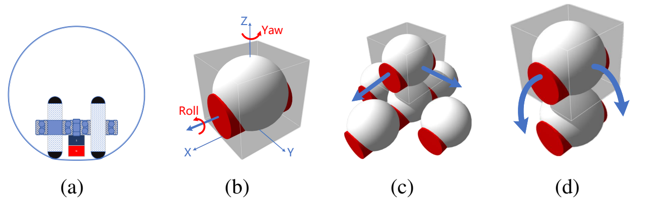


Fig. 1. The *rolling sphere* and two of its basic actions. (a)the sphere robot. The trolley in the lower part of the spherical metal shell has differential wheels and an electromagnet. (b) the simulation in Gazebo. The red cylinder indicates the orientation of the differential wheels, which are actually inside the sphere. (c) four possible sliding transitions; (d) four possible convex transitions. Their two opposite directions are not drawn.

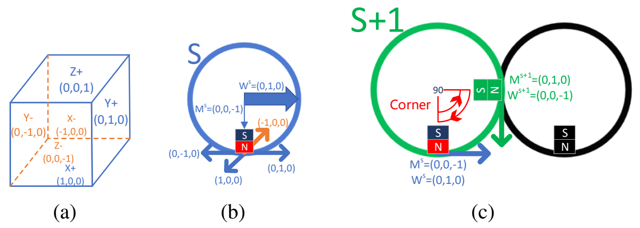


Fig. 2. Two direction vectors of each module and the *Corner* transitions. (a) Six normals on six faces representing six possible directions; (b) M^s and W^s, two direction vectors of each module at step S; (c) The *Corner* transitions that change the M^s of the module to the M^{s+1} in the next step $S+1$.

II. PROBLEM FORMULATION

A. The rolling sphere and its prerequisites

The *rolling sphere* consists of a rough metal spherical shell and a trolley in the lower half. The trolley has two differential wheels and an electromagnet, as shown in Fig. 1(a). In Fig. 1(b), the red cylinder simulated in Gazebo is protruding from the sphere for the convenience of indicating the orientation of the differential wheels, which are actually inside the sphere. We idealize this hardware and summarize it into the following three prerequisites that can be achieved:

(1) 3D positioning capability;
(2) 3D perception of the environment;
(3) The disturbance caused by detaching, sliding and convex transitions is negligible.

The prerequisites (1)(2) are used to generate the input signals of the SR algorithm, which are the position and orientation of each module and the OctoMap [19] of obstacles in the environment. Prerequisite (3) is a condition to realize four kinds of basic actions of each module. Since the *rolling sphere* has unlimited connection points [20], we call it a free-form robot.

In the discrete motion space, we use the normals of six faces of a cube, as shown in Fig. 2(a), to represent the pointing of the N pole of the magnet in the S^{th} step, namely M^s, and the forward directions of the differential wheels in the S^{th} step, namely W^s. There are six options for the direction of the magnet, as shown in the Eq. (1). Fig. 2(b) gives an example of $M^s = (0,0,-1)$. For the inner trolley inside the *rolling sphere*, it has only two driving forces, as shown in Fig. 1(b), yaw and roll. With the Yaw force, the trolley in the example of Fig. 2(b) can choose from four directions as the forward direction of the differential wheels W^s, as shown in the Eq. (2). With the roll force, the *rolling sphere* can perform four kinds of basic actions, $\{Pan, Corner, Sliding, Convex\}$.

$$\begin{aligned} M^s \in \{&x+ : (1,0,0); x- : (-1,0,0);\\ &y+ : (0,1,0); y- : (0,-1,0);\\ &z+ : (0,0,1); z- : (0,0,-1).\} \end{aligned} \tag{1}$$

$$\begin{aligned} &if\ M^s \in \{z+ : (0,0,1); z- : (0,0,-1).\}\\ &then\ W^s \in \{x+ : (1,0,0); x- : (-1,0,0);\\ &\qquad y+ : (0,1,0); y- : (0,-1,0).\},\ etc. \end{aligned} \tag{2}$$

Pan is defined as moving on the ground when no other modules block it. If blocked by other modules in front, the inner trolley climbs up to 90° along the inner spherical shell. Its magnet at this time will be attracted to that module, from $M^s = (0,0,-1)$ to $M^{s+1} = (0,1,0)$ shown in Fig. 2(c). Conversely, if the magnet of the module in the $(S+1)^{th}$ step, M^{s+1}, was originally attracted to other modules and the forward direction W^{s+1} chosen from Eq. (2) was blocked by the ground or other modules, the inner trolley would also climb down for 90°. These movements are called the $Corner$ transitions.

The last two kinds of basic actions are the $Sliding$ transitions and the $Convex$ transitions, as shown in Fig. 1(c)(d) and Fig. 6(c)(e). For anyone of the directions, W^s, when its movement is supported by adjacent modules, it is a $Sliding$ transition along the surfaces of two modules. If there is no support from adjacent modules, it is a $Convex$ transition along the center of a module. In reality, these two actions require the support of a certain 3D connection structure and sufficient friction to ensure that the entire modular robot will not be dragged by gravities or disturbances. Thus we set the prerequisite 3 to focus on path planning. Note that the *rolling sphere* only can choose W^S parallel to the face touched by the inner trolley. The sliding cube [] can choose directions based on any of the six faces of the cube.

Further, the $Corner$ transitions and the Pan transitions are combined to be the $Retreat$ transitions in Fig. 6(a). Section III-B will describe four kinds of the combined actions: $\{Retreat, Corner\ Retreat, Corner\ Sliding, Corner\ Convex\}$.

B. The detail of the reconfiguration problem

Reconfiguration can be roughly represented by Eq. (3) where the $CFG_{current}$ and CFG_{final} represent the current configuration and the final configuration. The definition of configuration should be clear-cut with an appropriate complexity for our modular robot. For example, the graph representation [21] is defined by pure connection relations, which can't distinguish configurations with different gravity distributions or functionalities. Whereas, it's too complex to use the transformations of every module as the configuration shown in Eq. (4) and (5). For this reason, we replace R_i^c in Eq. (5) by two range-limited vectors, M^s in Eq. (1) and the corresponding W^s in Eq. (2). Further, we limit the three coordinates of P, in Eq. (5) and (6), to only integer multiples of the radius of the sphere module which means the motion space is discrete. The discrete space facilitates the calculation of trajectory planning and collision detection. Now, the current configuration is defined as Eq. (7). The final configuration is defined as Eq. (8), given that only positions and connection relationships can be extracted from OctoMap of obstacles. The superscripts c and f indicate the current configuration and the final configuration, respectively.

$$Route(t) = Reconfiguration(CFG_{current}, CFG_{final}) \tag{3}$$

$$CFG_{current} = \{[Id_i, T_i^c] \mid i = 1, \ldots, n\} \tag{4}$$

$$T_i^c = \begin{bmatrix} R_i^c & \vdots & P_i^c \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_i^c \\ x_{21} & x_{22} & x_{23} & y_i^c \\ x_{31} & x_{32} & x_{33} & z_i^c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$P_i^c = [x_i^c, y_i^c, z_i^c]^T, i = 1, \ldots, n. \tag{6}$$

$$CFG_{current} = \{[Id_i, P_i^c, M_i^c, W_i^c] \mid i = 1, \ldots, n\} \tag{7}$$

$$CFG_{final} = \{[P_i^f, M_i^f] \mid i = 1, \ldots, n\} \tag{8}$$

Based on the prerequisites 1 and 2, each module can get the transformations of all modules, $CFG_{current}$ in Eq. (3), and a co-built shared OctoMap, as shown in Fig. 3(a). One of the ideas of decentralized computing [16] is that each module has the same input and calculation process, and then selects its own route from the same output according to its own Id, as shown in Fig. 3(b). As far as the algorithm is concerned, it can be just understood as centralized control.

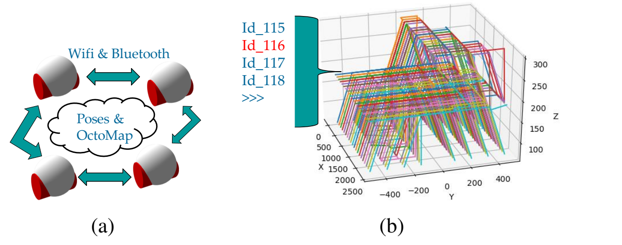


Fig. 3. The input and output. (a) Each module is input the Poses of all modules and the OctoMap of obstacles. (b) Select a route from the output based on its Id.

We mathematically define the problem of self-reconfiguration as Eq. (9), where P^c represents the set of 3D positions of all modules in the current configuration, as shown in Eq. (10). D_j represents the multiplication of n Manhattan distances between one of the $vacancies$ j in the final configuration and the n modules in the current configuration. In Eq. (11), the reason for using the multiplication for D_j is that the position of the j^{th} $vacancy$ in the final configuration can be occupied by any module i in the current configuration. The Manhattan distance is calculated by giving two weights $w_x, w_y \in (0, 1]$ to the $x - y$ plane, as shown in Eq. (12). Eq. (9) expresses that the purpose of the self-reconfiguration algorithm is to fill all the $vacancies$ $j = 1, \ldots, n$ in the final configuration by adjusting the 3D position of each module in the current configuration. The planned routes in Eq. (13) is the P^c at each step S of the minimization process in Eq. (9).

$$p^* = \arg\min_{P^c} \sum_{j=1}^{n} D_j \tag{9}$$

$$P^c = \{P_i^c \mid i = 1, \ldots, n\} \tag{10}$$

$$D_j = \left(\prod_{i=0}^{n} \left| P_i^c - P_j^f \right| \right) \tag{11}$$

$$\left| P_i^c - P_j^f \right| = w_x \times \left| x_i^c - x_j^f \right| + w_y \times \left| y_i^c - y_j^f \right| + \left| z_i^c - z_j^f \right| \tag{12}$$

$$Route(s) = \{P^c \mid s = 0, 1, \ldots\} \tag{13}$$

III. THE SELF-RECONFIGURATION ALGORITHM

A. The global priority and setting goals

This section uses a simple example to explain how to assign a goal position to each module in the current configuration. In Fig. 4, the four modules on the left represent the current configuration, and the four colored cubes called $vacancies$ on the right represent the final configuration, one of which is black and translucent. the corresponding feedback control scheme for one step is shown in Fig. 5.

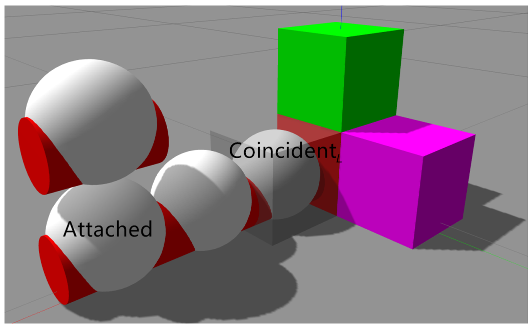


Fig. 4. The example explains the calculation of global priority and setting goals. Left: the current configuration; Right: the final configuration, including the black and translucent cube.

In Fig. 5, the sequence S_f represents the different $vacancies$ in the final configuration with the inner number indicating their global priority. S_f is obtained by layering $vacancies$ in the final configuration and sorting them according to their Manhattan distances from the center of this layer. Sorting within a layer can select the mode (maximum,

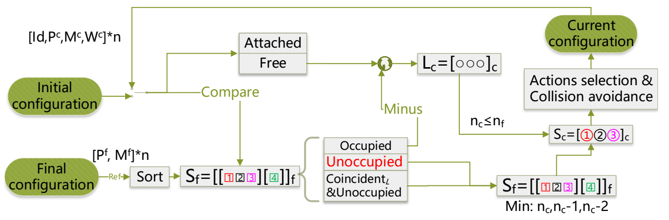


Fig. 5. The reconfiguration control scheme for one step. Different colors of the circles in S_c represent the target *vacancy* assigned to each movable module in the current configuration. The numbers in the squares or circles indicate global priorities.

minimum, or center) to calculate the Manhattan distance with each module according to different tasks. After comparing with the current configuration, delete occupied *vacancies* from the head of S_f until unoccupied *vacancies* appear. At this time, there may still exist occupied *vacancies* in S_f, such as the *vacancy* marked with $Coincident_L$ in Fig. 4, which is ranked behind the unoccupied red cubes in Fig. 4 in S_f.

To obtain the ordered module sequence, S_c in Fig. 5, we assign a *vacancy* in S_f as the goal position to each module in the unordered module list L_c, and use the global priority of this *vacancy* as the priority of the module to avoid collisions when selecting actions. The unordered module list L_c is obtained by removing the attached modules such as the module marked with *Attached* in Fig. 4, and the modules occupying one *vacancy* deleted from S_f. Thus the length of L_c, n_c, is less than the length of S_f, n_f. The $n_c < n_f$ guarantees that each module in L_c can be assigned a goal *vacancy* and global priority. For the S_c in Fig. 5, the number and the color represent the global priority and the goal *vacancy* assigned to the module separately.

The *vacancy* ranked in the i^{th} $(i \in [0, n_f - 1])$ position in S_f will be assigned to the nearest one of the remaining $n_c - i$ modules. For example, the *vacancy* represented by the red cube will select the one with the smallest Manhattan distance from the n_c modules, namely the module inside the black and translucent cube in Fig. 4. The reason for this is to minimize D_j in Eq. 11 and thus minimize the total distance in Eq. 9. For any module $i \in S_c$, its relationship with the other modules in the current configuration can be expressed as $[[high\,priority][i][low\,priority][Attached, Coincident_L]]_c$.

B. The candidate actions and collision avoidance

We calculate 4 candidate actions corresponding to four forward directions W^s in Algorithm 1, based on M^s in Fig. 2(b). If any W^s is blocked by immovable modules, $[Attached, Coincident_L]$, the *Corner* transition in Fig. 2(c) will change M^s and result in the second calculation of 4 candidate actions. Due to the second calculation of candidate actions, there are four types of combined actions as shown in the Fig. 6 created by combining the *Corner* transition in front.

To explain the concept of the calculation of 4 candidate actions for two times, we give an example shown in Fig. 7 as

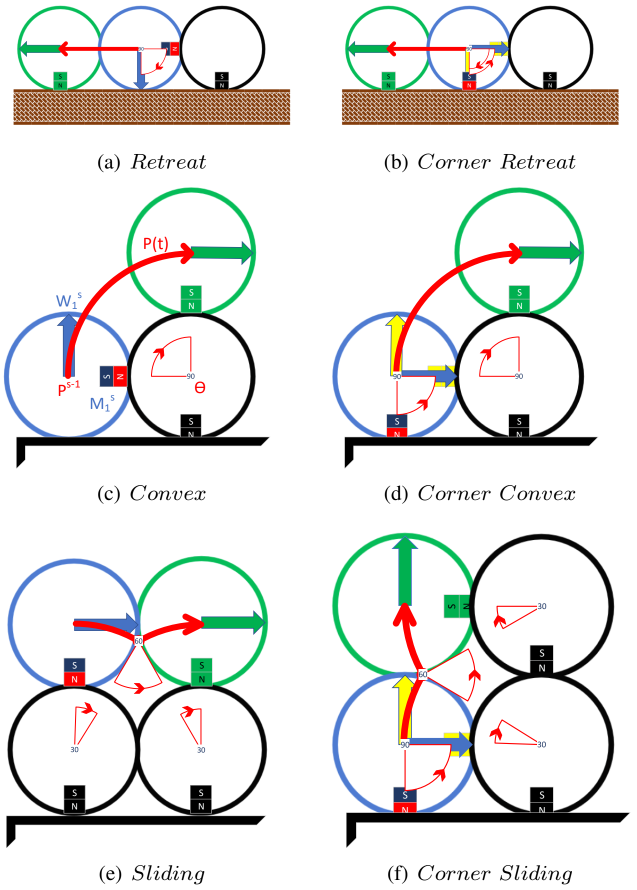


Fig. 6. 4 kinds of combined actions: $\{(a), (b), (d), (f)\}$. The green circle and arrow represent the goal position for this type of action and the expected forward direction of the wheel. The yellow arrow and magnet represent the result position of the first selected action before 4 new candidate actions are going to be calculated. The red arrow and fans represent the trajectory and the rotation angles of the type of action.

well as the corresponding code in Algorithm 2. In the three views, the yellow trajectories represent the first calculation of 4 candidate actions. The result of one of the candidate actions conflicts with the existing immovable module, the 3rd module in Fig. 7(d), so the direction of the magnet M^s is changed to point to that immovable module. Based on the new M^{s+1}, 4 new candidate actions represented by the red trajectories are calculated. The result of one action in 4 new candidate actions conflicts with the 1st module in Fig. 7(b) again. At this time, we no longer calculate thirdly, because the obstacle-crossing capability that adapts to the surface of different obstacles is mainly achieved by the coating of two-layer modules. If this self-reconfiguration algorithm aims to form dense shapes of multi-layer, more than three times' calculation is needed. Moreover, focusing on the calculation of two-layer modules can avoid falling into the local bucket minimum to ensure convergence.

There are three cases for collision avoidance. For each action, if its target position conflicts with the target position of the action selected by the higher priority module, it will be deleted in Algorithm 1. If the selected action conflicts with the current position of a lower priority module, then this

Algorithm 1 Calculate four candidate actions from four W^s

```
function $fun\_del\_collision(CFG_{current}, S_c)$
    if Collide with higher priority modules or obstacles then
        Delete this action.
    end if
end function

function $fun\_W\_to\_actions(CFG_{current}, S_c)$
    for $i \in S_c$ do
        for 4 choices of $W^s$ of module i do
            if Collide with the ground then
                Action type: $Retreat$
                $fun\_del\_collision$
            end if
            if The supporting module exist then
                Action type: $Sliding$
                $fun\_del\_collision$
            else
                Action type: $Convex$
                $fun\_del\_collision$
            end if
        end for
        Sort by the distances of these actions' result
        if $CFG_{current}[i].stop == True$ : then
            Action type: Stop
        end if
    end for
    return 4 actions or Stop
end function
```

Algorithm 2 Calculate four candidate actions for two times

```
function $fun\_two\_times(CFG_{current}, S_c)$
    for actions in $fun\_W\_to\_actions()$ do
        if Doesn't collide with $CFG_{current}$ then
            (Retreat/Sliding/Convex)
        else
            if Collide with the module $h_{low\_priority}$ then
                $CFG_{current}[h].stop = False$;
                (Retreat/Sliding/Convex)
            else
                Substitute the new $M^{s+1}$;
                for actions in $fun\_W\_to\_actions()$ do
                    if Doesn't collide with $CFG_{current}$ then
                        Corner+(Retreat/Sliding/Convex)
                    else
                        if Collide with the module $h_{low\_p}$ then
                            $CFG_{current}[h].stop = False$;
                            Corner+(Retreat/Sliding/Convex)
                        else
                            pass;
                        end if
                    end if
                end for
            end if
        end if
    end for
end function
```

lower priority module cannot choose to stop in Algorithm 2. If a low-priority module has not collected any action, it needs to request the right to stop, from the higher-priority module collided with him. Thus, all possible candidate actions are collected through Algorithms 1 and 2.

In order to minimize the sum of D_j in Eq. 14, each D_j should take the minimum value. Because the minimization of the product of Manhattan distances is mostly affected by the smallest of all distances, we choose to minimize the smallest one in Eq. 15 to obtain a large gradient. Then choose one of the collected candidate actions that further minimizes this distance, as shown in Eq. 16. Thus we have the remark 1.

$$\arg\min_{P^c} \sum_{j=0}^{n} D_j = \sum_{j=0}^{n} \arg\min_{P^c} \left(\prod_{i=0}^{n} \left| P_i^c - P_j^f \right| \right) \tag{14}$$

$$\left| P_{i^*}^c - P_j^f \right| = \min\{ |P_i^c - P_j^f| \mid i = 1, \dots, n \} \tag{15}$$

$$The\ selected\ action = \arg\min_{actions} \left| P_{i^*}^c - P_j^f \right| \tag{16}$$

Remark 1: The large gradient of self-reconfiguration is obtained by taking the minimum values in three steps' calculation as shown in Eq. (14), (15) and (16).

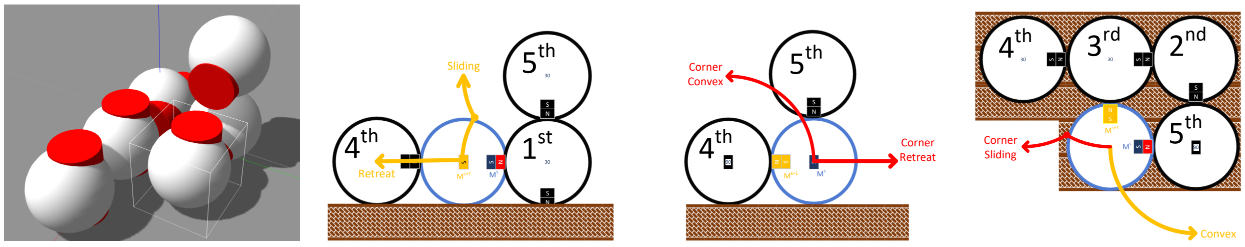
IV. THE OBSTACLE-CROSSING STRATEGY

A. Jointing multiple reconfigurations

The obstacle-crossing strategy is jointing multiple reconfiguration processes whose final or current configurations are adjusted to fit the obstacle's OctoMap. Let us take a simple 2D obstacle in Fig. 8 as an example. After OctoMap decoding, the brown obstacle in Fig. 8(a) is represented by the cubes of different sizes similar to Fig. 9(b). The $Z-Y$ plane of the coordinate system of OctoMap is shown in Fig. 8(a). This coordinate system is discretized using the diameter of the module as one unit. The $Z-Y$ plane is cut into green squares, as shown in Fig. 8(a). For each square containing obstacles, such as one of the squares $\{6, 7, 8, 9\}$ in Fig. 8(a), calculate whether the 8 squares surrounding it contain obstacles. For example, among the 8 squares surrounding square 6, $\{1, 2, 3, 4, 5\}$ has no obstacles, and $\{7, 8, 9\}$ contain obstacles. For square 8, there are only two squares $\{1, 2\}$ surrounding it that do not contain obstacles. $\{1, 2, 3, 4, 5\}$ and $\{1, 2\}$ are gathered increasingly to compose the $vacancies$ surrounding the obstacle. They will be used on the right side of Fig. 4 as the final configuration. The number of $vacancies$ extracted is guaranteed to be not less than the number of modules in the current configuration.

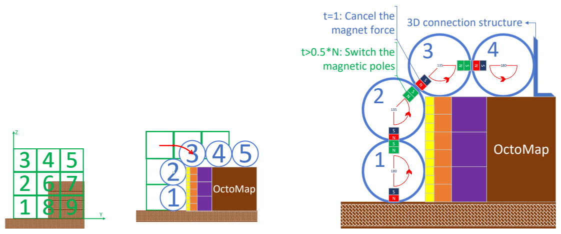
The obstacle's OctoMap contains cubes of various sizes, and they are impossible to match the squares in Figure 8(a). So for the calculated $vacancies$ surrounding the obstacle, we need to adjust their positions in turn following the connection order $\{1 \to 2 \to 3 \to 4 \to 5\}$, as shown in Fig. 8(b). For example, No. 1 $vacancy$ pans forward a distance until collision with the OctoMap. No. 3 $vacancy$ also pans forward the same distance as No. 1 but needs to rotate around the panned No. 2 $vacancies$ by an angle to collide with the OctoMap. Thus we have the remark 2.

Remark 2: The final configuration fits the obstacles represented by cubes of different sizes by adjusting the positions



(a) 3D view (b) front view (c) left view (d) top view

Fig. 7. An example explains the calculation of 4 candidate actions for two times. The yellow trajectory represents the first calculated candidate actions, and the red arrow represents the second calculated actions.


t=1: Cancel the magnet force
3D connection structure
t>0.5*N: Switch the magnetic poles
OctoMap
OctoMap

(a) (b) (c)

Fig. 8. The obstacle-crossing strategy. (a) Extract a layer of *vacancies* just surrounding the obstacle. (b) Adjust the position following the connection order to fit obstacles represented by cubes with different sizes. (c) Jointing two consecutive self-reconfiguration processes. To reverse the connection order, the electromagnet on the inner trolley demagnetizes at the beginning of the operation, and the magnetic pole is changed after half the time.

and angles of goal vacancies following the connection order.

After the above two steps, a single reconfiguration process can control the movement of MSRR from the initial configuration to the adjusted final configuration shown in Fig. 8(b). To overcome obstacles, we also need sequential operations to reverse the connection order from $\{1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5\}$ to $\{5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1\}$, as shown in the Fig. 8(c). For example, No. 1 in Fig. 8(c) must climb $180°$ upwards. When it climbs to $90°$, it changes the magnetic pole of the electromagnet to attract the upper magnet for upward traction. Next, at the moment No. 2 starts to act, its magnetism is canceled for leaving. We call those operations the $Jointing$ transitions. With the $Jointing$ transition to reverse the connection order, module No. 1 is not $Attached$ but $Free$ in the reconfiguration feedback control scheme in Fig. 5, so that it can climb up along No. 2 and No. 3 in turn. In other words, another reconfiguration process is happening.

B. Smoothing

Since the self-reconfiguration algorithm is based on discrete motion space, we need to calculate the corresponding continuous routes to navigate the trolley inside the sphere module. The method of calculating continuous routes of each type of action is similar. The information used includes the two direction vectors at the beginning and end of each action, $\{M_1^s, W_1^s, M_N^s, W_N^s\}$. For example, the cross product of two direction vectors can be used as a rotation axis. Note that $M_N^{s-1} = M_1^s$, which means the orientation of the electromagnet at the beginning of the action is the same as the orientation at the end of the previous step.

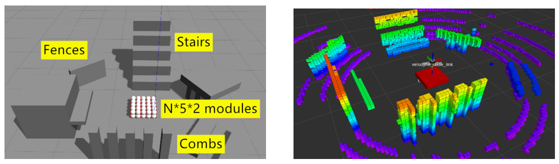
For different kinds of basic actions or combined actions collected by Algorithms 1 and 2, the rotation axis and rotation angle are used to calculate the increment $\triangle R$ between the current step s and the previous step s-1, so that the orientation in the current step can be calculated as $R_s = \triangle R \times R_{s-1}$. The translation in the current step $P(t), t = 0, \ldots, N$ is calculated based on the diameter d of the sphere module and the rotation angle. Eq. (17) and (18) show the smoothing of the $Corner$ transitions in the Fig. 6(c). The $|M^{s-1}|$ in Eq. (18) means taking the absolute value of each element, rather than the determinant of M^{s-1}. The smoothing of the $Jointing$ transitions is similar, but the calculation of fitting obstacles in Fig. 8(b) requires the collision detection with the OctoMap in order to calculate the panning distance or rotation angle of each goal $vacancy$ in turn.

$$\begin{aligned} Rotation\ Axis &= W_1^s \times M_N^{s-1} \\ Rotation\ Angle\ \Theta &= \frac{t}{N} \cdot 90° \end{aligned} \tag{17}$$

$$\begin{aligned} P(t) =& ([1,1,1]^T - |M_1^s|) \cdot (P^{s-1} + W_1^s \cdot d \sin\Theta) \\ &+ |M_1^s| \cdot (P^{s-1} + M_1^s \cdot d(1-\cos\Theta)) \end{aligned} \tag{18}$$

V. SIMULATION

We conduct two types of simulations in Gazebo, using ROS packages to control each module. In Gazebo, the position and orientation information is directly read from the topic $\backslash gazebo \backslash model\_states$, and the OctoMap of obstacles in the environment is created by a Lidar at the same starting position of the MSRR, as shown in Fig. 9.



(a) (b)

Fig. 9. The simulation scene. (a) A cuboid composed of $N \times 5 \times 2$ modules and three types of obstacles. (b) The OctoMap of the obstacles.

We define one step as the isometric time required for every action in the basic action set and the combined action set. The total steps are related to (1) the number of modules, (2) the

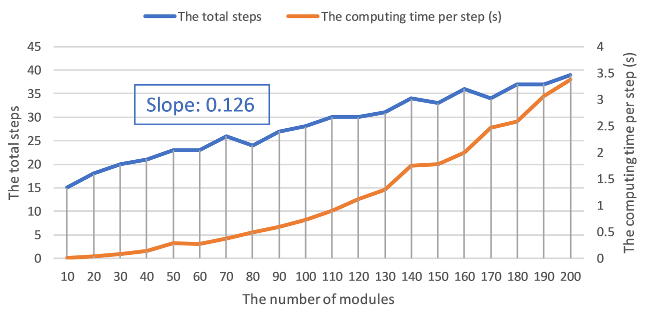


Fig. 10. The simulation results for $N \times 5 \times 2$ modules moving over a 4 module-diameter distance on flat ground. One step is the isometric time required for a single basic action or combined action.

morphological difference between the initial configuration and the final configuration, (3) the center distance between the initial configuration and the final configuration. Among them, the influence of variable 3 is obviously proportional. Our first type of simulation explores the relationship between the total steps and variable 1 under the control of variable 2. As shown in Figure 9(a), the initial configuration is a cuboid with length, width, and height of $N \times 5 \times 2 (N \in [1, 20])$, and the final configuration is also a $N \times 5 \times 2$ cuboid. The center distance is controlled to be 4 module-diameter distances in the direction of the width of the cuboid. The experimental results by adjusting the length N from 1 to 20 are shown in Fig. 10. Fig. 3(b) shows the routes before smoothing in the case of $N = 20$. The slope of the green curve, the required steps over the number of modules, in Fig. 10 is fitted as 0.126. This is an optimal value due to the parallel movement of many modules in one step, incorporated with the maximum gradient described in Remark 1 to achieve fast reconfiguration. The cost of this parallel movement is an increase of the amount of calculation, as shown in the yellow curve in Fig. 10.

Because the hardware drive of each modular robot is different, we can only get a rough sense of the comparison of the algorithms. let us compare the similar simulation result of the sliding cube [3]. The required steps of the sliding cube in the simulation of $125 (5 \times 5 \times 5)$ modules are 69 steps. Our simulation data are 31 steps for $130 (13 \times 5 \times 2)$ modules but 105 steps for $125 (5 \times 5 \times 5)$ modules. Our algorithm is slower than the sliding cube controlled by reinforcement learning in the self-reconfiguration of 5-layer configurations. Because the *rolling sphere* can only choose actions on one face but six faces as for the sliding cube, the *rolling sphere* cannot utilize the advantages of parallel movement when the configuration height is 5 layers as thoroughly as the sliding cube. When the rolling spheres compress the height and expand the length and width to facilitate parallel movement, a similar obstacle-crossing speed with the reinforcement learning method is obtained. The improvement of our algorithm over the sliding cube is that we can adapt to obstacles of different sizes using 2 layers configurations, as shown in Remark 2, instead of obstacles that are all composed of module-sized cubes.

TABLE I

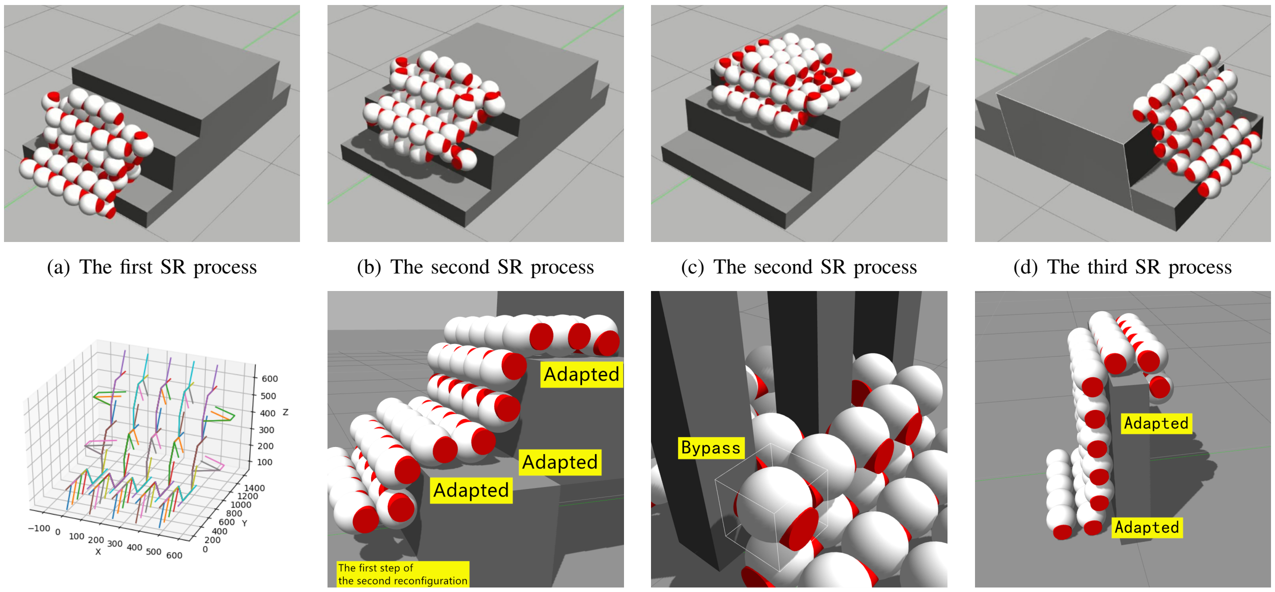
THE SIMULATION RESULTS

The obstacle type	The total steps	The computing time per step (s)
Stairs	53	0.3672 ± 0.01
Combs	40	0.2901 ± 0.01
Fences	46	0.3065 ± 0.01

The second type of action is to explore the influence of the morphological difference between the initial configuration and the final configuration using $50 (5 \times 5 \times 2)$ modules. Table I shows the simulation results for crossing three different types of obstacles. The final or initial configuration is now determined by the shape of the obstacle, as described in Remark 2. Take the stairs as an example. The obstacle-crossing strategy is jointing two reconfiguration processes. The first process starts from the initial configuration shown in Fig. 9 to the final configuration in Fig. 11(f) that adapts to obstacles. Fig. 11(e) shows all the routes before smoothing of the first reconfiguration process. After the adaptation to the different obstacles and the *Jointing* operation shown in Fig. 8, the configuration in Fig. 11(f) can be obtained. This configuration is adapted to the stair, such as the three corners of the stairs marked as *Adapted* in Fig. 11(f). The bottommost module in Fig. 11(f) starts to climb up along the other modules in order, which is the first step of the second self-reconfiguration process. The second process takes the final configuration of the first process as the initial configuration, and the configuration that can adapt to any subsequent obstacles as the final configuration. It can be found in Fig. 11(a)(b)(c)(d) that the entire obstacle-crossing processes makes full use of parallel movement and can well adapt to the shape of the stairs. Except for the obstacle types in Fig. 11(f)(g)(h), there is at least one obstacle that this algorithm cannot handle. That is, an over-hanging [3] obstacle similar to the edge of a table. The OctoMap of this obstacle is not enough to extract the final configuration that can be used to climb it.

VI. CONCLUSIONS AND FUTURE WORK

We propose an obstacle-crossing strategy based on a new class of sphere modular robots to overcome obstacles composed of cubes of different sizes. This obstacle-crossing strategy includes jointing multiple self-reconfiguration processes that have achieved a large gradient, and adjusting the discrete final configuration to fit the OctoMap of the obstacle. In this article, we detail the three formulas to obtain large gradients in the self-reconfiguration process and how to adjust the sphere modules that make up a chain connection order to fit obstacles composed of cubes of different sizes. Future work will focus on using distributed computing to reduce the computational load of each module and trying to overcome unusual obstacles such as overhanging obstacles and gaps.



(a) The first SR process (b) The second SR process (c) The second SR process (d) The third SR process

(e) Rough paths (f) Stairs (g) Combs (h) Fences

Fig. 11. The obstacle-crossing strategy. (a)(b)(c)(d) Three self-reconfiguration processes used to cross stairs. (e) The routes before smoothing of the first self-reconfiguration process. (f) The first step of the second self-reconfiguration process. In this step, the bottommost modules start to climb up along the other modules, which have adapted to the stair whose length and width are not an integer multiple of the module diameter. (g)(h) the other two obstacles.

References

[1] H. Ahmadzadeh and E. Masehian, "Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization," *Artificial Intelligence*, vol. 223, pp. 27–64, 2015.

[2] Z. Butler, K. Kotay, D. Rus, and K. Tomita, "Generic decentralized control for lattice-based self-reconfigurable robots," *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 919–937, 2004.

[3] R. Fitch and Z. Butler, "Million module march: Scalable locomotion for large self-reconfiguring robots," *The International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 331–343, 2008.

[4] P. Varshavskaya, L. P. Kaelbling, and D. Rus, "Automated design of adaptive controllers for modular robots using reinforcement learning," *The International Journal of Robotics Research*, vol. 27, no. 3-4, pp. 505–526, 2008.

[5] J. Phipps, "An analysis of the million module march algorithm applied to the atron robotic platform," 2011.

[6] G. Aloupis, S. Collette, E. D. Demaine, S. Langerman, V. Sacristán, and S. Wuhrer, "Reconfiguration of cube-style modular robots using o (logn) parallel moves," in *International Symposium on Algorithms and Computation*. Springer, 2008, pp. 342–353.

[7] A. Pamecha, I. Ebert-Uphoff, and G. S. Chirikjian, "Useful metrics for modular robot motion planning," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 4, pp. 531–545, 1997.

[8] T. Larkworthy and S. Ramamoorthy, "An efficient algorithm for self-reconfiguration planning in a modular robot," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 5139–5146.

[9] J. Liu, Y. Wang, B. Li, S. Ma, and D. Tan, "Center-configuration selection technique for the reconfigurable modular robot," *Science in China Series F: Information Sciences*, vol. 50, no. 5, pp. 697–710, 2007.

[10] M. Eden *et al.*, "A two-dimensional growth process," in *Proceedings of the fourth Berkeley symposium on mathematical statistics and probability*, vol. 4. University of California Press Berkeley, 1961, pp. 223–239.

[11] F. Hou and W.-M. Shen, "On the complexity of optimal reconfiguration planning for modular reconfigurable robots," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 2791–2796.

[12] K. Støy, "Controlling self-reconfiguration using cellular automata and gradients," in *Proceedings of the 8th international conference on intelligent autonomous systems (IAS-8)*, 2004, pp. 693–702.

[13] K. Stoy, "How to construct dense objects with self-recondfigurable robots," in *European Robotics Symposium 2006*. Springer, 2006, pp. 27–37.

[14] K. Stoy and R. Nagpal, "Self-reconfiguration using directed growth," in *Distributed autonomous robotic systems 6*. Springer, 2007, pp. 3–12.

[15] M. Yim, Y. Zhang, J. Lamping, and E. Mao, "Distributed control for 3d metamorphosis," *Autonomous Robots*, vol. 10, no. 1, pp. 41–56, 2001.

[16] T. K. Tucci, B. Piranda, and J. Bourgeois, "A distributed self-assembly planning algorithm for modular robots," 2018.

[17] P. Thalamy, B. Piranda, F. Lassabe, and J. Bourgeois, "Scaffold-based asynchronous distributed self-reconfiguration by continuous module flow," 2019.

[18] P. Thalamy, B. Piranda, and J. Bourgeois, "Distributed self-reconfiguration using a deterministic autonomous scaffolding structure," 2019.

[19] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, no. 3, pp. 189–206, 2013.

[20] G. Liang, H. Luo, M. Li, H. Qian, and T. Lam, "Freebot: A freeform modular self-reconfigurable robot with arbitrary connection point - design and implementation," 2020.

[21] 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*, vol. 3839. International Society for Optics and Photonics, 1999, pp. 246–257.