

5-2017

Algorithms for Modular Self-reconfigurable Robots: Decision Making, Planning, and Learning

Ayan Dutta

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

 Part of the [Computer Sciences Commons](#)

Algorithms for Modular Self-reconfigurable Robots: Decision Making, Planning, and Learning

By

Ayan Dutta

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Dr. Prithviraj Dasgupta

May, 2017

Supervisory Committee:

Prithviraj Dasgupta (chair)

Hesham Ali

Yuliya Lierler

Vyacheslav Rykov

Carl Nelson

ProQuest Number: 10271409

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10271409

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Algorithms for Modular Self-reconfigurable Robots: Decision Making, Planning, and Learning

Ayan Dutta, Ph.D.

University of Nebraska, 2017

Advisor: Dr. Prithviraj Dasgupta

Modular self-reconfigurable robots (MSRs) are composed of multiple robotic modules which can change their connections with each other to take different shapes, commonly known as configurations. Forming different configurations helps the MSR to accomplish different types of tasks in different environments. In this dissertation, we study three different problems in MSRs: partitioning of modules, configuration formation planning and locomotion learning, and we propose algorithmic solutions to solve these problems.

Partitioning of modules is a decision-making problem for MSRs where each module decides which partition or team of modules it should be in. To find the best set of partitions is a NP-complete problem. We propose game theory based both centralized and distributed solutions to solve this problem. Once the modules know which set of modules they should team-up with, they self-aggregate to form a specific shaped configuration, known as the configuration formation planning problem. Modules can be either singletons or connected in smaller configurations from which they need to form the target configuration. The configuration formation problem is difficult as multiple modules may select the same location in the target configuration to move to which might result in occlusion and consequently failure of the configuration formation process. On the other hand, if the modules are already in connected configurations in the beginning, then it would be beneficial to preserve those initial configurations for placing them into the target configuration as disconnections and re-connections are costly operations. We propose

solutions based on an auction-like algorithm and (sub) graph-isomorphism technique to solve the configuration formation problem.

Once the configuration is built, the MSR needs to move towards its goal location as a whole configuration for completing its task. If the configuration's shape and size is not known *a priori*, then planning its locomotion is a difficult task as it needs to learn the locomotion pattern in dynamic time – the problem is known as adaptive locomotion learning. We have proposed reinforcement learning based fault-tolerant solutions for locomotion learning by MSRs.

To my parents and my wife.

Author's Acknowledgements

First and foremost, I would like to express my deepest gratitude towards my advisor, Dr. Prithviraj Dasgupta, who has guided me throughout my tenure as a Ph.D. student. It was my pleasure to work with you over the last five years.

I would like to thank the members of my supervisory committee: Dr. Hesham Ali, Dr. Yuliya Lierler, Dr. Vyacheslav Rykov. Your suggestions during my dissertation topic proposal have improved the quality of my research work and have been invaluable in completing this dissertation.

I would also like to thank all the friends and colleagues in the CMANTIC lab. You all have helped me in one way or the other in completing my research. It was you who made my workplace better and motivated me in achieving my goal. I would also like to thank Dr. Carl Nelson for all of his invaluable comments and suggestions which helped me improving my research.

I would like to thank my undergraduate advisor, Dr. Krishnendu Mukhopadhyaya. You have introduced me to research and the world of robotics. You are the one who has motivated me towards pursuing Ph.D. Thank you for all your help.

Finally, I would like to thank my parents, my wife, and all other family members for their patience, inspiration and support during my Ph.D.

Grant Acknowledgment

Most of my research works were supported by the NASA-EPSCoR Grant # NNX11AM14A and College of IS&T (UNO).

Contents

1	Introduction	1
1.1	Problems Studied	3
1.1.1	Partitioning of Modules	3
1.1.2	Configuration Formation	3
1.1.3	Adaptive Locomotion Learning	5
1.2	ModRED MSR Platform	6
1.3	Contributions	9
1.4	Document Outline	10
2	Decision Making: Partitioning of MSR Modules	12
2.1	Background	14
2.2	A Centralized Solution: Bottom-Up CSG Search	15
2.2.1	Dynamic Partitioning of Modules: Preliminaries	15
2.2.2	Search Algorithm: <i>bottomUpCSGSearch</i>	21
2.2.3	Experimental Evaluation	30
2.3	A Distributed Approach: Spanning Tree Partitioning	39
2.4	Discussions	40
3	Planning: Configuration Formation	42
3.1	Background	44
3.2	General Problem Formulation	47

3.3	Simultaneous Configuration Formation and Information Collection	48
3.3.1	Algorithm Description	50
3.3.2	Experimental Evaluation	55
3.4	Distributed Configuration Formation From Initial Smaller Configurations .	58
3.4.1	Notations	58
3.4.2	Algorithms for Configuration Formation	62
3.4.3	Experimental Evaluation	76
3.5	Discussions	89
4	Learning: Adaptive Locomotion Learning	91
4.1	Background	91
4.2	General Problem Formulation	94
4.3	Locomotion Learning via Joint Action Learning	95
4.3.1	Q-Learning Based Approach for Distributed Locomotion Learning .	96
4.3.2	Experimental Evaluation	98
4.4	Locomotion Learning via Independent Action Learning	106
4.4.1	Goal Directed Reward Formulation	107
4.4.2	Normal-form Games	108
4.4.3	Independent Action Learning vs. Joint Action Learning	109
4.4.4	Game Theoretic Solution Using Independent Q-Learner	110
4.4.5	Experimental Evaluation	113
4.5	Discussions	118
5	Conclusion	120
5.1	Summary	120
5.2	Future Directions	121
5.3	Remarks	123

List of Figures

1.1	Three different types of MSRs: (left) Chain, (middle) Lattice, and (right) Hybrid.	2
1.2	Configuration formation problem in MSRs. Left: 5 modules are located at arbitrary positions and have to achieve the target configuration (red dotted lines), Right: Modules after achieving target configuration.	4
1.3	ModRED Modules [7]	7
1.4	Different parts in a single module of ModRED [21]	7
1.5	Electronic Architecture of ModRED Modules	8
1.6	Simulated ModRED Modules within Webots Robot Simulator	8
2.1	An example illustrating the configuration generation process (left) shows the initial arrangements of the modules, (middle) shows the best partitions calculated, (right) shows the final configurations of the modules.	14
2.2	Simulated model of 6 ModRED modules reconfiguring from snake to ring shape.	15
2.3	Coalition structure graph (CSG) with 4 agents	16
2.4	left: An illustration of the size-constrained value function; right: An illustration of the effect of this value function on finding higher utility nodes in the CSG.	17

2.5	Probability of a pair of MSRs to dock successfully with each other for (left) distance between the MSRs, (middle) angular difference between the MSRs, and, (right) environment noise values [36].	19
2.6	An abstraction of the CSG for $ A = 7$ agents showing the partitions of A that are inspected at each level. Note that at level l , A has different partitions with exactly l parts or subsets. With $n_{max} = 3$, the maximum number of C_n -s (coalitions of size 3) are at level $l_{n_{max}} = 3$	24
2.7	Illustrative example of the working procedure of the algorithm	26
2.8	Configuration generation process using our proposed <i>bottomUpCSGSearch</i> algorithm.	31
2.9	Comparison of run time of the <i>bottomUpCSGSearch</i> algorithm and actual space complexity against the number of modules	33
2.10	Comparison of number of nodes explored by different existing coalition structure generation algorithm and the proposed <i>bottomUpCSGSearch</i> algorithm.	34
2.11	a) Comparison of number of nodes generated with existing algorithms – for 12 modules, with varying n_{max} ; b) Run time comparison with <i>searchUCSG</i> algorithm; c) Run time comparison with <i>graphPartitioning</i> algorithm; d) Run time comparison with <i>BP</i> algorithm.	35
2.12	(a) - (c) Exploring the anytimeness nature of the proposed <i>bottomUpCSGSearch</i> algorithm – for 8, 10 and 12 modules; d) Effect of changing n_{max} on number of explored nodes – for 10 modules. . .	36
2.13	Comparison of total number of nodes in $l_{n_{max}} = 2$ vs. no. of nodes generated.	38
2.14	Ratios between total number of nodes, number of nodes generated and time taken to find optimal for $l_{n_{max}} = 2$ and 3.	38

2.15	(a) MST generation among modules, (b) distribution of integer partitions among modules.	39
3.1	(a) A singleton and three initial configurations consisting of 2, 6 and 8 modules respectively, and desired target configuration (marked with yellow dotted lines) (b) target configuration involving all 17 modules connected in ladder configuration; module numbers marked in white, yellow and red are retained between initial and target configurations.	43
3.2	Complicated configurations formed by ModRED II modules [49].	44
3.3	Artistic patterns formed by robot swarms [4].	46
3.4	Configuration formation problem in MSRs. Left: 5 modules are located at arbitrary positions and have to achieve the target configuration (red dotted lines), Right: Modules after achieving target configuration.	47
3.5	Illustration of how the target configuration T is modeled as a graph.	48
3.6	(a) Run times of EPS algorithm for different budgets; (b), (c) Nodes explored (shown in white color) by EPS algorithm, with $B = 45$ and 55 respectively.	55
3.7	(a) Comparison of estimated information collection between SA and auction algorithm.; (b) Comparison of no. of messages sent in planning phase with auction algorithm; (c) Comparison of planning times between SA and auction algorithm; (d) Effect of changing values of \mathcal{O} ; (e) Change in collected information over time; (f) Configuration formation by 10 modules: boxed $+$ and circled $*$ indicate the start and final locations respectively.	57

3.8	(a) A singleton and three initial configurations consisting of 2, 6 and 8 modules respectively, and desired target configuration (marked with yellow dotted lines) (b) target configuration involving all 17 modules connected in ladder configuration; module numbers marked in white, yellow and red are retained between between initial and target configurations.	58
3.9	(a) Graph abstraction of T ; (b) Graph abstraction of A_i	59
3.10	Illustration of eviction algorithm for 3 modules with $\mathcal{D}_{max} = 3$	65
3.11	(a) A scenario where the colored subgraphs of T are isomorphic to A_i , (b) A scenario where a subgraph of t is isomorphic to a subgraph of A_i . The red dotted box shows the maximal common subgraph between T and A_i ; the unmatched module a_3 is detached from A_i and allocated to spot s_1 by our block selection algorithm.	69
3.12	Illustration of acting phase: Dotted boxes represent the spots in T . First, the spot with the maximum betweenness centrality gets allocated (black spot). Next its neighbors get allocated (red spots) and finally neighbors of red spots get allocated (yellow boxes). (top) all modules are singletons; (bottom) red modules on the left side were part of an initial configuration. Therefore they occupied the spots at the same time even though two of the extreme red modules were not immediate neighbors of the central black module.	71
3.13	(a) Time to calculate MCS or IS vs. different initial configuration sizes, (b) Total planning time for different number of modules in environment.	77
3.14	(a) Distance traveled by modules to reach target configuration for different number of modules in the environment, (b) Number of messages exchanged between modules to select positions in the target configuration for different numbers of modules in the environment.	78

3.15	(a) Change in % of planning completion with % of time completion, for different no. of modules; (b) Change in no. of messages at different time steps, for 100 modules.	79
3.16	(a) Change in % of planning completion with % of time completion, for different no. of modules and $ S = 50$; (b) Change in no. of messages for different no. of modules and different no. of spots.	80
3.17	(a) Log scale comparison of planning phase execution time with auction algorithm; (b) Comparison of total traveled distances with auction algorithm.	81
3.18	(a) Log scale comparison of no. of messages with auction algorithm; (b) Change in % of planning completion with % of time completion and comparison with auction algorithm. 50 lines indicate 50 runs.	83
3.19	Cases showing configuration formation procedure along with corresponding planning times and number of disconnections required. Leftmost figure in each case shows the initial configurations and singletons, middle figure shows the MCS (or, IS) found (marked by dotted boxes) by executing our algorithms, rightmost figure shows the final formed target configuration with modules selecting spots (shown in a color-coded fashion).	85
3.20	A single ModRED module used for the configuration formation algorithm (a) CAD drawing, (b) hardware. Each module has 4 connectors which enables it to form branched configurations. (c) A 17-module branched, ladder configuration similar to Figure 3.8(b) that is capable of complex maneuvers and forming truss-like structures [7].	86
3.21	(a) Comparison of run times to elect a leader and map the topology of the ModRED configuration against the configuration size; (b) Change in run time to find MCS with different number of modules in the initial configuration.	87

3.22	(a) Change in run time of the spotAllocation() algorithm with different number of spots; (b) Change in run time of the auction algorithm with different number of spots.	88
4.1	a) Simulated ModRED Modules within Webots Robot Simulator and (b) Hardware of ModRED.	99
4.2	Performance comparison of our proposed approach when applied on ModRED configurations: (a) 2-module chain and against hand-coded gaits (b) 5-module chain and against the random approach.	101
4.3	Performance comparison of our proposed approach when applied on ModRED configurations: (a) 4-module and (b) 5-module chains against the random approach.	102
4.4	Performance of our proposed approach when applied on Yamor configurations: (a) 10-module, (b) 12-module and (c) 14-module chains. . .	103
4.5	(a) Maximum distance traveled in any direction from the start location by different configurations, (b) Maximum speed achieved by different configurations and (c) Average number of messages sent by each module in different configurations.	103
4.6	Snapshot of inchworm locomotion performed by (top) ModRED and (bottom) Yamor configurations using our proposed approach.	104
4.7	Performance of our proposed algorithm after (a) the end and (b) the middle module becomes non-operational.	105
4.8	(a) Performance of our proposed algorithm after two end modules become non-operational. (b) Comparison of performance of our algorithms for rolling locomotion against the same by using hand-coded gaits.	105
4.9	Snapshot of rolling locomotion by 2-module ModRED chain using our approach.	106

4.10	Different configurations used for our experiments. Snapshots are captured within Webots simulator.	114
4.11	Webots snapshot of inchworm locomotion performed by a 3-module ModRED chain.	114
4.12	Change in distance from goal over time for configurations a) M3I and b) Y12. The faint lines denote multiple runs and the bold blue line indicates the average line.	115
4.13	Average speeds achieved by different configurations along with standard deviations. Comparison against the single-agent Q-learning based (SAQL) adaptive locomotion work is shown [19].	117
4.14	Comparison of configuration's performances before and after the failure of the end module.	117
4.15	Comparison of configuration's performances before and after the failure of the middle module.	118

List of Tables

2.1	Complexity Comparison	29
2.2	Ratio of values of best coalition structure found using our algorithm to the optimal value and corresponding running times.	32
3.1	Planning times and the numbers of disconnected modules (average and standard deviation) in the configuration formation process, where all initial configurations have same sizes ($ S = \mathbb{A} = 100$).	81
4.1	Actions for inchworm locomotion	100
4.2	Actions for rolling locomotion	100
4.3	A 2-player normal-form game's payoff matrix	109

Chapter 1

Introduction

Modular self-reconfigurable robots (MSRs) are composed of multiple homogeneous or heterogeneous modules which can change their connections with each other to form different configurations or shapes [90]. The main advantage of using MSRs is that the modules can change the connections among themselves to form different shapes and transition from one shape to another depending on the current environment and the current task. This configuration adaptability affords a high degree of dexterity and maneuverability to MSRs and makes them suitable for robotic applications such as inspection of engineering structures like pipelines [40], extra-terrestrial surface exploration [17], information collection [32], forming truss-like structures for support [100] etc.

An excellent overview of the state of the art MSRs and related techniques is given in [100]. Based on architectural properties, modular robots can be divided into three main categories [100]:

1. **Chain:** In this type of MSR architecture, modules are connected together in a two-dimensional graph topology. This type of configuration can fold-up to become space filling, but the underlying architecture is planar [100].
2. **Lattice:** This type of architectures have modules that are arranged and connected in some regular, three-dimensional pattern, such as a simple cubic or hexagonal grid.

Modules are controlled in a parallel fashion. This type of configurations have a liquid-flow like locomotion pattern where each module behaves as a molecule of the liquid [41, 1].

3. **Hybrid:** This type of architecture is a combination of both chain and lattice type. Modules tend to form large connected network in hybrid configurations [100].



Figure 1.1: Three different types of MSRs: (left) Chain, (middle) Lattice, and (right) Hybrid.

In the MSR literature, it has been seen that most of the work is done to solve the *self-reconfiguration problem* in MSRs – Given a set of modules connected in a certain configuration, search for a set of actions for each of the modules such that following the actions they will transform into a new configuration (or, shape) while maximizing some given objective function, as well as reducing the time and cost expended to identify and achieve the new configuration [52]. When an MSR encounters an obstacle while moving, or when its assigned task requires it to take a certain shape, it requires to dynamically change or reconfigure from an existing configuration to a new configuration, so that it can continue to perform its operations autonomously. As the space of possible action set for modules is exponential in the number of modules involved, conventional search algorithms are unsuitable to solve the reconfiguration planning problem within a reasonable amount of computation time and space. The self-reconfiguration is known to be a NP-complete problem [51]. We look at three other fundamental problems in MSRs which are discussed as follows.

1.1 Problems Studied

1.1.1 Partitioning of Modules

First, we address the partitioning problem in MSRs for configuration generation which can be described as follows: how to identify the *best* partitioning of modules for forming any configuration, while maximizing some pre-defined objective function. Finding the best partitioning of modules for forming shapes or configurations is a non-trivial problem, as the number of possible partitions is exponential with the number of modules involved. Moreover, the computation becomes further complicated if we include uncertainty in the mobility and connections of modules, which are practical considerations for any physical modular robot operating in an unknown environment.

To address these challenges, we have proposed a search algorithm while using concepts from cooperative game theory [69], called coalitions and coalition structures. A *coalition* is a group of autonomous agents (modules in our scenario) which work together towards achieving a common goal. From the MSR perspective, a coalition is a configuration formed by a set of modules which are connected and can maneuver as a single entity. A *coalition structure* corresponds to disjoint and exhaustive sets of coalitions (connected modules) which represent all shapes or configurations formed by a set of MSRs. In this work, we have modeled the best partition search problem as the best coalition structure generation problem, as finding the best coalition structure will give us the best partition of modules, i.e., the best set of possible teams or coalitions of modules.

1.1.2 Configuration Formation

Once the partitioning of modules is complete, i.e., each module has made a decision of with which other modules it should form the configuration, it moves on to do that task – configuration formation. This is a fundamental problem for modular robots and also a pre-requisite of MSR self-reconfiguration and it can be defined as follows: given a set

of modules as singletons and/or in connected configurations, how to form a user-specified target configuration using all or some of the initial modules while optimizing some criterion such as reducing the cost of movement and the number of connections and disconnections among modules [32]. In a recent survey on MSRs [1], authors have reported that although configuration formation is a very fundamental and difficult problem to solve in MSRs, there has not been extensive work done on this particular topic. Most of the studies on this topic are limited to the MSR platform on which the proposed approach has been deployed - therefore in most of the cases, they cannot be generalized to all types of MSRs. In our research, we try to overcome this limitation by proposing algorithms which can be generalized to configuration formation using any type of MSR. Not only in modular robotic systems, our work can also be easily extended for automated formation of pre-defined shapes in industries such as furniture [72].

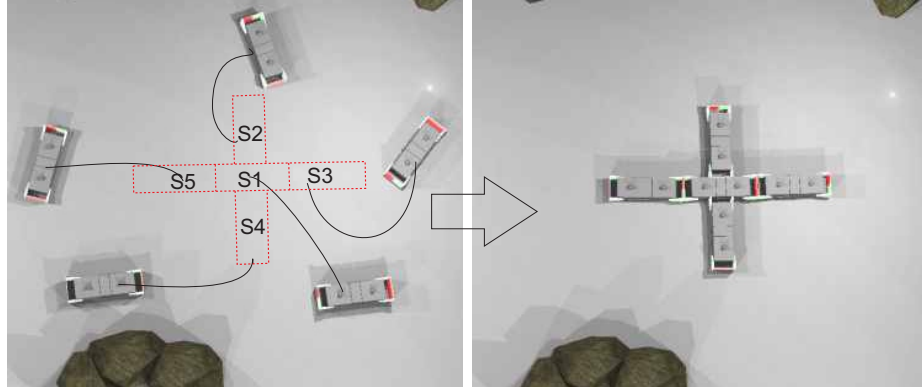


Figure 1.2: Configuration formation problem in MSRs. Left: 5 modules are located at arbitrary positions and have to achieve the target configuration (red dotted lines), Right: Modules after achieving target configuration.

Configuration formation is a way to fulfill shape-formation function, in which modules aggregate autonomously to a final shape or configuration. In the context of MSRs, configuration formation enables modular robots to transform into any desired configuration. For task completion using MSRs, configuration formation is a very important operation. As a motivating example, we consider a scenario where a set of singleton modules are

collecting information, e.g., temperature from an environment. To access a specific region of the environment, e.g., an elevated region, they need to form a certain shape (configuration) such as a legged configuration, which allows them to navigate the elevation. To get into this new shape, all the singleton modules will plan their paths from their current locations to appropriate positions in the target configuration to form the target legged configuration.

1.1.3 Adaptive Locomotion Learning

After the target configuration is formed, the built configuration needs to move to different locations to complete the tasks at hand, such as exploration or information collection. If the size (number of modules) and the topology of the configuration are known, then coordinated locomotion for all the member modules can be planned *a priori* in a deterministic manner [85]. But the difficulty arises when either the size or the shape, or both, is unknown. In that case, determining the control sequences for locomotion is impossible.

In [100], the authors have mentioned that developing algorithms for large-scale manipulation and adaptive locomotion is one of the most difficult future challenges for MSR researchers. To solve this problem, we propose two adaptive locomotion algorithms which learn the best control sequence for all the modules in the configuration. Our approaches look into this particular problem from a machine learning perspective [42]. Machine learning enables the modules in the configuration to learn the control sequences which suits their task (locomotion in this case) the best. Particularly, in our research, we investigate reinforcement learning algorithms for locomotion learning purposes [91]. In layman terms, reinforcement learning can be best described as the following:

”Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most

reward by trying them.” [91]

In our particular locomotion learning scenario, a ‘situation’ is modeled as the size and the shape of the configuration and an ‘action’ is modeled as an available locomotion action to each module. ‘Reward’ is modeled as the distance traveled by the configuration, i.e., higher distance covered by the configuration earns higher reward and vice-versa [19]. Modules discover (by learning) which actions (control sequences) earn them higher rewards and they perform those actions repeatedly to earn more reward and consequently accomplish locomotion (towards the goal).

1.2 ModRED MSR Platform

The ModRED (*Modular Robot for Exploration and Discovery*) is a homogeneous modular robot system and has been developed as part of the NASA sponsored ModRED project for efficient maneuvering over unstructured surfaces such as can be experienced during planetary surface exploration [21]. In this section, we briefly discuss the characteristics of the ModRED MSR. The robot system is characterized by dexterity of modules and a distributed control architecture. The novel kinematic arrangement of the robot modules is supported by a rotary plate genderless single-sided docking mechanism (RoGenSiD) [50]. It enables a module to detach itself from a faulty module which is essential for sustaining the robot system’s functionality by means of self-healing.

Each of the ModRED modules has 4 DOF - 3 rotational and 1 prismatic. The module has five distinct segments - two end brackets containing the docking interfaces and three box-shaped segments housing the actuators, transmission, circuit components and power source, as shown in Fig. 1.4. The module is capable of producing pitch, yaw, and roll and one extension DOF. The four independent DOF are characterized by specific ranges to meet the requirements for generating gaits [21] and reconfiguration. The docking brackets have a rotation range of $\pm 90^\circ$. Relative to the central box segment, one end segment has an

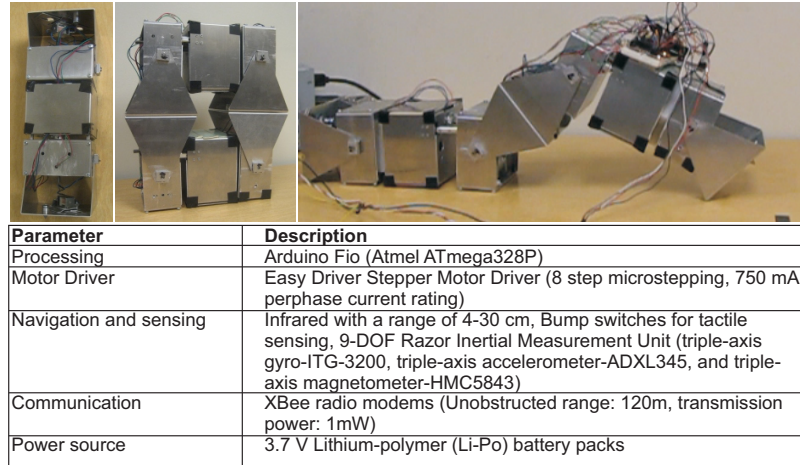


Figure 1.3: ModRED Modules [7]

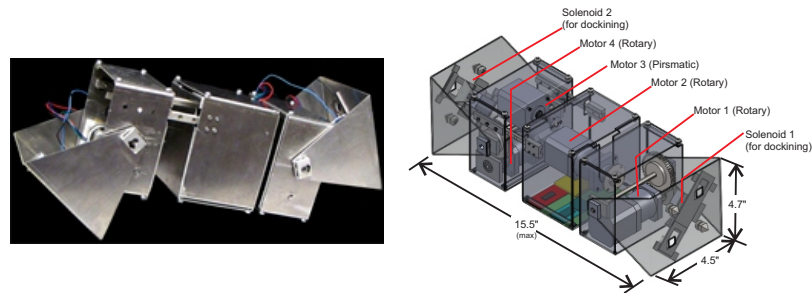


Figure 1.4: Different parts in a single module of ModRED [21]

infinite angle of twist whereas the other end box segment has a 0-1 inch (0-25mm) linear range of displacement. To allow maximum dexterity, all 4 DOF are made independent, *i.e.*, they have individual actuators. The motivation behind such a design was to keep the design simple with minimal transmission mechanisms. This would result in enhanced robustness by minimizing the parts count and thus minimizing the probability of failure for an overall module. Stepper gear motors are used for the 3 rotational DOF whereas a stepper linear actuator (lead-screw mechanism) is used for the prismatic DOF.

As an autonomous system, each of the ModRED modules is equipped with necessary electronics to give them such autonomy, *i.e.*, two ATmega328P microcontrollers, rechargeable lithium-polymer battery packs, XBee modules to enable wireless communication among modules, one inertial measurement unit (9-DOF Razor), an array

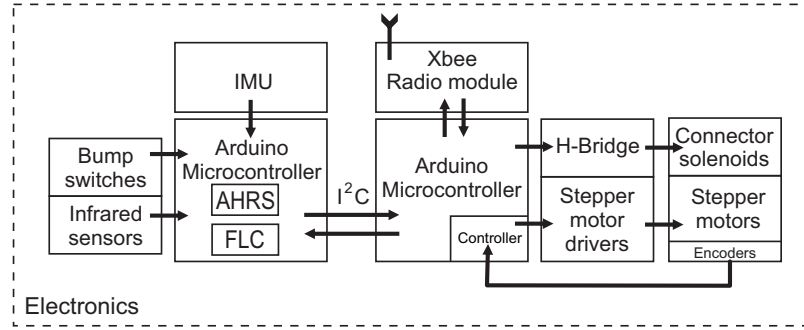


Figure 1.5: Electronic Architecture of ModRED Modules

of infrared sensors for proximity sensing and bump sensors for tactile sensing, as shown in Fig. 1.5. An accurate 3D model of ModRED has been created using the Webots platform to simulate the execution of different gaits before implementation in the robot, as shown in Fig. 1.6. Some of the algorithms described in this dissertation have been implemented on different simulated ModRED configurations and some of the algorithms have been implemented directly on the ModRED hardware. Although, our algorithms presented here are not only restricted to ModRED modules, they can be used for any other chain-type modular robotics platform.

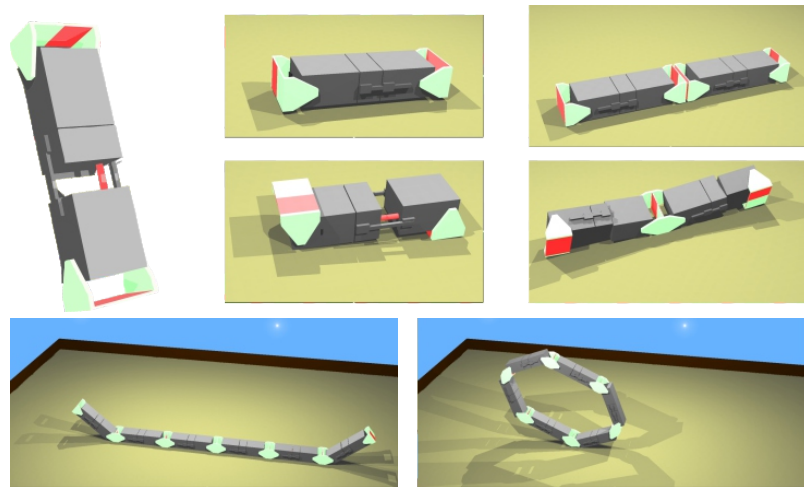


Figure 1.6: Simulated ModRED Modules within Webots Robot Simulator

1.3 Contributions

Our contributions can be summarized as follows:

- The partitioning problem has been formalized and modeled as a coalition structure search problem. Both centralized and distributed solutions have been proposed to solve this problem. To the best of our knowledge, we are the first one to solve this problem as a coalition structure search problem.
- There has been a very little work done for solving the configuration formation problem in MSRs [1] and most of the proposed approaches are difficult to generalize to all types of MSR platforms. In our research, we generalize this by solving the problem using both centralized and distributed planning approaches.
- A novel problem of simultaneous configuration formation and information collection has been addressed and solved.
- Solutions have been proposed for the configuration formation problem when modules are initially part of any arbitrary-shaped configurations. To the best of our knowledge, the configuration formation problem has not been solved for the case when initially modules can be connected in some arbitrary configurations and they need to be placed in the target configuration with as low number of disconnections in their initial configurations as possible. We have solved this problem in a distributed manner using concepts like (sub)graph isomorphism and maximum common subgraph detection. Although, (sub)graph isomorphism and maximum common subgraph detection have been used in the literature [51, 70] for MSR self-reconfiguration to detect the maximum portion in the initial configuration which does not need to be reconfigured to transform into the goal configuration, but they have not been used for MSR configuration formation.
- A reinforcement learning based solution has been proposed for an MSR's locomotion

leaning where the correlation among the neighboring modules' actions have been taken into account to learn from for better coordination. To the best of our knowledge, this work is the first to take the inter-module action-relation into account for locomotion learning which is unquestionably a major performance-affecting factor in MSR locomotion.

- We have also proposed a multi-agent learning based solution for MSRs' adaptive locomotion learning. Our work is the first to model the MSR locomotion problem as a multi-agent learning problem and solve it.

1.4 Document Outline

This dissertation is structured as follows:

- **Chapter 2**

First, we discuss the partitioning problem in MSRs. We mainly discuss a centralized approach which we have proposed for size-constrained partitioning. Then, we briefly mention a novel distributed solution to solve the same problem.

- **Chapter 3**

In this chapter, we discuss the configuration formation problem in MSRs. We describe our proposed semi-decentralized and distributed solutions which solve the configuration formation problem. We also evaluate our proposed approaches in simulation as well as by implementing them on ModRED hardware.

- **Chapter 4**

Here, we discuss the adaptive locomotion learning problem in MSRs. We describe our proposed reinforcement learning based solutions which solve the problem. We show the effectiveness of our proposed approaches by doing simulated experiments on ModRED and Yamor MSR platforms within the Webots simulator.

- **Chapter 5**

In this chapter, we summarize the main contributions and findings in this dissertation and discuss a few research directions that we plan to pursue in the future.

Chapter 2

Decision Making: Partitioning of MSR Modules

Our approach of solving the partitioning problem in MSRs is based on a game-theoretic formulation, called the coalition structure search [73]. Coalition structure generation and searching for the best coalition structure are well-known NP-complete problems [81] and exhaustively searching for the set of all possible coalition structures becomes computationally prohibitive even for a relatively small number of coalitions. Coalition structure generation problems have been proposed for real-world scenarios like voting, or for task completion while maximizing resource allocation criteria [87]. However the direct implementation of the previously developed algorithms for the best coalition structure generation for virtual agents, such as in [74, 73], is infeasible, due to mechanical constraints, communication constraints, and uncertainty in robots' movements as well as in the environment.

Each configuration or shape needs a specific number of modules in it, denoted by n_{max} , to form the configuration [51]. In [51], the authors have mentioned that for reconfiguration from one shape to another, both shapes require exactly the same (n_{max}) number of modules present, connected together. If n_{max} modules are together forming a shape, then the

probability of forming the desired shape is higher. On the other hand, if there are more or fewer modules than n_{max} present to form the desired shape, then the formation is not completely successful - either the shape will not be complete (if there are $< n_{max}$ modules) or it will be bigger in size and also different in shape from the desired configuration (if there are $> n_{max}$ number of modules present). Therefore having more or fewer modules will lead to undesired configuration formation. To incorporate this criterion into our approach, we have proposed a variant of the classical coalition structure generation problem called *size-constrained* coalition structure generation. In this approach, each coalition's worth or value is determined by the number of modules it has - if there are exactly n_{max} modules then the coalition will receive the highest value, else its value will be diminished. Many real-world domains, such as sports teams or judging committees, are constrained by a similar maximum size determined by the game or the competition rules. As a motivating example, we consider a scenario where a set of singleton ModRED modules [7] are dropped from an aircraft in an extra-terrestrial environment. The task for them is to form shapes (or configurations) to inspect parts of the environment, such as volcanic craters. To access a specific region of the environment, not any configuration of any size can be formed due to size and shape restrictions. Let us assume that the maximum size any configuration can have is n_{max} . Now the modules need to find the best partition among themselves which also restricts the maximum size of any configuration to n_{max} .

An example of the working procedure of our algorithm is shown in Figure 2.1, where the number of modules $|A| = 4$ and n_{max} is varied between 2 and 3. Initially modules are randomly distributed as singletons (Figure 2.1 (left)). By using our proposed algorithm, modules decide what coalitions they should form among themselves, as shown in Figure 2.1 (left), with red and yellow lines. Figure 2.1 (middle) and (right) show these coalitions being realized as MSR configurations, while using any of the available MSR reconfiguration techniques [51]. Thus by appropriate partitioning of the modules, we can find the best configurations.

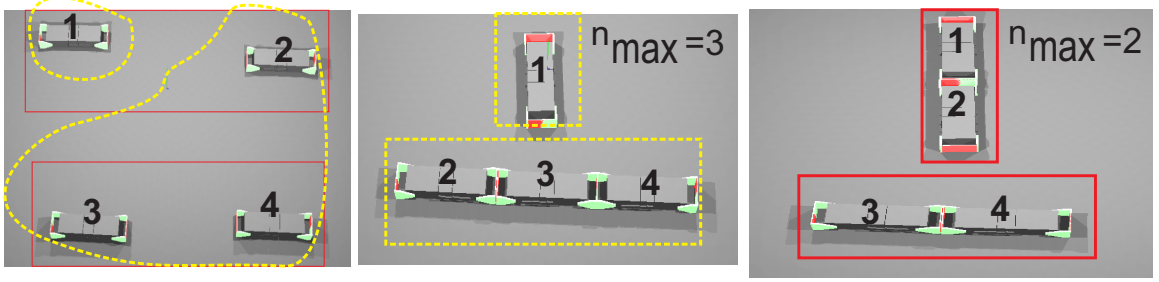


Figure 2.1: An example illustrating the configuration generation process (left) shows the initial arrangements of the modules, (middle) shows the best partitions calculated, (right) shows the final configurations of the modules.

2.1 Background

We discuss previous similar studies in this section. The literature is divided into the following subsections.

Coalition Structure Search: Cooperative or coalition game theory gives a set of techniques that can be used by a group of agents to form teams or coalitions with each other [69, 81]. A coalition structure graph (CSG) is usually used to represent the set of all possible coalition structures among a set of agents in a systematic and hierarchical manner. Searching for the best coalition structure within a CSG is a NP-complete problem [81].

Heuristic [80] and anytime [101] algorithms were proposed to solve the problem of finding the best coalition structure. Though heuristic algorithms scale up well with higher numbers of agents, the algorithms take a significant amount of time to find a good *solution*. Anytime algorithms alleviate this problem, but they can end up searching the whole search space, which is infeasible, due to large time complexity ($O(n^n)$). Sandholm [81] has proved that all possible coalitions can be found in the first two levels of the CSG and proposed an anytime algorithm exploiting this property of the CSG. Rahwan [73] proposed an anytime, dynamic programming based approach called IDP which reduced the space complexity from a previous dynamic programming based approach [78]. Genetic algorithms have been used to implement these types of heuristic solutions [82]. Shehory and Kraus [84] have proposed a decentralized greedy algorithm which takes into account only those coalitions

which have size less than a permitted value. In addition to the techniques in these solutions, our approach considers uncertainty while forming coalition structures, as it is an essential aspect of modular robot configuration generation.

Configuration Generation: In one of our earlier works [34], we proposed an anytime algorithm for configuration generation which takes the maximum value of permitted coalition size (n_{max}) as input and significantly reduces the time and space complexity compared to previously proposed configuration generation techniques, to find the optimal coalition structure, similar to [84]. In this work only those coalitions are searched which have sizes less than or equal to n_{max} . In another earlier study, we used a graph partitioning approach for configuration generation [27] without taking uncertainty into account. In our more recent work, we have used CSG to find the best coalition structure [36]. The work described here is an extension of our earlier work described in [35].

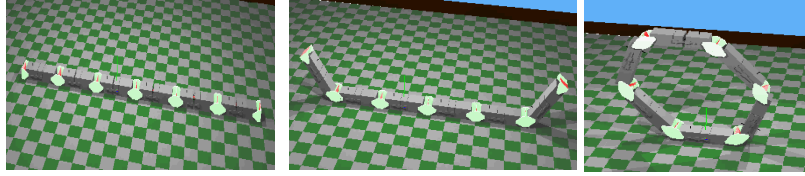


Figure 2.2: Simulated model of 6 ModRED modules reconfiguring from snake to ring shape.

2.2 A Centralized Solution: Bottom-Up CSG Search

2.2.1 Dynamic Partitioning of Modules: Preliminaries

Let A be a set of modules. $A_i = \{a_{i_1}, a_{i_2}, a_{i_3}, \dots, a_{i_{|A_i|}}\}$ where $\{a_{i_j}, a_{i_{j+1}}\}, j = \{1, \dots, |A_i| - 1\}$ is the set of physically coupled modules in A_i . When $|A_i| = 1$, the MSR is a single module - we call it a singleton. Let $\Pi(A)$ be the set of all partitions of A . By a partition of a set A we will mean a collection of nonempty, pairwise disjoint sets whose union is A . The sets into which A is partitioned are called the classes of the partition [97].

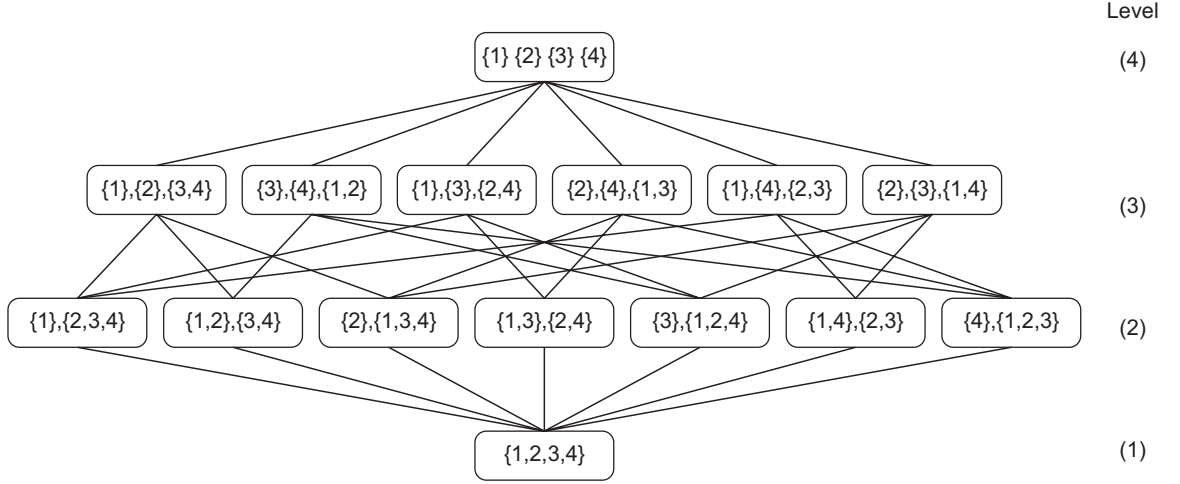


Figure 2.3: Coalition structure graph (CSG) with 4 agents

The number of partitions of set A is exponential in $|A|$, and this number is denoted by *bell numbers* $B_{|A|}$ [97], where $B_{|A|} = |\Pi(A)|$. Also, let $CS(A) = \{A_1, A_2, \dots, A_k\} \in \Pi(A)$ denote a specific partition of A , which is called a coalition structure; note that $A_i \subseteq A$ is the i -th MSR in that coalition structure. A systematic way to go about analyzing the partitions in $\Pi(A)$ is provided by a hierarchical graph structure called a coalition structure graph (CSG) [69]. A CSG with 4 agents is shown in Figure 2.3. CSG nodes are organized into levels. *Level l* indicates that every node in level l in the CSG has exactly l subsets or coalitions as its members. CSGs offer a structured way of exploring coalition structures because a node at level $l + 1$ can be generated by breaking up a coalition from a node at level l . We assume that initially all the modules are singletons ¹.

In a CSG, each partition $CS(A) \in \Pi(A)$ is called a coalition structure and appears as a node in the CSG. The parts or subsets of a partition are called coalitions, denoted by A_i . Each coalition A_i has a value associated with it that can be referred to as a virtual reward received by the agents in that coalition for coming together to perform the task at hand. The value function is denoted by $Val : A_i \rightarrow \mathbb{R}^+$. The value function assigns to each coalition A_i a real positive number corresponding to a virtual reward that the coalition can

¹This work is published in [37]

obtain for performing its assigned task. Our value function is modeled in a way such that it is beneficial for agents to form coalitions up to a certain coalition size n_{max} but this benefit starts diminishing for coalition sizes that are larger than n_{max} .

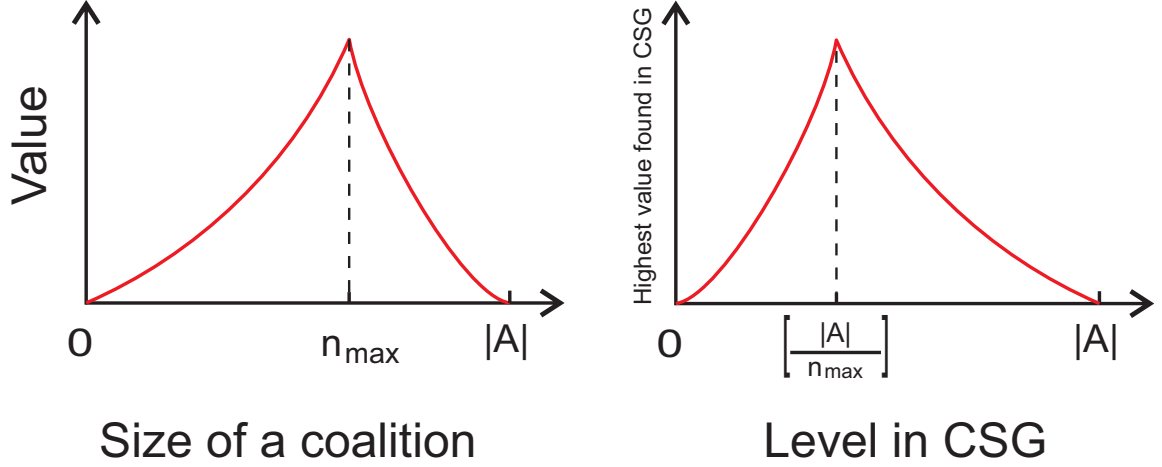


Figure 2.4: left: An illustration of the size-constrained value function; right: An illustration of the effect of this value function on finding higher utility nodes in the CSG.

Size-Constrained Value Function: Each configuration needs exactly n_{max} number of modules to form that particular configuration. But if n_{max} number of modules are not present there, then forming that particular shape is impossible. To incorporate this, we have developed a size-constrained value function, which is illustrated in Figure 2.4 (left) and represented by:

$$Val(A_i) = \begin{cases} |A_i|^k, & \text{if } |A_i| \leq n_{max} \\ e^{-(|A_i| - n_{max})} \times n_{max}, & \text{otherwise} \end{cases} \quad (2.1)$$

where k is any integer > 1 . The above value function is super-linear, i.e., $Val(|A_i| + 1) - Val(|A_i|) > Val(|A_i|) - Val(|A_i| - 1)$; which ensures that larger coalitions are better and obtain higher rewards, up to a size of n_{max} , as it is taking the tally towards the number of modules required to form that configuration. n_{max} denotes the maximum allowable size of a coalition; it is given as input to our algorithm and it does not change the

operation of the algorithm. Our value function gives preference to larger coalitions only up to a certain size n_{max} , beyond which larger coalitions are penalized by yielding lower values. The preferred size of a coalition is relevant to the task assigned to the MSR. The value of a coalition structure, $CS(A)$, is given by the summation of the values of coalitions comprising it, i.e., $Val(CS(A)) = \sum_{A_i \in CS(A)} Val(A_i)$. Evidently, if forming coalitions incurred no cost, the most suitable partition for a set of agents is the one that maximizes $Val(CS(A))$. For example, $Val(\{1, 2\}, \{3, 4\}) = Val(\{1, 2\}) + Val(\{3, 4\})$. The size of coalitions $\{1, 2\}$ and $\{3, 4\}$ is 2. In equation 2.1, if we assume $k = 2$ and $n_{max} = 2$, then $Val(\{1, 2\}, \{3, 4\}) = 2^2 + 2^2 = 8$.

An illustrative example scenario, where the size-constrained value function can be used, is shown in a snapshot of ModRED from the Webots simulator in Figure 2.2 (right). If the desired motion of the MSR is to cross an obstacle while in a ring shape, which needs 6 modules to give the MSR sufficient traction for its rolling motion, then the value of n_{max} is set to 6. As the focus of this work is on determining the best coalition structure, the problem of determining the optimal n_{max} is not considered further in this work.

Uncertainty in Configuration Formation. Unexpected motion and alignment of the robot modules can cause ModRED's behavior to deviate from ideal operation. Following [36], we have considered three major sources of uncertainty under this category that could affect the mobility of the modules and consequently the configuration generation process. The uncertainty model is summarized below:

(i) *Distance uncertainty* is the uncertainty arising out of the distance required to be traversed by a pair of MSRs before docking with each other. As the modules do not know the features of the terrain such as obstacles between them beforehand, successful alignment and docking of the modules' end connectors becomes more uncertain with higher distance between them. The distance uncertainty is modeled as a half-Gaussian distribution $\mathcal{N}(\mu_{du}, \sigma_{du})$ as shown in Figure 2.5 (left).

(ii) *Alignment uncertainty* is the uncertainty arising out of the angle each MSR in a

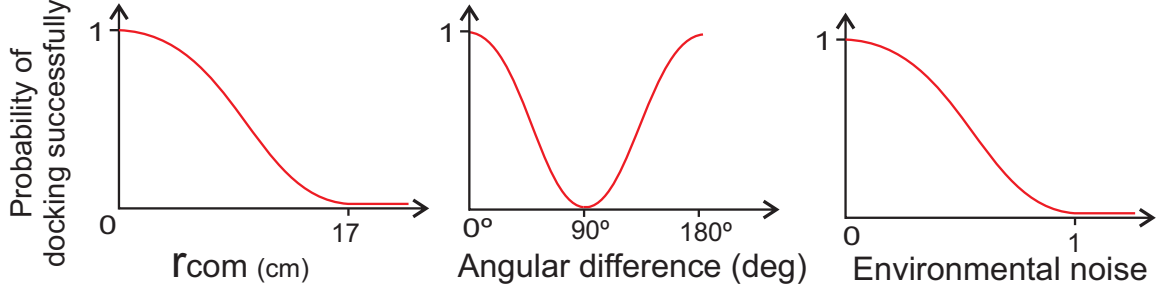


Figure 2.5: Probability of a pair of MSRs to dock successfully with each other for (left) distance between the MSRs, (middle) angular difference between the MSRs, and, (right) environment noise values [36].

pair of docking MSRs needs to rotate before they can align with each other. As ModRED modules have docking faces at two ends [50], if the rotational difference between them is close to 0° or 180° , then it is easier for them to align and dock; they are most unaligned when the rotational difference between them is close to 90° . Alignment uncertainty between two modules is modeled as a Gaussian distribution, $\mathcal{N}(\mu_{au}, \sigma_{au})$, as shown in Figure 2.5 (middle).

(iii) *Environment uncertainty* is the uncertainty arising from the operational conditions in the environment due to factors such as terrain conditions, surface friction, etc. that affect the movement of a pair of MSRs while moving towards and docking with each other. The uncertainty is modeled as a multi-variate half-Gaussian distribution $\mathcal{N}(\mu_{eu}, \sigma_{eu})$, as shown in Figure 2.5 (right).

To combine the Gaussians representing the motion uncertainties, a weighted mean with variance is considered [67], where weights are inverse of the variance estimates.² These weights are denoted by w_{du} , w_{au} , and w_{eu} respectively. The weighted mean of the three Gaussians then gives the total motion uncertainty, expressed as a probability, for forming any coalition A_i by connecting its member modules, as given below:

²According to the central limit theorem, any sum and/or average of samples from any random distribution with finite mean and standard deviation will always be approximately Gaussian.

$$prob(A_i) = \frac{(w_{du} \cdot p_{du} + w_{au} \cdot p_{au} + w_{eu} \cdot p_{eu})}{w_{du} + w_{au} + w_{eu}}, \quad (2.2)$$

where $p_{du} \in \mathcal{N}(\mu_{du}, \sigma_{du})$, $p_{au} \in \mathcal{N}(\mu_{au}, \sigma_{au})$ and $p_{eu} \in \mathcal{N}(\mu_{eu}, \sigma_{eu})$, and, $w_{du} = \frac{1}{\sigma_{du}^2}$, $w_{au} = \frac{1}{\sigma_{au}^2}$, $w_{eu} = \frac{1}{\sigma_{eu}^2}$.

Expected Cost functions. When a set of modules need to form a configuration, they move towards each other, the end modules align with the other's docking faces and perform the connection operation [50] to finally form the configuration. To perform these operations, modules have to spend a considerable amount of battery power. Therefore the modules which are going to form a new configuration should be chosen in such a way that they expend less energy in moving and aligning. We have represented the energy expended by modules as the cost to generate configurations. While searching for the best coalition structure, modules not only need to find the coalition structure which has highest value, but also which incurs minimum cost.

Let $cost(A_i)$ denote the cost of forming a coalition A_i . $cost(A_i)$ is defined as the cost of connecting singleton modules in a chain configuration. We denote the expected cost of forming coalition A_i as: $\overline{cost}(A_i) = cost(A_i) \times (1 - prob(A_i))$. This indicates that if the probability of modules connecting together in a coalition is higher, then the corresponding cost will be lower and vice-versa. Going further, we denote the expected cost of coalition structure $CS(A)$ as: $\overline{cost}(CS(A)) = \sum_{A_i \in CS(A)} \overline{cost}(A_i)$.

For notational convenience, we denote expected utility of a coalition structure $CS(A)$ as $\overline{U}(CS)$ where $\overline{U}(CS) = Val(CS(A)) - \overline{cost}(CS(A))$. Then, the optimal coalition structure is given by $CS^* = arg \max_{CS \in \Pi(a)} \overline{U}(CS)$. Based on the above formulation, we can now formally define the partitioning problem for modular robots as follows:

Definition 1 Coalition Size-Constrained Partitioning Problem: *Given a set of modules A and an initial coalition structure $CS_{old}(A) = \{A_1^{old}, A_2^{old}, \dots, A_k^{old}\}$ in which they are deployed (e.g., all singletons), find a new partition (or coalition structure)*

$CS(A) = \{A_1^{new}, A_2^{new}, \dots, A_{k'}^{new}\}$ such that the following objective is maximized:

$$\max_{CS(A) \in \Pi(A)} \overline{U}(CS)$$

An example of the configuration generation using the objective function is shown in Figure 2.1. Figure 2.1 (left) shows an initial configuration of ModRED, where the modules are randomly distributed as singletons. n_{max} is given as input, varied between 2 and 3. Following the proposed *bottomUpCSGSearch* algorithm, modules determine the best configuration, which happens to be $\{\{1\}, \{2, 3, 4\}\}$ when $n_{max} = 3$ and $\{\{1, 2\}, \{3, 4\}\}$ when $n_{max} = 2$. Modules then move and align themselves to form the new coalitions and finally the planned configurations are formed (Figure 2.1.(middle), (right)).

In the rest of the chapter, for the sake of legibility, we slightly abuse notations by referring to expected utility and expected cost as utility and cost respectively. Finally, we use two notations for convenience in our CSG search algorithm - a coalition of size n_{max} is denoted by C_n and the level in the CSG that contains the maximum number of coalitions of size n_{max} is denoted by $l_{n_{max}}$, where $l_{n_{max}} = \lfloor \frac{|A|}{n_{max}} \rfloor$.

2.2.2 Search Algorithm: *bottomUpCSGSearch*

Size-based Partitioning of CSG. Our value function assigns the highest value to the coalitions which have size n_{max} . From coalition size 0 to n_{max} , the value of a coalition increases in a super-linear fashion, as discussed earlier, whereas beyond size n_{max} , the value of a coalition decreases exponentially (Figure 2.4 (left)). As the utility of a coalition structure depends on the values of its member coalitions, the coalition structure with member coalitions having sizes n_{max} will most likely be part of the best coalition structure. Our algorithm is designed towards exploiting this property of the value function. The objective of our proposed CSG search algorithm is to target the search towards nodes (coalition structures) in the CSG that include coalitions of size n_{max} (C_n).

Sandholm [81] has proved that in level 2 of CSG, we can get all the possible coalitions. This means after expanding the bottom most node of the CSG, we can encounter all the

Algorithm 1: Searching for the best coalition structure

```

1 bottomUpCSGSearch( $v_{l_1}, n_{max}$ )
   Input:  $v_{l_1}$ : node at  $l = 1$  of CSG,
    $n_{max}$ : max. allowable size of a coalition
   Output:  $CS^*$ : Node with highest expected utility  $U^*$ 
2  $C_n$  is a coalition, where  $|C_n| = n_{max}$ .
3  $l_{n_{max}}$ : Lowest level with maximum number of  $C_n$ .
4  $l_{curr} \leftarrow 1$ ;  $OPEN \leftarrow v_{l_1}$ ;  $CLOSED = \{\emptyset\}$ .
5 while  $OPEN$  is non-empty do
6    $U^* \leftarrow \max_{v \in OPEN} \bar{U}(v)$ 
7   for every  $v \in OPEN$  do
8     for every  $v_{child} \in children(v)$  do
9       if  $v_{child} \ni C_n$  then
10        add  $v_{child}$  to CLOSED;
11       else
12         if  $l_{curr} < l_{n_{max}}$  then
13           if  $Val(v_{child}) > U^*$  then
14             add  $v_{child}$  to CLOSED;
15           else
16             start a DFS on subtree of  $v_{child}$  up to level  $l_{n_{max}}$ .
17             At each level  $l_{dfs}$  of dfs do
18               if (max. exp. util. of nodes generated at  $l_{dfs}$ )  $\geq U^*$  then
19                 add  $v_{child}$  to CLOSED;
20               exit DFS;
21           else
22             if  $\exists child(v_{child}) : Val(child(v_{child})) \geq U^*$  then
23               add  $child(v_{child})$  to CLOSED;
24    $OPEN \leftarrow CLOSED$ ;
25    $CLOSED \leftarrow \{\emptyset\}$ ;
26    $l_{curr} \leftarrow l_{curr} + 1$ ;
27 return  $CS^*$ ;

```

coalitions with size n_{max} . And we have already discussed earlier that coalitions with size n_{max} earn the highest value; therefore coalition structures with these coalitions in them will have higher chance to be the best coalition structures. This is the main insight of our approach. Therefore, a search starting from the bottom and going upwards in the CSG (bottom-up CSG search), unlike our previous approach [36], will be faster and more efficient. Possible sizes of the member coalitions in any coalition structure can be found by the integer partitions [5] of the total number of modules $|A|$. For example, there are 5 possible integer partitions of the number 4, which are (4), (3, 1), (2, 2), (2, 1, 1), (1, 1, 1, 1). From the coalition structure perspective, each integer partition indicates the sizes of the member coalitions in any coalition structure. For example, in Figure 2.3, level 1 of the CSG consists of only 1 node (coalition structure), which is $\{1, 2, 3, 4\}$. This coalition structure consists of only one coalition of size 4, which is the first integer partition of the number 4 – (4). In level 2, all the coalition structures have 2 coalitions in them. Either these coalitions have sizes 3 and 1 (such as $\{\{2\}, \{1, 3, 4\}\}$) or 2 and 2 (such as $\{\{1, 2\}, \{3, 4\}\}$). Thus the nodes in a CSG can be clustered according to their underlying integer partitions.

An illustration of the partitioned CSG with $|A| = 7$ and $n_{max} = 3$ is shown in Figure 2.6. This is a size based partitioning graph of the CSG. Each node represents the partition size of a coalition structure. For example, node (7) represents that the coalition structure corresponding to this partition has only one coalition in it with size 7; similarly the node (3, 3, 1) represents all the coalition structures which have 3 coalitions in them with sizes 3, 3 and 1 respectively. As n_{max} is 3, the highest valued coalition structures can be found under the (3, 3, 1) partition. Coalition structures corresponding to that partition can be generated from coalition structures of two different partition sizes, viz., (4, 3) and (1, 6). Note that all the coalition structures of partition size (4, 3) have one coalition of size n_{max} but that is not the case for (1, 6); however, both of the partitions can generate coalition structures of highest possible value. Figure 2.4 (right) shows that because of our proposed value function, the highest valued coalition structures, i.e., the coalition structures which have a

maximum number of n_{max} -sized coalitions in them, can be found in level $\lfloor \frac{|A|}{n_{max}} \rfloor (l_{n_{max}})$ in the CSG. Our proposed algorithm takes these factors into account and the search is designed accordingly.

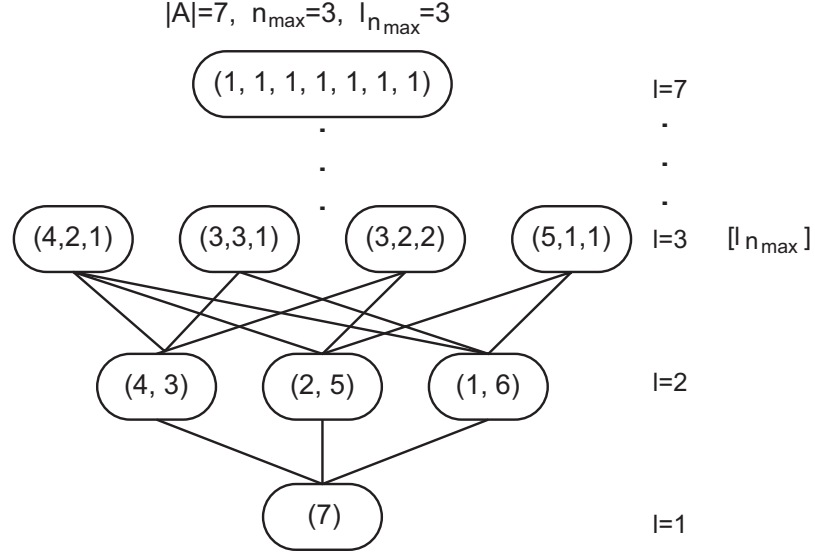


Figure 2.6: An abstraction of the CSG for $|A| = 7$ agents showing the partitions of A that are inspected at each level. Note that at level l , A has different partitions with exactly l parts or subsets. With $n_{max} = 3$, the maximum number of C_n -s (coalitions of size 3) are at level $l_{n_{max}} = 3$.

Discussion of the *bottomUpCSGSearch* algorithm. In [73], the authors have shown that if it is impossible for some node in the CSG to lead to the best coalition structure, then those unpromising nodes can be pruned right away and the search process can be made faster. However, identifying these unpromising nodes is a challenge. We have employed two different pruning strategies to reduce the search space by eliminating the unpromising nodes as soon as we encounter them. The main search procedure is shown in Algorithm 1. As the name suggests, the search (called *bottomUpCSGSearch*) for the best coalition structure starts from the bottom-most node of the CSG. The bottom-most node (coalition structure) contains only one coalition in it - the grand coalition, i.e., every module is part of this coalition. All the nodes which need to be explored further are kept in a data structure called *OPEN*. Initially the bottom-most node of the CSG is kept in *OPEN*. At every

level, all the children nodes of the nodes stored in the *OPEN* data structure are generated.

³ If a child node contains a C_n (a coalition having size n_{max}), it is immediately added to *CLOSED* for future expansion (lines 10 – 11). If a child node does not contain any C_n , it might still lead to the optimal coalition structure. To detect the suitability of a generated child node not including C_n , we check if the current level, l_{curr} , being explored in the CSG is less than $l_{n_{max}}$.

Let U^* denote the highest utility found. As utility is the difference between the value and cost of a coalition structure, if the value of any coalition structure is less than U^* , then the utility of that coalition structure will always be less than U^* . This is the main insight behind developing our first pruning strategy, the *fitness test*. This pruning strategy is applied only if the current level $l_{curr} < l_{n_{max}}$. For any newly generated coalition structure CS in level $l_{curr} (< l_{n_{max}})$, we first check whether the value of this coalition structure exceeds U^* or not. If $Val(CS) \geq U^*$, then it would make sense to explore the node CS further; therefore CS is added to the set *CLOSED* for future expansion (lines 13 – 15). On the other hand, if $Val(CS) < U^*$, i.e., the value of the coalition structure is already below the best utility found thus far, then it is evident that the utility of this node will be less than U^* . Therefore exploring all of its children nodes for further expansion and inspection will not be beneficial. These are *unpromising* nodes. For this type of nodes, we only explore their descendants up to depth $l_{n_{max}}$. While performing this depth-first search (DFS) of the unpromising node CS , if any of the descendants has a value $> U^*$, then we add CS to *CLOSED* for future expansion, as it can lead to a node that has utility better than the maximum utility obtained till then (lines 16 – 20). Otherwise, we just continue the search along the child (expanded node) that has the highest utility amongst all its generated siblings. If none of the descendants of CS , till level $l_{n_{max}}$, has higher value than the current highest utility, then CS is automatically pruned.

Figure 2.4 (right) suggests that up to level $l_{n_{max}}$ the value of coalition structure

³Children nodes of node v in level l of the CSG, denoted by $children(v)$, are the nodes connected via edges to node v in the CSG in level $l + 1$.

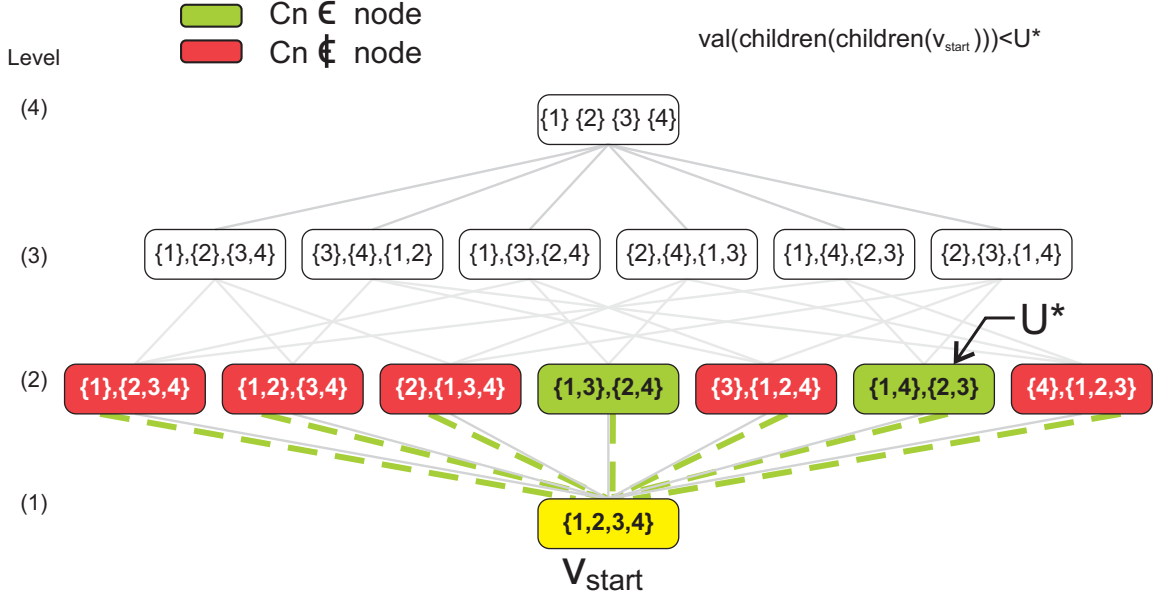


Figure 2.7: Illustrative example of the working procedure of the algorithm

increases, and beyond $l_{n_{max}}$ level, the value decreases. Therefore for expanding the nodes in any level $l_{curr} \geq l_{n_{max}}$, we use a more conservative approach to accommodate nodes for future expansion. As the values of the nodes (coalition structures) are descending beyond level $l_{n_{max}}$, it is evident that if any node in level $l(> l_{n_{max}})$ has value lower than U^* , then none of its descendants will have higher value than U^* . Thus if we encounter any node beyond level $l_{n_{max}}$, having lower value than U^* , then that node can be immediately pruned. On the other hand, if the node's value is greater than U^* , then the node is added to the *CLOSED* data structure for future expansion (lines 24 – 25).

The search proceeds through successive levels, until all nodes that exceed the best found value of the expected utility U^* have been explored. The best utility node found by the search algorithm is returned as CS^* , the node corresponding to the optimal coalition structure.

An example of the working of our algorithm with $|A| = 4$, $n_{max} = 2$ is shown in Figure 2.7. Here, the maximum valued coalition structure occurs at level $l = 2$ and is marked with U^* . The search algorithm expands the bottom node $\{1, 2, 3, 4\}$ and adds all its children

to *CLOSED*. However, in the next iteration, U^* is set to the maximum valued node in *OPEN* (which is also the maximum valued node in the entire CSG). Consequently, no other node in *OPEN* or its descendants has a value $\geq U^*$, and no other node is expanded by the search algorithm. The search terminates returning the node CS^* that corresponds to utility U^* .

Theoretical Analysis of *bottomUpCSGSearch* Algorithm

Establishing Worst Case Bound. As our value function (Eqn. 2.1) assigns the highest value to coalitions of size n_{max} , when A is partitioned into coalitions of sizes n_{max} , $Val(CS_{nmax}) = \sum_{A_i \in CS(A)} Val(A_i)$ has the highest value, where CS_{nmax} is a coalition structure which contains the maximum number of n_{max} sized coalitions in it. We denote ideal utility (when there is no cost to reform the coalition) as $U_{ideal} = Val(CS_{nmax})$. We call this set of coalition structures *ideal* coalition structures. For any coalition structure, CS , let $\alpha_{CS} = \frac{\bar{U}(CS)}{U_{ideal}}$. α_{CS} is dependent on the cost of forming coalition structure CS and gives a worst case bound. Let CS_1 denote the first coalition structure generated by the algorithm (with at least one C_n). $\alpha_{CS_1} = \frac{\bar{U}(CS_1)}{U_{ideal}} = \frac{Val(CS_1) - \overline{cost}(CS_1)}{Val(CS_{nmax})} = \frac{Val(CS_1) - \overline{cost}(CS_1)}{\beta \cdot Val(CS_1)} = \kappa(1 - \frac{\overline{cost}(CS_1)}{Val(CS_1)})$, where $\beta \geq 1$, $\kappa \leq 1$ and $\kappa = \frac{1}{\beta}$. Note that α_{CS_1} denotes the initial worst case bound on α_{CS} . With time we keep on improving this bound. This demonstrates the anytime property of our algorithm. The anytime property is very important from a practical aspect because even if the algorithm terminates prematurely, it still gives a solution which is guaranteed to be within a certain bound from the optimal.

Theorem 1 *The worst case bound α_{CS} is a function of cost of reconfiguration. (Proof follows from the earlier discussion.)*

Lemma 1 *If $n_{max} > 1$, a bottom-up search in the CSG (starting from $l = 1$) can establish a worst case bound more quickly than a top down-CSG search (starting from $l = |A|$).*

Proof: A worst case bound can be established by a CSG search algorithm as soon as a coalition structure with a coalition of size n_{max} is generated by it. Let l_{exp} denote the number of levels explored by a CSG search algorithm when it generates a coalition structure with a coalition of size n_{max} . For a bottom-up search starting from $l = 1$, the first coalition with size n_{max} is encountered at $l = 2$, where A is partitioned into two subsets of size n_{max} and $|A| - n_{max}$ respectively. Then $l_{exp}^{bottomup} = 2 - 1 = 1$. In contrast, in a top-down search starting from $l = |A|$, the first coalition structure that has a coalition of size n_{max} is encountered at level $l = |A| - n_{max}$, which gives $l_{exp}^{bottomup} = |A| - (|A| - n_{max}) = n_{max}$. Clearly, if $n_{max} > 1$, $l_{exp}^{bottomup} > l_{exp}^{topdown}$ and the bottom-up search explores fewer levels and generates a worst case bound more quickly than a top-down search.

Lemma 2 *The bottomUpCSGSearch algorithm does not remove any optimal coalition structure while removing unpromising nodes.*

Proof: (by contradiction) Suppose that, a node \hat{v} got pruned by *fitness test* and consequently the optimal coalition structure also got pruned. That means either \hat{v} was the optimal coalition structure or it could have generated the optimal coalition structure in future levels. \hat{v} cannot be the optimal coalition structure, because if $U(\hat{v}) > U^*$, we would not have deleted \hat{v} . If this happened in level l_{curr} , where $l_{curr} < l_{n_{max}}$, then before pruning \hat{v} , we have generated children nodes with the highest value of \hat{v} for successive levels up to level $l_{n_{max}}$ and none of its successor nodes have met the criterion $val(child(\cdot)) > U^*$. Also, after level $l_{n_{max}}$ the value of coalitions encountered in successive levels starts decreasing (Figure 2.4). And if the pruning happened where $l_{curr} \geq l_{n_{max}}$, then it was only because $val(child(\hat{v})) < U^*$. So \hat{v} could not have contributed to finding the optimal node. Hence proved.

Theorem 2 *The bottomUpCSGSearch algorithm is anytime.*

Proof: At every level of a CSG, our algorithm generates the nodes first which contain at least one coalition with size n_{max} . From Theorem 1, we can say, from the first generated

Algorithms	Complexity
Original complexity	$O(A ^{ A })$
Sandholm [81]	$O(3^{ A })$
IP algo. [74]	$O(2^{ A })$
Graph Partitioning [27]	$O(\log A)$
BP algo. [34]	$O(n_{max}^{n_{max}})$
bottomUpCSGSearch algo.	$O(\sum_{j=2}^{\lfloor \frac{ A }{n_{max}} \rfloor} S(n, j))$

Table 2.1: Complexity Comparison

coalition structure, this algorithm will be within a bound. We only admit a coalition structure if it increases this worst case bound further or it has potential to do so in future levels. Thus, this worst case bound successively increases with number of levels and eventually reaches the optimal utility.

Theorem 3 *The bottomUpCSGSearch algorithm finds the optimal coalition structure.*

Proof : Due to its anytime property, the *bottomUpCSGSearch* algorithm only admits coalition structures that have a higher utility than the previously inspected coalition structures. It also prunes unpromising coalitions that cannot be part of an optimal coalition structure (by Lemma 2 and Theorem 2). This ensures that our algorithm never accepts a coalition structure that has a lower utility than a previously seen coalition structure, neither does it prune a probable candidate node for optimal coalition structure. Hence, it finds the optimal coalition structure eventually.

The *completeness* of the proposed algorithm also follows from Theorem 3 which guarantees that it always finds the optimal coalition structure.

Complexity Analysis. The best case for this algorithm will be where $l_{n_{max}} = 2$ and all the nodes without C_n in it got pruned. In the worst case scenario, all the nodes in the graph will be generated and that will give us a complexity of $O(B_n)$, where $n = |A|$ and B_n denotes n th *Bell Number*. But in an average case, nodes will be generated only between $l = 2$ and $l_{n_{max}}$. The lower bound of the average time complexity will be $S(n, 2)$, where

$S(n, k)$ denotes Stirling Number of the second kind and $S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} [k, j] j^n$ and the upper bound of average case complexity will be $\sum_{j=2}^{\lfloor \frac{|A|}{n_{max}} \rfloor} S(n, j)$.

In Table 2.1, we have provided average case complexities for existing coalition structure search algorithms, in agent coalition formation and MSR configuration generation. Here, $|A|$ is the number of agents or modules and n_{max} is the maximum allowed size of a coalition. Sandholm's anytime solution [81] has a time complexity of $3^{|A|}$, which first searches the bottom-most 2 layers of the CSG to search through all possible coalitions. In [74], the authors' anytime solution, based on integer partitioning, has an improved time complexity of $2^{|A|}$. A graph partitioning algorithm [27] for coalition structure formation solves the 0 – 1 integer linear programming problem (which is part of the graph clustering technique, a well known NP-Complete problem), by relaxing it to a general linear programming problem; thus the solution is found in sub-linear time ($O(\log|A|)$). Our previously proposed BP algorithm's [34] complexity depends solely on the value of n_{max} .

2.2.3 Experimental Evaluation

In this section, we will describe various experiments that we performed to check the performance of our proposed search algorithm for dynamic partitioning of modules for configuration generation through extensive simulations.

Experimental Setup

We consider a setting where a set of $|A| = [4, \dots, 12]$ modules are present in the system. n_{max} has been varied through $\{2, 4, 5, 6\}$. The size of the environment is $10 \text{ m} \times 10 \text{ m}$. The initial positions of the modules are drawn from uniform distribution $\mathcal{U}[(0m, 10m), (0m, 10m)]$. Initial orientations of the modules are drawn from uniform distribution $\mathcal{U}[0, \pi]$. Noise values are also drawn from uniform distribution $\mathcal{U}[0, 1]$. The

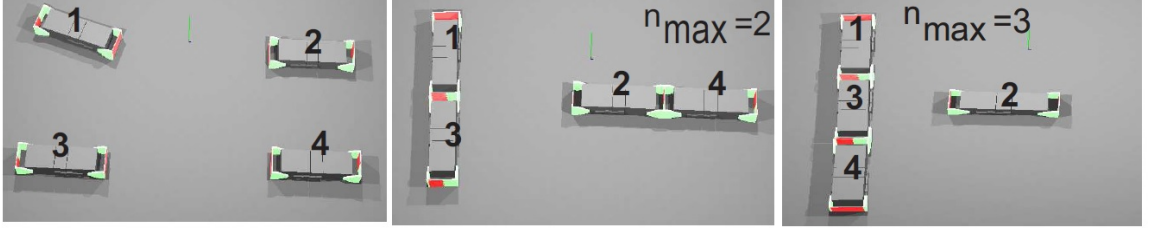


Figure 2.8: Configuration generation process using our proposed *bottomUpCSGSearch* algorithm.

simulations for *bottomUpCSGSearch* algorithm are run on a desktop PC, with 24GB of RAM and an Intel(R) Xeon(R) CPU with 2 processors.

Figure 2.8 shows an instance of the experimental setup. Initially 4 modules, $\{1, 2, 3, 4\}$ are randomly distributed. The objective of the modules is to find the coalition structure that gives the highest utility calculated from previously discussed functions. To do this, each module runs the *bottomUpSearchCSG* algorithm, where each module starts the search from the bottom-most node in the CSG. As we assume that all the modules have other modules' position and orientation information, they first calculate the utility of the coalition structure where all the modules are connected together (bottom-most node). Whenever any node in the CSG is searched by a module, first its value is calculated using equation 2.1 and then the cost of forming this coalition structure from the initial coalition structure is calculated. Once the value and the cost of a node (i.e., a coalition structure) are calculated, utility can be calculated from them, to be used in algorithm 1. Finally all the modules find the best partition to form by using the *bottomUpSearchCSG* algorithm.

In Figure 2.8, two different sets of configurations, for different n_{max} values, formed by the modules are shown. In Figure 2.8, with $n_{max} = 2$, the best coalition structure found is $\{\{1, 3\}, \{2, 4\}\}$, whereas with $n_{max} = 3$, the best coalition structure found is $\{\{1, 3, 4\}, \{2\}\}$. The main metrics reported in this article are the time calculated and number of nodes explored in the CSG while searching for the best configuration, with different numbers of modules present in the environment.

No. of agents	Ratio to opt. value	Runtime (ms)
4	1	1
6	1	2
8	1	7
10	1	16
12	1	33

Table 2.2: Ratio of values of best coalition structure found using our algorithm to the optimal value and corresponding running times.

Simulation Results

Solution quality and run time

In the first set of experiments, we analyzed the effect of the main concept of our algorithm, i.e., finding the best coalition structure possible from the CSG. As shown in Table 2.2, for $4 \leq |A| \leq 12$, our algorithm was able to do a search in the space of all coalition structures and find the optimal coalition structure for all values of $|A|$. For higher values of $|A|$ the exhaustive search (complexity $O(|A|^{|A|})$) becomes prohibitive. The value of $l_{n_{max}}$ is fixed to 2 for this test. These data show that our algorithm takes a fraction of a second to find the best coalition structure. With 6 modules, the algorithm takes only 2 milliseconds, whereas for 12 modules, the run time of the algorithm is 33 milliseconds. When $l_{n_{max}}$ is fixed to 3, run time of the algorithm with 12 modules, increased to 667 ms. Figure 2.9 shows how actual space complexity for our algorithm and the run time to explore the nodes change with the number of modules. It is evident from this figure that even though space complexity is increasing exponentially with increasing number of modules, our algorithm maintains the run time within a reasonable value and finds the optimal coalition structure. This also shows that our algorithm scales well with number of modules.

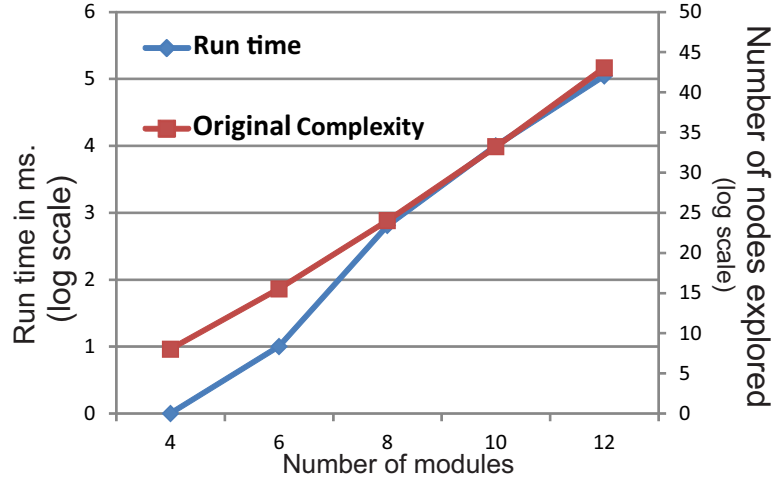


Figure 2.9: Comparison of run time of the *bottomUpCSGSearch* algorithm and actual space complexity against the number of modules

Comparisons with existing algorithms

Figure 2.10 shows the comparison between the number of nodes explored by our proposed algorithm with the existing coalition structure search algorithms proposed in [81] and [74]. These algorithms are anytime in nature and have worst case time complexities $O(3^{|A|})$ and $O(2^{|A|})$ respectively. As these algorithms do not necessarily support size-constrained value functions, the numbers of nodes explored in the CSG by them are higher than our proposed algorithm. The graph is shown on a logarithmic (\log_2) scale. The value of n_{max} is set to 2 for $|A| = 4$ modules, 4 for $|A| = 6$ modules and 6 for $|A| \geq 8$ modules. This graph shows that our algorithm is able to prune more of the unpromising search space than the other two algorithms and consequently its time and space complexities also reduce.

In our next set of experiments, we compared the number of nodes generated by our algorithm for 12 modules against previously studied algorithms for coalition structure generation in [81] and [74]; n_{max} is varied between 4 and 6. Figure 2.11 (a) shows the number of nodes generated in all the algorithms on a logarithmic (\log_2) scale. As can be seen, our algorithm explores 2^3 times fewer nodes when $n_{max} = 4$ and 2^8 times fewer nodes when $n_{max} = 6$, than the algorithm in [81]. It also explores 2 times fewer nodes in

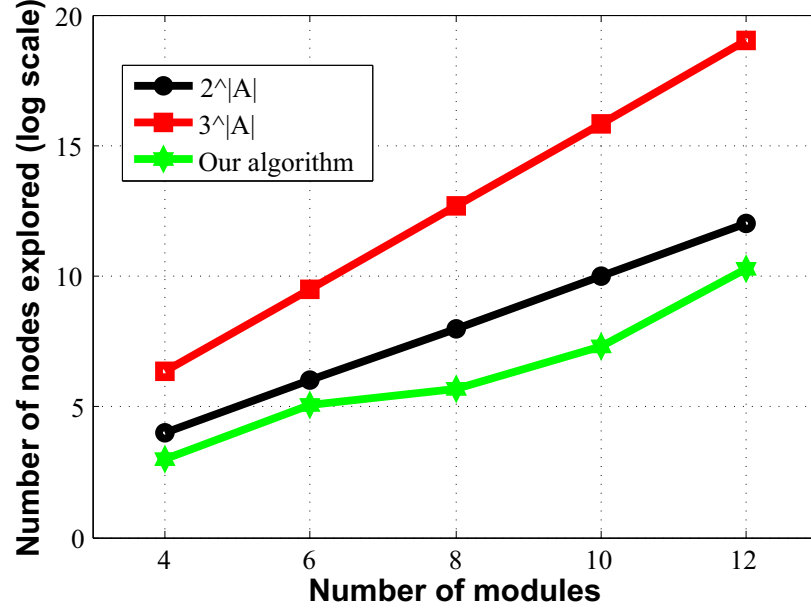


Figure 2.10: Comparison of number of nodes explored by different existing coalition structure generation algorithm and the proposed *bottomUpCSGSearch* algorithm.

the case of $n_{max} = 6$ than the algorithm in [74]. This illustrates that using a CSG search algorithm for the unconstrained coalition formation problem can become inefficient for the coalition size-constrained coalition formation problem, as the number of agents increases. It is also understood that the algorithm proposed in this chapter is a function of n_{max} rather than of $|A|$.

Next we have compared the run time of our algorithm with existing algorithms where coalition structure search algorithms have been used for configuration generation in ModRED [7]. The first algorithm we compared with is *searchUCSG*. Figure 2.11 (b) shows the comparison of our algorithm's running time with *searchUCSG* [36], where the authors proposed a top-down, heuristics-based search algorithm to search the CSG. It shows that for 12 agents, the *searchUCSG* algorithm took 3.81×10^5 milliseconds whereas our proposed *bottomUpCSGSearch* algorithm took only 33 milliseconds to find the optimal solution – a run time improvement in the order of 10^4 ms. The main reason behind this improvement in time is that we start the search from the bottom of the CSG and therefore we reach the level $l_{n_{max}}$, where nodes with maximum number of n_{max} -sized coalitions are stored, faster

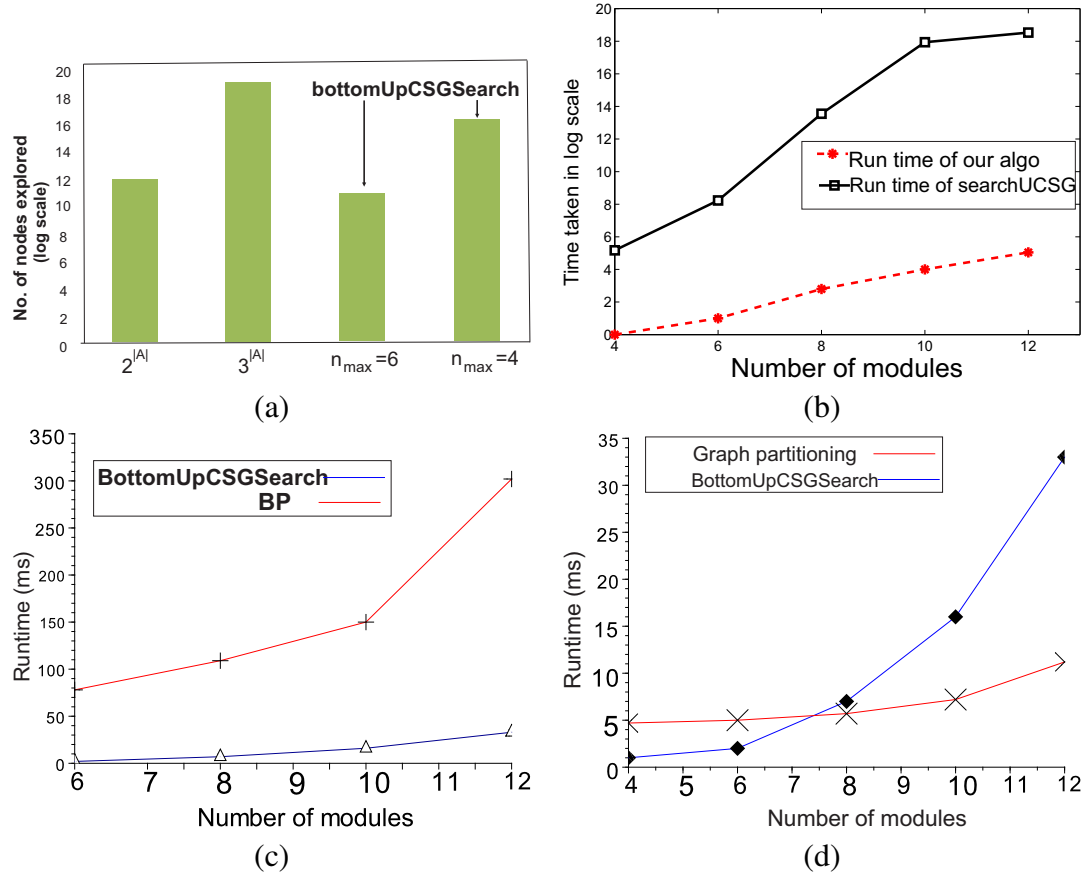


Figure 2.11: a) Comparison of number of nodes generated with existing algorithms – for 12 modules, with varying n_{max} ; b) Run time comparison with *searchUCSG* algorithm; c) Run time comparison with *graphPartitioning* algorithm; d) Run time comparison with *BP* algorithm.

than the *searchUCSG* algorithm.

Our next comparison was done with the *BP* algorithm, where the coalition size is constrained by n_{max} , using a *hard* constraint, i.e., no coalition with size more than n_{max} was generated. The *BP* algorithm partitions the coalitions of different sizes into multiple blocks and searches through these blocks to form the coalition structures. Our *bottomUpCSGSearch* algorithm takes less run time than the *BP* algorithm (Figure 2.11 (c)). For example, by using the *BP* algorithm, for 6 and 12 modules, the run times are 78 and 302 milliseconds respectively, whereas for the same number of modules *bottomUpCSGSearch* takes 2 and 33 milliseconds run time respectively – an improvement of almost 10 times.

Lastly, we compared *bottomUpCSGSearch* with a *graph partitioning* algorithm [27]. This algorithm restricts the problem to a weighted graph where coalition utilities are calculated by summing pairwise utilities (edge weights), making it a polynomial problem. On the other hand, our algorithm tries to deal with the original NP-hard problem without any assumption or relaxation. This comparison is shown in Figure 2.11 (d). Being a NP-hard problem, our algorithm performs comparably against the polynomial-relaxed *graph partitioning* algorithm - for 12 modules, our *bottomUpCSGSearch* algorithm takes 33 milliseconds while the *graph partitioning* algorithm takes 11 milliseconds - only 3 times worse than a polynomial solution.

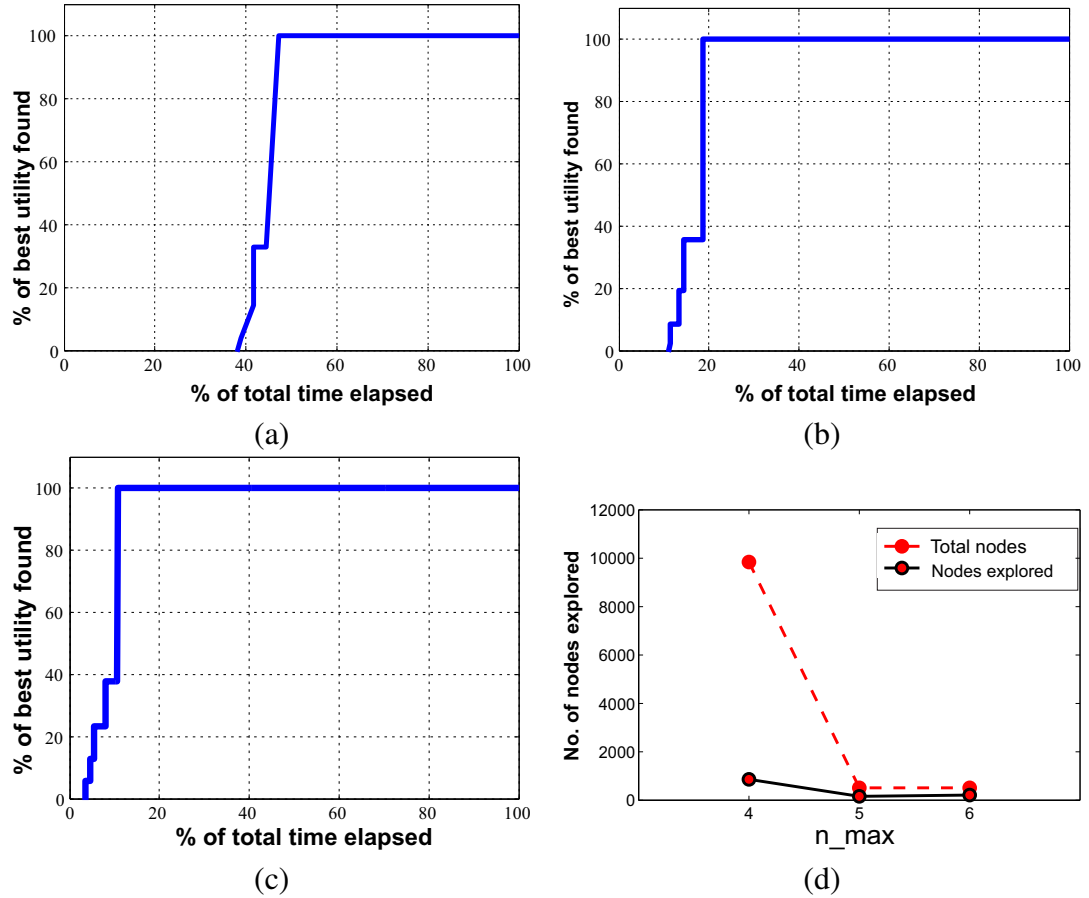


Figure 2.12: (a) - (c) Exploring the anytimeness nature of the proposed *bottomUpCSGSearch* algorithm – for 8, 10 and 12 modules; d) Effect of changing n_{max} on number of explored nodes – for 10 modules.

Empirical evaluation of the anytimeness property

In the next set of experiments, we have tested the anytimeness property of our proposed algorithm. Figure 2.12 (a)-(c) shows the anytimeness nature of our algorithm, for 8, 10 and 12 modules respectively. The x-axis denotes the percentage of total time elapsed and the y-axis denotes the percentage of the best utility found in that particular time-step. For these tests, $l_{n_{max}}$ is set to 2. As can be seen from these figures, for 8 modules (Figure 2.12 (a)), the best coalition structure is found in about 40% of the total time, whereas for 12 modules (Figure 2.12 (c)), the best coalition structure is found in about 10% of the total time. One should note that as the total run times of the algorithm for 8, 10 and 12 modules are not the same, therefore 10% of the total time for 12 modules is not necessarily less clock time than 40% of the total time for 8 modules.

Experiments with n_{max}

For the next set of experiments, we varied n_{max} through 4, 5, 6 while keeping $|A|$ fixed at 10 (Figure 2.12 (d)) and as can be seen, compared to the total number of nodes possible in those levels, we were able to restrict the number of explored nodes within a polynomial bound and still achieved optimal utility every time. Also this figure shows that for $n_{max} = 4$, more nodes are explored than when $n_{max} = 5$ or 6. When $n_{max} = 4$, then $l_{n_{max}} = 3$, whereas when $n_{max} = 5$ or 6, then $l_{n_{max}} = 2$, instead of 3. So, we can see that when n_{max} is set to 4, then more need to be explored in order to reach the *best* level. On the other hand, when n_{max} is set to 5 or 6, to reach the *best* level, fewer nodes are explored. This is why our algorithm's complexity depends on both $|A|$ and n_{max} – more specifically on the value of $l_{n_{max}} (= \lfloor \frac{|A|}{n_{max}} \rfloor)$.

We have also experimented with the effect of the set of $\{|A|, n_{max}\}$ on the number of explored nodes in the CSG. We experimented with different values of $|A|$ and n_{max} while maintaining $l_{n_{max}}$ at 2. The different combinations of $|A|$ and n_{max} that were used are: $\{4, 2\}$, $\{6, 3\}$, $\{8, 4\}$, $\{10, 5\}$ and $\{12, 6\}$. From Figure 2.13, it is evident that though the

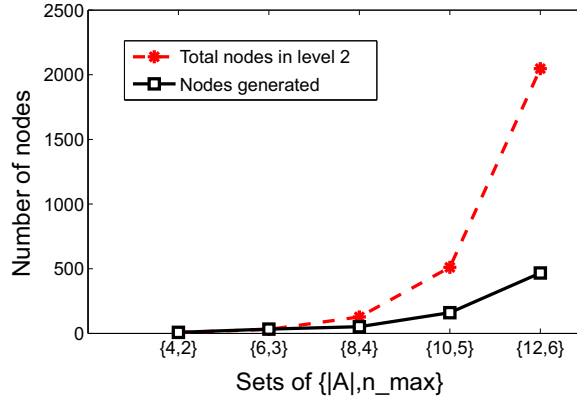


Figure 2.13: Comparison of total number of nodes in $l_{n_{max}} = 2$ vs. no. of nodes generated.

total number of nodes in level 2 increased exponentially, still our algorithm was able to keep the number of nodes generated polynomial with respect to the increasing values of $l_{n_{max}}$, while finding the optimal coalition structure.

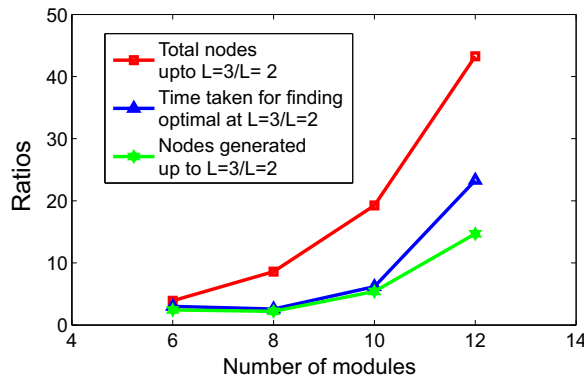


Figure 2.14: Ratios between total number of nodes, number of nodes generated and time taken to find optimal for $l_{n_{max}} = 2$ and 3.

For the next set of experiments, we took the ratios between the total number of nodes up to level 3 and level 2, between the number of nodes explored in those levels and also between the total time taken for exploring nodes in those two levels. This is shown in Figure 2.14, which shows that the ratio of total number of nodes possible in level 2 and 3 is exponentially increasing. But in the case of our algorithm, the ratio of total number of nodes explored in those levels increased in a polynomial fashion. Therefore the graph for the ratio of time taken for exploring nodes in those two levels did not increase exponentially.

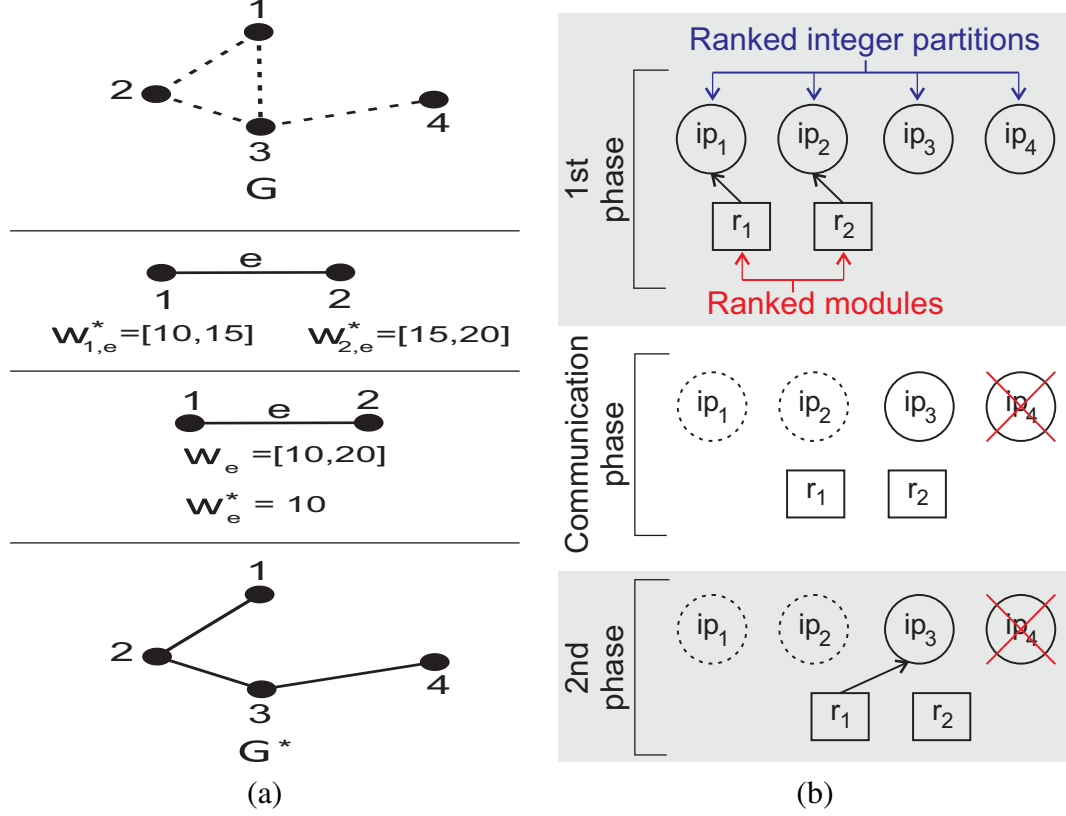


Figure 2.15: (a) MST generation among modules, (b) distribution of integer partitions among modules.

2.3 A Distributed Approach: Spanning Tree Partitioning

In this section, we briefly discuss the distributed approach that we have proposed to solve the partitioning problem. For more details, the readers are referred to [39]. This work proposes a novel technique that attempts to reduce the search space by first constructing a minimal reachability graph between modules that are within communication range of each other, in the form of a minimum spanning tree (MST). We have noticed that the partitions or coalition structures (shown in Figure 2.3) can be categorized into groups according to the sizes of their subsets and these groups can be maintained using all possible integer partitions of n . Then, the possible combinations of only those modules that are connected in the MST, up to a specific size, n_{max} , which is determined from the maneuverability constraints of modules, are explored using an integer partitioning based technique to

find the combination or configuration of modules that gives the highest expected utility. The computations are performed in a distributed manner by the modules and portions of the problem that are solved by each module are allocated to them in proportion of their available energy (battery) for computation. Figure 2.15 shows an illustration of our proposed approach. The top-most graph in Figure 2.15(a) shows the reachability graph – if two modules are within each other’s communication range, then there is an edge between them. Next, a MST of this graph is calculated based on each module’s uncertain realization of edge weights originating from it. After the MST is built, the integer partitions (ip) are distributed among modules. Each module only searches through the coalition structures corresponding to its allocated integer partitions and only the coalition structures are considered which contain coalitions that are connected in the MST. We have performed simulated experiments and shown that our proposed technique is able to rapidly identify the highest utility configuration for different number of modules and performs significantly better in terms of time and space complexity than previously existing techniques for MSR configuration identification and coalition structure search algorithms. We have also empirically shown that our proposed algorithm scales better than existing techniques.

2.4 Discussions

This chapter discusses a very unique problem in modular robotics – partitioning a set of modules (robots) for forming the best utility teams (configurations). Our proposed algorithms are one of the very first approaches to solve this problem. We have experimented with different values of n_{max} , maximum size that an MSR can have, and shown that even with different n_{max} values, our algorithm scales well.

To solve the problem, we have proposed two dynamic partitioning techniques for configuration generation in modular robots. Our centralized *bottomUpCSGSearch* algorithm models the problem as a CSG search problem for finding the best coalition

structure and intelligently generates, searches and prunes nodes in the CSG. We have also provided analysis on worst case guarantee that our proposed algorithm provides. We have shown that our algorithm is anytime and complete, and also provided a complexity analysis for the algorithm. We have empirically shown that our algorithm scales well with the number of modules.

On the other hand, our distributed minimum spanning tree partitioning approach takes into account different constraints of the real world such as communication, and battery power and finds the optimal configuration for the current situation. To the best of our knowledge, this work is the first approach to solve the constrained partitioning problem in a distributed manner.

Chapter 3

Planning: Configuration Formation

Over the last decade, the self-reconfiguration problem has been studied most extensively by MSR researchers [1]. It has been proved to be a NP-complete problem [51]. Several approaches based on graph theory [52, 6] and control theory [77, 61] have been proposed. In our research, we look at a fundamental problem and a pre-requisite of the self-reconfiguration problem, called the configuration formation problem, where modules self-aggregate to form different configurations. We study the configuration formation problem from two different aspects:

1. **Initially all the modules are singletons** (Figure 1.2): Given a set of singleton modules initially distributed arbitrarily within the environment and a desired target configuration involving those modules, how can each module(s) select an appropriate position or spot in the target configuration to move to, so that, after reaching the position, it can readily connect with adjacent modules and form the shape of the desired target configuration. Figure 1.2 shows an illustration of the configuration formation problem with 5 ModRED modules [7]. The initial positions of the modules and the target configuration are shown in Figure 1.2.(left), while Figure 1.2.(right) shows the final configuration, where the modules have been allocated to their respective spots and they have moved there. The configuration formation problem is

non-trivial as the desired spots of different modules in the target configuration could conflict with each other, resulting in occlusions and leading to failed attempts to achieve the target pattern. Additionally, it is beneficial for the modules to reduce the energy (battery) expenditure, and it would make sense for robots to solve the problem in a way that reduces the traveled distances to their desired positions, so that the cost of locomotion is also reduced. To address these issues, we have proposed both distributed and centralized algorithms using theories ranging from bipartite graph matching [48] to auctions [10].

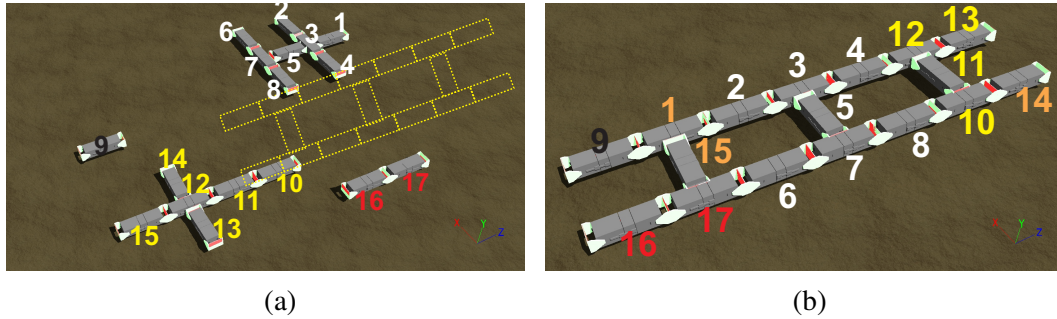


Figure 3.1: (a) A singleton and three initial configurations consisting of 2, 6 and 8 modules respectively, and desired target configuration (marked with yellow dotted lines) (b) target configuration involving all 17 modules connected in ladder configuration; module numbers marked in white, yellow and red are retained between initial and target configurations.

2. **Initially some modules are connected with each other forming (small) configurations** (Figure 3.1): Given a set of singleton modules and set of modules connected in (multiple) configurations which are initially distributed arbitrarily within the environment and a desired target configuration involving those modules, how can the modules be allocated to spots in the target configuration such that along with the cost of moving from initial locations to the goal locations, the initially connected configurations can be preserved as much as possible to reduce the disconnections in the initial configurations and re-connections in the target configuration. The reason for preserving the initial configurations and thus reducing

the number of disconnections and re-connections is that the connecting two modules using their end-connectors is a very costly (energy-wise) and difficult task [50]. This problem is non-trivial as the modules might already be connected in initial configurations that do not correspond to parts of the target configuration. Moreover, multiple modules from different initial configurations might end up selecting the same most-preferred position in the target configuration, leading to failed attempts to achieve the target configuration. To address these challenges, we propose a decentralized algorithm that allows modules from initial configurations to select suitable positions in a target configuration using a technique based on subgraph isomorphism which aims to reduce the cost of movement as well as the number of disconnections and re-connections.

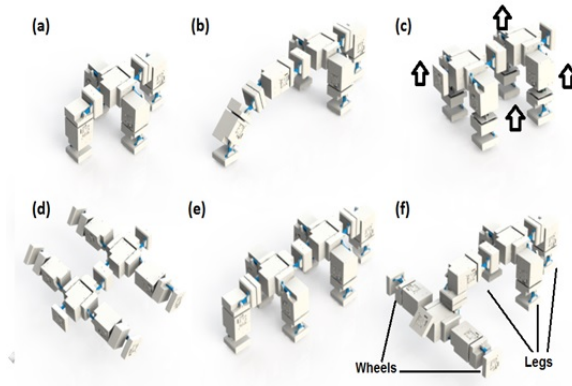


Figure 3.2: Complicated configurations formed by ModRED II modules [49].

3.1 Background

First, we briefly review different categories of self-reconfiguration strategies. The self-reconfiguration problem in MSRs can be classified along three main directions, as described below:

- **Search-based Reconfiguration:** In this technique, the objective is to find a path, or a sequence of moves among the modules within their configuration space, that takes the MSR from an initial configuration to a goal configuration. This technique requires *a priori* knowledge of the goal configuration [15, 18].
- **Control-based Reconfiguration:** This technique uses local movement rules for each module and eventually allows the whole MSR to evolve towards the goal configuration. This technique also requires *a priori* knowledge of the goal configuration [77].
- **Task-driven Reconfiguration:** This technique is not focused on achieving a specific goal configuration; rather it tries to get the modules into a configuration that will enable the MSR to perform its assigned task most effectively. In this technique, the MSR's target configuration is selected dynamically based on current conditions (e.g. environment, assigned task, battery power, etc.). This technique offers flexibility in selecting intermediate configurations and gives higher robustness as compared to techniques which require *a priori* knowledge. This technique has recently shown considerable success [61] in MSR self-reconfiguration.

In our research, we have looked into the configuration formation problem which can be imagined as a pre-requisite step for the self-reconfiguration process.

Configuration formation is the way of autonomously aggregating modules together to form a target pattern. This enables the modules to form the desired pattern. In a recent survey, authors have found that configuration formation problem has not been studied extensively in the literature [1] and the solution approaches proposed so far are not always easy to generalize to all MSRs. A few studies on configuration formation (by means of programmable self-assembly) can be found for self-actuated modular robots [58], and for modules that lack innate actuation ability, like stochastically-driven modules in liquid environments [93]. But these approaches can not be generalized to the ModRED MSR

directly. Some complicated configurations formed by ModRED II MSRs are shown in Figure 3.2.

In swarm robotic systems, configuration formation is known as pattern formation or self-assembly. As the swarm robots are usually not equipped with connectors to connect with other robots, instead they aggregate nearby to form different patterns. In [96], the authors have defined self-assembly as the following: “*self-assembly can be defined as a reversible process by which pre-existing discrete entities bind to each other without being directed externally*”. There are many studies on autonomous self-assembly of robot swarms. Alonso-Mora *et al.* [4] have solved the problem of forming artistic patterns by miniature swarm robots where they are initially distributed arbitrarily (spatially) in an environment and their final objective is to aggregate in such a way that they form the given pattern. They have used Voronoi partitioning to divide the region, and then each robot is allocated to unique partitions to go to for forming the desired shapes (shown in Figure 3.3).

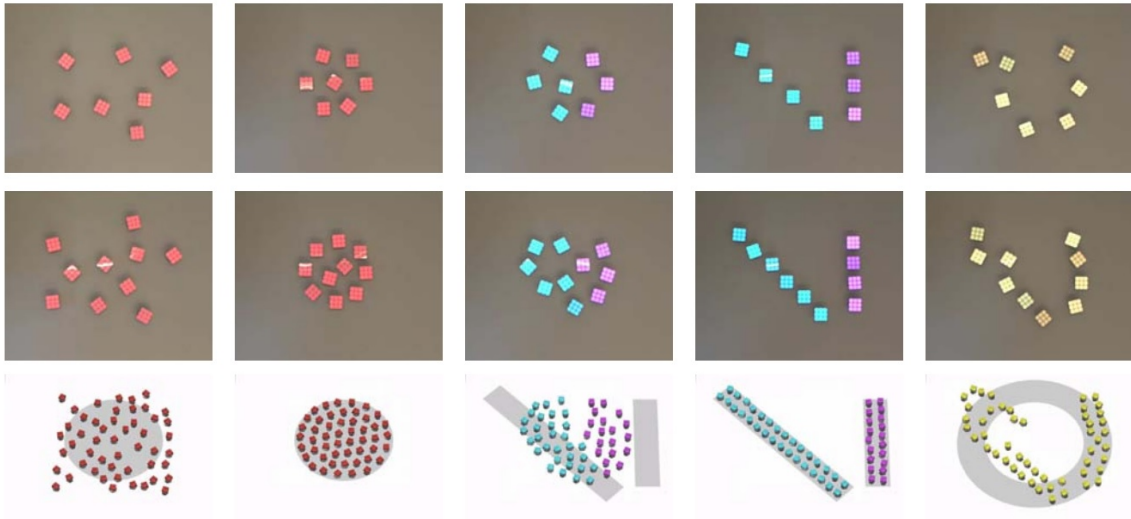


Figure 3.3: Artistic patterns formed by robot swarms [4].

Similarly, distributed algorithms for shape construction using swarm robots [95] have been proposed to solve the problem of arranging blocks to certain positions in a target shape. These blocks are then carried by the robots to those locations to form the shape.

Unlike other approaches, in this work, the authors have solved the self-assembly problem in 3D. Specific patterns, such as circle formation by asynchronous robots, have been studied in [31, 28]. In [43], the authors have proposed distributed approach to solve the self-assembly problem in S-bots swarm robotic systems. Recently, researchers have proposed a programmable self-assembly based approach for a swarm containing a thousand robots [79]. Not only in swarm robotic systems, self-assembly is a phenomenon which can be observed in nature [96] that uses similar principles across length scales ranging from nano-size DNA particles [98] to aggregation of social insects [23].

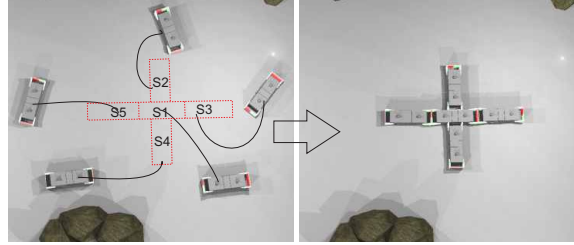


Figure 3.4: Configuration formation problem in MSRs. Left: 5 modules are located at arbitrary positions and have to achieve the target configuration (red dotted lines), Right: Modules after achieving target configuration.

3.2 General Problem Formulation

Let $\mathbb{A} = \{a_1, a_2, \dots\}$ denote a set of robot modules. Each $a_i \in \mathbb{A}$ has an initial pose denoted by $a_i^{pos} = (x_i, y_i, \theta_i)$, where (x_i, y_i) denotes the location of a_i and θ_i denotes its orientation within a 2D plane corresponding to the environment. Each module has a unique identifier. For the purpose of navigation, each module uses a map of the environment; the map is decomposed into grid-like cells using a cellular decomposition technique. We assume that initially all the modules are within each others' communication range.

In the variant of configuration formation problem studied in this research work, singleton robot modules, starting from arbitrary initial locations, are required to get into a specified target configuration. The target configuration is represented as a graph, denoted

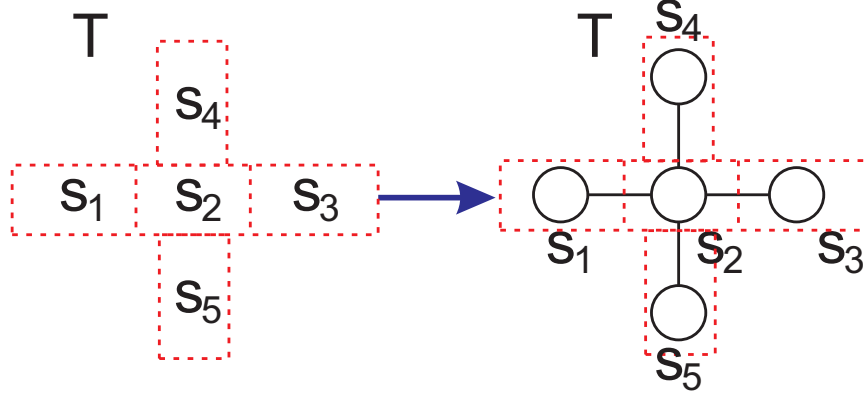


Figure 3.5: Illustration of how the target configuration T is modeled as a graph.

by $G_T = (V_T, E_T)$, where $V_T = \{s_1, s_2, \dots\}$ is the set of vertices and $E_T = \{e_{ij} = (s_i, s_j)\}$ is the set of edges. Each vertex in V_T is referred to as a *spot* that a module needs to occupy. Each spot $s_i \in V_T$ is specified by its pose and its neighboring spots in the target configuration, $s_i = (s_i^{pos}, neigh(s_i))$, where $neigh(s_i) \subset V_T$. An illustration is shown in Figure 3.5.

3.3 Simultaneous Configuration Formation and Information Collection

In this particular work, we impose another criterion on top of the configuration formation, which is, modules need to plan their paths from their initial locations to the spots in the target configuration in such a way that the paths are maximally informative. For information collection purposes a robot needs to sense the region it is situated in with its sensors. We discretize the information collection procedure, by using \mathcal{C} to denote the set of information collection locations or cells in the environment. \mathcal{C} can be decomposed into two disjoint subsets, O and U , corresponding to the cells that are visited and not visited by the robots. Note that, $V_T \subset \mathcal{C}$. Robot a_i 's path from its current location to a spot, s_j , in a target configuration is defined as an ordered sequence of cells it visits, i.e.,

$P_{ij} = \{c_1, c_2, \dots, s_j\} \subseteq \mathcal{C}$. Cost of a path P_{ij} is defined by the number of cells present in that path, i.e., $cost(P_{ij}) = |P_{ij}| - 1$.

To model the environmental phenomena generating the information, we have used Gaussian processes (GP) [44, 47]. Modeling the environment as a GP requires the assumption that all the sampling locations \mathcal{C} in the environment have a joint Gaussian distribution. A GP can be defined by its mean $m(\cdot)$ and its co-variance (*kernel*) $k(\cdot, \cdot)$ functions. Given a set of measurements X_O , we can predict the information measurement in the rest of the unobserved locations U , conditioned on X_O . A GP can be specified by the following equations [44]:

$$\mu_{U|O} = \mu_U + \Sigma_{UO}\Sigma_{OO}^{-1}(X_O - \mu_O)$$

$$\sigma_{U|O}^2 = \Sigma_{UU} - \Sigma_{UO}\Sigma_{OO}^{-1}\Sigma_{OU}$$

where $\mu_{U|O}$ is the conditional mean and $\sigma_{U|O}^2$ is the variance. Σ_{UO} is the co-variance matrix, with an entry for every location $o \in O$. Following GP formulations, the objective of informative path planning is to plan a path which maximizes the entropy, where entropy is given by:

$$H(U|O) = \frac{1}{2} \log(2\pi e \sigma_{U|O}^2) \quad (3.1)$$

The main idea behind entropy maximization is to select the locations in the environment which have the highest amount of uncertainty.

We have modeled the path planning with information collection problem as an instance of the bounded-cost search problem [89]. In this problem, the evaluation function for a cell is called its *potential*. The potential of a cell c is defined in our problem as $u(c) = \frac{B-g(c)}{h(c)}$, where $g(c)$ is the cost of moving from the start cell(location) to cell(location) c , $h(c)$ is the estimated cost of moving from cell(location) c to the goal location, and B is the budget that corresponds to the maximum of number of cells in any module's path from

its current position to the goal location, i.e., maximum allowable path length. From this it follows that the cost of the path used by module a_i to occupy spot s_j is budget-limited to B , i.e., $cost(P_{ij}) \leq B, \forall a_i \in \mathbb{A}, s_j \in V_T$. The informativeness of path P_{ij} is computed as $inf(P_{ij}) = \sum_{\forall c_k \in P_{ij}} H(c_k)$.

For finding the path from every module's current location to its goal position in the target configuration, a best-first technique is used which explores nodes with larger *entropic potential* ($hu(\cdot)$) values, defined as $hu(c) = u(c) + H(c|O)$. Formally we can define the studied problem as follows: Given a set of singleton modules \mathbb{A} and a set of spots V_T representing the target configuration, find a suitable allocation $f : \mathbb{A} \rightarrow V_T$ such that $\forall a_i \in \mathbb{A}, P_{ij} = \arg \max_{P \in \Pi, s_j \in V_T} inf(P)$ and $cost(P_{ij}) < B; \forall a_k \neq a_i, f(a_i) \neq f(a_k)$, where Π denotes the set of all possible paths from a_i 's current location to the goal location.

3.3.1 Algorithm Description

The solution approach is divided into two phases - a *planning phase*, where modules select spots in the target configuration and an *acting phase*, where modules move to their selected spots.

Planning Phase

In the beginning of the planning phase, all the modules broadcast their positions and orientations. We assume that each module autonomously and independently plans its paths to all the spots, and a module is aware of only its local planning information for any spot. Consequently, multiple modules could have identical maximum informative paths for the same spot and end up choosing it to move to. This could result in occlusions to each other, and, in the worst case, a failure of the configuration process. To avoid such a situation, we propose an additional coordination mechanism by employing a centralized supervisor to resolve conflicts between modules for the same spots in a structured manner, without incurring a high computational overhead.

Computing Informative Paths using Entropic Potential Search (EPS) Algorithm:

Our proposed planning mechanism operates in two phases, as shown in Algorithm 2. In the first phase, called the *computation* phase, each module a_i first calculates informativeness of the paths from its current location to each of the spots in V_T , using the Entropic Potential Search algorithm (EPS) (Algorithm 3). This is a modified version of the PTS algorithm proposed by Stern et al. [89]. The algorithm employs a greedy best-first technique to explore the cells with high entropic potential values. The EPS algorithm takes a module's current location and one of the positions in the target configuration as input, along with the bounded cost (budget) B . A data structure, called *OPEN*, is maintained for holding nodes for further exploration. Another data structure, called *CLOSED*, is maintained for holding the nodes which have been explored already.

In each iteration, the node, n_{max} , with the highest entropic potential value is expanded. If the current neighbor cell, n_n , of n_{max} is already in *OPEN* with smaller or equal $g(\cdot)$ value, then n_n is ignored. Because we assume the heuristic function, $h(\cdot)$, to be admissible, it is necessary to check whether $g(n_n) + h(n_n)$ surpasses B . If $g(n_n) + h(n_n) > B$, then n_n is pruned, as it can never be a part of the required bounded cost solution. If n_n is the goal cell, then the search procedure terminates. Otherwise, n_n is pushed back into *OPEN*, if the entropy value of cell n_n , $H(n_n)$, is greater than 0, and the search continues¹. This way we never explore a cell which does not guarantee to have any entropy value. Once EPS is terminated either we find a path with cost lower than B which is also highly informative or EPS returns null to notify that no such path with cost lower than B exists.

Every module individually runs the EPS algorithm for every spot $s_j \in V_T$. Each module sends its list of spots with computed informativeness to a supervisor node for the following *allocation* phase.²

Allocation: During the *allocation* phase, the supervisor waits until it receives the sorted

¹Initial cells of the modules have been treated as obstacles and therefore restricted to be added to *OPEN*.

²The supervisor could be a centralized external entity or one of the modules with higher computational capabilities elected using a leader election protocol.

lists of spots from all the modules. Then it proceeds to allocate spots in rounds, while allocating one spot in each round, starting from s_1 . In round j , spot s_j is allocated to the module a_i that has the highest informative path $inf(P_{ij})$ to s_j . If a module is allocated in a certain round, it is not considered for allocation in subsequent rounds. In case every available module's path cost exceeds budget B , it means that there is no module available that can occupy s_j while remaining within the battery constraint. In such a case, the module that has the lowest cost path P_{ij} among the conflicted modules for spot s_j is allocated to s_j . A similar strategy is used even if all the modules have the same informative paths for a specific spot, where path cost is below B . If ties still remain after applying the above strategy, they are broken at random. At the end of the allocation phase, the supervisor sends the list of allocated spots to all the modules.

Algorithm 2: Spot Allocation (SA) Algorithm

1 Phase 1: Computation Phase by Modules

2 Each module a_i will do the following:

3 For all spots $s_j \in V_T$, execute *pathFormation()* algorithm and find a set of paths, P , to all spots.

4 Send the list of spots along with the informativeness values of all paths to all spots to the supervisor.

5 Phase 2: Spot Allocation by Supervisor

6 wait until ranked list of slots recd. from all modules

7 **for** each spot s_j **do**

8 $winners \leftarrow \arg \max_{a_i \in \mathbb{A}} inf(P_{ij})$

9 **if** only one module a_i in *winners* **then**

10 $winner \leftarrow a_i$

11 **else**

12 // more than one winner module: multiple modules with same informativeness for s_j

13 $winner \leftarrow \arg \min_{a_i \in winners} cost(P_{ij});$

14 // ties are broken randomly

15 add (*winner*, s_j) to $f(\cdot)$;

16 remove *winner* from \mathbb{A} and remove s_j from V_T ;

17 Send set of spot allotments $f(\cdot)$ to every module a_i .

Algorithm 3: Entropic Potential Search (EPS) Algorithm

```

1 pathFormation()
   Input:  $B$ : Budgeted cost;  $c_{curr}$ : Current cell of the module and  $s_k$ : A node in the
           target configuration.
   Output:  $P_{ik} \in \Pi$ : Generated path for module  $r_i$ .
2  $OPEN \leftarrow c_{curr}$ .
3  $CLOSED \leftarrow \{\emptyset\}$ .
4 while  $OPEN$  is not empty do
5    $n_{max} \leftarrow \arg \max_{n \in OPEN} hu(n)$ 
6   for each neighbor  $n_n$  of  $n$  do
7     if  $n_n$  is in  $OPEN$  or  $CLOSED$  and  $g(n_n) \leq g(n_{max}) + cost(n_{max}, n_n)$ 
8       then
9         Continue with the next neighbor of  $n$ .
10       $g(n_n) \leftarrow g(n_{max}) + cost(n_{max}, n_n)$ 
11      if  $g(n_n) + h(n_n) \geq B$  then
12        Continue to the next successor of  $n_{max}$ 
13      if  $n_n = s_k$  then
14        return the best path to  $s_k \rightarrow P_{ik}$ 
15      if  $n_n \in OPEN$  then
16        Update the  $g(n_n)$  value of  $n_n$  in  $OPEN$ 
17      else
18        if  $H(n_n) > 0$  then
19          Insert  $n_n$  to  $OPEN$ 
20  Insert  $n_{max}$  to  $CLOSED$ 
21 return Null // no solution exists which has lower cost than  $B$ 

```

Acting Phase

In the acting phase, the modules move to their respective allocated spots in a sequential manner. No module is allowed to move until all the spots are allocated using the allocation phase. In the absence of a proper order of modules to occupy spots, deadlock situations might arise. For example, in Figure 3.4, if all the spots except S_1 are assumed first, then when the module which has selected the spot S_1 arrives, it will not be able to move to S_1 , unless other modules disconnect and make space for it to move. To avoid repeated connects and disconnects between modules, we allow the module which has selected the spot with the highest betweenness centrality measure in G_T [13], first to occupy its position (ties are

broken at random). Once it is in its proper position, it will broadcast a message to notify that it has concluded locomotion, to all other modules. Next the spots neighboring the center spot will be occupied by modules and so on. Techniques described in [38] can be used for locomotion of the modules.

Each module, a_i , maintains a list of its visited cells, $C_{V_i} \in \mathcal{C}$, while moving towards its goal position in the target configuration. In a GP, with a newly added set of visited cells, the estimated entropy of the unobserved cells gets updated as given by Equation 3.1. To incorporate this change and also to gain maximum information from the environment, modules need to update their paths, whenever possible. Modules update their initially calculated paths by following Algorithm 4. After visiting \mathcal{O} new cells, each module executes the EPS algorithm with its remaining budget.

Algorithm 4: Movement Strategy Of Modules

Input: B_r : Remaining budget; c_{curr} : Current cell of the module and $s_k \in V_T$: Goal position in the target configuration.

- 1 $\bar{P}_{ik} \subset P_{ik}$: Module a_i 's remaining path from c_{curr} to s_k .
- 2 Update the set of visited cells, C_{V_i} .
- 3 **if** module a_i has visited \mathcal{O} cells **then**
- 4 Execute *pathFormation*(B_r, c_{curr}, s_k) to find a new path, P_{ik}^* .
- 5 **if** $inf(P_{ik}^*) > inf(\bar{P}_{ik})$ and $cost(P_{ik}^*) \leq B_r$ **then**
- 6 Follow the new path P_{ik}^*
- 7 $\bar{P}_{ik} \leftarrow P_{ik}^*$
- 8 **else**
- 9 Follow initially generated path \bar{P}_{ik}
- 10 **if** module a_i has reached its goal position $s_k \in V_T$ **then**
- 11 Broadcast REACHED message

If a new path from the module's current cell to the goal position can be found while remaining within the budget constraint and improving the informativeness, then the module selects it to move towards its allocated spot. Otherwise it follows the earlier path \bar{P}_{ik} . Once a module reaches its goal position in the target configuration, it broadcasts a REACHED message to notify other modules. Modules are allowed to move exclusively in the order of the centrality of selected spots; ties are broken at random.

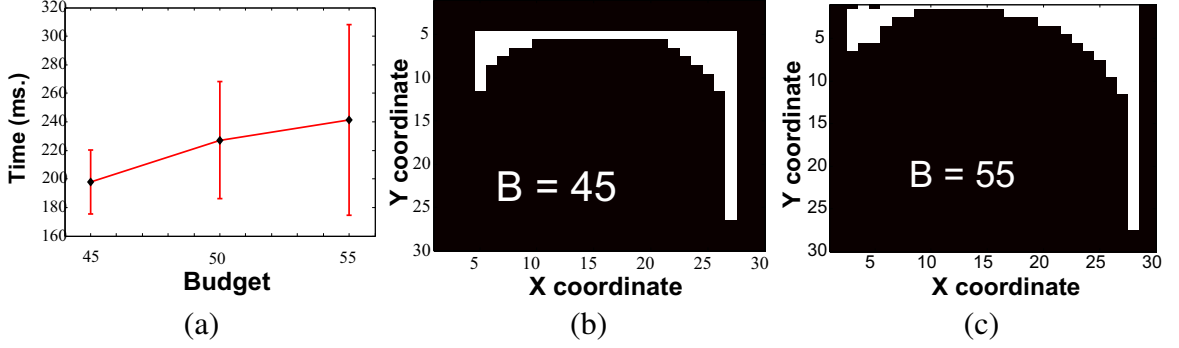


Figure 3.6: (a) Run times of EPS algorithm for different budgets; (b), (c) Nodes explored (shown in white color) by EPS algorithm, with $B = 45$ and 55 respectively.

3.3.2 Experimental Evaluation

Experimental Settings

We have implemented the algorithms in simulation on a desktop PC (Intel Core i5 -960 3.20GHz, 6GB DDR3 SDRAM). The environment is divided into a 30×30 , 4-connected grid structure. Each cell in the environment is represented by its centroid. The information value of each cell in the environment is drawn from $U[1, 10]$. We have tested instances where random target configurations, in forms of graphs, have been generated of sizes, $|V_T| = 10$ through 50 , inside the environment. Each node in the target configuration has between 1 to 4 neighbor nodes and each edge between two neighbor nodes has unit distance. In all the cases, $|\mathbb{A}| = |V_T|$. Each module is modeled to be a cube of size 1 unit \times 1 unit \times 1 unit; their initial cells are drawn uniformly from $\mathbb{U}[(0, 29), (0, 29)]$. Similar to [65], 40% of total cells and their corresponding ground truth data has been provided to the modules to learn the mean and covariance structure of GP through maximum likelihood estimation. Budget, B , has been set to 45 cells unless otherwise mentioned. We have used Manhattan Distance (\mathcal{MD}) for calculating cost of a path. Each singleton module runs the SA algorithm and then moves to its allocated or selected spot in G_T . Each test is run 5 times.

We have also compared the performance of the SA algorithm with an auction algorithm [10], which is a classical assignment algorithm. For implementing the auction algorithm,

each module is modeled as a bidder and each spot is modeled as an item, which modules are bidding for.

Experimental Results

First, we have tested the run times of the EPS algorithm for different budget amounts. For fixed start and goal locations, B is varied through $[45, 50, 55]$, where $\mathcal{MD}(\text{start}, \text{goal}) > B$. The result is shown in Figure 3.6.(a). We can see that with increasing amount of budget, the run time also increases, as the algorithm needs to search for more possible paths in the search space. Figure 3.6.(b) and (c) show the cells explored by the EPS algorithm for $B = 45$ and 55 respectively in a particular instance. We have observed that, with $B = 55$, on an average the EPS algorithm explored about 50% more cells in the environment than with $B = 45$, which also can be noticed in Figure 3.6.(b) and (c). Next, we compared the performances of the proposed SA algorithm and the auction algorithm. In terms of estimated information collection, both the allocation algorithms performed almost equally (Figure 3.7.(a)). In terms of total number of messages sent by the modules in the planning phase, the SA algorithm outperformed the auction algorithm (Figure 3.7.(b)). For 50 modules, using the auction algorithm, modules have sent about 10^4 times more messages. Figure 3.7.(c) shows that auction algorithm takes significantly higher time (with 50 modules, the auction algorithm takes 3 times more) than the proposed SA algorithm.

Next, we have varied the value of \mathcal{O} between $\frac{B}{2}$ and $\frac{B}{10}$ to evaluate the effect of frequency of path updates on the information gain and time taken to run the algorithm. This test has been performed with 1 module only. The result is shown in Figure 3.7.(d). We observe that although with increasing number of path updates, the module earned up to 88% extra information than estimated, the running time also increased considerably. For example, with $\mathcal{O} = \frac{B}{2}$, run time is 20 ms., whereas with $\mathcal{O} = \frac{B}{10}$, run time increased to 860 ms. In Figure 3.7.(e), we have shown how with acting phase completion, the percentage of total information collected by the modules changes. Finally, Figure 3.7.(f) shows an

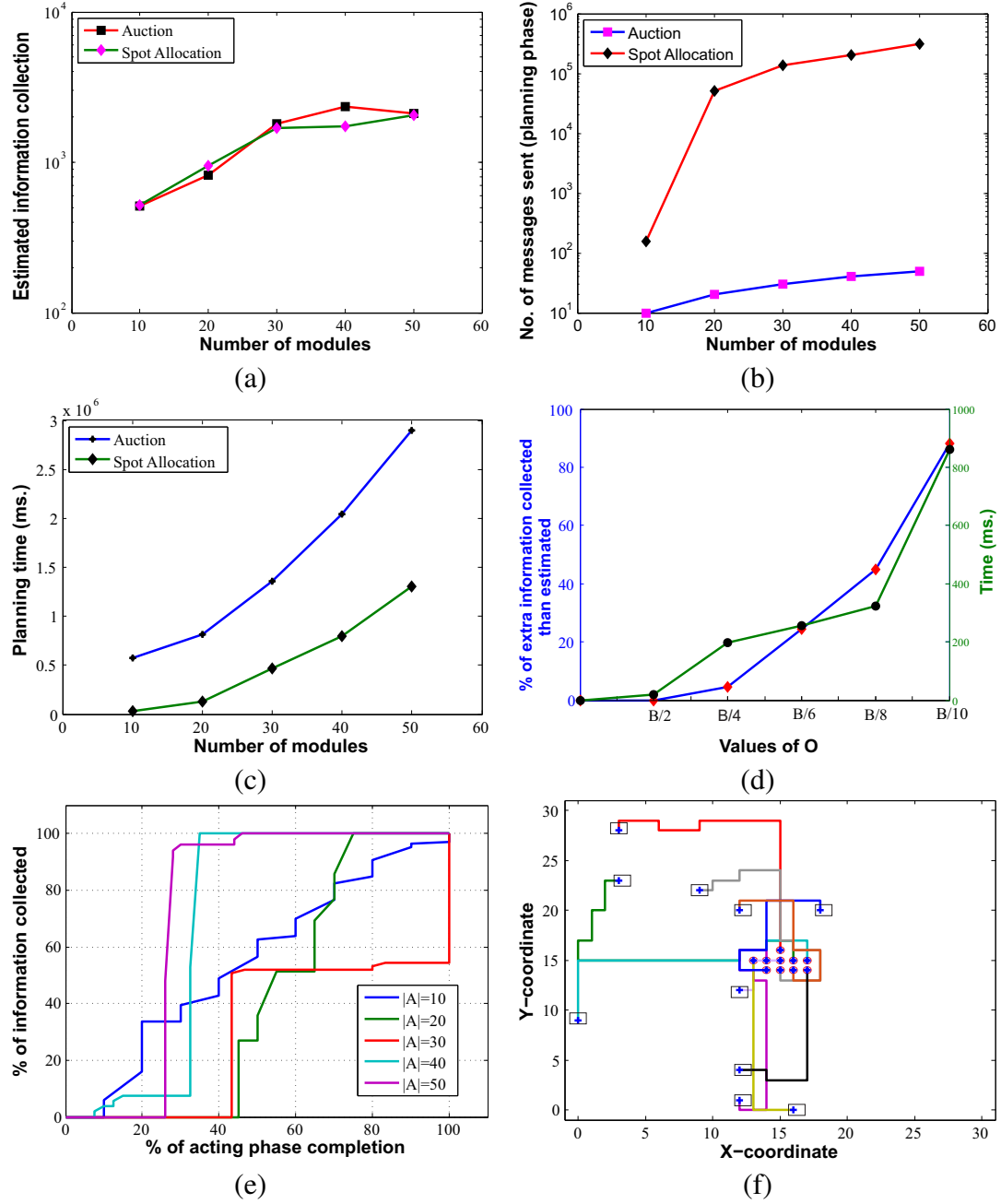


Figure 3.7: (a) Comparison of estimated information collection between SA and auction algorithm.; (b) Comparison of no. of messages sent in planning phase with auction algorithm; (c) Comparison of planning times between SA and auction algorithm; (d) Effect of changing values of O ; (e) Change in collected information over time; (f) Configuration formation by 10 modules: boxed + and circled * indicate the start and final locations respectively.

instance of the configuration formation procedure. In this experiment, 10 modules start from arbitrary locations in the environment (boxed + marked points) and form the target configuration, by following the maximally possible informative paths from their initial locations to the allocated goal spots in the target configuration (circled * marked points).

3.4 Distributed Configuration Formation From Initial Smaller Configurations

Unlike our previous configuration formation work, in this work, we assume that the modules can either be singletons or can be part of any arbitrary shaped configuration in the beginning. An illustration is shown in Figure 3.8.

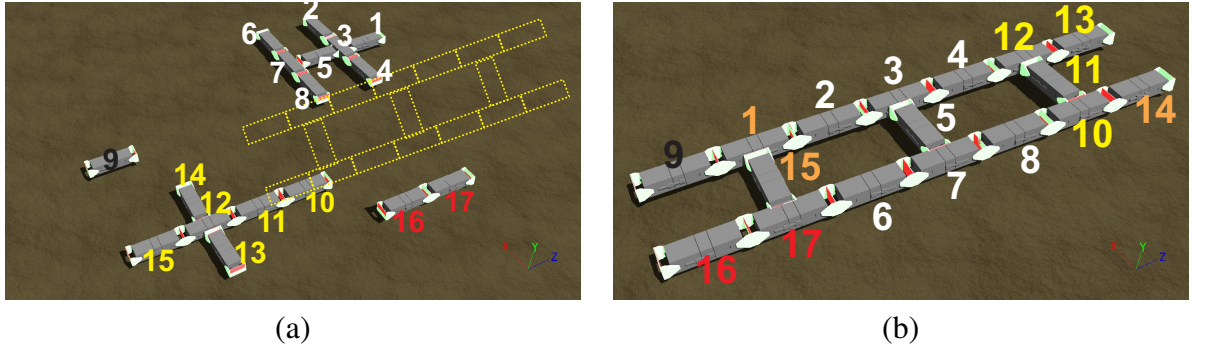


Figure 3.8: (a) A singleton and three initial configurations consisting of 2, 6 and 8 modules respectively, and desired target configuration (marked with yellow dotted lines) (b) target configuration involving all 17 modules connected in ladder configuration; module numbers marked in white, yellow and red are retained between initial and target configurations.

3.4.1 Notations

A configuration is a set of modules that are physically connected. A configuration is denoted as $A_i = \{a_1, a_2, \dots, a_j\} \subseteq \mathbb{A}$. The topology of configuration A_i is denoted as a graph, $G_{A_i} = (V_A, E_A)$, where $V_A = A_i$ and $E_A = \{e_{kj} = (a_k, a_j) : a_j \text{ and } a_k \text{ are}$

physically connected in A_i . Each configuration has a module that is identified as a leader [8] and the leader's pose is used to represent the configuration's pose.

Visual representations of the two graph structures used here have been shown in Figure 3.9. Even though we have modeled the initial and target configurations as graphs, for testing purposes, we have used only tree configurations. In the rest of the chapter, for the sake of legibility, we have slightly abused the notation by using T instead of G_T to denote the target configuration and S instead of V_T to denote the spots in the target configuration. Let $cost^{loc}()$ denote the locomotion cost from a_i^{pos} to s_j^{pos} , $cost^{dock}$ denote the cost of docking a_i with modules in neighboring spots of s_j and $cost^{undock}$ denote the un-docking costs of a_i from neighboring modules in A_i .

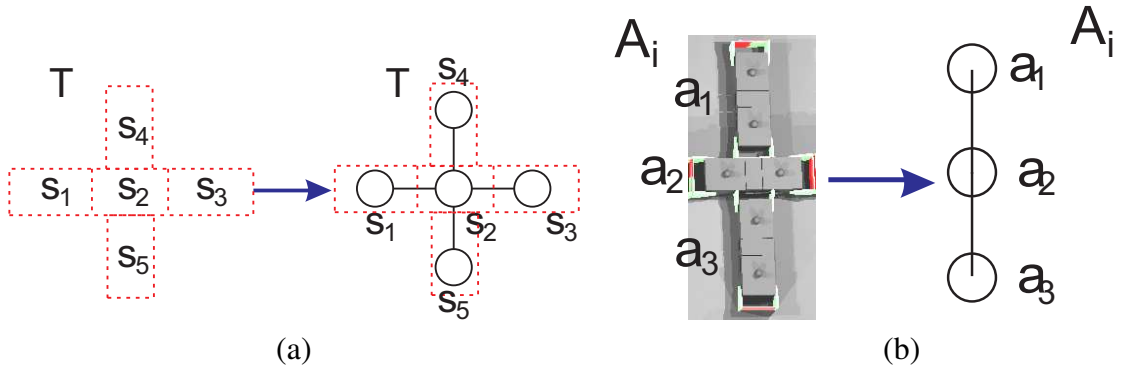


Figure 3.9: (a) Graph abstraction of T ; (b) Graph abstraction of A_i .

Problem Setup

To formulate the configuration formation problem as a utility maximization problem, we first represent the utility of a single module to occupy a single spot in the target configuration, and then extend that representation to a set of modules connected as a configuration to occupy a set of adjacent spots in the target configuration. A single module's utility for a spot is given by the value of the spot to the module minus the costs or energy expended by the module to occupy the spot. As reported in [56], the locomotion of an MSR is significantly affected by the locomotion of the module(s) in the MSR that

has more neighbors in the MSR's configuration. For example, for the configuration shown in Figure 3.8.(a), module 12's position at the center of the 6-module configuration is more critical than the other modules for locomotion as it has more neighbors. If module 12 becomes un-operational at any point of time, then four of its connected neighbor modules need to un-dock to get rid of module 12 and then reconnect again to continue working. On the other hand, if any of the terminal modules (e.g., 14) becomes un-operational, then that particular module can be detached from the MSR body with just one un-dock operation.

To capture this position dependency, we have used a concept from graph theory called the betweenness centrality [13] to denote the value of spot s_i , given by:

$$Val(s_i) = \sum_{s_i \neq s_j \neq s_k} \frac{\sigma_{s_j s_k}(s_i)}{\sigma_{s_j s_k}} \quad (3.2)$$

where $\sigma_{s_j s_k}$ is the total number of shortest paths between any pair of nodes s_j and s_k in G_T and $\sigma_{s_j s_k}(s_i)$ is the number of shortest paths between s_j and s_k which go through s_i .

The cost to a module a_i located at a_i^{pos} to occupy spot s_j at s_j^{pos} , is calculated as a sum of a_i 's locomotion costs to reach and occupy spot s_j , and any costs to undock and re-dock with neighboring modules before and after it occupies the spot [27]. This is denoted as the following:

$$cost_{a_i}(s_j) = cost^{loc}(a_i^{pos}, s_j^{pos}) + \sum_{a_k \in neigh(s_j)} cost^{dock}(a_i, a_k) + \sum_{a_{i'} \in neigh(a_i)} cost^{undock}(a_i, a_{i'}) \quad (3.3)$$

Note that energy requirements for locomotion of a module are generally higher than those for docking the module with another module as locomotion requires continuous power to all motors and much higher torques than docking; also, docking two modules requires aligning their docking ports first, which takes more energy than un-docking two modules.

When a set of modules is connected in configuration A_i , the cost of occupying a set of

spots $S_j \subseteq V_T$ in the target configuration is given by:

$$cost_{A_i}(S_j) = \sum_{s_l \in S_j, a_k \in A_i} cost_{a_k}(s_l) - f_{rwd}(|A_i|) \quad (3.4)$$

where $f_{rwd}(|A_i|) = \frac{|A_i|-2}{|\mathbb{A}|}$ is a reward function for retaining connections between modules in the existing configuration A_i while being allocated to the target configuration. Because $f_{rwd}(|A_i|)$ increases (and $cost_{A_i}()$ decreases) with the size of A_i , it is cost-wise better to break smaller configurations than to break larger configurations to fit into the target configuration. So, the reward function ensures that keeping the initial configuration intact in the target configuration, whenever possible, results in lower cost. Using the above formulation, it can easily be seen that when A_i can fit entirely into V_T (i.e., $S_j = V_T$), $cost_{A_i}(S_j) < \sum_{s_j \in S_j, a_i \in A_i} cost_{a_i}(s_j)$.

The utility of a spot to a module determines how profitable or beneficial that spot is for the module if it finally ends up occupying that spot. The utility of module a_i for spot s_j is given by

$$U_{a_i}(s_j) = Val(s_j) - cost_{a_i}(s_j) \quad (3.5)$$

Similar to the cost function described above, the utility for initial configuration A_i to occupy a set of spots $S_j \subseteq V_T$ is given by the sum of the utilities of the individual modules comprising A_i to occupy spots in S_j ,

$$U_{A_i}(S_j) = \sum_{s_l \in S_j} Val(s_l) - cost_{A_i}(S_j) \quad (3.6)$$

Using the above formulation, the spot allocation problem has to assign modules to spots so that each module is allocated to the most eligible (highest utility earning) spot and no two modules are assigned to the same spot.

Formally, we can define the objective function as follows: Given a set of modules \mathbb{A} in a set of initial configurations, and a set of spots S representing the target configuration, find

a suitable allocation $P^* : \mathbb{A} \rightarrow S$ such that

$$P^* = \arg \max_{\forall P} \left(\sum_{a_i \in \mathbb{A}, s_j \in S} U_{a_i}(s_j) + \sum_{A_i \subseteq \mathbb{A}, S_j \subseteq S} U_{A_i}(S_j); \right. \\ \left. \forall a_k \neq a_i, \quad P^*(a_i) \neq P^*(a_k). \right. \quad (3.7)$$

Note that, if two modules a_i and a_k both have the same highest utility for spot s_j , then only one of them can be allocated to and occupy s_j . In the next section, we describe our spot selection algorithm that provides a suitable allocation of modules to spots for the above utility maximization problem.

3.4.2 Algorithms for Configuration Formation

We divide the problem into two phases - a *planning phase*, where modules select spots in the target configuration, and an *acting phase*, where modules move to their selected spots and connect with other modules.

Planning Phase

In the beginning of the planning phase, all the modules broadcast their positions and orientations. After having this information, each module calculates the location corresponding to the center target configuration T in the environment, as the mean of all spots' positions. However, a specific desired location can also be given as an input to the modules by the user.³ Singleton modules then rank themselves according to their distances from the center of T ; the rank of a configuration is calculated using the distance of the configuration's leader from the center of T . Singletons and configurations select spots in T based on their rank. Because $cost^{loc}$ has the most significant contribution to the cost function, the distance-based rank ensures that modules and configurations with lower costs

³A common coordinate system can be maintained by modules for localizing themselves following the model described in [92].

(higher utilities) get to select spots in T first. We describe the spot selection techniques in the planning phase in two parts - spot selection by singleton modules and spot selection by configurations.

Algorithm 5: Spot Allocation Algorithm for Singleton Modules.

```

1 procedure: spotAllocation()
   Input:  $S$ : set of spots,  $\bar{S}$ : set of (spot, selector) pairs;  $a_{curr}$ : module currently
       selecting spot.
2  $S_{sort} \leftarrow$  Sort  $S$  in descending order of utility of spots
3 for each  $s_j \in S_{sort}$  do
4    $\mathcal{D} \leftarrow 0$ ;
5   if ( $s_j$  is not selected by another module)  $\vee$  ( $(s_j$  is selected by module
        $a_{block} \notin A_i \subseteq \mathbb{A}) \wedge (evict(a_{curr}, a_{block}, \mathcal{D}) = TRUE)$ ) then
6     Select spot  $s_j$  for  $a_{curr}$ ;
7     Broadcast updated set of spot-selector pairs  $\bar{S}$ ;
8     return;
9 Broadcast NO_SPOT_FOUND message;
```

Algorithm 6: Eviction algorithm used by modules to select alternate spots.

```

1 procedure: evict( $a_{curr}, a_{block}, \mathcal{D}$ )
   Input:  $a_{curr}$ : module currently selecting spot  $s_{curr}$ ;  $a_{block}$ : the module which has
       already selected  $a_{curr}$ 's best spot  $s_{curr}$ ;  $\mathcal{D}$ : current recursion depth.
2 if  $\mathcal{D} < \mathcal{D}_{max}$  then
3    $s_{block} \leftarrow \arg \max_{s_i \in S \setminus s_{curr}} U_{a_{block}}(s_i)$ ;
4    $s_{curr'} \leftarrow \arg \max_{s_i \in S \setminus s_{curr}} U_{a_{curr}}(s_i)$ ;
5   if ( $U_{a_{curr}}(s_{curr}) + U_{a_{block}}(s_{block}) > U_{a_{curr}}(s_{curr'}) + U_{a_{block}}(s_{curr})$ ) then
6     if  $s_{block}$  is not selected by any module then
7       return TRUE;
8     else
9        $//a'_{block} \notin A_i \subseteq \mathbb{A}$  is the module occupying  $s_{block}$ 
10      return evict( $a_{block}, a'_{block}, \mathcal{D} + 1$ );
11 return FALSE;
```

Spot Selection by Singleton Modules

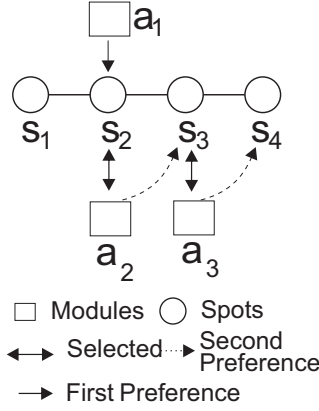
A singleton module a_{curr} selects a spot to occupy using Algorithm 5. a_{curr} first sorts the spots in order of its expected utility $U_{a_{curr}}(s_j), \forall s_j \in S$. If a spot s_j has not already been selected by another module, or, if it has been selected by another singleton module (module that is not part of a configuration) that can be evicted using the *evict* method, then a_{curr} selects s_j and broadcasts the updated spot-selector pairs to all other modules. If a_{curr} cannot evict the module currently occupying its highest utility spot, then it successively reattempts spot selection using the spots for which it has the next highest utilities. If none of the spots in S can be selected by a_{curr} , it broadcasts a NO_SPOT_FOUND message to all other modules.

Eviction Strategy: The *evict method* is used by module a_{curr} to cancel the selection of spot s_{curr} done previously by another singleton module a_{block} . Note that eviction can be done only for a singleton module, and not for modules that are part of configurations, as breaking existing configurations will incur additional time as well as costs for docking and un-docking modules. The method first checks the expected combined utility between a_{curr} and a_{block} for selecting their most (conflicting) and second-most preferred spots. If this combined utility is greater when a_{curr} selects s_{curr} and a_{block} selects its next highest utility spot that it can occupy, then a_{curr} evicts the selection of s_{curr} by a_{block} , as shown in the *evict()* method in Algorithm 6. To limit excessively long cycles of eviction, we have allowed at most \mathcal{D}_{max} successive evictions. An illustration of the eviction process with $\mathcal{D}_{max} = 3$ is shown in Figure 3.10.

Block Allocation by Modules Connected in a Configuration

Preliminaries: Following are some definitions which will be needed in explaining our proposed approach.

Definition 2 *Graph Isomorphism* [75]: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are



1. a_1 finds s_2 to be its highest utility spot,
2. a_1 checks if it can evict a_2 occupying s_2 ,
3. a_2 then checks its next highest utility spot, s_3 ,
4. a_2 checks if it can evict a_3 , occupying s_3 ,
5. a_3 checks its next highest utility spot s_4 ,
6.
 - if s_4 is free, a_3 selects s_4 , a_2 selects s_3 , a_1 selects s_2 ; return TRUE.
 - if s_4 is not free, $\mathcal{D}_{max} = 3$ is reached; return FALSE.

Figure 3.10: Illustration of eviction algorithm for 3 modules with $\mathcal{D}_{max} = 3$.

isomorphic if there is a one-to-one mapping between the nodes and edges in G_1 and G_2 .

Formally, this bijection relationship exists - $f : V_1 \rightarrow V_2$.

Loosely speaking, if two graphs are isomorphic, then they will have same number of nodes and if any two nodes in one graph are adjacent, then those nodes will be adjacent in the other graph as well. Graph isomorphism is one of those problems which are neither solvable in polynomial time nor can they be proved to be NP-complete; rather they belong to an ‘intermediate’ class [59]. Unfortunately, from an algorithmic point of view, even if a problem cannot be proved to be NP-complete, being outside of the P-class makes it difficult to solve anyway (in the worst-case scenario). Even though graph isomorphism is a well-known notorious problem to solve [59], there are efficient linear time algorithms available for tree isomorphism [2].

Definition 3 *Subgraph Isomorphism [94]: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are subgraph isomorphic if any subgraph G'_1 of G_1 ($G'_1 \subseteq G_1$) is isomorphic to G_2 .*

Usually, in case the of subgraph isomorphism, one graph is larger in size than the other and the problem becomes to find a subgraph of the larger graph which is isomorphic to the smaller graph. As can be understood, there can be multiple isomorphic subgraphs available

Algorithm 7: Block Allocation Algorithm that a set of modules connected in configuration A_{curr} uses to select a set of maximally adjacent spots in the target configuration.

```

1 blockAllocation( $A_{curr}, \bar{S}$ )
   Input:  $\bar{S}$ : Set of (spot, selector) pairs;  $A_{curr}$ : Set of modules connected together as a
           configuration and currently selecting spots.
2  $T_{sub} \leftarrow$  Set of all subgraphs of  $T$ , which are isomorphic to  $A_{curr}$ .
3 if  $T_{sub} == \{\emptyset\}$  then
4    $T_{sub} \leftarrow$  Set of all maximum common isomorphic subgraphs of  $T$  and  $A_{curr}$ .
5 for each  $t_k \in T_{sub}$  in descending order of utility  $U_{A_i}(t_k)$  do
6   if No spot in  $t_k$  has been selected yet then
7     Select  $t_k$ ;
8     Broadcast updated set of spot-selector pairs  $\bar{S}$ ;
9   else
10     $S_{block} \leftarrow$  set of spots  $\in t_k$  already selected by  $\{a_{block}\} \subseteq \mathbb{A} \setminus A_{curr}$ 
11     $s^i \leftarrow$  spot matched to  $a_i \in A_{curr}$  but already selected by  $a_{block} \in \mathbb{A} \setminus A_{curr}$ 
12    if  $\text{evict}(a_i, a_{block}) = \text{TRUE}$  for every  $s^i \in S_{block}$  then
13      Select  $t_k$ ;
14      Broadcast updated set of spot-selector pairs  $\bar{S}$ ;
15    else
16      if all  $t_k \in T_{sub}$  have been checked then
17        for each  $a_i \in A_{curr}$  where  $\text{evict}(a_i, a_{block}) = \text{FALSE}$  and  $s^i \in t_k$  do
18          Disconnect  $a_i$  from  $A_{curr}$ 
19           $A_{curr} \leftarrow A_{curr} \setminus a_i$ ;
20          spotAllocation( $a_i, \bar{S}$ );
21          Broadcast updated set of spot-selector pairs  $\bar{S}$ ;
22      if selected  $t_k$  is MCS of  $A_{curr}$  then
23        for every  $a_i \in A_{curr}$ , where  $s^i \notin t_k$  do
24          Disconnect  $a_i$  from  $A_{curr}$ 
25           $A_{curr} \leftarrow A_{curr} \setminus a_i$ ;
26          spotAllocation( $a_i, \bar{S}$ )
27        Broadcast updated set of spot-selector pairs  $\bar{S}$ ;
28
```

in the smaller graph. This problem is a well-known NP-complete problem [22]. However, there are approximation algorithms proposed in the literature which solve the problem in polynomial-time for certain graph structures like trees [83]. We are also interested in the isomorphic subgraphs which are also maximum in size, which leads us to our next

definition.

Definition 4 *Maximum Common Subgraph (MCS):* Given $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a MCS is a subgraph consisting of the largest number of edges isomorphic to both G_1 and G_2 .

The problem of finding a MCS between two graphs is a combinatorially intractable NP-complete problem [75] for which no algorithm of polynomial-time complexity exists for the general case. For finding all possible MCSs having k nodes in a pair of graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the total number of comparisons we have to do is:

$$\frac{V_1!V_2!}{(V_1 - k)!(V_2 - k)!k!} \quad (3.8)$$

As can be seen in this equation, even with small values of V_1 , V_2 and k , computational comparisons can reach an astronomical value. Although it is a computationally difficult problem to solve in graphs, we should mention that polynomial-time approximation algorithms for finding maximum common subtrees can be found in the literature [3], [76].

As discussed earlier, our main objective is to place the initial configurations (A_i) into the target configuration (T) with least number of disconnections between the modules present in A_i . We have modeled A_i and T as graphs. Therefore if G_{A_i} is an isomorphic subgraph of T , then A_i can readily be allocated to T (provided the spots are free). On the other hand, if G_{A_i} is not an isomorphic subgraph of T , then we look for a MCS so that we can preserve most of the connections in A_i while allocating it to T while the rest of the modules in A_i which are not part of that MCS are detached from it.

Algorithm Description: The technique used by configuration A_{curr} to select a set of connected spots in the target configuration T is given by the *blockAllocation* procedure shown in Algorithm 7. The algorithm is executed on l_{curr} , the leader of configuration A_{curr} , selected using techniques in [8].

To place A_{curr} into T without breaking the connections between its modules, we have to find if T , or a subgraph of T , is isomorphic to A_{curr} . An example of this problem is shown in Figure 3.11(a) that shows all possible subgraphs of T which are isomorphic to the configuration A_i using different colors. This problem requires finding the isomorphic subgraphs (IS) [22] of T . However, if A_{curr} is not isomorphic to T or a subgraph of T , then A_{curr} cannot be placed into T without breaking its connections and, thus, changing its shape. In such a scenario, our objective is to reduce the number of connections that are removed between A_{curr} 's modules. For this, we have to find the maximum number of modules in A_{curr} , which can be placed directly into T , without first disconnecting them. An example is shown in Figure 3.11(b), where the red dotted boxes indicate the maximum common subgraphs of T and A_i , which are isomorphic.

This problem is an instance of the *maximum common subgraph (MCS) isomorphism* problem as discussed earlier [75], where, given two graphs T and A_{curr} , the goal is to find the largest subgraph which is isomorphic both to a subgraph of T and A_{curr} . If $|V_{A_{curr}}| > |V_T|$, then we find the maximum size subgraph of T which is isomorphic to $A'_{curr} \subseteq A_{curr}$ and allocate the spots to matched modules, using a similar technique as in the *blockAllocation* algorithm. On the other hand, if $|V_T| = |V_{A_{curr}}|$ and $G_T, G_{A_{curr}}$ are isomorphic, then A_{curr} can be allocated to T ; otherwise, we find the MCS between A_{curr} and T which can be readily allocated to T while the rest of A_{curr} can be allocated following the proposed *blockAllocation* algorithm.

Our algorithm first finds subgraphs of G_T that are isomorphic to G_A . If there are no isomorphic subgraphs, it checks for maximal common isomorphic subgraphs. These subgraphs are stored in set T_{sub} (lines 2 – 4). As modules want to maximize the utility earned from the allocation, the subgraphs t_k within T_{sub} are ordered by utility to A_{curr} . The algorithm then inspects each subgraph t_k . If all the spots in t_k are free, then t_k is selected by A_{curr} and l_{curr} broadcasts a message to notify every module in \mathbb{A} about this selection (lines 6 – 9). On the other hand, if any spot $s^i \in t_k$ is already selected by a singleton a_{block} , A_{curr}

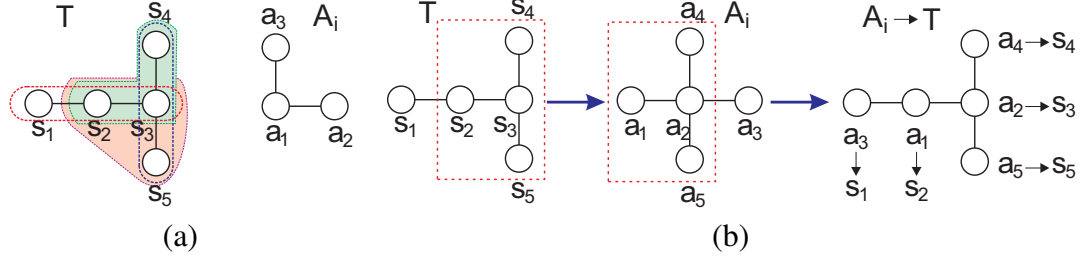


Figure 3.11: (a) A scenario where the colored subgraphs of T are isomorphic to A_i , (b) A scenario where a subgraph of t is isomorphic to a subgraph of A_i . The red dotted box shows the maximal common subgraph between T and A_i ; the unmatched module a_3 is detached from A_i and allocated to spot s_1 by our block selection algorithm.

checks to see if it can evict a_{block} using the *evict()* method. If eviction is successful, t_k is selected for A_{curr} and the updated set of spot-selector pairs are broadcast to all modules in \mathbb{A} (lines 11 – 15). If eviction is not successful, it means that some modules in A_{curr} could not occupy some spots in the target configuration (or its subgraph) as some other modules that did not belong to configuration A_{curr} had already selected those spots. In this case, the modules of A_{curr} that could not find a spot in t_k will be disconnected from A_{curr} . Single spot selection algorithm is then used to select other spots in t_k for these modules (lines 17 – 21).

Finally, because selection of t_k by a configuration A_{curr} is done by means of matching modules of A_{curr} to unique spots in t_k , if t_k is an MCS of A_{curr} (i.e., $|V_{t_k}| < |V_{A_{curr}}|$), then some of the modules in A_{curr} will not be matched to any spot in t_k . Those unmatched modules will disconnect from A_{curr} , become singletons and will execute the *spotAllocation()* algorithm, in the order of their distances from the center of T , to get allocated to a spot (lines 22–24). Note that all other modules in A_{curr} whose matched spots in t_k were free to occupy, will occupy the matched spots while retaining their configuration. The updated set of spot-selector pairs are broadcast to all modules.

Algorithm 8: Movement strategy for the modules to assume appropriate spots in T .

```

1 procedure: MoveToSpots()
  Input:
     $S_{sort} \leftarrow$  Sorted  $S$  in descending order of betweenness centrality values.
     $a_c \leftarrow$  The module which is allocated to the central spot (highest betweenness
    centrality).
     $s_c \leftarrow$  The central spot.
     $A' \leftarrow$  Set of modules which have already assumed their allocated spots.
     $\bar{A} \leftarrow$  Set of modules which will take neighboring spots of the modules in  $A'$ .
2 if  $a_c$  is a singleton then
3   | Move to  $s_c$ .
4   |  $A' \leftarrow A' \cup a_c$ .
5 else
6   |  $//a_c \in A_i$ 
7   |  $A_i$  moves to  $T$  and therefore  $a_c$  is allocated to  $s_c$ .
8   |  $A' \leftarrow A' \cup A_i$ .
9 while configuration is not completely formed do
10  | for each  $a' \in A'$  do
11  |   | Notify other modules of the spot that  $a'$  has assumed.
12  |   | Update  $\bar{A}$ .
13  |   | Clear  $A'$  ( $\leftarrow \emptyset$ ).
14  |   | for each  $\bar{a} \in \bar{A}$  do
15  |   |   | Move to  $T$  and assume allocated spots.
16  |   | Update  $A'$ .

```

Acting Phase

After the planning phase is finished and all the spots in the target configuration have been selected by modules, the modules have to move to their respective selected spots. Note that no module moves until all the spots are selected. If there is no proper order of modules for assuming spots, then a deadlock situation might arise. For example, in Figure 3.8(b), if all the modules occupy their spots before module 5 does, assuming module 5 is a singleton, then it will be difficult for module 5 to occupy its spot properly, unless other modules give it space for moving. But then they will have to align themselves again, which is a difficult task. To avoid this, the module which has selected the spot with highest betweenness centrality value (or, central spot), will move first and assume its position. Once it is in

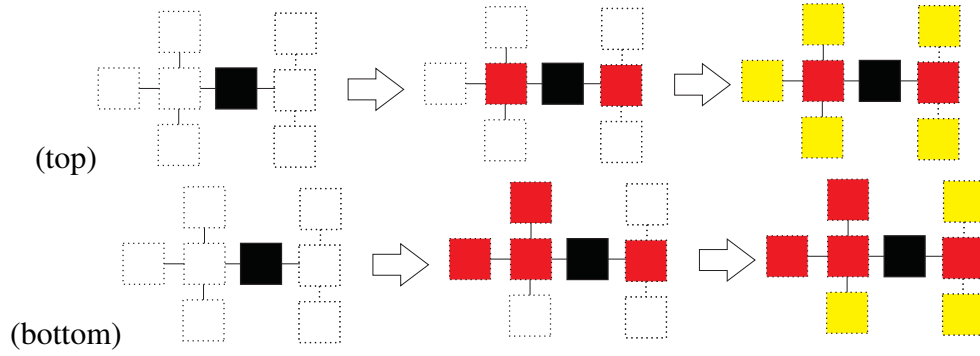


Figure 3.12: Illustration of acting phase: Dotted boxes represent the spots in T . First, the spot with the maximum betweenness centrality gets allocated (black spot). Next its neighbors get allocated (red spots) and finally neighbors of red spots get allocated (yellow boxes). (top) all modules are singletons; (bottom) red modules on the left side were part of an initial configuration. Therefore they occupied the spots at the same time even though two of the extreme red modules were not immediate neighbors of the central black module.

its proper position, it will broadcast a message to notify this to all other modules. Next the spots neighboring the center spot will be filled and so on. The procedure is shown in Algorithm 8.

If some spot s_i is allocated to a module which is part of an initial configuration A_j , then the whole configuration moves together to assume the allocated spots. As the initial configuration is a connected graph, therefore s_i 's neighbors and their neighbors will get filled up by this. Next, the spots adjacent to A_j 's allocated set of spots and the empty spots which are closer to s_i which did not get assumed because of A_j 's allocation will be assumed. Similar inside-out growth approaches have been proved to be very effective in mitigating the challenges like hole covering, deadlock avoidance etc. in swarm robotic self-assembly [79][95][31] and also in our earlier work of configuration formation in modular robots from singleton modules [32]. Techniques described in [38] can then be used for locomotion of the modules.

Theoretical Analysis of Algorithms

In this section, we provide the theoretical analysis of our proposed algorithms for singleton and initial configuration allocation.

Theorem 1 *spotAllocation and blockAllocation algorithms are complete when sufficient numbers of modules are available to form desired target configurations.*

Proof: We prove the completeness of the algorithms by showing that there is no empty spot or hole in the target configuration when the number of modules is at least equal to the number of spots in the target configuration T , i.e., when $|\mathbb{A}| \geq |S|$. A hole exists in T if there is a spot s_h that is not occupied by any module. This can happen because of two conditions: 1) No module has selected s_h , or, 2) module a_h , which selected s_h , could not reach its spot because another module blocked the path to its selected spot by occupying a spot that was further from the center of T than the selected spot. We show that these two conditions cannot arise. If $|\mathbb{A}| \geq |S|$, then because of the recursive approach in the *evict* method of Algorithm 5, each module will try to select a spot in T , as long as there are available spots. This guarantees that condition 1 never arises as at least one module a_h will select s_h . Condition 2 will never arise because, as described in Section 3.4.2, modules' priority to move is based on the betweenness centrality of their selected spots, and spots nearer to the center of T are occupied first, followed by outer ones. In other words, no module will occupy an outer spot before its neighboring spot, that is nearer to the center of the target configuration, gets occupied. Consequently, T cannot have a hole. Hence proved.

Lemma 3 *Any module a_i allocated to any spot s_j before the *evict()* method will still be allocated to some spot s_k after the execution of the *evict()* method even if $s_k \neq s_j$.*

Proof: We prove this by contradiction. Let us assume that as a result of the *evict()* method, a_i will be allocated to a null spot, s_k , i.e., $s_k = NULL$. But according to our proposed eviction strategy, if a_i does not get a spot to be allocated to, then the module which is

trying to evict it will not be able to do that and as a result, a_i will still be allocated to its spot $s_j \neq NULL$. Therefore, the *evict()* method will not reduce the number of modules that are already allocated to some spots.

Corollary 1 *The number of modules allocated to unique spots in the target configuration increases monotonically over time.*

Lemma 4 *Eviction of module is eligible iff the total utility earned by the modules increases.*

Proof: We prove this by contradiction. Let's assume that module a_i evicts another module a_j from spot s_k and then a_j selects its next highest utility spot s_l . If U^* and U' denote the total utility earned by all the modules with and without this eviction and $s_{k'}$ denote a_i 's next highest utility spot, then we assume $U^* < U'$. For the sake of simplicity, let's assume that s_l was not selected by any other module before and no other modules are executing the *evict()* function. So, $U^* = U_{a_i}(s_k) + U_{a_j}(s_l) + U^{rest}$ and $U' = U_{a_i}(s_{k'}) + U_{a_j}(s_k) + U^{rest}$. From algorithm 6, we can guarantee that eviction is possible iff $U_{a_i}(s_k) + U_{a_j}(s_l) > U_{a_i}(s_{k'}) + U_{a_j}(s_k)$ and therefore $U^* > U'$. Hence our initial assumption was incorrect and it's proved that the *evict()* function maximizes the total utility.

Theorem 2 *spotAllocation algorithm returns a Pareto-optimal allocation between modules and spots, i.e., any module's earned utility cannot be improved without making another module's utility worse.*

Proof: Let $s_{i,k}$ denote the k -th highest utility spot for module a_i . Because each module orders the spots based on utilities, it follows that $U_{a_i}(s_{i,k}) > U_{a_i}(s_{i,k+1})$. Consider two modules a_i and a_j that have the highest utility for the same spot s (i.e., $s_{i,1} = s_{j,1} = s'$, but $U_{a_i}(s') > U_{a_j}(s')$). Also, assume that a_j has selected spot s' first. Now, if the *spotAllocation* allocates a_i to its next best spot, $s_{i,2}$ and a_j remains at s' , then the total utility is $U^1 = U_{a_i}(s_{i,2}) + U_{a_j}(s')$. On the other hand, if the *spotAllocation* method evicts

a_j from s' and allocates it to its next best spot $s_{j,2}$ (assuming it is free), then the total utility becomes $U^2 = U_{a_i}(s') + U_{a_j}(s_{j,2})$. From Algorithm 5, if eviction is possible, then $U^2 > U^1$. On the other hand, if eviction does not happen, then it implies $U^1 > U^2$. For any other allocation strategy that does not do eviction even if $U^2 > U^1$, then the total utility earned by the alternate allocation strategy is always less than the utility earned by the *spotAllocation* algorithm. From the above equations, we can conclude that, if any two modules a_i and a_j have same ranking for a particular spot, s' , then one of the modules will be allocated to that spot and the other will be pushed to its next highest utility spot, i.e., its earned utility reduces, and no other allocation would increase their utilities as well as the overall utility. Hence the allocation strategy is Pareto-optimal.

Lemma 5 *Both spotAllocation and blockAllocation algorithms are deterministic in nature, i.e., no two modules will be allocated to the same spot as a result of our proposed strategy.*

Proof: We divide the proof into two following scenarios.

Case I

Let us assume that a singleton a_i selects a spot s_j which is already allocated to another singleton module a_k and also a_k 's allocation does not change due to this, i.e., both a_i and a_k are now allocated to s_j . But according to Algorithm 5, a_i will first try to evict a_k from s_j and then it can be allocated. If a_k cannot be evicted, then a_i will not select s_j . Also, following Lemma 3, we can guarantee that if a_k is evicted, then it will be allocated to some spot $s_l \neq s_j$. On the other hand, if a_k is a member of an initial configuration, then a_i cannot evict it anyway; rather it will look for the next best available spot. Therefore it is not possible that both a_i and a_k will be allocated to the same spot s_j .

Case II

If $a_i \in A_m$, and a_k is a singleton module, then a_i has permission to evict a_k if all other required conditions are satisfied. Following the similar logic as before, we can guarantee that if a_k is evicted by $a_i \in A_m$, then a_k will be allocated to some spot $s_l \neq s_j$. If a_k

cannot be evicted, then a_i will look for different spot (different isomorphic subgraph or as a singleton module if detached). A similar thing will happen if $a_k \in A_l$. Therefore, we can guarantee that if a_i is part of an initial configuration A_m , it will not be allocated to the same spot s_j with a_k .

Hence proved.

Theorem 3 *As \mathcal{D}_{max} approaches $|S|$, the total utility earned by the modules (U) approaches the optimal utility U^* .*

Proof: If there is no conflict among the modules about their best spots, i.e., each module's highest utility spot is unique, then the *spotAllocation* algorithm allocates highest utility spots to all the modules and thus achieves the optimal utility. But if there is a conflict among modules for the same spots, then the eviction method is invoked. From Algorithm 5, we can conclude that the total utility earned by the modules increases by successively calling the evict method. For $\lim_{\mathcal{D}_{max} \rightarrow |S|}$, any subsequent evictions will consequently increase the total utility. If eviction fails, then that means the total utility cannot be improved any further. Thus, every time the eviction method is invoked it will increase the total utility, going towards the optimal utility.

Theorem 4 *The proposed configuration formation process converges with time.*

Proof: Following Theorem 1, Lemma 3, and Corollary 1, we can guarantee that the configuration formation process will converge over time.

Note on complexity. The *spotAllocation* algorithm (Algo. 5) has a time complexity given by $O(|S|^{\mathcal{D}_{max}})$ where $|S|$ is the number of spots in the target configuration and \mathcal{D}_{max} is the depth up to which the eviction of modules is allowed. In the *blockAllocation* algorithm (Algo. 7), target configurations are considered to be trees and finding all possible isomorphic subtrees in the target configuration has a polynomial worst case time complexity of $O((|S||A_i|)^{d+1})$ [22], where $|A_i|$ and $|S|$ are the number of modules and

spots in initial and target configurations respectively, and d is the maximum branch factor of either configuration.

3.4.3 Experimental Evaluation

Settings

We have implemented the spot allocation algorithm on a desktop PC (Intel Core i5 -960 3.20GHz, 6GB DDR3 SDRAM). We tested instances where random numbers of singletons and initial configurations with sizes between 2 and 10 modules need to be allocated to target configurations with between 10 and 100 spots. In all cases, unless otherwise mentioned, the total number of modules in the environment is equal to the total number of spots in the target configuration. Each module is modeled as a cube of size 1 unit \times 1 unit \times 1 unit. The modules are placed at random locations within a 16 unit \times 16 unit environment, their initial orientations are drawn from a uniform distribution in $\mathbb{U}[0, \pi]$, and the initial positions of singletons and leaders of the initial configurations are drawn uniformly from $\mathbb{U}[(0, 15), (0, 15)]$. For all the tests, \mathcal{D}_{max} has been set to 3. Changing the value of \mathcal{D}_{max} from 3 to 10 affected the algorithm’s performance (both time and quality wise) negligibly; therefore this is not included in the results.

Extracting ‘better’ isomorphic subgraphs: Initial and target configurations were restricted to be trees based on the connections the modules in our MSR platform are capable of, although our algorithms can be applied for any other kinds of graphs as well. As there can be numerous subtrees present in the target configuration, which are isomorphic to the initial configuration and finding all possible isomorphic subtrees can take considerable time, we set an upper bound, MAX , on the number of isomorphic subtrees that the *blockAllocation* algorithm (Algo 7) will check. MAX is set to 20; different values of $MAX = 10, 30$, or 40 did not change the performance of the algorithm. To get higher utility isomorphic subtrees, first the nodes in the target configuration are sorted in descending order of betweenness centrality values, because if the costs to occupy

two different spots are the same, then higher betweenness centrality (spot value) indicates higher utility of the spot. For every node in the sorted list of spots, every node in the current configuration A_i is made the root of A_i once and checked for subtree isomorphism with target configuration T while making each node in T the root once, for every possible tree in A_i . The checking of isomorphic subtrees between A_i and T is stopped as soon as the first MAX isomorphic subtrees are found. All results are averaged over 50 runs.

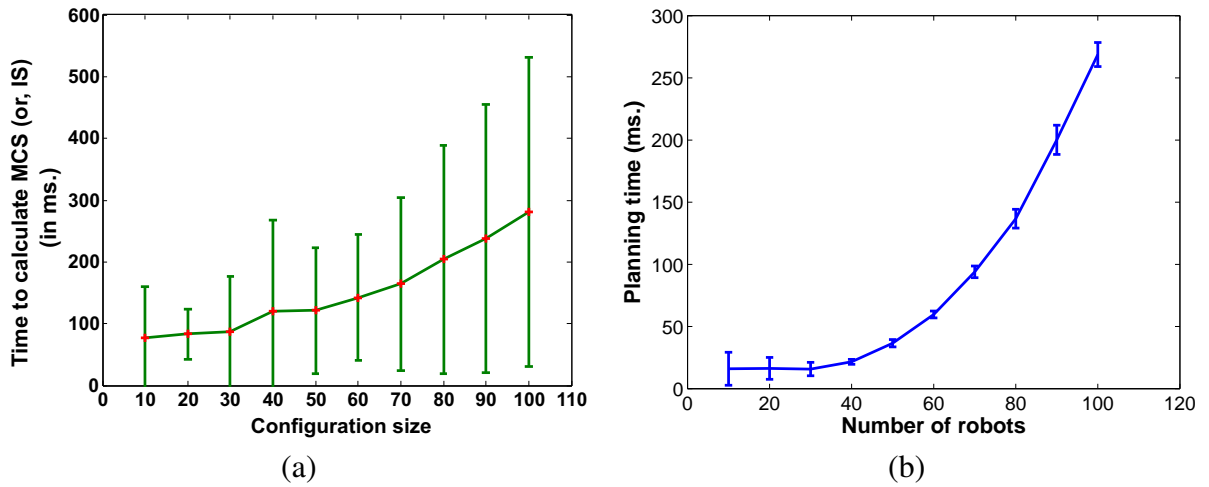


Figure 3.13: (a) Time to calculate MCS or IS vs. different initial configuration sizes, (b) Total planning time for different number of modules in environment.

Results

Performance Analysis of Our Approach: First we have shown how much time it takes to find MAX number of MCS (or, IS). The result is shown in Figure 3.13(a). The x -axis denotes the size of a single configuration and the y -axis denotes the time in milliseconds to find MAX number of MCS (or, IS) of that configuration in the target configuration. For this test, total spots in the target configuration have been set to 100. Though the run time increases with the size of the initial configuration, which can be expected because of the complexity results shown in [83] for finding isomorphic subtrees, still it was always well within a reasonable bound. In the next set of experiments, we have focused on

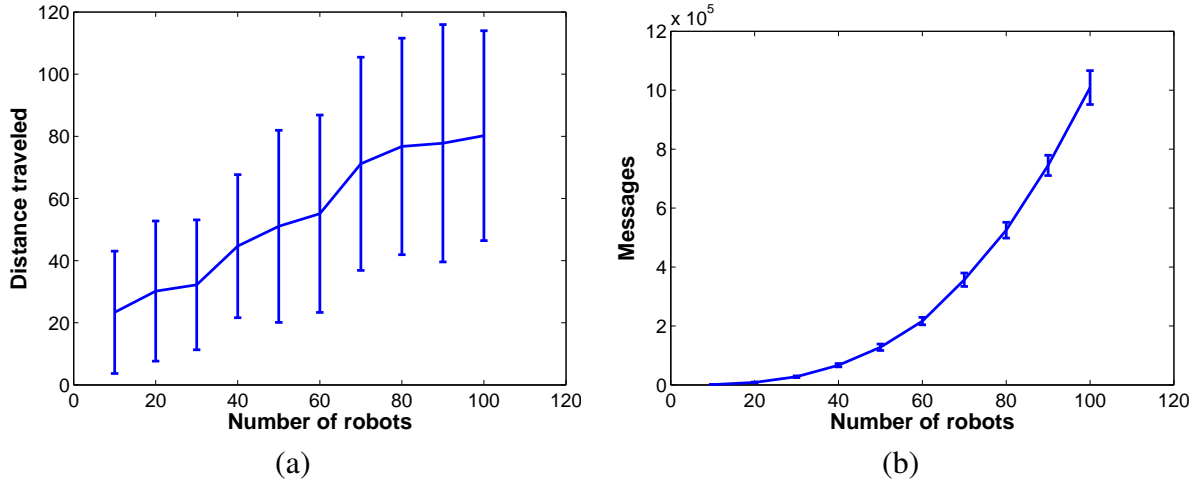


Figure 3.14: (a) Distance traveled by modules to reach target configuration for different number of modules in the environment, (b) Number of messages exchanged between modules to select positions in the target configuration for different numbers of modules in the environment.

the main contribution of this work - how to construct a modular robotic system from an initial set of singletons and configurations. Figure 3.13(b) shows how the planning time changes with different numbers of modules; the y -axis denotes the total planning time in milliseconds and the x -axis denotes the number of modules. It can be noted from this plot that though for a small set of modules, time change is almost constant, as the configuration size as well as the number of modules increases, elapsed time increases in a polynomial fashion. This elapsed time indicates only the planning phase execution time of the modules. Figure 3.14(a) shows how with increasing number of modules the total distance traveled by them changes. This metric is calculated by adding the distances traveled by each module from their initial positions to their respective spots in T . The figure shows that the total distance traveled by the modules increases linearly. We have also calculated the total number of messages passed among modules while the configuration formation process is occurring. Figure 3.14(b) shows how the number of total messages changes with the number of modules. As can be expected, with a higher number of modules in the environment, the number of messages increases in a polynomial fashion. We are also interested in understanding the completion rate of the planning phase. The percentage of

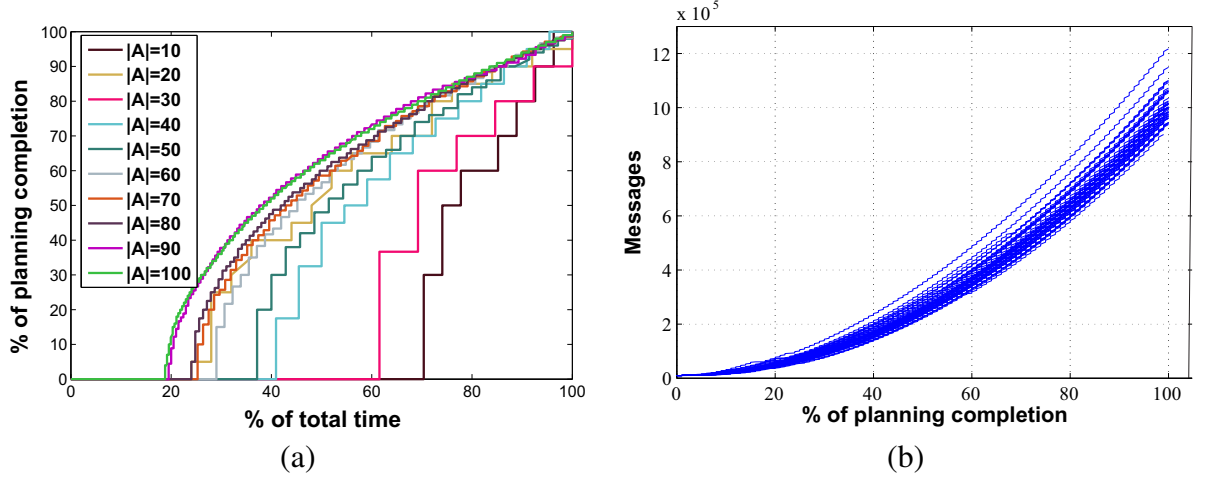


Figure 3.15: (a) Change in % of planning completion with % of time completion, for different no. of modules; (b) Change in no. of messages at different time steps, for 100 modules.

planning phase completion indicates what percentage of the total modules are allocated to their spots in T . Figure 3.15(a) shows the planning completion rate for different numbers of modules between 10 and 100. We can see that with increasing numbers of modules, the completion rate increases and is more evenly distributed over time. For instance, with $|A| = 10$, after 70% time completion, only 30% of planning has been completed, whereas with $|A| = 100$, 30% of planning gets completed after only 25% of time completion. The relationship between planning phase completion and the number of passed messages for 100 modules has been shown in Figure 3.15(b). All the graphs from 50 runs have been plotted. We observe that the message count is increasing almost-linearly with completion rate. For the next set of experiments, we have kept the number of spots, $|S|$, fixed at 50 and we have varied the number of modules between $[50, 100]$. Figure 3.16(a) shows planning completion rate for different numbers of modules. We can see that with increasing number of modules, completion rate increases and is more evenly distributed over time. This behavior is similar to what we have seen in Figure 3.15(a). Although in Figure 3.15(a), for most of the module sets, the planning phase completes almost at the end of their respective time-lines, in the case of Figure 3.16(a), we can notice that the planning phase finishes at

different stages of their time-lines, for different numbers of modules. As an example, for $|\mathbb{A}| = 100$, the planning phase almost converges at 50% the of total elapsed time, whereas for $|\mathbb{A}| = 50$, it takes almost 100% time to converge. Figure 3.16(b) shows the comparison

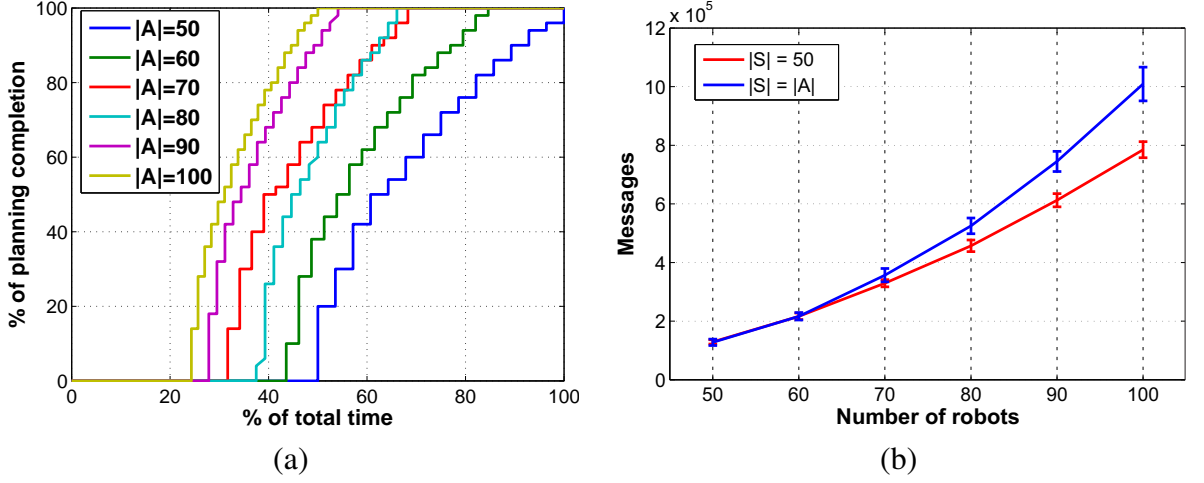


Figure 3.16: (a) Change in % of planning completion with % of time completion, for different no. of modules and $|S| = 50$; (b) Change in no. of messages for different no. of modules and different no. of spots.

of the number of passed messages by the different numbers of modules, between the cases where $|S| = 50$ and $|S| = |\mathbb{A}|$. It can be observed from this figure that with same number of modules, fewer messages are passed if there are fewer spots than modules, i.e., if $|S| < |\mathbb{A}|$. For example, with, $|\mathbb{A}| = 100$ and $|S| = 50$, 8×10^5 messages are passed, whereas with $|S| = 100$ and keeping $|\mathbb{A}|$ fixed to 100, the number of messages increases to 10×10^5 . This result shows that the total number of messages depends on both the number of modules and spots. Next we have run experiments to check how the subgraph isomorphism technique used in this work helps to reduce the number of disconnections from initial configurations. For this test, we have kept $|S| = |\mathbb{A}| = 100$. Initially all modules were part of some smaller configurations and each initial configuration has the same size. We have varied the sizes of each initial configuration between $[10, 20, 25, 50]$ and thus in these cases the number of initial configurations have been varied between $[10, 5, 4, 2]$. The planning times and number of modules required to be disconnected for these cases are shown in Table 3.1. As can be

Size of All Initial Configurations	Planning Time (ms.)	No. of Modules Disconnected
10	171.48 (avg.) 15.13 (std.)	0.12 (avg.) 0.32 (std.)
20	166.66 (avg.) 12.88 (std.)	4.32 (avg.) 3.56 (std.)
25	172.10 (avg.) 11.30 (std.)	8.76 (avg.) 4.85 (std.)
50	218.28 (avg.) 19.57 (std.)	29.68 (avg.) 5.33 (std.)

Table 3.1: Planning times and the numbers of disconnected modules (average and standard deviation) in the configuration formation process, where all initial configurations have same sizes ($|S| = |\mathbb{A}| = 100$).

seen, with increasing size of initial configurations, the number of disconnected modules increases. This is because the probability of finding isomorphic subgraphs in T decreases with increasing size of initial configurations. But the low numbers of disconnected modules show that it is always beneficial, in terms of number of connections detachments and re-attachments, to use our proposed approach than to break all initial configurations into singletons and then form the target configurations with them.

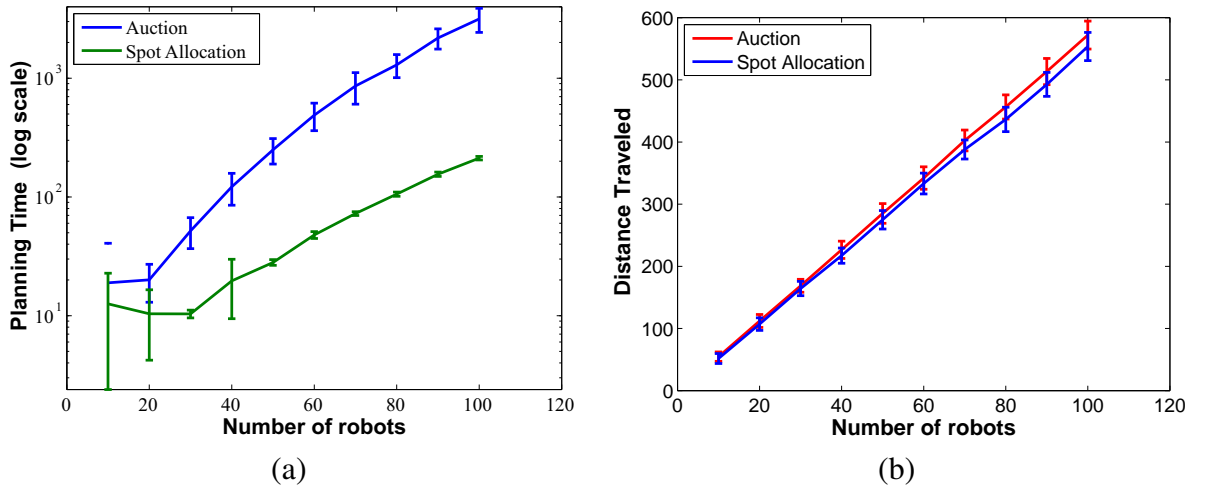


Figure 3.17: (a) Log scale comparison of planning phase execution time with auction algorithm; (b) Comparison of total traveled distances with auction algorithm.

Comparison with Auction-based Allocation. We have also compared our approach for MSR configuration formation with an auction algorithm [10] that finds an optimal assignment between spots and modules. Using the auction mechanism a group of modules bid for a set of spots. First the modules bid for their most preferred spots; conflict among modules for the same spot is resolved by revising bids in successive iterations. The assignment is done in a way such that the utility is maximized. The auction algorithm does not take connected configurations of modules during allocation. Therefore only for the tests which compare the performances of our algorithm against the auction algorithm, initially all the modules are considered to be singletons.

A log scale comparison of planning times between spot allocation and the auction algorithms is shown in Figure 3.17(a). As can be seen from this graph, with increasing the number of modules, the difference between planning times of these two algorithms increases, i.e., our proposed algorithm's performance gets better with increased number of modules compared to the auction algorithm. Comparison of distances traveled by the modules using our algorithm and the auction algorithm is shown in Figure 3.17(b). As we can see in this plot, in most of the cases total traveled distance by the modules is the same. But with higher numbers of modules, using the proposed spot allocation algorithm modules travel less distance than by using the auction algorithm. Thus the spot allocation algorithm assigns the spots to the modules in very nominal time, keeping the cost for movement almost the same (or less in some cases), compared to the auction algorithm. A log scale comparison of number of messages generated, by the spot allocation and auction algorithms, is shown in Figure 3.18(a). This figure indicates that the spot allocation algorithm generates fewer messages than the auction algorithm, which helps to reduce the communication overhead. Figure 3.18(b) compares the completion rates of planning phases of the auction and spot allocation algorithms - the x -axis denotes the percentage of total time elapsed. This result indicates that completion rate of the auction algorithm is higher, even though the auction algorithm takes longer than the spot allocation algorithm.

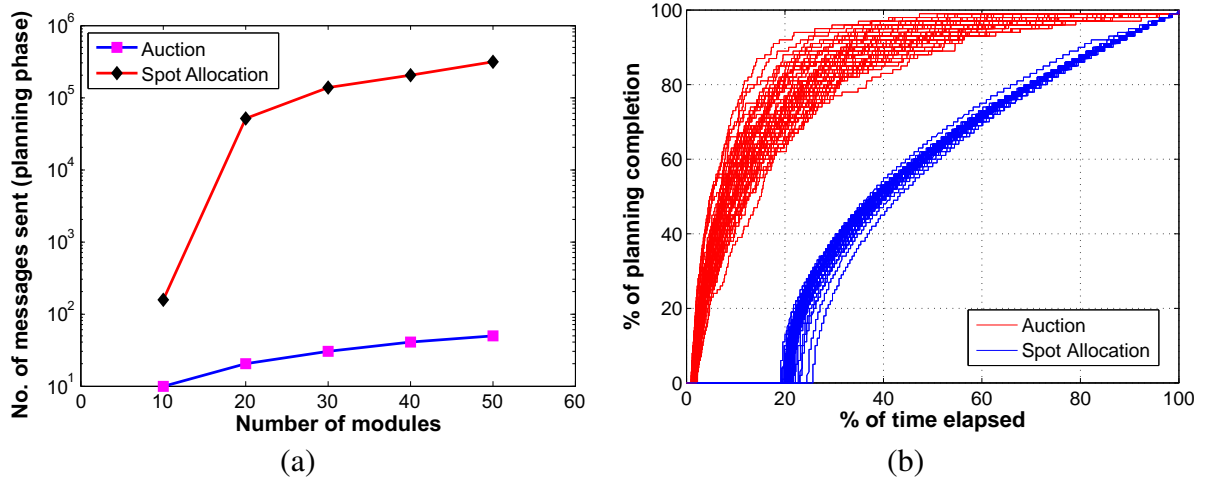
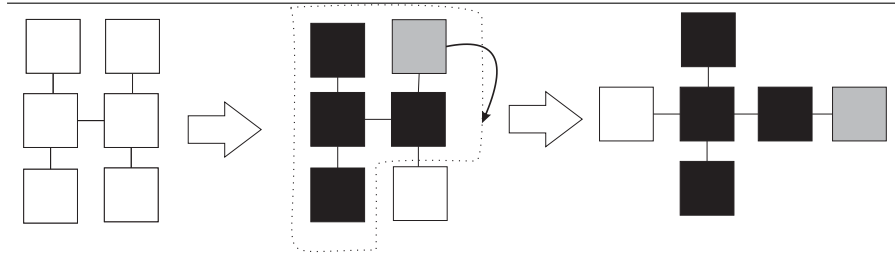


Figure 3.18: (a) Log scale comparison of no. of messages with auction algorithm; (b) Change in % of planning completion with % of time completion and comparison with auction algorithm. 50 lines indicate 50 runs.

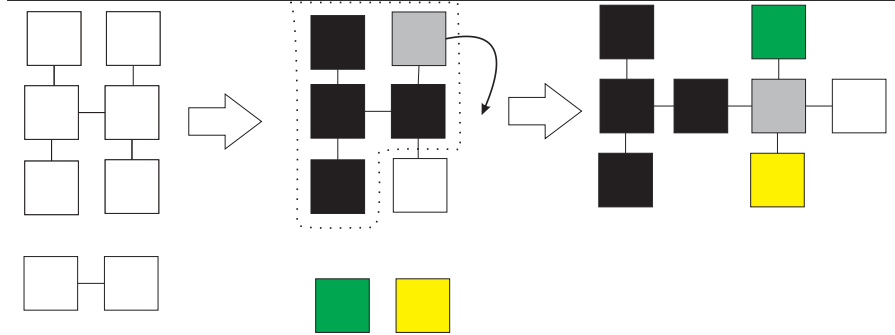
Case Studies

In this section, we have shown 8 specific cases of the configuration formation process that are shown in Figure 3.19. Each of the initial and target configurations used for this set of experiments have been shown to be feasible and stable for the ModRED MSR in [49]. To show the generalization of our approach, we have used both tree and graph structured MSR configurations as opposed to only tree configurations used in the previous sections. This was also made possible due to not-so-large configurations used here. Squares represent the modules and the links between two squares denote the connection between those two modules.

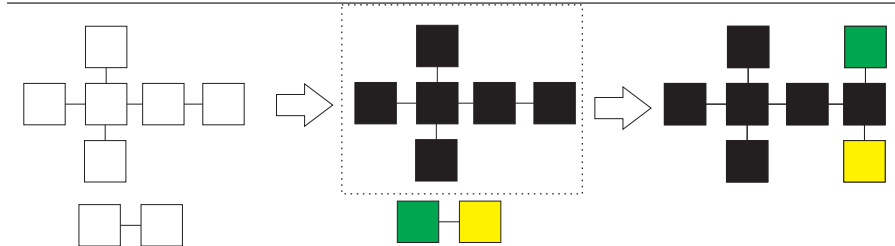
For each case illustrated, the left-most diagram shows the initial configurations and/or singletons, the middle diagram shows the detected MCS (or, IS) and the diagram on the right shows the final formed configuration. The modules are color-coded to show the final allocations. MCS (or, IS) are shown with dotted boxes. Grey-colored modules represent the modules that remain connected to the same neighboring module between initial and target configurations, but only change the connector through which they are connected. Although this operation requires one un-docking and one re-docking operation, it consumes less



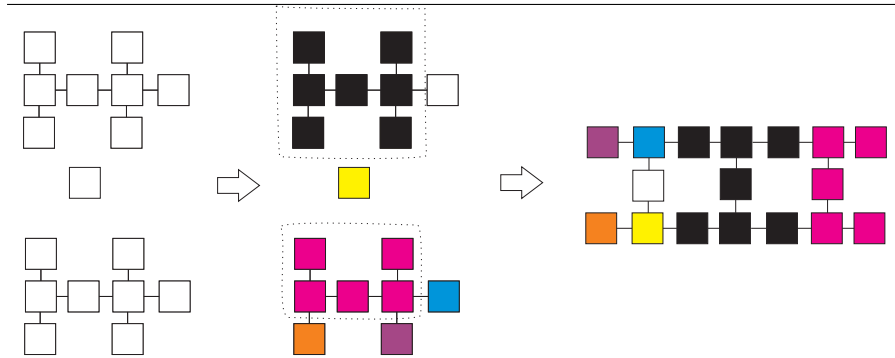
Case 1: Planning Time: 110 ms., No. of disconnections: 1



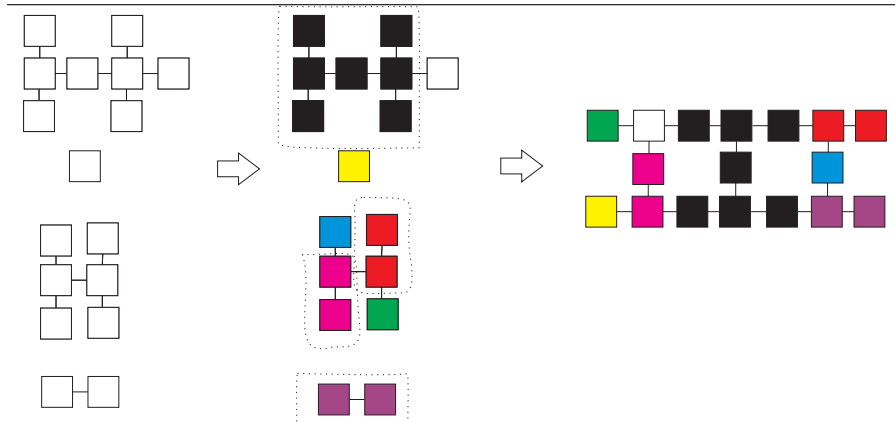
Case 2: Planning Time: 113 ms., No. of disconnections: 2



Case 3: Planning Time: 111 ms., No. of disconnections: 1



Case 4: Planning Time: 170 ms., No. of disconnections: 4



Case 5: Planning Time: 182 ms., No. of disconnections: 3

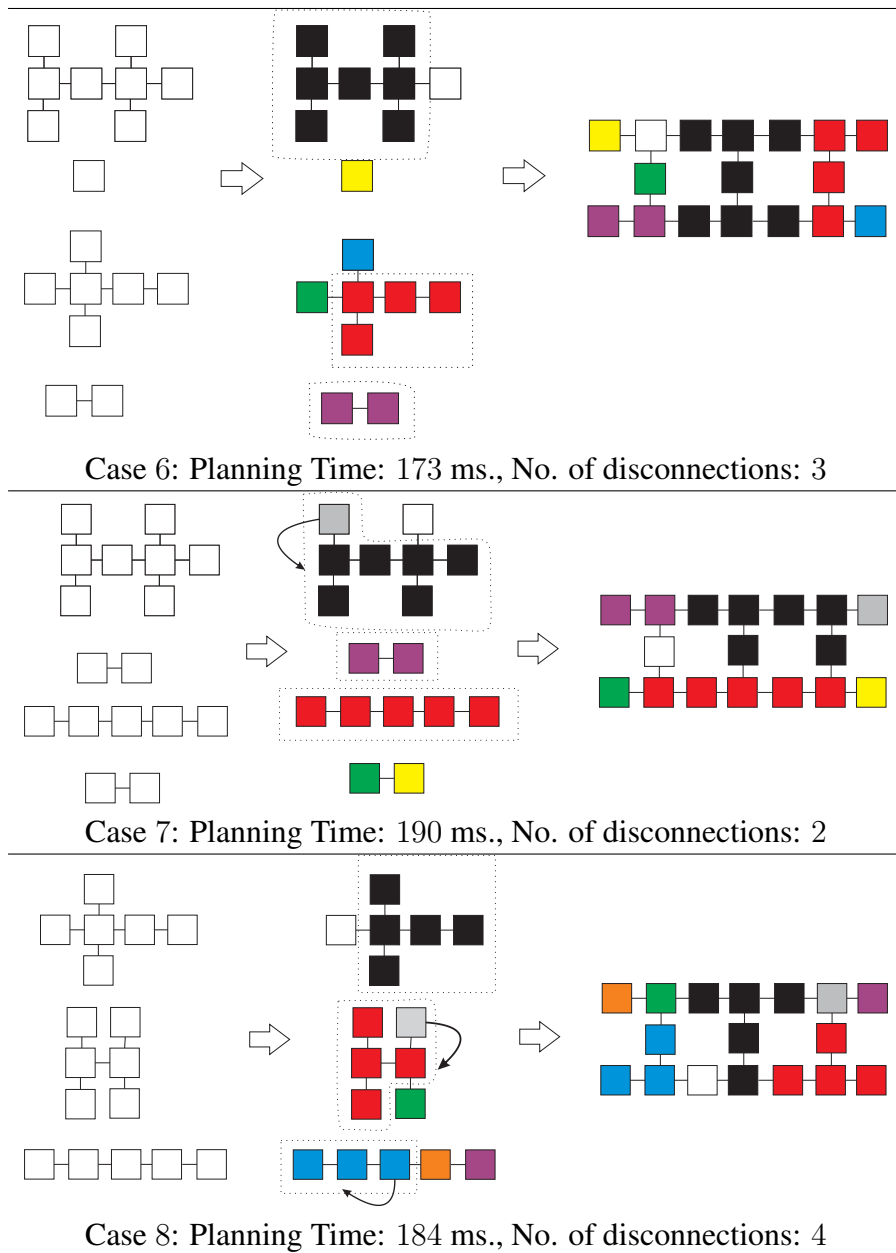


Figure 3.19: Cases showing configuration formation procedure along with corresponding planning times and number of disconnections required. Leftmost figure in each case shows the initial configurations and singletons, middle figure shows the MCS (or, IS) found (marked by dotted boxes) by executing our algorithms, rightmost figure shows the final formed target configuration with modules selecting spots (shown in a color-coded fashion).

energy than if the module were to be connected to a non-neighbor module. The planning time and number of disconnections for each case are provided alongside each configuration formation case in Figure 3.19. We can see that each of the test cases requires less than 200 milliseconds of planning time. Target configurations are also formed with relatively few link disconnections (maximum being 4).

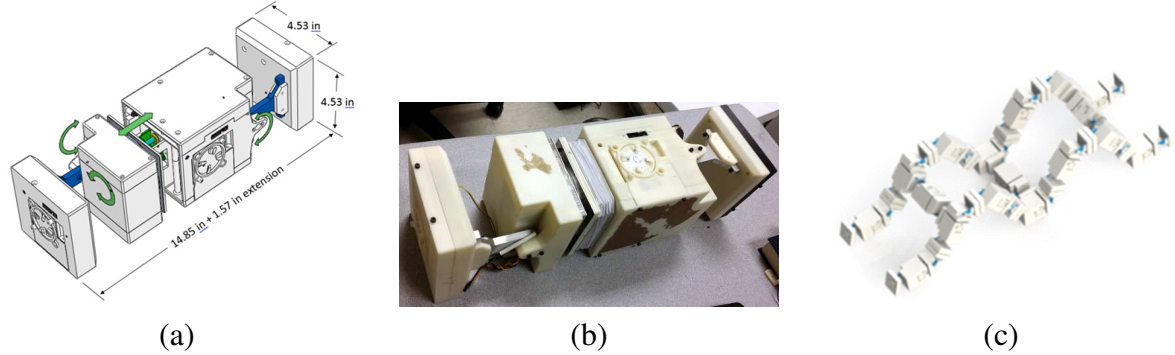


Figure 3.20: A single ModRED module used for the configuration formation algorithm (a) CAD drawing, (b) hardware. Each module has 4 connectors which enables it to form branched configurations. (c) A 17-module branched, ladder configuration similar to Figure 3.8(b) that is capable of complex maneuvers and forming truss-like structures [7].

Hardware Experiments with ModRED II MSR

The main objective of hardware experiments is to show how much time it takes for the singleton modules and the leader modules to do the local computations. We have chosen the ModRED II modular self-reconfigurable robot platform [49] for experimental purposes. Each ModRED II module is a 4-DOF robot (similar to its predecessor ModRED I [7]) with four connectors (unlike its predecessor which has only two connectors, one at each end). Due to its four in-built connectors, ModRED II is able to form more complex configurations compared to ModRED I. For more details on ModRED II hardware architecture and features, readers are referred to [49]. Each ModRED II module also houses a BeagleBone Black, a Linux based computer, on-board. It has 512MB DDR3 RAM and 4GB 8-bit eMMC on-board flash storage. It is also equipped with a AM335x 1GHz ARM® Cortex-A8 processor.

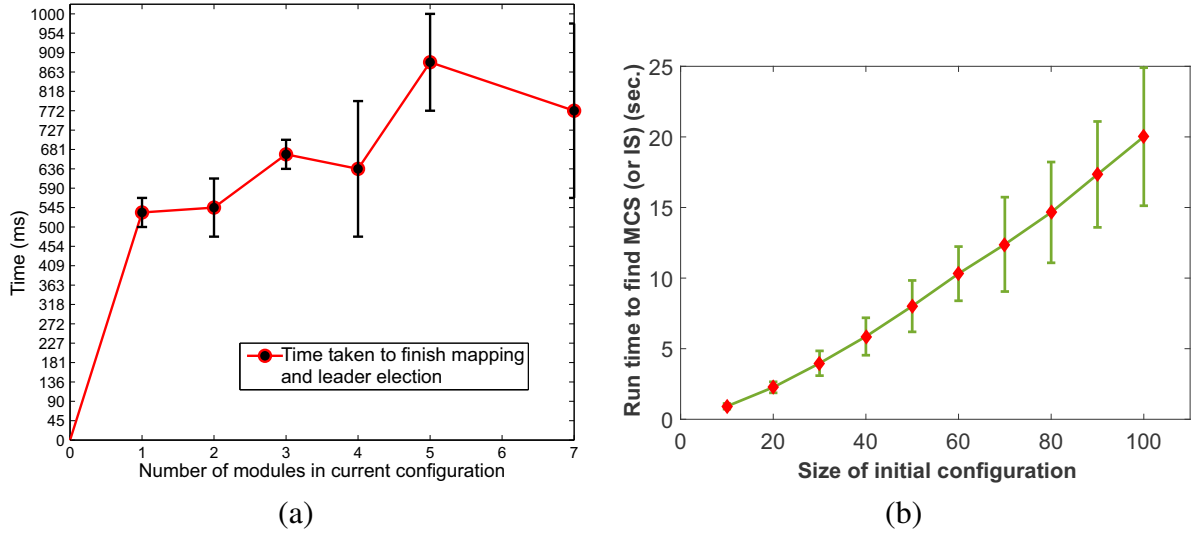


Figure 3.21: (a) Comparison of run times to elect a leader and map the topology of the ModRED configuration against the configuration size; (b) Change in run time to find MCS with different number of modules in the initial configuration.

As we have mentioned earlier that our main objective is to show how much on-board computation is needed by the singletons and the leader modules, we have used a single ModRED II module for our experiments which alternatively worked as a singleton and a leader module. For these experiments, we have implemented our algorithms on the Beaglebone Black Processor inside the ModRED II module and collected the results. We have also compared our algorithms' performance against the auction algorithm's performance by implementing the auction algorithm on the ModRED II as well. As we ran hardware tests on a single module, the reported run time results only consider the computational time, and do not include the communication time between modules.

First, the ModRED module acted as a leader. In our earlier work [8], we have shown how much time it takes to elect a leader and to map the topology of the configuration for varying sizes of the configuration. This result is reproduced here in Figure 3.21(a) to show how much pre-processing will be needed before we can start executing our proposed algorithms. This shows that with 7 modules present in the configuration, it takes less than a second of time to elect the leader and map the topology of the configuration.

Next, the elected leader module searches for *MAX* MCS (or IS) for the given target configuration. For this test, we have provided the topology of the initial configuration and also the target configuration to the leader module. Similar to the simulation results, these configuration trees have been generated randomly. Figure 3.21(b) shows how with the increasing size of the configuration, run time to search the MCS changes. Note that the size of the target configuration was set to 100 for all the cases here. We can observe that the run time increases almost linearly in fashion even though it took more time than simulated experiments. We noticed that running the *blockAllocation()* algorithm along with this took a negligible amount of extra time, so that result is not included. The main reason behind this is that calculating the possible MCSs is the most computationally intensive component in our proposed *blockAllocation()* algorithm.

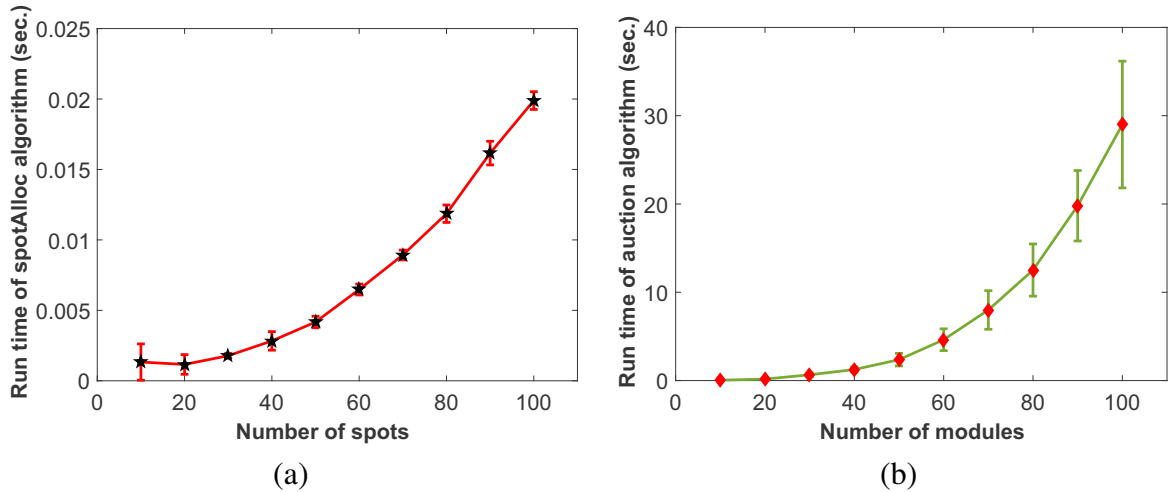


Figure 3.22: (a) Change in run time of the *spotAllocation()* algorithm with different number of spots; (b) Change in run time of the auction algorithm with different number of spots.

Next, we have implemented the *spotAllocation()* algorithm on the singleton ModRED module. The result is shown in Figure 3.22(a). This result shows that a singleton module will take much less time (0.019 sec.) even with 100 spots. Finally, we implemented the auction algorithm on the ModRED module and in this case, the ModRED module acts as a centralized auctioneer agent. The number of spots is set equal to the number of

modules in this case. The result of this test is shown in Figure 3.22(b). We can notice that with increasing numbers of modules, the run time of the auction algorithm increases significantly. For example, with 100 spots, the ModRED module takes 29 sec. to run the auction algorithm whereas it takes only 0.019 sec. to run the *spotAllocation()* algorithm, a 1526-times improvement.

3.5 Discussions

Our main objective was to find an efficient solution for the configuration formation problem where initially, modules could be either singletons or part of an already connected configuration. We have argued that as docking and un-docking of modules are costly operations, these operations should be minimized by preserving the initially formed configurations as much as possible. We have proposed subgraph isomorphism based checking and allocation algorithms that retain maximal portions of connected modules while forming a target configuration. Even though our solution approaches are studied and tested for 2D planar configurations, we believe that in the future this technique can be extended to 3D scenarios. From our results, we can notice that our both of our solutions produced good results consistently, both in terms of planning time, distance traveled and number of connections/disconnections among modules, given the combinatorially intractable nature of the used techniques. Although most of the results reported here are produced using tree-like MSR structures, our case studies show that even with graph structures, our methods are able to produce considerably good results especially in terms of number of disconnections among the initially formed configurations.

As allocating modules to target spots is an instance of the classical bipartite graph matching problem, algorithms like Hungarian matching can also be used for the allocation process (at least for singleton modules) [60]. As our second approach is distributed in nature, a relevant issue is the scalability of the number of messages passed between

modules for synchronizing intermediate calculations of the algorithms. As the modules need to reach consensus about allocation in a distributed manner, they need to continuously exchange information about the current state of the allocation process with each other. A semi-centralized method, where part of the decision is made by a central supervisor (as discussed in Section 3.3), can be used to mitigate this problem [32]. However, this increases the risk of potential failure of the whole process if the supervisor fails. Finally, we have tested our approaches with homogeneous modules only, but it remains an open research problem for future researchers to investigate the configuration formation problem with heterogeneous modules where initially modules can be part of different configurations instead of just singletons. A preliminary study done by us on self-assembling heterogeneous agents can be found in [30]. Besides modular robotics, we believe that our proposed approach can be used for parts assembling in the manufacturing and automobile industries where smaller portions (initial configurations and/or singletons) of objects, can be brought together and assembled to form a large object (target configuration).

Chapter 4

Learning: Adaptive Locomotion

Learning

After the configuration is formed, the MSR needs to move from its current location as a whole to go to a goal position to complete some task. The task can be exploration, monitoring or information collection. If the topology of the configuration is pre-defined, then its locomotion can be planned beforehand. But if the configuration is arbitrarily formed depending on the current task, then we cannot plan the locomotion for that particular configuration *a priori*. Therefore generating the locomotion pattern on-the-fly depending on the current configuration's topology is a very important problem in MSRs. In a recent study, the authors have identified this problem as one of the most important and challenging to be accomplished by future researchers [1].

4.1 Background

Unlike the configuration formation planning problem, locomotion planning in modular robotic systems has been studied in the literature extensively. Despite that, it has remained a big challenge for MSR researchers to find a solution which quickly adapts to the shape of the current configuration of the MSR and consequently, the configuration learns to

move towards its goal within a short amount of time [1]. Most of the research in MSR locomotion tried to develop pre-defined locomotion plans for any particular configuration, one of the first of which is found in [99]. In this work, the authors have proposed (pre-defined) locomotion plans for the Polypod MSR which can take various shapes like snake, rolling track and hexapod. Many of these works on locomotion planning in MSRs have proposed solutions based on gait tables. Gaits are synchronized patterns of locomotion used by animals, humans and machines such as robots [71]. Since [99], many researchers have used gait tables for the purpose of locomotion in modular robotic systems. In the context of MSRs, a gait control table can be defined as the following:

“A gait control table is a two-dimensional table of actuator actions (e.g., a goal position or to act as a spring), where the columns are identified with an actuator id and the rows are identified with a condition for transition to the following row (e.g., a time-stamp or a sensor condition) [20].”

Usually, gait control tables are executed in a cyclic order to produce periodic locomotion patterns in MSRs. Hand-coded locomotion patterns using gait control tables for ModRED MSR have been proposed in [21]. Though popular, this can be mostly useful for chain type modular robotic systems [19]. In [19], the authors have proposed a novel reinforcement learning based technique for gait adaption for locomotion learning in MSRs. They have applied their proposed locomotion strategy on seven different shaped configurations (containing up to 12 modules) and they have shown that for six of those tested configurations, the locomotion strategy converges in most of the trials.

Not only for self-reconfigurable robots, gait tables have been used for locomotion pattern generations for fixed-shape robots such as biped [46], hexapod [63] etc. The main difference between MSR gait tables and these approaches is that these gait tables are for fixed-shape configurations and therefore the gait control tables can be hand-coded before deployment. But in the case of MSRs, as the configuration shape might not be known *a priori*, therefore fixed gait control tables are not useful. Also it has been found that learning

gait tables for MSRs is a computationally hard task to accomplish as the search-space for finding the best gait table increases exponentially with the action space of each module [19]. Recently, researchers have proposed a learning strategy for a fixed-shape hexapod robot which learns to adapt its locomotion even after some part of the robot's body has been damaged [25].

CPGs: Central pattern generators (CPGs) have also been studied for locomotion planning in MSRs [55]. According to [14], “*CPGs are neural networks that can produce rhythmic patterned outputs without rhythmic sensory or central input and they underlie the production of most rhythmic motor patterns*”. The goal of the CPG is to generate synchronized oscillations (locomotion patterns/rhythms) in connected oscillators (modules) [88]. CPGs have also been used for locomotion control in monolithic robots [53]. Locomotion in MSRs using CPGs have been first studied by Kamimura et al. for their M-TRAN MSR [56]. In [88], the authors have proposed a CPG-based approach for locomotion learning in Yamor, a chain-type modular robotic system. In this work, along with the basic CPG, the authors have proposed an online optimization technique which is executed in parallel with CPG generation. The authors have also mentioned that it is more beneficial to use CPGs for locomotion learning than gait control tables as it provides better handling of synchronization among modules. Also they have mentioned that CPGs offer smoother changes in produced oscillations (locomotion patterns) than gait tables. But CPGs are mostly useful when the locomotion is periodic [53]. Not only for MSRs, CPGs have been used for locomotion planning in other types of robots as well, e.g., swimming-pattern planning in a fish-like robot [24]. Similar to CPGs, approaches based on biological phenomena, such as hormone-based controllers for MSR locomotion planning have also been studied [45].

Synchronization: Synchronization plays a very important role in locomotion planning. If the movements of multiple modules are not synchronized, then the locomotion is not synchronized and therefore the locomotion is not proper, i.e., either the configuration will

not move towards the correct direction and/or the speed of the configuration will be very slow [85]. Several different approaches for maintaining synchronization among modules in a configuration have been proposed. A biology-inspired hormone based approach has been proposed in [85] where modules pass hormones (information-coded) among themselves in the configuration and can detect any change in the topology of the current configuration; this is also used for synchronization of their actions in a distributed manner. In a leader-follower approach, every module detects its local leader module and coordinates its actions with that leader module only and thus the synchronized behavior flows through the configuration [86, 26]. On the other hand, in a master-slave approach, a central leader is elected for the whole configuration (called the master) and control is passed to all other ‘slave’ modules [16]. In a recent work, Baca et al. [9] proposed a master-slave approach for configuration discovery in modular robots where the master is elected using a bully algorithm [66].

Multi-agent Learning: An excellent overview of multi-agent learning algorithms can be found in a recent survey [11]. Researchers have come up with algorithms to solve problems in both cooperative [57] and competitive [12] multi-agent systems. Various approaches, like independent action learners [57], joint-action learners [64], and optimization techniques (e.g., gradient ascent) [12], have been proposed to solve different multi-agent learning problems. We have proposed an independent action learning algorithm to solve the locomotion learning problem in MSRs in Section 4.4.

4.2 General Problem Formulation

Let $M = \{m_1, m_2, \dots, m_N\}$ denote the set of N modules connected together forming a certain configuration. The configuration is connected, i.e., any two modules in the configuration are connected either physically or through other modules. $neigh(m_i)$ denotes the set of neighboring modules, i.e., the modules which are physically connected to

m_i . Each module has a unique identification (ID). Module m_i 's position and orientation are denoted by (x_i, y_i, θ_i) . We assume that each module knows the topology of the configuration [8]. We also assume that each module is able to calculate its own position using a GPS or an overhead tracking system.

Each module performs an action by actuating its motors. An action a_j is a vector which specifies the actuation provided to each of the K motors present in the module, i.e., $a_j = \{ac_1, ac_2, \dots, ac_K\}$. Each module is provided a library of actions, A , from which it can choose its action at any given time-step. The actions present in the library are known beforehand and given as an input by the user. Each module, m_i , receives a reward by performing an action a_j in a specific time-step t , denoted by $R_i(a_j, t)$. Reward can be calculated as the Euclidean distance traveled by a module since its last time step, given by:

$$R_i(a_j, t) = ||p_i(t) - p_i(t-1)|| \quad (4.1)$$

where $p_i(t)$ is the position of the module m_i at time-step t . Let a_i^{best} denote the best (highest reward earning) action. We have modeled the learning strategy as a stateless Q-learning approach [91, 57, 19]. Let $Q(a_j)$ denote the Q -value of an action a_j . The Q -value provides an estimate of the usefulness of executing any action in the next iteration, and this value is updated after each learning cycle according to the reward received for the action. Also, let ϵ denote the ratio between exploration of new actions and exploitation of past high reward-earning actions.

4.3 Locomotion Learning via Joint Action Learning

First, we propose a reinforcement learning-based adaptive locomotion strategy which learns the best action for each module. We have observed that each module's locomotion performance (e.g., distance traveled) is highly dependent not only on the action that particular module is taking, but also on the actions taken by its neighboring modules (i.e.,

the modules directly connected to it). This observation led us to build our learning strategy in such a way that it does not only learn from its own previous actions, but it also learns from the correlation of that past action with the neighboring modules' actions at that particular time-step.

4.3.1 Q-Learning Based Approach for Distributed Locomotion Learning

In this approach for solving the locomotion learning problem, we try to capture the following idea: each module not only learns from its own current and past actions, but also from the relationship among the actions performed by its neighboring modules at any particular time-step. The pseudo-code of our proposed approach is shown in Algorithm 9.

Algorithm 9: Q-Learning Based Distributed Locomotion Learning Algorithm

```

1 Perform every action  $a_j \in A$  in sequential order.
2 Receive corresponding rewards  $R_i(a_j, t)$ .
3  $Q(a_j) \leftarrow R_i(a_j, t), \forall a_j \in A$ .
4  $A_{DS} \leftarrow$  Data structure for storing best action-pair.
5 Loop
6    $a_i^{best} \leftarrow \arg \max_{a_k \in A} Q(a_k)$ .
7   With  $\epsilon$  probability,  $a_{nxt} \leftarrow a_i^{best}$ .
8   With  $(1 - \epsilon)$  probability,  $a_{nxt} \leftarrow a_{rand}$ .
9    $a_{pres} \leftarrow$  Prescribed best action for  $a_{nxt}$  to be send to neighbor modules.
10  Send  $\{a_{nxt}, a_{pres}, R_i(a_{nxt}, t)\}$  to the neighboring modules.
11  Receive similar message from the neighbor modules:  $\{a'_{nxt}, a'_{pres}, R_j(a_{nxt}, t)'\}$ .
12  Let  $a''_{pres}$  be the already prescribed action stored with  $m_i$  in  $A_{DS}$  for  $a'_{nxt}$  with
    corresponding reward  $R_i(a'_{nxt}, t)''$ .
13  if  $R_j(a_{nxt}, t)' > R_i(a_{nxt}, t)''$  then
14     $\lfloor$  With  $\tau$  probability,  $a_{nxt} \leftarrow a'_{pres}$ ./*switch to prescribed action*/
15  Perform the action  $a_{nxt}$  and receive reward  $R_i(a_{nxt}, t)$ .
16  Update Q-value:  $Q(a_{nxt}) = Q(a_{nxt}) + \alpha \cdot (R_i(a_{nxt}, t) - Q(a_{nxt}))$ .
17  Update  $a_i^{best}$  and  $\{a'_{nxt}, a''_{pres}, R_i(a'_{nxt}, t)''\}$  if necessary.
```

First each module, $m_i \in M$, performs every action, a_j , available in the action library, A , in a sequential order and calculates the rewards, $R_i(a_j, t)$, for all the actions. Q -values of all the actions are also initialized to these reward amounts, i.e., $Q(a_j) = R_i(a_j, t)$. Next, the

modules follow the modified Q-learning strategy as shown in Algorithm 9. In the beginning of every learning iteration, each module finds out which action has maximum Q-value associated with it so far, i.e., the best action. This information is local to every module, i.e., each module has its local best action, a_i^{best} , calculated by the following equation: $a_i^{best} \leftarrow \arg \max_{a_k \in A} Q(a_k)$. We have observed that the behavior of the modules are highly coupled and if the actions taken by them are not synchronized in any way, then the modules take a longer time to reach the final learned behavior [1]. To alleviate this problem, each module communicates its next action information with its neighboring modules before performing any action. Each module chooses its best action seen so far a_i^{best} , to execute in the next learning iteration with ϵ probability or a random action a_{rand} with $(1 - \epsilon)$ probability. Each module sends this calculated next action for the next iteration, a_{next} , to the neighboring modules. Similarly, it receives the next actions of its neighboring modules for the next iteration.

Joint-Action Learning

Each module m_i maintains an action-pair data-structure A_{DS} which can be imagined as a $|A| \times 3$ matrix. Each row in A_{DS} contains an action $a_k \in A$ (first column), a prescribed best (highest reward earning) action $a_{pres} \in A$ for a_k (second column) and corresponding reward received by m_i by performing a_{pres} : $R_i(a_{pres}, t)$ (third column). $a_k \in A_{DS}$ represents the action performed by a neighboring module, $m_j \in \text{neigh}(m_i)$, at any particular time-step t . a_{pres} represents the corresponding action from the action set performed by m_i at that time-step t and which earned m_i a reward of $R_i(a_{pres}, t)$. For any module m_i , A_{DS} contains only one copy of each action $a_k \in A$ in its first column, even if multiple neighboring modules might have performed that action. Only the corresponding best action performed by m_i at those time-steps and the reward earned by it (column 2 and 3) change over time. Initially all the prescribed actions in A_{DS} and the corresponding rewards earned for those actions are unknown and therefore initialized to a null value. Over

time, when the modules start communicating their current actions, this data structure gets populated accordingly. While communicating, each module not only sends its next action, a_{next} , but also the information about the corresponding prescribed best action available with it, a_{pres} , for its neighboring modules and the reward received for that action pair, $R_i(a_{next}, t)$.

Once the module receives this information from its neighboring modules, it checks whether any of the neighboring modules has any prescribed best action for that current learning iteration or not. Only if the prescribed action received from the neighboring module, a_{pres} , has been shown to earn higher reward than the prescribed action already available with the module, then the module selects a_{pres} as its next action with τ probability or keeps a_{next} to be its next action with $(1 - \tau)$ probability. After the next action is decided, the module performs that action and receives reward for that action. Following [57, 91], each module then updates the Q-value of the action performed following the update equation:

$$Q(a_{next}) = Q(a_{next}) + \alpha \cdot (R_i(a_{next}, t) - Q(a_{next})) \quad (4.2)$$

where $\alpha \in [0, 1]$ is the learning rate. Also, it updates the best action and action-pair data structure A_{DS} as necessary.

4.3.2 Experimental Evaluation

Settings

We have mainly implemented our proposed adaptive locomotion learning strategy on simulated ModRED modules within the Webots robot simulator [68]. Each ModRED module is a 4-DOF robot with connectors in both ends. For more details on ModRED hardware architecture and features, readers are referred to [7]. Simulated ModRED modules within the Webots simulator and actual ModRED hardware are shown in Table 1. We have tested our approach on ModRED chain configurations having 2, 3, 4, 5 modules

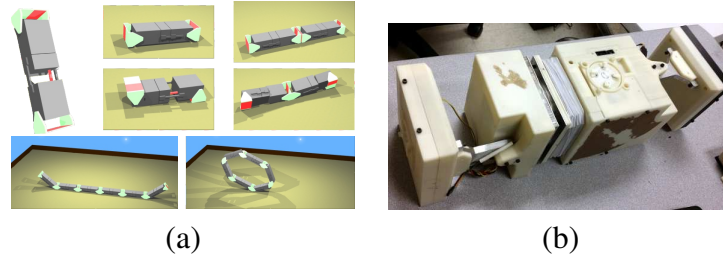


Figure 4.1: a) Simulated ModRED Modules within Webots Robot Simulator and (b) Hardware of ModRED.

(denoted by M2, M3, M4, and M5 respectively).¹

ModRED actions: Each ModRED module can have 54 unique actions [21]. In this work, we have shortlisted 10 of them for inchworm locomotion and 5 actions for rolling motion which have been used for our testing purpose. These actions have also been used in [21] for creating hand-coded gaits. More actions can always be used for more robust locomotion behavior, but at the same time it will slow down the learning process exponentially.

In [21], the authors have described hand-coded gait tables for ModRED's locomotion (inchworm and rolling). As described earlier, ModRED has 4 degrees of freedom and consequently 4 motors. Tables 2 and 3 summarize the action set used for ModRED modules for inchworm and rolling locomotion. Two end connectors (front (FC) and rear (RC) connector) can go up (0.7 rad), down (-0.7 rad) or stay in neutral (0 rad) positions represented as +1, -1 and 0 respectively in the action table. Similarly, the translational motor (T) of ModRED helps to extend or contract the body of the module and is represented by +1 and -1 in the gait tables. The rotational degree of freedom (R) of the module either rotates the module in clockwise (+1: 6.28 rad) or anti-clockwise (-1 : -6.28 rad) directions or just stays neutral (0: 0 rad). For inchworm locomotion, the rotational motor is always inactive (0 throughout the column). After one action is executed, the module calculates its local reward for that particular performed action using Equation 4.1.

¹Because each module has 4 DOF, testing with ModRED modules becomes computationally intensive with more than 5 modules. Testing with larger configurations is reported for a 1-DOF robot called Yamor.

	FC	T	R	RC
Action 1	0	-1	0	0
Action 2	-1	-1	0	1
Action 3	-1	1	0	1
Action 4	1	-1	0	-1
Action 5	-1	-1	0	0
Action 6	0	1	0	-1
Action 7	1	1	0	1
Action 8	-1	-1	0	-1
Action 9	0	-1	0	-1
Action 10	-1	1	0	0

Table 4.1: Actions for inchworm locomotion

	FC	T	R	RC
Action 1	0	-1	0	0
Action 2	-1	-1	0	1
Action 3	-1	-1	1	1
Action 4	1	-1	0	-1
Action 5	1	-1	1	-1

Table 4.2: Actions for rolling locomotion

We have also tested our approach on Yamor modular robots, which is a 1-DOF robot [29]. We have used the sample of Yamor’s simulated model provided in Webots. For more details on Yamor hardware, readers are referred to [29]. We have tested on 10, 12 and 14 Yamor chain configurations (denoted by Y10, Y12, and Y14 respectively). Each Yamor module has 3 available actions: go up (+0.5 rad), go down (−1.57 rad) or stay neutral (0 rad).

We have compared our proposed approach against a random locomotion approach. Modules select a random action in every iteration and execute that. Random action strategies have been shown to provide steady performance in ATRON and M-TRAN robots [19]. The main performance metrics shown here are: distance traveled from the start point by the front module, maximum speed achieved by the configuration and number of messages passed between modules. The faint lines in the plots (Figure 4.2, 4.3, 4.4, 4.8) denote multiple runs and the bold red/blue lines indicate the best-fit line. Three variables in the Q-learning approach, α , ϵ and τ , have been set to 0.1, 0.8 and 0.9 respectively for all the tests (these values were determined experimentally). In our tests, each configuration runs Algorithm 9 for learning different types of locomotion (such as inchworm and rolling motions). Each test has been run for 30 minutes (clock-time) and 10 times each. Videos can be found here: <https://youtu.be/8YiAj5xF8ag> and

https://youtu.be/zjKkNW_r0ZI.

Results

Inchworm Locomotion: As the hand-coded gait tables for the 2-module ModRED chain configuration are already available [21], we first compare the performance (distance metric) of our proposed algorithm applied on the 2-module ModRED chain configuration against the same using the hand-coded gaits. The result is shown in Figure 4.2(a). This result shows that using our approach, the ModRED configuration was able to move further than by using hand-coded gaits. The reason for this is the hand-coded gaits were built in such a way that no part of the chain is dragged along the ground, but our proposed approach learns from all available actions - does not necessarily restrict some modules being dragged. That is why our approach performed better.

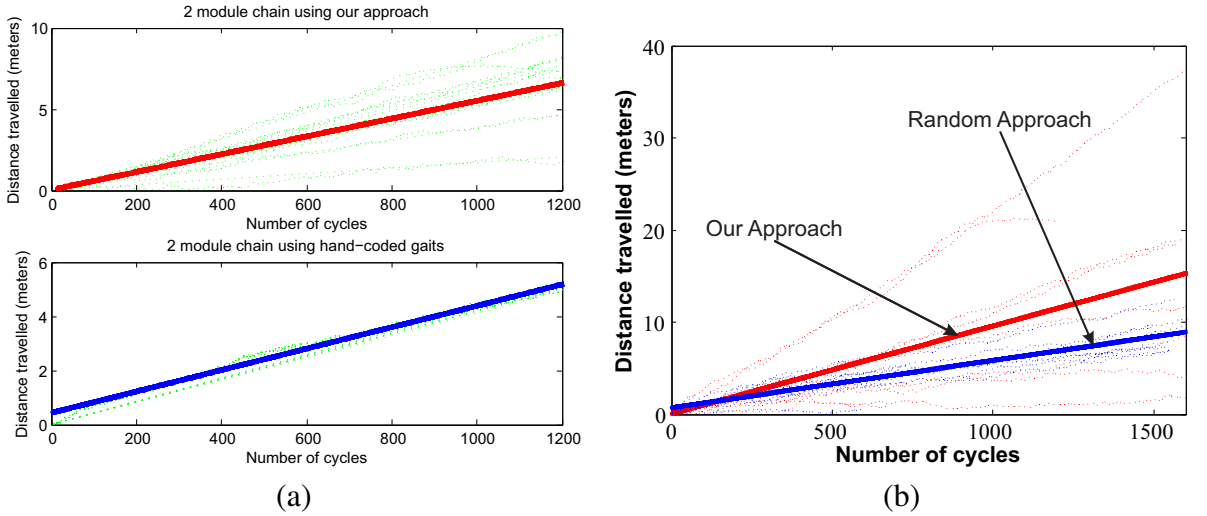


Figure 4.2: Performance comparison of our proposed approach when applied on ModRED configurations: (a) 2-module chain and against hand-coded gaits (b) 5-module chain and against the random approach.

Next we compare our algorithm's performance (distance metric) on M3, M4 and M5 against the random approach (Figure 4.2(b) and 4.3). Although in almost all of the cases, the random approach initially performed better than our approach, over time our algorithm

was able to perform better than the random approach. In some of the plots one might notice that the distance metric (y-axis) decreases sometimes. The reason for this is we have calculated the distance from the start point - not the total distance traveled. The reward is based on distance traveled (regardless of the direction). In some cases, after traveling towards a certain direction for some time, the configurations started to move towards the start direction again. This behavior of the configurations caused the dips in distance metric in some of the plots.

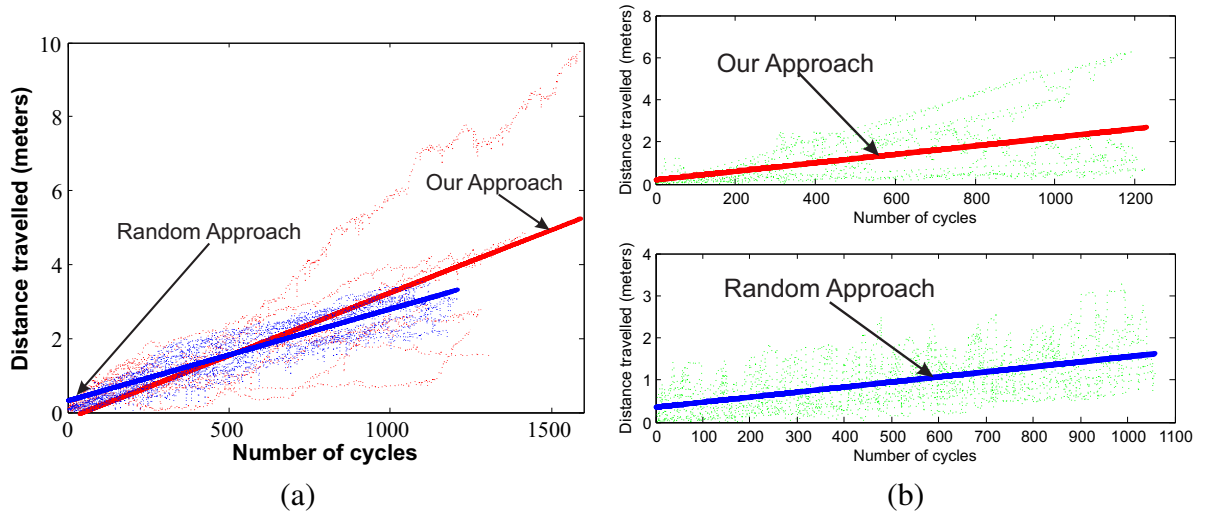


Figure 4.3: Performance comparison of our proposed approach when applied on ModRED configurations: (a) 4-module and (b) 5-module chains against the random approach.

Next we show the result of applying our proposed approach for locomotion learning in the Yamor modular robot [29]. As Yamor modules have fewer DOFs and each Yamor module can perform a very limited set of actions (only 3), we could test on longer Yamor chain configurations than we could do with ModRED within Webots. The performance (distance metric) of our proposed approach when applied on different Yamor configurations has been shown in Figure 4.4. The results show that Yamor configurations could travel longer distances than ModRED configurations. We believe the main reason for this is the smaller size and lighter Yamor modules compared to ModRED modules. Also due to larger chain sizes, we have noticed some abrupt jumps during locomotion which helped the

configurations to travel a longer distance in one learning cycle.

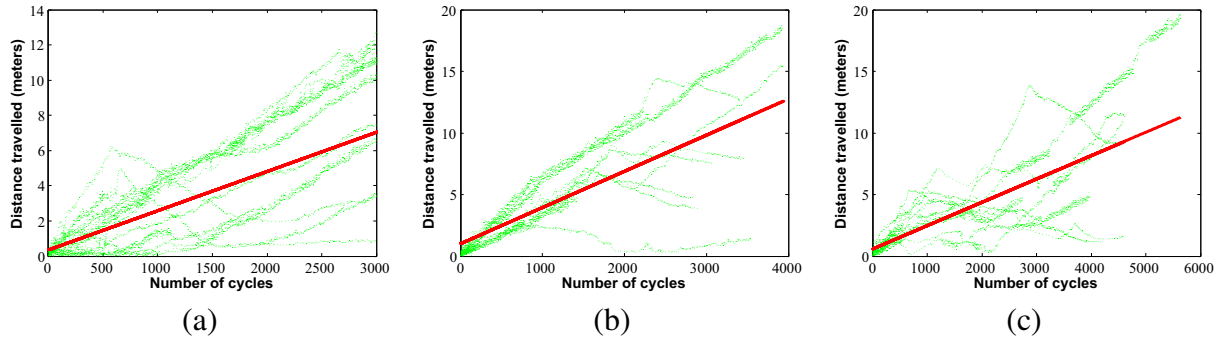


Figure 4.4: Performance of our proposed approach when applied on Yamor configurations: (a) 10-module, (b) 12-module and (c) 14-module chains.

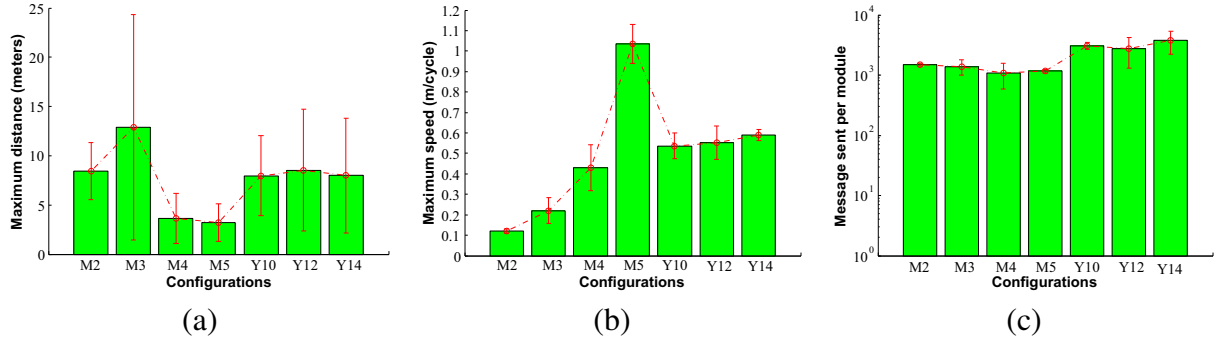


Figure 4.5: (a) Maximum distance traveled in any direction from the start location by different configurations, (b) Maximum speed achieved by different configurations and (c) Average number of messages sent by each module in different configurations.

Next we summarize the results for maximum distance traveled in any direction from the start location by different configurations (Figure 4.5(a)), maximum speed achieved (Figure 4.5(b)) and the number of messages sent by each module in different configurations (Figure 4.5(c)). As can be seen in Figure 4.5(a), maximum distance has been traveled by M3, but at the same time we can also observe a very high variance in that performance. As we have run each test for 30 minutes, in the case of (larger) ModRED configurations, M4 and M5, due to the slow simulation, we notice a low overall distance traveled. However, these configurations achieved the maximum speeds among all ModRED configurations (Figure 4.5(b)). In terms of maximum achieved speed, the Yamor configurations performed very

similarly, though we can notice a slight increase in maximum speed in Y14. This makes us believe that the longer chain sizes and corresponding (rare) abrupt jumps during inchworm locomotion were the reasons for larger configurations achieving higher maximum speed in both ModRED and Yamor. Figure 4.6 shows the snapshots of inchworm locomotion in ModRED and Yamor configurations (M3 and Y12 respectively). Green and red marks denote the start location in those pictures.

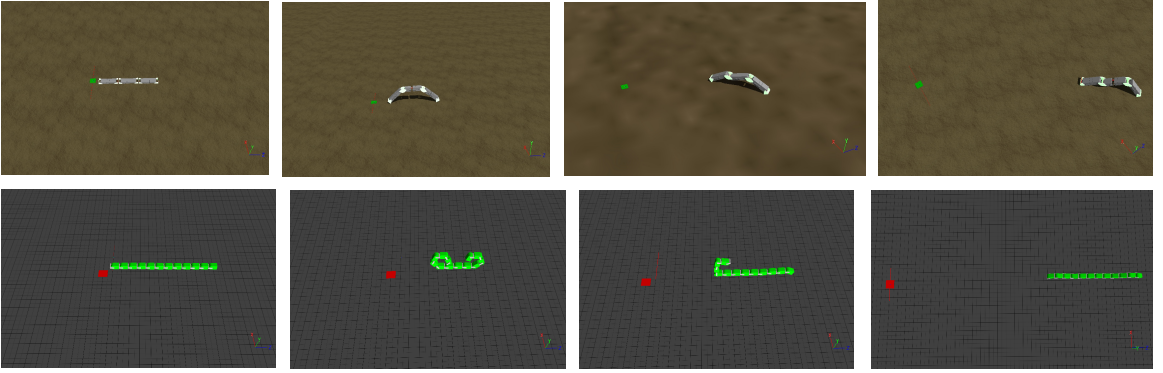


Figure 4.6: Snapshot of inchworm locomotion performed by (top) ModRED and (bottom) Yamor configurations using our proposed approach.

Fault Adaptation: We are also interested in understanding how our proposed approach adapts itself if some module becomes faulty during the operation. We have tested the fault adaptability of our approach on M3 and have performed three adaptability tests: how the algorithm performs when the middle, end or last two modules stop working (i.e., stop moving and communicating). For the first 30 minutes in the experiment, all modules were functional, and then for the next 30 minutes one of the above situations occurs. Depending on the relative position of the faulty module within the configuration, the performance of the algorithm (distance metric) varies. For example, when the middle module becomes faulty, we can see a very nominal change in the distance traveled by the configuration: the slope of the best-fit lines are almost the same (Figure 4.7(b)). But when the end module stops working (irrespective of the middle module’s faulty condition), then the total amount of distance traveled by the configuration drops drastically (Figure 4.7(a))

and 4.8(a)). But even with any type of fault, we can notice that our algorithm performs steadily and was able to adapt to any module's fault in the configuration.

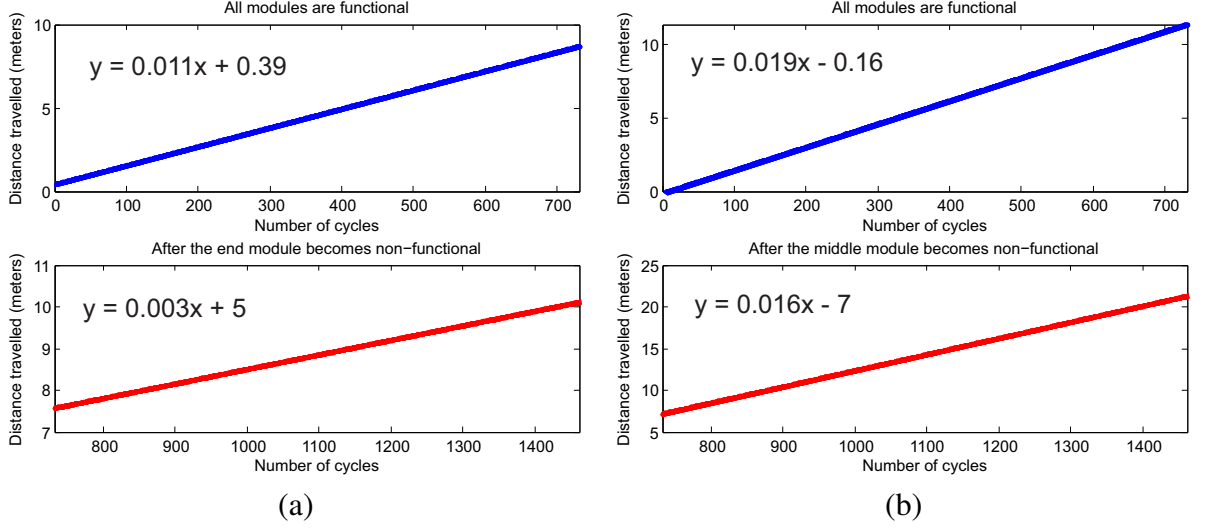


Figure 4.7: Performance of our proposed algorithm after (a) the end and (b) the middle module becomes non-operational.

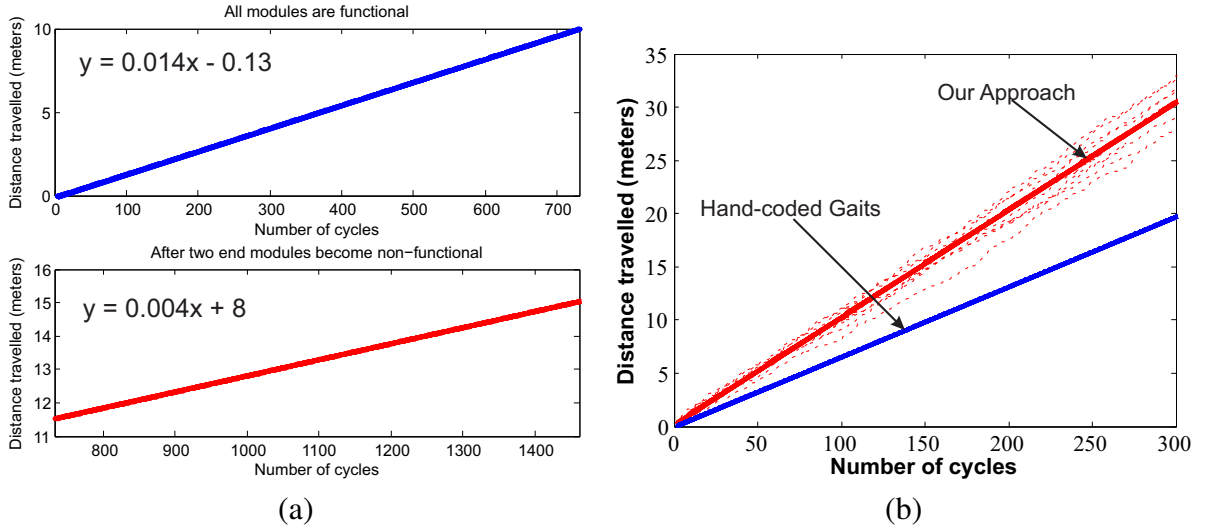


Figure 4.8: (a) Performance of our proposed algorithm after two end modules become non-operational. (b) Comparison of performance of our algorithms for rolling locomotion against the same by using hand-coded gaits.

Rolling motion of ModRED: We also present preliminary results of applying our proposed approach on a ModRED 2-module chain configuration for rolling locomotion.

We compare the performance of our approach applied on the 2-module ModRED chain for rolling motion against the hand-coded gaits for ModRED proposed in [21]. The result is shown in Figure 4.8(b). This figure shows that using our proposed approach the ModRED configuration was able to travel almost double the distance than by using the hand-coded gaits.



Figure 4.9: Snapshot of rolling locomotion by 2-module ModRED chain using our approach.

4.4 Locomotion Learning via Independent Action Learning

In our previous work, we have seen that although our approach achieves good locomotion performance, it also imposes a high communication overhead which in turn might become a drawback if the correct synchronization is not achieved. Also, our previous approach did not necessarily move the configuration in any particular direction similar to other existing approaches [19]. To solve these challenges, we have proposed a game-theoretic solution which formulates the modules' actions in every round as a normal-form game and the objective of the modules is to learn the actions in that game which helps the configuration to travel faster when executed. For the learning purpose, we use a multi-agent reinforcement learning framework. In a single-agent setting, each module executes an action in an environment and the feedback from the environment is taken into account while deciding the future actions. On the other hand, in a multi-agent learning setting,

multiple modules interact with the environment and also with each other to learn their best actions simultaneously.

Multi-agent learning is an inherently complex problem as a change in one module's best action may in turn change the best actions of all other modules in the configuration. As the modules are executing and learning their actions to achieve a common goal (i.e., movement of the whole configuration), their actions must coordinate in a sense that any individual module's local action should not obstruct the configuration from achieving a better performance. An MSR consists of multiple modules where each of them is learning to move simultaneously and each one's actions are affecting the overall locomotion performance of the configuration; that is why the MSR locomotion learning problem can be solved using a multi-agent learning technique. In this work, we bring these two domains together so that any arbitrary-shaped configuration can learn to move faster. To the best of our knowledge, this work is the first one to employ a game theory-based multi-agent learning solution for solving the locomotion learning problem in MSRs.

4.4.1 Goal Directed Reward Formulation

Each module, m_i , receives a reward for contributing towards the configuration's locomotion, denoted by $R_i(a_j, \tau)$ where a_j is an action taken at iteration τ . Each configuration is given a goal location G which is characterized by (x_G, y_G) . The objective of the modules is to learn a locomotion pattern so that they can achieve a goal-directed locomotion. In goal-directed locomotion, the whole configuration moves towards a fixed direction, unlike in [19, 38] where modules can move in any arbitrary direction. In our particular scenario, before the learning process begins, a leader module is elected [8] which calculates the distance traveled by the configuration. The leader module is elected at the beginning of the process and does not change during the mission. Each module's local reward can be calculated as a fraction of the total Euclidean distance traveled by the leader

module towards the goal since its last iteration. This can be formulated as follows:

$$R_j(a_i, \tau) = \frac{||D_{\tau-1} - D_\tau||}{N} \quad (4.3)$$

where $D_{\tau-1}$ and D_τ denote the distances between the goal location G and the leader module before and after taking the local action a_i . Each module receives $\frac{1}{N}$ -th of the reward earned by the whole configuration, which can be seen as its contribution to the goal-directed locomotion objective.

4.4.2 Normal-form Games

Definition 5 (*Normal-form game [87]*) A normal-form game is a tuple (M, A, R) , where:

- M is a finite set of N agents;
- $A = A_1 \times \cdots \times A_N$, where A_i is a finite set of actions available to the i -th agent.
- $R = (R_1, \cdots, R_N)$ where $R_i : A \rightarrow \mathbb{R}$ is a real-valued reward (or payoff). R_i is called the reward function or the payoff function for the i -th agent.

Normal-form games are one-shot interactive games among agents. In the game, all the involved agents (modules in our case) simultaneously play one action (locomotion actions in our case) and each one of them consequently receives a reward (Euclidean distance traveled in our case) for taking those actions, after which the game ends. The game is played repeatedly, but every time as a stand-alone interaction. Therefore, there is no state-transition function involved between two consecutive games played. The payoff to the agents can be visualized using an N -dimensional matrix, an example of which is shown in Table 4.3. In this example, row actions are of agent 1 and column actions are of agent 2. If both the agents play action a_2 , they both will receive a reward of 5, whereas for all other possible action combinations their earned reward is less than that. Note that, as per our reward setting (Eq. 4.3), $R_i = R_j, \forall m_i, m_j \in M$.

Within normal-form games, we are mostly interested in the cooperative normal-form games where multiple agents execute their actions for a global common goal (such as the locomotion of the configuration in our problem). From a multi-agent learning perspective, the objective of the modules is to learn the best (possibly joint) action-reward relation so that such actions are played which earn higher reward. In the normal-form game setting, each module performs an action, receives a corresponding reward and plays the game again, and by playing the game multiple times, they should learn which actions are better.

Table 4.3: A 2-player normal-form game’s payoff matrix

Actions	a_1	a_2
a_1	1	5
a_2	2	0.5

4.4.3 Independent Action Learning vs. Joint Action Learning

Two main classes of multi-agent learning approaches emerge from the literature [11]: 1) Independent Action Learning, and 2) Joint Action Learning. Independent action learning algorithms extend the single-agent learning algorithms where each agent learns its own best actions based on the interactions with other agents. As an agent in the system is solely concerned about modifying its own actions so that they match the other agents’ best actions and consequently overall reward gain increases, these algorithms scale up very easily [11]. This work employs an independent learner.

Our earlier work on locomotion learning of MSRs [38] was more geared towards the joint-action learning. These algorithms are inherently more complex than the independent action learning algorithms, as they try to learn the best joint-action pattern for all the agents involved. As can be understood, even with a small set of ten agents and each agent having only ten actions, the number of possible joint-actions reaches an astronomical value ($|A|^N = 10$ billion) which in turn becomes very difficult to learn (both time and space complexities grow exponentially). Even though these algorithms have higher complexities,

they usually provide better performance guarantees. Another drawback of this approach is that the agents need to continuously communicate to be updated about other agents' actions and then they need to synchronize those actions which is a very difficult task to accomplish in modular robotic systems. Even though these algorithms come with these drawbacks, they are proven to provide better means of coordination among the agents and consequently better performance guarantees.

4.4.4 Game Theoretic Solution Using Independent Q-Learner

This solution approach is modeled as a normal-form game where each module independently learns the best action-reward structure of the game using a Q-learning technique. Q-learning, a popular form of reinforcement learning technique, has been used before for locomotion learning in MSRs [19, 38]. A Q-learner maintains a data-structure containing the Q-values for each action in the library. The Q-value provides an estimate of how useful one action is. After an action, a_i , is performed by module m_j and its corresponding reward is received following the normal-form game structure, the Q-value, $Q_j(a_i)$, of that action, a_i , is updated using the following equation:

$$Q_j(a_i) \leftarrow Q_j(a_i) + \alpha(R_j(a_i, \tau) - Q_j(a_i)) \quad (4.4)$$

Note that the Q-value of any action is local to a module, i.e., the estimate of how good an action is can be different for different modules. $\alpha \in [0, 1]$ is the learning rate in Eq. 4.4. As normal-form games do not have a transition function, an agent's payoff function is just a probability distribution over its actions [54].

Q-learning is guaranteed to converge to the optimal solution in the case of a single-agent scenario. But in the case of our multi-agent setting, convergence to an optimal solution is not always guaranteed, and the action selection strategy plays an important role in convergence. Two main classes of action selection strategies can be found in the

literature, as follows:

(Greedy Selection) In this strategy, the best action seen so far, i.e., the action which yields maximum expected reward, is exploited heavily. Therefore, each module performs the best action a_{best} with a high probability (e.g., $> 90\%$). On the other hand, other actions are also explored from time to time to find out whether there is any action available or not besides the best action which can yield better reward than the best action. Mathematically, the probability of any action selection ($pr(a_i)$) can be determined using the following equation [54]:

$$pr(a_i) = \begin{cases} \epsilon & \text{if } a_i = a_{best} \\ 1 - \epsilon & \text{otherwise} \end{cases} \quad (4.5)$$

(Boltzmann Selection) In this strategy [54], the probability of any action is calculated as:

$$pr(a_i) = \frac{e^{\frac{ER(a_i)}{T(\tau)}}}{\sum_{a \in A} e^{\frac{ER(a)}{T(\tau)}}} \quad (4.6)$$

where $ER(a_i)$ denotes the expected reward of action a_i (see the next section for details) and $T(\tau)$ is the temperature variable at any learning iteration τ which controls the exploration and exploitation ratio. We start with a very high value of T which motivates the modules to explore the un-attempted actions. With time, the value of T is decreased. As after several phases of exploration, best actions are most likely to emerge, therefore the weight of exploration is turned down.

Our proposed strategy uses both the greedy and Boltzmann action selection strategies depending on the value of T (lines 5 – 8 in Algorithm 10). We initialize T with a very high value (MAX_T) and exponentially decrease the value of T until it reaches a certain lower limit (MIN_T). The following equation has been used to calculate the value of T at any iteration τ [57]:

$$T(\tau) = e^{-s\tau} MAX_T + 1 \quad (4.7)$$

where s controls the decay parameter. $+1$ in the above equation makes sure that $ER(\cdot)$ does not get divided by a 0 in Equation 4.6. Once the temperature reaches the lowest limit (i.e., MIN_T), we know that the modules have done enough exploration of actions and they are more certain about which actions are better than the others (in terms of how much distance the configuration can travel by executing them). Therefore, at that point (i.e., when $T < MIN_T$), we switch our action selection strategy to the greedy strategy from the initial Boltzmann strategy. As the modules have better idea about the ‘good’ actions now, therefore they exploit these ‘good’ actions more rather than exploring other actions.

Expected Reward and Heuristic

In this work, we use the *Frequency Maximum Q-value* (FMQ) Heuristic that has been proposed in [57] for calculating the expected reward of any action. The FMQ heuristic follows a similar idea as the optimistic assumption which has been proposed in [62] for multi-agent learning. The optimistic assumption idea implies that an agent will always take the best action expecting that other agents will also play their best actions accordingly. For example, in the normal-form game shown in Table 4.3, agent 1 should always take action a_1 provided agent 2 is playing action a_2 - and over time, the optimal joint action (a_1, a_2) will emerge even in an independent learner. On the other hand, if both the agents take action a_2 , then they will both receive a very low reward.

In classical single-agent Q-learning, as soon as a low-reward joint-action (a_2, a_2) is played, the Q-value of a_2 is updated (e.g., Q-value goes down) even though the optimal joint action (a_1, a_2) contains a_2 as agent 2’s best action. To alleviate this problem, the FMQ heuristic proposes that even though in one round one particular action is yielding lower reward, that might not necessarily be the fault of that agent’s action. As this is a multi-agent scenario, even if one agent is taking its optimal action, other agents’ worse action choices may harm the overall reward yield. That is why the expected reward of any action a_i is changed based on the maximum reward earned by that action so far, $R_{max}(a_i)$,

and how many times the maximum reward has been earned by playing that action, i.e., the frequency of $R_{max}(a_i)$, denoted by $f(R_{max}(a_i))$. Formally, the expected reward, $ER(a_i)$, can be calculated using the following equation [57]:

$$ER(a_i) = Q(a_i) + w * f(R_{max}(a_i)) * R_{max}(a_i) \quad (4.8)$$

where w is a weight that determines how influential the FMQ heuristic is in the action selection strategy. This calculated expected reward value is then used in Eq. 4.6. Note that, if $w = 0$, then FMQ heuristic actually boils down to the classical Q-learning algorithm, where $ER(a_i) = Q(a_i)$. As according to our strategy, each module is exploring different actions in the beginning, enough such exploration increases the probability that the optimal joint-action will be encountered once by the modules. The overall locomotion learning procedure is shown in Algorithm 10.

Algorithm 10: Multi-agent Q-Learning Based Locomotion Learning Algorithm

```

1  $\tau \leftarrow 0$ .
2 Loop
3    $\tau \leftarrow \tau + 1$ .
4   Update  $T(\tau)$  using Eq. 4.7.
5   if  $T(\tau) > MIN\_T$  then
6     select an action  $a_j$  using the Boltzmann action selection strategy (Eq. 4.6)
7   else
8     select an action  $a_j$  using the greedy action selection strategy (Eq. 4.5)
9   Perform  $a_j$  and receive the reward,  $R_i(a_j, \tau)$ .
10  Update  $Q(a_j)$  and  $ER(a_j)$ .
```

4.4.5 Experimental Evaluation

Settings

We have implemented our proposed game theory-based locomotion learning strategy on simulated ModRED and Yamor modules within the Webots robot simulator. We have tested our approach on three different types of configurations - 1) configurations where modules

have only inchworm motion, 2) configurations where modules have only rolling motion, and 3) configurations where some modules have inchworm motion and the others have only rolling motion (called hybrid). Three tested configurations are shown in Fig. 4.10. ‘M’ or ‘Y’ indicates if the configuration is formed by ModRED or Yamor modules. The number indicates how many modules are present in that configuration. ‘I’, ‘R’, and ‘H+’ indicate that the configuration follows inchworm, rolling or hybrid motion. In the case of inchworm or rolling motion, modules are always connected in a chain configuration.²

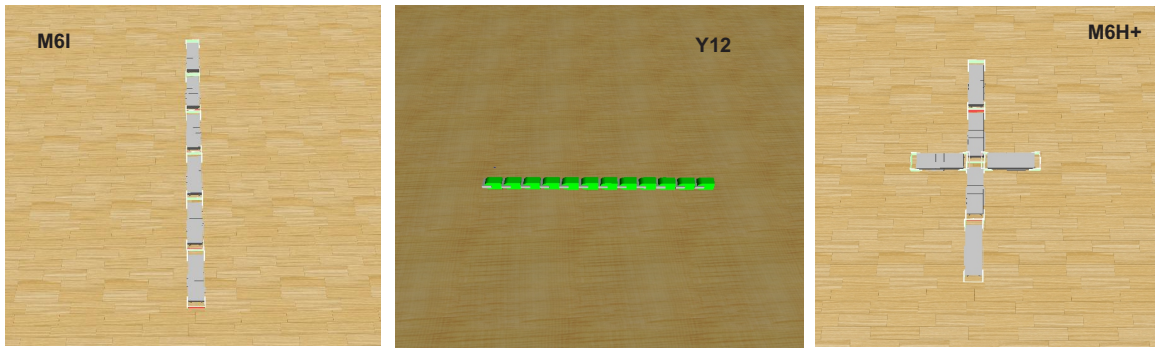


Figure 4.10: Different configurations used for our experiments. Snapshots are captured within Webots simulator.

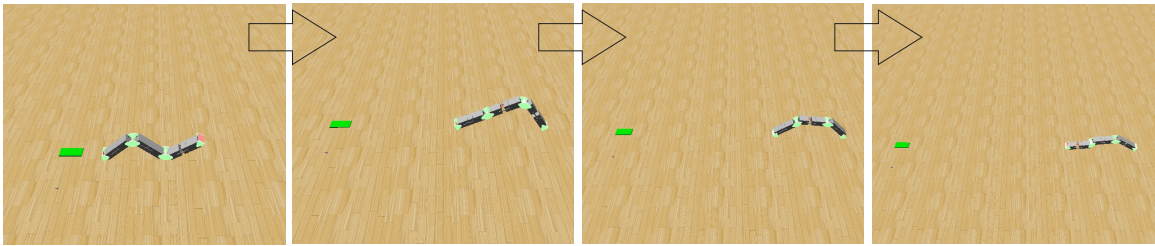


Figure 4.11: Webots snapshot of inchworm locomotion performed by a 3-module ModRED chain.

For our algorithm, α is set to 0.9 and 0.1 for the Boltzmann and the greedy action selection strategies respectively. The decay parameter s is set to 0.01, and MAX_T and w are set to 500 and 5 respectively. As only the leader module sends one message in every learning cycle, and all other modules only receive that, therefore the

²Because each module has 4 DOF, testing with ModRED modules becomes computationally intensive with more than 6 modules. Testing with larger configurations is reported for a 1-DOF robot called Yamor.

communication complexity is constant ($O(1)$) and does not depend on N . We have compared the performance of our approach against the single-agent Q-learning (SAQL) approach proposed in [19]. This approach employs a Q-learning algorithm on each of the modules in the configuration and a greedy action selection strategy is used. For this case, α and ϵ are set to 0.1 and 0.9 respectively. Each test case has been run for 30 minutes and results have been averaged over 5 runs.

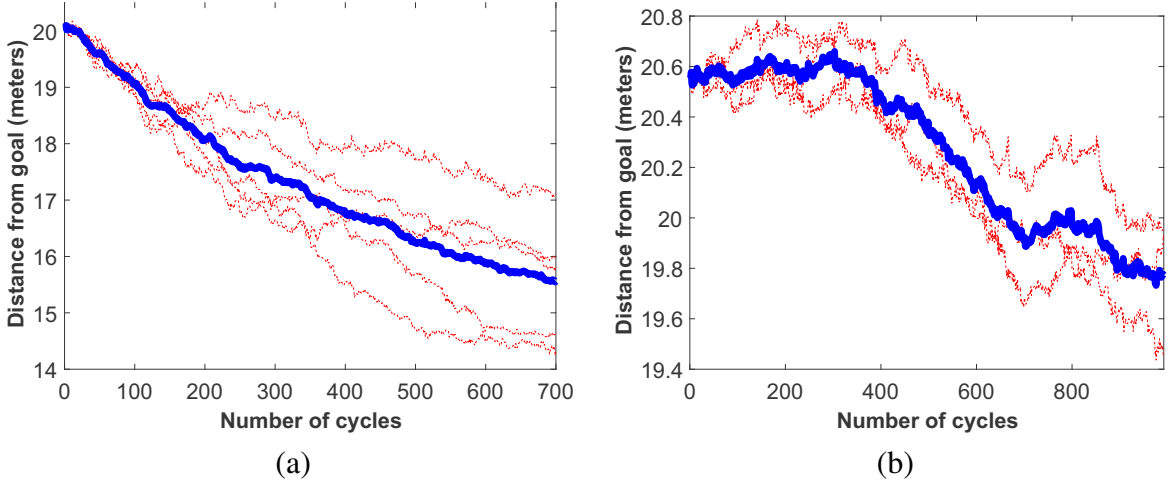


Figure 4.12: Change in distance from goal over time for configurations a) M3I and b) Y12. The faint lines denote multiple runs and the bold blue line indicates the average line.

Results

A 3-module ModRED chain's locomotion towards the goal direction using our proposed approach has been shown in Figure 4.11. Next, we discuss our quantitative results. The faint lines in the plots (Fig. 4.12, 4.14, 4.15) denote multiple runs and the bold blue lines indicate the average line. First, we show that for different chain configurations, the distance to goal changes with time. This result is shown in Fig. 4.12. We show this result for 3-module ModRED and 12-module Yamor chains. Because of smaller chain size and consequently much smaller set of possible action sequences (1000 compared to 531441) to learn from, M3I could travel more distance than Y12 in 30 mins. time and it achieved 2.17

times higher average speed than Y12. Note that the SAQL approach does not necessarily push the configuration towards a specific direction; rather the configuration can move in any direction.

Comparison with a single-agent Q-learner: Next, we compare another performance metric, average speed achieved by the configurations, against the SAQL approach. The result is shown in Fig. 4.13. Six different configurations with different locomotion patterns have been tested. It can be seen that in almost all of the cases our proposed approach outperforms the SAQL approach. In the case of the M6I configuration, our approach performs 7.86 times better than the SAQL approach. One interesting thing we have observed here is that if each module's action library is small in size (e.g., 3 actions for Yamor modules and 5 actions for ModRED rolling), then SAQL performs better than our approach. But when the number of available actions is large (e.g., 10 in ModRED inchworm motion), then the SAQL approach is outperformed. Even then, our proposed approach achieved 88% and 72% of the average speed achieved by the SAQL approach for Y12 and M2R respectively. On average, across all 6 tested configurations, our proposed approach achieved 2.82 times higher speed than the SAQL approach.

Fault tolerance: Finally, we test the fault-tolerant nature of our proposed approach. We also wanted to more precisely pinpoint which modules are more important to the configuration in terms of their contribution towards the locomotion of the configuration. To do this, we have implemented two different test cases on a 3-module ModRED chain (M3I) which has a unique middle and end module. For this experiment, we let the configuration follow our algorithm for the first 5 mins. with all modules working. Next, we disable either the end or the middle module and let the configuration move for 25 mins. and then we stop. We study how the distance traveled by the configuration changes with and without module failures.

In the first test case, we disable the end module. The result is shown in Fig. 4.14. The best linear-fit line's equation is also shown. This slope of the best-fit line shows that

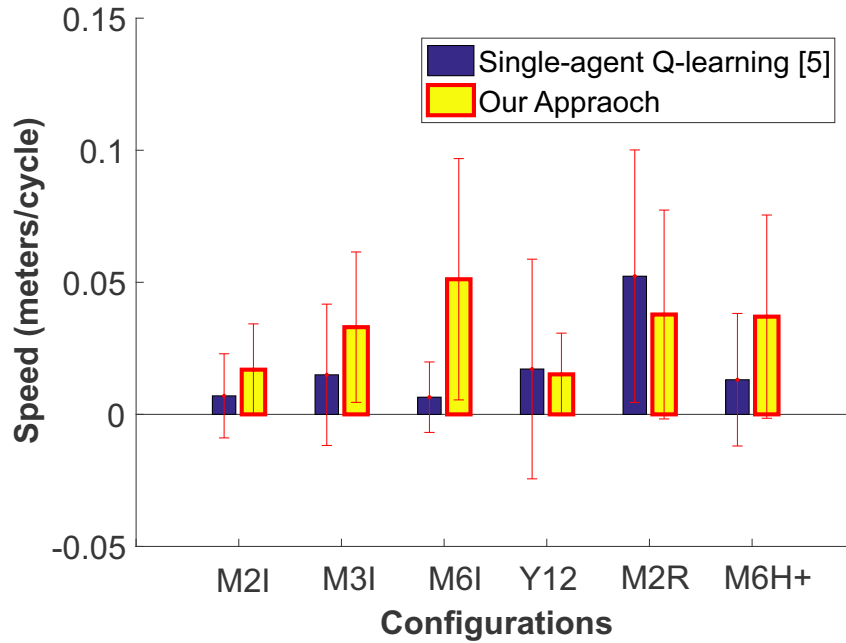


Figure 4.13: Average speeds achieved by different configurations along with standard deviations. Comparison against the single-agent Q-learning based (SAQL) adaptive locomotion work is shown [19].

even though the performance of the configuration was affected by the module failure, using our proposed strategy, the other two modules could adapt their locomotion patterns and therefore the configuration continued to move towards the goal direction.

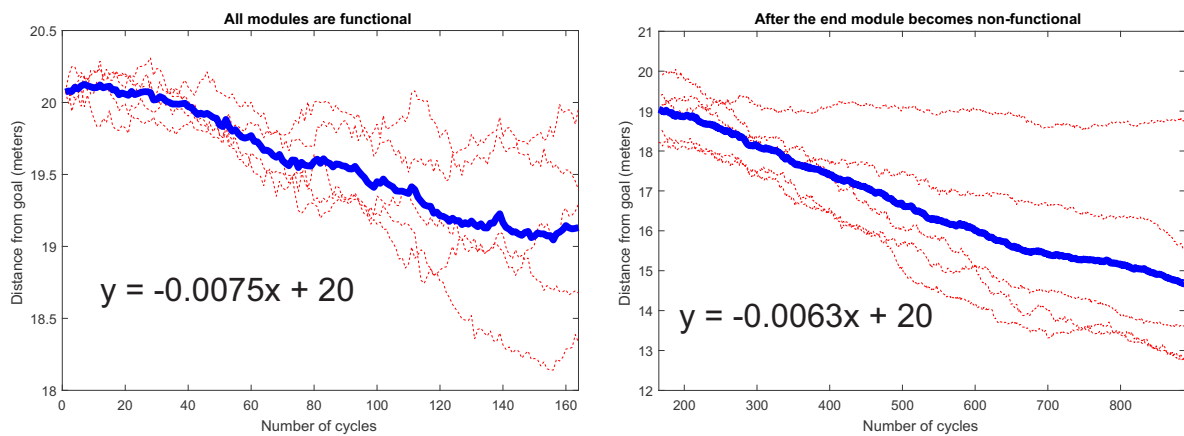


Figure 4.14: Comparison of configuration's performances before and after the failure of the end module.

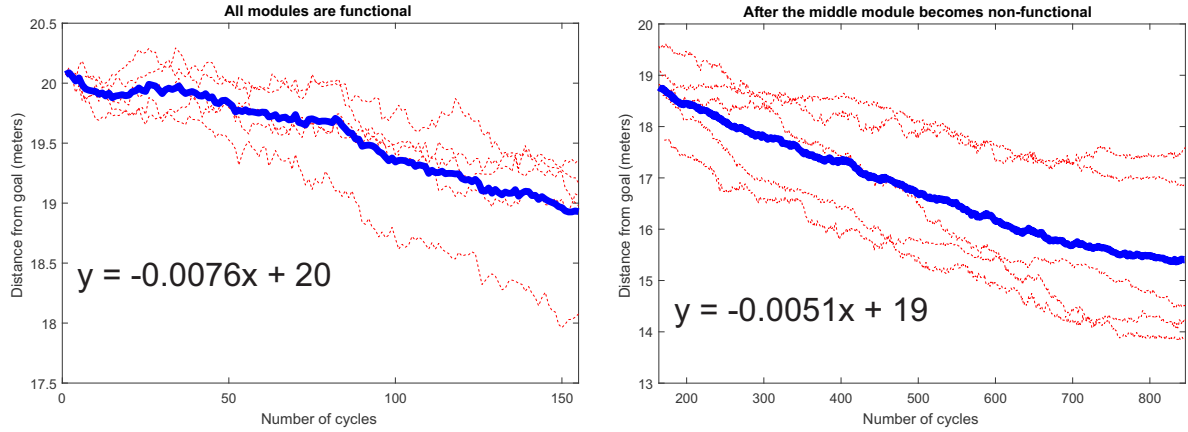


Figure 4.15: Comparison of configuration's performances before and after the failure of the middle module.

In the second test case, we disable the middle module. The result is shown in Fig. 4.15. The performance of the configuration before the module failure is pretty much similar to that of the previous case. But the slope of the best-fit line reduces a little from the case when the end module failed. This phenomenon can be explained by the fact that as the modules do not have any explicit coordination among them, it was easier for them to continue to pull the end module when it stopped working, but on the other hand when the middle module stopped working, then it stopped pulling the end module. But still, using our approach the whole configuration continued to move towards the goal direction.

4.5 Discussions

We have proposed two different approaches to solve the adaptive locomotion learning problem in MSRs. Although the two proposed solutions are very different in their approaches to solve the problem, a common theme has been used in both the cases: as one module's bad action can lead to worse performance of the whole configuration even if the other modules are taking better actions, each module needs to take this inter-module action-relation into account.

To do this, both joint-action and independent action learners have been proposed. As can be imagined, the joint-action learner needs more communication among the modules to get the information about all the neighboring modules' actions. This communication overhead has been mitigated in the independent learning approach. Also, using the joint learner, each module needs to maintain a complicated data structure of all the best joint-action sequences for all of its own actions which might be difficult to maintain if there are more actions available to each of the modules. On the other hand, the independent learner does not need to save any other modules' information which helps it to run using a very low memory space.

Both of our proposed approaches are fault-tolerant in nature. If one or more modules become faulty during the MSR's mission, then it has been empirically shown that the configurations were still able to move forward by adapting the locomotion pattern using our proposed approaches. This characteristic adds more robustness to our solutions. Moreover, our independent learning approach was able to push the tested configurations towards a specific goal direction unlike the joint learner and other existing approaches [19]. In the future, it would be interesting to see how the change in the size of the neighborhood of each module from which it is learning the best joint-action, affects the performance of the configuration in terms of speed and distance traveled. Currently, we are implementing these two proposed approaches on real ModRED hardware. Our goal is to test these algorithms on a 2-module ModRED chain and observe the difference in performance with the simulation results.

Chapter 5

Conclusion

5.1 Summary

In our research, we have proposed several algorithms for solving three different types of problems in MSRs which involved decision making, planning, and learning. We have solved three very important and fundamental problems in MSRs, namely, partitioning of modules, configuration formation planning and adaptive locomotion learning.

Our proposed distributed configuration formation problem not only forms the configuration from singleton modules, it also offers a technique which is able handle the situation where the modules can be in any arbitrary configuration in the beginning. To the best of our knowledge, we are the first ones to solve the configuration formation problem where the modules can start as a part of already connected arbitrary shaped configurations. Most of the previous works on this topic cannot be generalized to all types of MSRs. Our work mainly aims to generalize the configuration formation in MSRs by proposing algorithmic solutions which do not depend on the characteristics of the MSR platform used.

We have also proposed a novel problem, named simultaneous configuration formation and information collection problem, where initially randomly distributed singleton modules

plan the paths to the spots in the target configuration in such a way that the paths are maximally informative. This work is a novel approach which merges two different existing problems – information collection in multi-robot systems and configuration formation in modular robots. Our work is also the first one to solve this novel problem.

For locomotion learning purposes, we have proposed a reinforcement learning based adaptive locomotion learning strategy which learns from the module’s past actions as well as from the correlation between its own actions with its neighboring modules’ actions. As modules are physically connected with each other, therefore one module’s action selection strategy impacts the overall locomotion performance of the configuration. Almost none of the related works on this topic took this inter-module action-relation into account. We have also proposed a game theoretic multi-agent learning approach for adaptive locomotion learning in MSRs. To the best of our knowledge, this work is one of the first to address the issue and solve it using multi-agent learning where each module learns the best sequence of actions among all the modules by playing a normal-form game. At the same time, modules only spend a constant communication cost to achieve the goal.

5.2 Future Directions

We now present some future directions of our research.

- **Path planning in MSRs:** Path planning is one of the most important problems for any mobile robot – how to move from point A to point B by covering the least distance. As we have already solved the locomotion learning problem in MSRs, the next natural step for us would be to solve the path planning problem. We plan to use a sampling-based path planning approach, e.g., PRM or RRT, as the base of our solution approach. We also plan to use a bipartite graph matching based coordination strategy similar to what has been proposed in one of our recent works [33].
- **Information collection using MSRs:** MSRs are usually sent where humans cannot

go and we expect the modules to collect as much information as possible for us. If different modules can be equipped with different sensors then they can actually collect different types of information. But that also brings challenges like distribution of the area among the modules, communication constraints, and redundant information collection. We plan to extend our current information collection approach for heterogeneous sensor-equipped modules.

- **Self-repair of MSR modules:** Animals like lizards can reproduce and/or repair their body parts such as tails if that is harmed. We plan to solve the self-repair problem in MSRs by getting inspired from nature. As we have discussed earlier, one or more modules can become faulty during the MSR's mission. Therefore, it would be very useful for the configuration if it can get rid of the faulty module(s) and attach working modules in its place. We plan to solve this problem in future.
- **Self-disassembly of MSR configuration:** We have solved the problem where multiple modules need to come together and physically attach to form a specific shape/configuration. But it is also very common in nature to reach a shape by getting rid of different body parts. We plan to look at this problem next where a larger configuration needs to get rid of some of its constituting modules to reach the target shape. This is a reverse self-assembly problem, known as the self-disassembly problem.
- **Other applications:** We also plan to use MSRs for some real-world applications such as object manipulation. One example of object manipulation is an MSR configuration pushing an object towards a specific direction. If solved, this can be used for aged persons or patients at home who cannot move much otherwise. The MSR configuration can be used to help them get objects. Along similar lines, MSRs can be used for monitoring of environments or human activity.

5.3 Remarks

In this dissertation, we have introduced a series of algorithms which solve three fundamental problems in modular self-reconfigurable robots. The solutions presented here have a broad range of applications including exploration, extra-terrestrial applications, and information collection. Our algorithms have been implemented in simulation and also have been proved to be feasible to be deployed on ModRED hardware. We plan to extend these solutions to solve problems like path planning and information collection using modular self-reconfigurable robots.

Bibliography

- [1] H. Ahmadzadeh and E. Masehian. Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization. *Artificial Intelligence*, 223:27–64, 2015.
- [2] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
- [3] T. Akutsu. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 76(9):1488–1493, 1993.
- [4] J. Alonso-Mora, A. Breitenmoser, M. Rufli, R. Siegwart, and P. Beardsley. Multi-robot system for artistic pattern formation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4512–4517. IEEE, 2011.
- [5] G. E. Andrews. *The theory of partitions*. Number 2. Cambridge university press, 1998.
- [6] M. Asadpour, A. Sproewitz, A. Billard, P. Dillenbourg, and A. J. Ijspeert. Graph signature for self-reconfiguration planning. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 863–869. IEEE, 2008.
- [7] J. Baca, S. Hossain, P. Dasgupta, C. Nelson, and A. Dutta. Modred: Hardware design and reconfiguration planning for a high dexterity modular self-reconfigurable robot for extra-terrestrial exploration. *Robotics and Autonomous Systems*, 62(7):1002–1015, 2014.
- [8] J. Baca, B. Woosley, P. Dasgupta, A. Dutta, and C. Nelson. Coordination of modular robots by means of topology discovery and leader election: Improvement of the locomotion case. In *Distributed Autonomous Robotic Systems*, pages 447–458. Springer, 2016.
- [9] J. Baca, B. Woosley, P. Dasgupta, and C. Nelson. Real-time distributed configuration discovery of modular self-reconfigurable robots. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 1919–1924. IEEE, 2015.
- [10] D. P. Bertsekas. The auction algorithm for assignment and other network flow problems: A tutorial. *Interfaces*, 20(4):133–149, 1990.

- [11] D. Bloembergen, K. Tuyls, D. Hennes, and M. Kaisers. Evolutionary dynamics of multi-agent learning: A survey. *J. Artif. Intell. Res.(JAIR)*, 53:659–697, 2015.
- [12] M. Bowling and M. Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.
- [13] U. Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.
- [14] D. Bucher, G. Haspel, J. Golowasch, and F. Nadim. Central pattern generators. *eLS*, 2000.
- [15] Z. Butler and D. Rus. Distributed motion planning for 3d modular robots with unit-compressible modules. In *Algorithmic Foundations of Robotics V*, pages 435–451. Springer, 2004.
- [16] A. Castano, A. Behar, and P. M. Will. The conro modules for reconfigurable robots. *Mechatronics, IEEE/ASME Transactions on*, 7(4):403–409, 2002.
- [17] S. Chien. The eo-1 autonomous sciencecraft and prospects for future autonomous space exploration. 2005.
- [18] G. Chirikjian, A. Pamecha, and I. Ebert-Uphoff. Evaluating efficiency of self-reconfiguration in a class of modular robots. *Journal of robotic systems*, 13(5):317–338, 1996.
- [19] D. J. Christensen, U. P. Schultz, and K. Stoy. A distributed strategy for gait adaptation in modular robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2765–2770. IEEE, 2010.
- [20] D. J. Christensen, U. P. Schultz, and K. Stoy. A distributed and morphology-independent strategy for adaptive locomotion in self-reconfigurable modular robots. *Robotics and Autonomous Systems*, 61(9):1021–1035, 2013.
- [21] K. D. Chu, S. Hossain, and C. Nelson. Design of a four-dof modular self-reconfigurable robot with novel gaits. In *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 747–754. American Society of Mechanical Engineers, 2011.
- [22] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.
- [23] N. Correll and A. Martinoli. Modeling and designing self-organized aggregation in a swarm of miniature robots. *The International Journal of Robotics Research*, 30(5):615–626, 2011.
- [24] A. Crespi, D. Lachat, A. Pasquier, and A. J. Ijspeert. Controlling swimming and crawling in a fish robot using a central pattern generator. *Autonomous Robots*, 25(1-2):3–13, 2008.

- [25] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.
- [26] A. K. Das, R. Fierro, V. Kumar, J. P. Ostrowski, J. Spletzer, and C. J. Taylor. A vision-based formation control framework. *Robotics and Automation, IEEE Transactions on*, 18(5):813–825, 2002.
- [27] P. Dasgupta, V. Ufimtsev, C. Nelson, and S. M. G. Mamur. Dynamic reconfiguration in modular robots using graph partitioning-based coalitions. In *AAMAS*, pages 121–128. Valencia, Spain, 2012.
- [28] S. Datta, A. Dutta, S. G. Chaudhuri, and K. Mukhopadhyaya. Circle formation by asynchronous transparent fat robots. In *Distributed Computing and Internet Technology*, pages 195–207. Springer, 2013.
- [29] J. Dietsch, R. Moeckel, C. Jaquier, K. Drapel, E. Dittrich, A. Upegui, and A. Jan Ijspeert. Exploring adaptive locomotion with yamor, a novel autonomous modular robot with bluetooth interface. *Industrial Robot: An International Journal*, 33(4):285–290, 2006.
- [30] A. Dutta. Self-assembly in heterogeneous multi-agent system using constrained matching algorithm. In *Web Intelligence (WI), 2016 IEEE/WIC/ACM International Conference on*, pages 351–358. IEEE, 2016.
- [31] A. Dutta, S. G. Chaudhuri, S. Datta, and K. Mukhopadhyaya. Circle formation by asynchronous fat robots with limited visibility. In *Distributed Computing and Internet Technology*, pages 83–93. Springer, 2012.
- [32] A. Dutta and P. Dasgupta. Simultaneous configuration formation and information collection by modular robotic systems. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5216–5221. IEEE, 2016.
- [33] A. Dutta and P. Dasgupta. Bipartite graph matching-based coordination mechanism for multi-robot path planning under communication constraints. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017.
- [34] A. Dutta, P. Dasgupta, J. Baca, and C. Nelson. A block partitioning algorithm for modular robot reconfiguration under uncertainty. In *Mobile Robots (ECMR), 2013 European Conference on*, pages 255–260. IEEE, 2013.
- [35] A. Dutta, P. Dasgupta, J. Baca, and C. Nelson. A bottom-up search algorithm for dynamic reformation of agent coalitions under coalition size constraints. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 329–336. IEEE, 2013.
- [36] A. Dutta, P. Dasgupta, J. Baca, and C. Nelson. searchucsg: a fast coalition structure search algorithm for modular robot reconfiguration under uncertainty. *Robotica*, 32(2):225–244, 2014.

- [37] A. Dutta, P. Dasgupta, and C. Nelson. A bottom-up search algorithm for size-constrained partitioning of modules to generate configurations in modular robots. In *Web Intelligence*, volume 14, pages 67–82. IOS Press, 2016.
- [38] A. Dutta, P. Dasgupta, and C. Nelson. Distributed adaptive locomotion learning in modred modular self-reconfigurable robot. In *International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 1–13, 2016.
- [39] A. Dutta, R. Dasgupta, J. Baca, and C. Nelson. Spanning tree partitioning approach for configuration generation in modular robots. In *The Twenty-Eighth International Flairs Conference*, 2015.
- [40] F. Enner, D. Rollinson, and H. Choset. Motion estimation of snake robots in straight pipes. In *International Conference on Robotics and Automation*, pages 5148–5153, 2013.
- [41] R. Fitch and Z. Butler. Million module march: Scalable locomotion for large self-reconfiguring robots. *The International Journal of Robotics Research*, 27(3-4):331–343, 2008.
- [42] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.
- [43] R. Groß, M. Bonani, F. Mondada, and M. Dorigo. Autonomous self-assembly in swarm-bots. *Robotics, IEEE Transactions on*, 22(6):1115–1130, 2006.
- [44] C. Guestrin, A. Krause, and A. P. Singh. Near-optimal sensor placements in gaussian processes. In *Proceedings of the 22nd international conference on Machine learning*, pages 265–272. ACM, 2005.
- [45] H. Hamann, J. Stradner, T. Schmickl, and K. Crailsheim. A hormone-based controller for evolutionary multi-modular robotics: From single modules to gait learning. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.
- [46] I. A. Hiskens. Stability of hybrid system limit cycles: Application to the compass gait biped robot. In *IEEE Conference on Decision and Control*, volume 1, pages 774–779. IEEE; 1998, 2001.
- [47] G. Hitz, A. Gotovos, F. Pomerleau, M.-E. Garneau, C. Pradalier, A. Krause, and R. Y. Siegwart. Fully autonomous focused exploration for robotic environmental monitoring. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 2658–2664. IEEE, 2014.
- [48] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

- [49] S. Hossain, C. Nelson, K. D. Chu, and P. Dasgupta. Kinematics and interfacing of modred: A self-healing capable, four-dof modular self-reconfigurable robot. *JMR*, 13:1256, 2014.
- [50] S. Hossain, C. Nelson, and P. Dasgupta. Rogensid: A rotary plate genderless single-sided docking mechanism for modular self-reconfigurable robots. In *ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages V06BT07A011–V06BT07A011. American Society of Mechanical Engineers, 2013.
- [51] F. Hou and W.-M. Shen. On the complexity of optimal reconfiguration planning for modular reconfigurable robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2791–2796. IEEE, 2010.
- [52] F. Hou and W.-M. Shen. Graph-based optimal reconfiguration planning for self-reconfigurable robots. *Robotics and Autonomous Systems*, 62(7):1047–1059, 2014.
- [53] A. J. Ijspeert. Central pattern generators for locomotion control in animals and robots: a review. *Neural Networks*, 21(4):642–653, 2008.
- [54] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.
- [55] A. Kamimura, H. Kurokawa, E. Yoshida, S. Murata, K. Tomita, and S. Kokaji. Automatic locomotion design and experiments for a modular robotic system. *Mechatronics, IEEE/ASME Transactions on*, 10(3):314–325, 2005.
- [56] A. Kamimura, H. Kurokawa, E. Yoshida, K. Tomita, S. Kokaji, and S. Murata. Distributed adaptive locomotion by a modular robotic system, m-tran ii. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2370–2377. IEEE, 2004.
- [57] S. Kapetanakis and D. Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. *AAAI/IAAI*, 2002:326–331, 2002.
- [58] E. Klavins. Programmable self-assembly. *Control Systems, IEEE*, 27(4):43–56, 2007.
- [59] J. Kobler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012.
- [60] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.
- [61] H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo, and S. Murata. Distributed self-reconfiguration of m-tran iii modular robotic system. *The International Journal of Robotics Research*, 27(3-4):373–386, 2008.

- [62] M. Lauer and M. Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.
- [63] M. A. Lewis, A. H. Fagg, and G. Bekey. Genetic algorithms for gait synthesis in a hexapod robot. *Recent trends in mobile robots*, pages 317–331, 1994.
- [64] M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, volume 157, pages 157–163, 1994.
- [65] K. H. Low, J. Chen, J. M. Dolan, S. Chien, and D. R. Thompson. Decentralized active robotic exploration and mapping for probabilistic field classification in environmental sensing. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 105–112. International Foundation for Autonomous Agents and Multiagent Systems, 2012.
- [66] Q. E. K. Mamun, S. M. Masum, and M. A. R. Mustafa. Modified bully algorithm for electing coordinator in distributed systems. *WSEAS Transactions on Computers*, 3(4):948–953, 2004.
- [67] P. Meier. Variance of a weighted mean. *Biometrics*, 9(1):59–73, 1953.
- [68] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.
- [69] R. Myerson. *Game Theory: Analysis of Conflict*. Cambridge, Massachusetts: Harvard University Press, 1997.
- [70] C. Nelson. A framework for self-reconfiguration planning for unit-modular robots. *Ph.D. Dissertation*, 2005.
- [71] J. Nutt, C. Marsden, and P. Thompson. Human walking and higher-level gait disorders, particularly in the elderly. *Neurology*, 43(2):268–268, 1993.
- [72] J. OShea, Z. Bandar, and K. Crockett. Systems engineering and conversational agents. In *Intelligence-Based Systems Engineering*, pages 201–232. Springer, 2011.
- [73] T. Rahwan and N. Jennings. An improved dynamic programming algorithm for coalition structure generation. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1417–1420, 2008.
- [74] T. Rahwan, S. Ramchurn, N. Jennings, and A. Giovannucci. An anytime algorithm for optimal coalition structure generation. *J. Artif. Intell. Res. (JAIR)*, 34:521–567, 2009.
- [75] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 16(7):521–533, 2002.

- [76] S. W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM Journal on Computing*, 6(4):730–732, 1977.
- [77] M. Rosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai. Scalable shape sculpturing via hole motions. In *IEEE Intl. Conf. Rob. and Auton.*, pages 1462–1468, Orlando, FL, 2006.
- [78] M. H. Rothkopf, A. Pekeč, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management science*, 44(8):1131–1147, 1998.
- [79] M. Rubenstein, A. Cornejo, and R. Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.
- [80] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.
- [81] T. Sandholm, K. Larson, M. Andersson, O. Shehory, and F. Tohme. Coalition structure generation with worst case guarantees. *Artificial Intelligence*, 111(1-2):209–238, 1999.
- [82] S. Sen and P. S. Dutta. Searching for optimal coalition structures. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 287–292. IEEE, 2000.
- [83] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Theory of Computing and Systems, 1997., Proceedings of the Fifth Israeli Symposium on*, pages 126–131. IEEE, 1997.
- [84] O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1):165–200, 1998.
- [85] W.-M. Shen, B. Salemi, and P. Will. Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots. *Robotics and Automation, IEEE Transactions on*, 18(5):700–712, 2002.
- [86] W.-M. Shen and P. Will. Docking in self-reconfigurable robots. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 2, pages 1049–1054. IEEE, 2001.
- [87] Y. Shoham and K. Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.
- [88] A. Sproewitz, R. Moeckel, J. Maye, and A. J. Ijspeert. Learning to move in modular robots using central pattern generators and online optimization. *The International Journal of Robotics Research*, 27(3-4):423–443, 2008.
- [89] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, and K. Goldberg. Potential-based bounded-cost search and anytime non-parametric a. *Artificial Intelligence*, 214:1–25, 2014.

- [90] K. Stoy, D. Brandt, and D. J. Christensen. *Self-Reconfigurable Robots: An Introduction*. The MIT Press, 2010.
- [91] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.
- [92] I. Suzuki and M. Yamashita. Agreement on a common x-y coordinate system by a group of mobile robots. In *Intelligent Robots*, pages 305–321, 1996.
- [93] M. T. Tolley and H. Lipson. Fluidic manipulation for scalable stochastic 3d assembly of modular robots. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2473–2478. IEEE, 2010.
- [94] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.
- [95] J. Werfel and R. Nagpal. Three-dimensional construction with mobile robots and modular blocks. *The International Journal of Robotics Research*, 27(3-4):463–479, 2008.
- [96] G. M. Whitesides and B. Grzybowski. Self-assembly at all scales. *Science*, 295(5564):2418–2421, 2002.
- [97] H. S. Wilf. *generatingfunctionology*. Elsevier, 2013.
- [98] H. Yan, S. H. Park, G. Finkelstein, J. H. Reif, and T. H. LaBean. Dna-templated self-assembly of protein arrays and highly conductive nanowires. *Science*, 301(5641):1882–1884, 2003.
- [99] M. Yim. *Locomotion with a unit-modular reconfigurable robot*. PhD thesis, Citeseer, 1994.
- [100] M. Yim, W.-M. 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.
- [101] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.