4-2011

Dynamic Reconfiguration in Modular Self-Reconfigurable Robots Using Multi-Agent Coalition Games

Zachary Ramaekers
University of Nebraska at Omaha

Dynamic Reconfiguration in Modular Self-Reconfigurable Robots using Multi-Agent Coalition Games

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree

M.S in Computer Science

University of Nebraska at Omaha

By

Zachary Ramaekers

April 2011

Supervisory Committee:

Dr. Prithviraj Dasgupta
Dr. Carl Nelson
Dr. Jong-Hoon Youn

Dynamic Reconfiguration in Modular Self-Reconfigurable Robots using Multi-Agent Coalition Games

Zachary Ramaekers, M.S.

University of Nebraska, 2011

Advisor: Dr. Prithviraj Dasgupta

In this thesis, we consider the problem of autonomous self-reconfiguration by modular self-reconfigurable robots (MSRs). MSRs are composed of small units or modules that can be dynamically configured to form different structures, such as a lattice or a chain. The main problem in maneuvering MSRs is to enable them to autonomously reconfigure their structure depending on the operational conditions in the environment. We first discuss limitations of previous approaches to solve the MSR self-reconfiguration problem. We will then present a novel framework that uses a layered architecture comprising a conventional gait table-based maneuver to move the robot in a fixed configuration, but using a more complex coalition game-based technique for autonomously reconfiguring the robot. We discuss the complexity of solving the reconfiguration problem within the coalition game-based framework and propose a stochastic planning and pruning based approach to solve the coalition-game based MSR reconfiguration problem. We tested our MSR self-reconfiguration algorithm using an accurately simulated model of an MSR called ModRED (Modular Robot for Exploration

and Discovery) within the Webots robot simulator. Our results show that using our coalition formation algorithm, MSRs are able to reconfigure efficiently after encountering an obstacle. The average "reward" or efficiency obtained by an MSR also improves by 2-10% while using our coalition formation algorithm as compared to a previously existing multi-agent coalition formation algorithm. To the best of our knowledge, this work represents two novel contributions in the field of modular robots. First, ours is one of the first research techniques that has combined principles from human team formation techniques from the area of computational economics with dynamic self-reconfiguration in modular self-reconfigurable robots. Secondly, the modeling of uncertainty in coalition games using Markov Decision Processes is a novel and previously unexplored problem in the area of coalition formation. Overall, this thesis addresses a challenging research problem at the intersection of artificial intelligence, game theory and robotics and opens up several new directions for further research to improve the control and reconfiguration of modular robots.

Acknowledgements

First, I would like to thank the NASA Nebraska Space Grant Consortium for their funding of my research during 2010 and 2011. I would also like to thank Dr. Dasgupta and Dr. Nelson for giving me the opportunity to work with them in the advancement of the ModRED project.

I would again like to thank Dr. Dasgupta for all of the time and effort he spent working with me and keeping me going in the right direction. We spent countless hours reviewing research and talking through ideas. Without him, none of this would have been possible.

Finally, I would like to thank my family for all of their support. First to my parents, who have always pushed me to never underestimate myself and have always allowed me to follow my dreams. And finally to my fiancée Kristina, who has always encouraged me, been patient with me, and kept me level headed.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Self-reconfigurable robots (Stoy, Brandt, & Christensen, 2010) are robots that are able to change their shape in order to adapt to a new environment or perform a new task. These robots are designed to be highly adaptable and capable of performing many different tasks using the same set of parts configured in different ways. Modular self-reconfigurable robots (MSRs) are a class of self-reconfigurable robots that are made up of functionally simple modules that are capable of working together. On its own, each module is capable of performing very limited operations, but when connected with other modules, they can adapt their shape to accomplish complex tasks. Each module in an MSR is easy to maneuver and the entire MSR's movement can be specified through a series of movements for each module comprising the MSR. Another key advantage of MSRs is that the individual modules are very simple robots that are inexpensive to manufacture. Hence, it is economical to use MSRs in place of expensive robots that are custom-made for performing specific tasks. In spite of their simple and inexpensive construction, and easy maneuverability, a principal challenge in MSRs is how to change their shape autonomously so that they can continue their operation after encountering obstacles or occlusions that impede their movement. In this thesis, we address this problem, called the *dynamic self-reconfiguration* for MSRs – how to find a set of rules that allows an MSR to dynamically change its current configuration and get into a new

configuration so that it can continue its operation efficiently. This problem is challenging because a fixed set of rules does not work for all situations. For example, a rule that tells the MSR to form a long, linear chain-shape to cross a chasm would not be appropriate when the MSR needs to climb a hill, possibly by forming a ring shape. Therefore, in the self-reconfiguration problem, the MSR needs to perceive its current environment to determine how many modules to connect together, and the configuration or shape those modules should get into, so that the MSR can perform its assigned task most efficiently.

The MSR self-reconfiguration problem falls under the category of autonomous robotic control problems that deals with how to autonomously provide each module or robot with intelligence, so that it can perform the tasks assigned to it autonomously, without requiring constant human intervention (Russel & Norvig, 2010; Siegwart & Nourbaksh, 2004). To solve this problem, researchers have proposed using software entities called agents that are situated on the robot. An agent is programmed to perform intelligent behavior, based on the sensory inputs that the robot receives, and helps the robot to make decisions and perform appropriate actions. In robotic systems composed multiple robots or multiple modules, agents situated on different robots need to interact with each other so that the multi-robot system can behave as a coordinated entity. The branch of artificial intelligence called multi-agent systems provides techniques for multiple autonomous software agents to interact with each other to achieve a common goal or to pursue individual interests (Multi-Agent Systems, 2010). In the case of MSRs, each module is provided with an agent that determines the actions for the module so that the MSR can perform its task efficiently. For the modules to be able to work together in a coordinated manner, they must be able to communicate with one another, and they must also be able

to determine on their own what groups of modules would be best to join together. In the area of multi-agent systems, a significant body of research has been done on team formation between multiple agents using coalition game theory (Shoham & Leyton-Brown, 2009; Ray, 2008). 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. The rules of coalition games ensure that the agents have incentive to remain together in the teams determined by the game's rules and do not arbitrarily change teams. This feature called stability is particularly essential for MSR self-reconfiguration because it ensures that modules that are determined to form a new configuration will remain together and not try to leave the new configuration and attempt to combine with other modules. However, there are also several research challenges in using coalition game theory in MSRs that are outlined below:

1) In coalition game theory, the assimilation of agents into teams and the communication between agents is assumed to be free of cost. However, for MSRs, modules have to physically move to each other's proximity so that they can dock with each other. Communication between modules also expends their battery power.
2) In coalition games, the order between the agents within a coalition or a team does not matter. For example, if four agents have IDs 1, 2, 3 and 4, respectively, the coalition {1, 2, 3, 4} is the same as the coalition {4, 2, 1, 3}, or for that matter, any other permutation among the agent IDs. In contrast, in MSRs, the order between neighboring modules does impact the formation of the MSR because the modules have to physically connect with each other.

3) Most of the existing solution techniques in coalition games are computationally intensive and are calculated using powerful desktop computers. On the other hand, for MSRs, the computations have to be done within limited computational capabilities available on each MSR.
4) Finally, in coalition games, the values or utilities the agents calculate for determining how much they benefit by participating in a coalition is assumed to be free from uncertainty. In contrast, in MSRs, due to the presence of noise in the robot's sensor readings, the perception of the robot's environment done by the robot (e.g., its location coordinates in a 2-D plane) is not 100 percent certain. Because of this, the coalition game solution techniques have to be modified so that their calculations can be done with uncertain values.

In this thesis, we have addressed these research challenges by developing appropriate techniques to integrate coalition games with MSR control. One of the fundamental contributions of this thesis is the novel combination of coalition game theory with planning under uncertainty using Markov Decision Processes (MDPs).

To illustrate the operation of our MSR, while using coalition game theory-based self-reconfiguration techniques, we have used the domain of robotic exploration of initially unknown environments. Robotic exploration is encountered in many applications of unmanned robotic systems such as unmanned search and rescue, surveillance and reconnaissance for homeland security applications, space exploration and even for agricultural and domestic applications such as automated crop harvesting, automated lawn mowing, etc. In each of these application domains, MSRs may provide improved

fielding and maneuver capabilities because they are cheaper to manufacture, easier to deploy and more dexterous to manipulate and move. For our research, we have used an accurately simulated version of the MSR called ModRED (Modular Robot for Exploration and Discovery) within the Webots robot simulation software. ModRED is being currently developed by our collaborators in the Mechanical Engineering Department at the University of Nebraska, Lincoln. In contrast to previously developed MSRs, most of which have a maximum of three degrees of freedom (DOF), ModRED offers improved dexterity by having an additional 4th DOF per module. The improved dexterity allows ModRED to maneuver itself efficiently in tight spaces as well as to rapidly self-reconfigure when its motion is impeded by obstacles.

Our experimental results show that coalition game theory-based algorithms can be successfully used to dynamically self-reconfigure ModRED into different configurations. We have compared our results while using three different heuristics for our algorithm and shown that MSR modules using our techniques to self-reconfigure receive on average 2-10 % more reward as compared to MSR modules using a previously existing algorithm for determining coalitions. To the best of our knowledge, the research results and insights gained from the field of coalition game theory have not been used to date to understand the problem of self-reconfiguration in MSRs.

The rest of this document is structured as follows: In Chapter 2, we explore the work related to MSR development and summarize recent research in the area of coalition structure generation. In Chapter 3, we discuss a modular self-reconfigurable robot currently being developed that introduces a new problem in the area of coalition structure

generation. Chapter 4 introduces our approach to solving the optimal coalition structure generation problem using a modified Markov Decision Process. The summary of our experimentation results is presented in Chapter 5, and Chapter 6 defines the future work to be done in the project. Finally, Chapter 7 concludes and summarizes the thesis.

Chapter 2

Related Work

In this chapter, we will introduce work currently applicable to the area of MSR development. In the first section we will begin by introducing some key concepts in the area of MSRs. In the second section we will discuss work in the area of multi-agent systems. The third section introduces the coalition structure generation problem. Finally, we will discuss the Markov Decision Process (MDP) and why it is useful for modeling situations with uncertainty.

2.1 Modular Self-Reconfigurable Robots (MSRs)

Self-reconfigurable robots can be defined as simple robots that can connect together autonomously to change their shape and adapt for a given task. Modular self reconfigurable robots (MSRs) are a class of self-reconfigurable robots which are composed of identical modules (Stoy, Brandt, & Christensen, 2010). Individually, a single module is capable of performing very simple movements and taking sensor readings from its sensors. The modules also have the capability of communicating with one another in order to decide how best to group themselves together into different configurations. MSRs require connectors that allow the modules to dock and undock with each other dynamically based on their decided configurations. While MSRs follow these main rules, there are three main categories that different types of MSRs fall into.

The first set of self-reconfigurable robots is known as the *lattice* type robot. In the lattice type robot, the modules of the MSR are all connected together at all times to form a lattice structure. The modules are usually incapable of moving on their own, but they do have a limited set of actuators that allow for movements of simple parts. A key distinguishing feature of lattice type MSRs is that the actuators have discrete or binary motions which limit the modules to a finite set of states. When the modules are connected together and they move their actuators in a coordinated set of motions, the entire robot can then move. In order for the robot to change its shape, the individual modules perform a set of disconnects, moving an actuator, and reconnecting. These modules are always connected to one another and therefore do not have to rely on communication to figure out where they are in relation to one another. These modules are very simple on their own, but through communication and coordinated motions the modules can work together to perform complex motions.

The second type of self-reconfigurable robots is known as *chain* type robots. Chain type robots are similar to lattice robots, but chain robots only connect to one another in a front-to-back or side-to-side configuration. These robots are capable of moving on their own (based on their kinematic design), and therefore in order to connect to one another the modules must communicate and move towards each other to connect together. Another key distinguishing feature of chain MSRs is the ability of the actuators to move in continuous motion, compared to the lattice type MSRs where the actuators move in distinct, binary motions. Chain robots are good for locomotion and capable of being more independent compared to the lattice robots.

The final type of robot is known as the *hybrid* robot. The hybrid robot combines the capabilities of the lattice robot as well as the chain robot. The modules are capable of moving on their own and communication with one another to find each other and move together. The modules also have the capability of performing lattice type reconfiguration without having to communicate after they have been connected together initially.

MSRs have been planned for different application domains of robotic systems. NASA is currently working on robotic systems where multiple robots work together to achieve a common goal. The ATHLETE rover (Townsend, J; Biesiadecki J; Collins, C; Jet Propulsion Lab., California Inst. of Technology, 2010) and the Tetrahedral Walker (Curtis, S; Brandt, M; Bowers, G; Brown, G; NASA Goddard Space Flight Center, 2007) are two such next generation rover systems. Another potential application of MSRs is for search and rescue operations. The ability of MSRs to traverse through environments that most robots could not travel through and then reconfigure into a shape that could help a person could become lifesaving. There could also be applications of MSRs being a household robot to help perform different tasks around the house. Having robots that can work together to complete a task provides many advantages over current rover technologies, and MSRs allow for an unprecedented amount of adaptability to advance this concept.

2.1.1 Self-Reconfiguration Problem in MSRs

The key issue with MSRs is how to form teams of modules that can work together to accomplish an assigned task. But after a task has been completed, the robot will then be assigned a new task to complete and it is likely that the current configuration of the

modules will not suffice to complete the next task. The modules therefore need to reconfigure themselves into a new configuration that will allow them to work towards their next task. This is a complex problem in MSRs, deciding how to go from one configuration to another. Three main approaches have evolved to solving this problem: search-based reconfiguration, control-based reconfiguration, and task-driven reconfiguration.

Search-based reconfiguration uses known search algorithms to try to find a path between a current state and a goal state. The states in this case are the current configuration and the needed configuration for the next task. The path that is determined is the series of movements that need to be completed to arrive at the goal state. Control-based reconfiguration is a less strict process that again tries to go from a current configuration to a goal configuration. In control-based, the modules perform movements related to their local position that lets the robot "evolve" to a goal configuration. Each module performs simple movements to try to make the entire robot get closer to its goal configuration. The final approach is known as task-driven reconfiguration. Task-driven reconfiguration is not focused on getting into a specific configuration as the previous two methods, but it simply wants to find any configuration that will meet its needs.

We have decided to use task-driven reconfiguration to address the reconfiguration problem. Task-driven reconfiguration eliminates the need to specifically know what modules need to be in what position in the final configuration. The robot is done when it finds a configuration that meets its needs. This method is most suitable to meet our needs in the MSR reconfiguration problem. By using task-driven reconfiguration, we take

advantage of advances being made in multi-agent systems which provides us a set of rules for the modules to work together in a coordinated manner.

2.2 Multi-Agent Systems

Multi-agent systems (MAS) consist of multiple agents that work in an environment to either maximize the collective utility value of all the agents or to maximize their individual utility values (Shoham & Leyton-Brown, 2009). In the area of MAS, we use the term *coalition* to define the concept of how we want the modules in an MSR to group together into a certain configuration. Coalitions can be defined in the following way:

> *"Coalitions in general are goal-directed and short-lived; they are formed with a purpose in mind and dissolve when that purpose no longer exists, or when they cease to suit their designed purpose, or when the profitability is lost as agents depart." (Horling & Lesser, 2005)*

Within the context of an MSR, we posit that a coalition represents a set of modules that are connected together in a certain configuration. Because the MSR has to change its configuration and its size to perform its intended task, the coalition has to be adapted dynamically. When a module or a set of modules that are part of an existing coalition decide that they have reached a point where they need to form a new coalition, possibly with other modules in their vicinity to perform their task better, the modules communicate with one another to decide which subsets of modules should join together. This process is known as *coalition structure generation*, which was formally defined by Sandholm (1999) as:

> *"The formation of coalitions by agents such that agents within each coalition coordinate their activities, but agents do not coordinate between*

> *coalitions. Precisely, this means partitioning the set of agents into exhaustive and disjoint coalitions. This partition is called a coalition structure (CS).”*

While there has been much research in the area of coalition structure generation, we have been unable to find research that incorporates a level of uncertainty into the coalition structure generation problem. The solutions we have seen assume there is a known and certain utility value assigned to each coalition structure. The problem with this approach is that determining an exact value for each coalition structure takes too much time and consumes too many resources on the modules to be viable. We will explore the current research being done in the area of coalition structure generation, and propose a modified Markov Decision Process (MDP) as a model that can be used to determine a near-optimal solution to the coalition structure generation problem without exhaustively searching through an entire coalition structure graph.

2.3 Coalition Structure Generation (CSG)

In systems where agents need to work together to form teams such as MSRs, a key component of the process is figuring out what modules should work together to accomplish a task in the most efficient manner. A form of game theory known as *coalition game theory* deals with how to partition the set of players into teams known as coalitions. The end of result of dividing the agents into coalitions is known as the *coalition structure*, and the process of creating the coalition structure is the *coalition structure generation* (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999).

In the coalition structure generation problem, the agents are divided so that each agent is in one and only one coalition. Given a set of agents, where *n* is the number of agents, there are a possibility of 2^n-1 coalitions ranging in size from 1 to *n*. In the case of three agents, there are seven possible coalitions: { {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3} }. Given that there are *n* agents, the number of possible coalition structures that use all of the agents is given by a *second order Sterling value*. The Sterling value is defined as $\sum_{i=0}^{n} Z(n,i)$, where $Z(n,i) = i*Z(n-1,i) + Z(n-1,i-1)$, and $Z(n,n) = Z(n,1) = 1$. Again with the three agents, the possible number of coalition structures is five: { [{1},{2},{3}], [{1},{2,3}], [{2},{1,3}], [{3},{1,2}], [{1,2,3}] }. In the worst case scenarios, the number of possible coalition structures is $O(n^n)$ (Sandholm, 1999). Figure 1 shows the calculations for the number of coalitions and coalition structures given a number of agents.

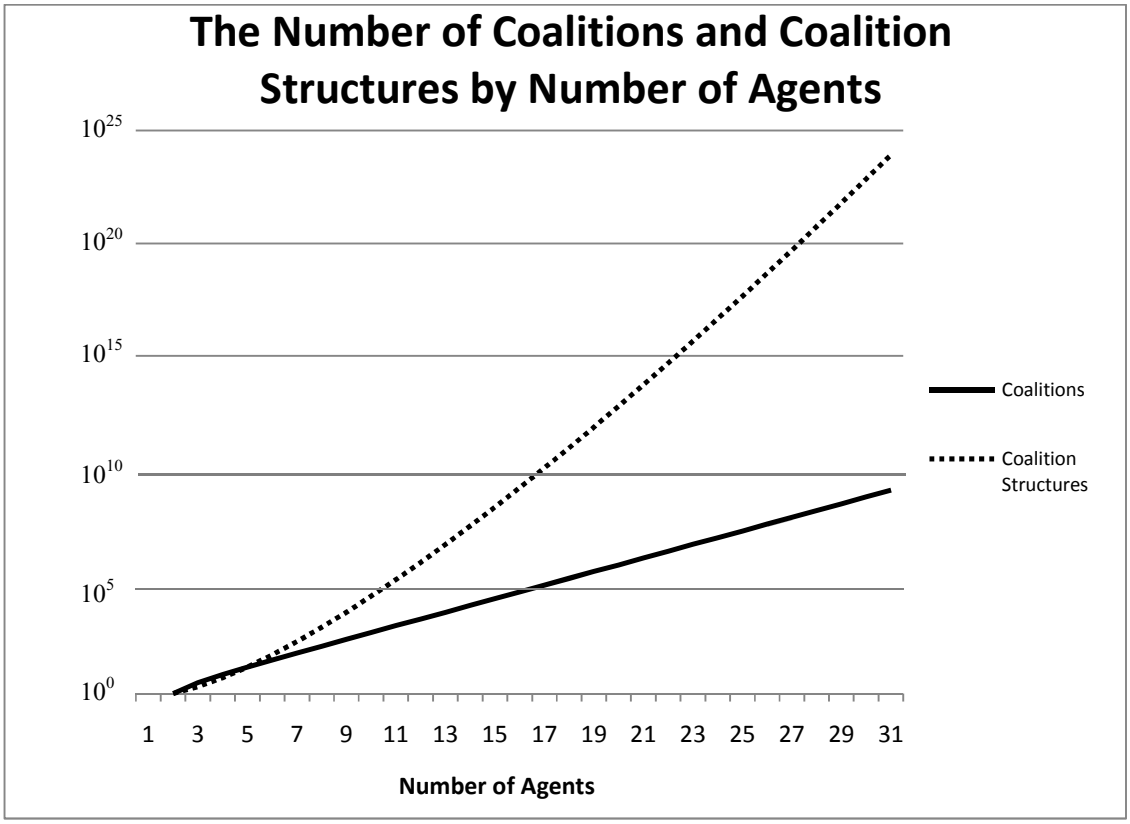

Figure 1. The number of coalitions and coalition structures by the number of agents. Y-Axis is a logarithmic scale.

In order to help organize the possible coalition structures that can be generated from a set of agents, a *coalition structure graph* can be generated to help visualize how the coalitions can be structured. The coalition structure graph is a visualization of how the possible coalition structures are related to one another. Figure 2 shows a coalition structure graph for a four-agent scenario. The bottom node of the graph represents level one. The number of the level corresponds to how many coalitions are present in each node at that level of the graph (i.e., level one has one coalition per node, level two has two coalitions per node, etc). The maximum number of levels in the graph is *n*, which represents the case where each module is in its own coalition.

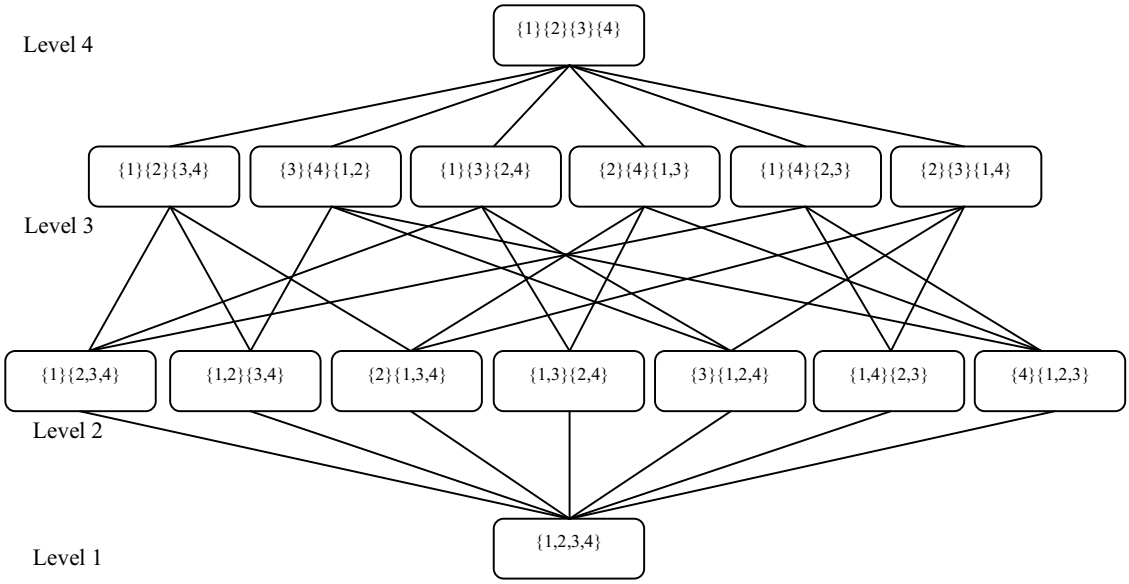

Figure 2. Coalition Structure Graph for 4 agents.

In the coalition structure graph, the edges between nodes represent an association between the nodes. If you follow an edge from level *n* to level *n-1*, it represents two of the coalitions in the node at level *n* joining together to form a single coalition in the node at level *n-1*. For example, referring to the node in level 3 in Figure 2 which contains the coalitions {{1}{2}{3,4}}, if you follow the edge from that node to the node in level 2 containing the coalitions {{1}{2,3,4}}, the edge represents coalition {2} and coalition {3,4} joining together to form coalition {2,3,4}. If you follow an edge from at node at level *n* to a node at level *n+1*, the edge represents a coalition in the node at level *n* splitting into two coalitions at level *n+1*.

2.3.1 Optimal Coalition Structure

The ultimate goal of enumeration of the possible coalition structures is to determine the coalition structure that gives us the optimal utility value. In order to determine which

coalition structure gives us the best value, we need to define how we value a coalition structure.

We first define a coalition S as a subset of all possible agents, and the set of agents that make up coalition S is known as the set A. The value of a coalition is called v_S, which is calculated in some fashion that is consistent across all possible S. A coalition structure is denoted as CS, where each CS has a distinct set of S that account for all possible agents. For simplicity, we assume that the value of a coalition structure, $V(CS)$, is additive, i.e. $V(CS) = \sum_{S \in CS} v_S$. After defining the value of a coalition structure, we can define the optimal coalition structure as $CS^* = max_{CS \in M} \; V(CS)$, where M is the set of all possible coalition structures.

The process of finding the optimal coalition structure is trivial in cases where the number of possible coalition structures is low. However, as stated previously, the number of coalition structures becomes $O(n^n)$ which creates a very computationally and time complex problem to solve. There has been research done to try to solve the optimal coalition structure problem in the cases where the number of coalition structures becomes too large to enumerate.

2.3.1.1 Approximate Coalition Structure Generation

Sandholm et al. (1999) proposed an algorithm which generates a coalition structure that is guaranteed to be within a certain bound of the optimal coalition structure while searching through the minimum possible number of coalition structures.

The goal of their algorithm is to search through a subset, N, of all possible coalition structures, M. They define the optimal coalition structure value they find by searching through N as $CS^*_N = arg\ max_{CS \in N}\ V(CS)$. They claim that $V(CS^*_N)$ is always within bound k of $V(CS^*)$, where CS^* is the optimal coalition structure of M, where $k = min(\kappa)$ and $\kappa \geq \frac{V(CS^*)}{V(CS^*_N)}$.

After conducting their analysis, they found that they can bind $k = n$, where the number of nodes searched is 2^{n-1}. They create this bound when they search only level one and level two of the coalition structure graph. The reason for this is that within level one and level two of the graph, all possible coalitions can be observed. The *grand coalition* is observed at level one, and all other possible coalitions are present in a node at level two. According to their research, no other algorithm can search less than 2^{n-1} coalition structures and guarantee the bound they have. While their research does provide a mathematically bounded way to find a near optimal coalition structure, it does restrict the number of possible coalitions in their result to having only a single coalition or at most two coalitions.

2.3.1.2 Pruning for Optimal Coalition Structure Generation

Rahwan (2007) proposed an anytime solution to coalition structure generation that guaranteed bounds on the optimal solution while being able to prune part of the search space. In order to be able to prune the state space of the coalition structure generation problem, Rahwan proposed to group coalitions together that have similar structure. This grouping was done through the use of integer partitions on the number of agents. For

example, with four agents the coalition structures with a single coalition and four agents are grouped together, the coalition structures with a coalition of three agents and a coalition of a single agent are grouped together, etc.

The algorithm begins by enumerating all possible coalitions and their values and storing them in lists according to the number of agents in the coalition. When doing this, the algorithm also keeps track of the maximum, average, and minimum v_S for each coalition size. For every possible integer partition of the number of agents, the algorithm determines the maximum, average, and minimum values for the coalition structure group. To do this, they calculate the maximum value for each coalition size in integer partition and add them together to get the maximum, and the minimum is calculated as the sum of the average values for each coalition length in the coalition structure group. For example, if we assume $max(L_i)$ is the maximum v_S for any coalition that contains i agents, and $avg(L_i)$ is the average v_S of all coalitions of length i, we can easily represent mathematically what the algorithm says. Assuming we have 16 agents that we want to find the optimal coalition structure for, we first find all coalitions, calculate their v_S, and determine $max(L_i)$ and $avg(L_i)$ for all i from 1 to 16. For a given coalition structure group such as $G = \{5,4,4,1,1,1\}$, we can calculate the max(G) = $max(L_5) + 2*max(L_4) + 3*max(L_1)$, and $avg(G) = avg(L_5) + 2*avg(L_4) + 3*avg(L_1)$. Instead of using $min(G)$ as the lower bound for the range, we use $avg(G)$ because in most cases the actual values of a $V(CS)$ will be above $avg(G)$, and by using $avg(G)$ compared to $min(G)$ we reduce the range of values which helps with pruning.

To do the pruning, we start by calculating *LB*, which is defined as *LB = max(avg(G))* for all *G*. We then prune off all *G* where the *max(G) < LB*. After pruning all possible *G*, we start with the *G* having the maximum *max(G)* and determine the *max(V(CS))* for all *CS* in *G*. We can then prune all *G* that have a *max(G)* less than the current *max(V(CS))*. We repeat this procedure until we have reached a point where there are no more *G* left. At this point, we have found the optimal coalition structure.

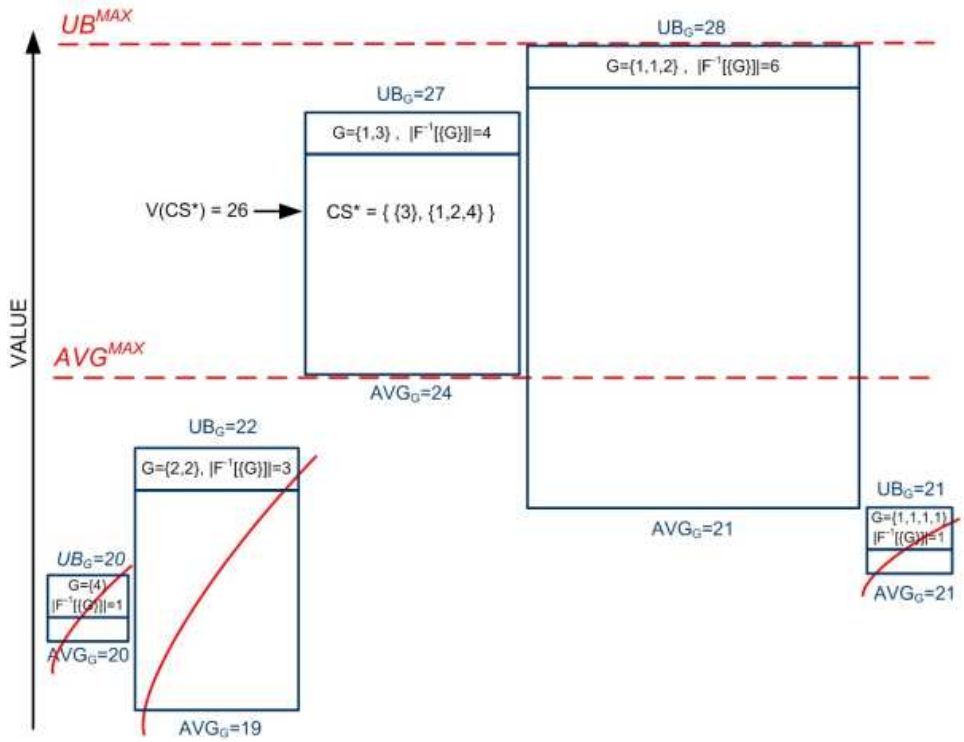

Figure 3. A visual representation of *G* pruning from Rahwan (Rahwan, 2007).

The optimal CSG calculation techniques described above deals with situations where the value of a coalition structure is certain. In other words, the coalition structure's value does not change while the optimal CSG is being calculated or while the agents are 'getting together' to form the coalition. In contrast, in the domain of MSRs there can be situations where the value of a configuration or coalition structure is uncertain because of

the noise and limited range of the MSR- modules' sensors. For example, a coalition's perceived value might change because of obstacles in the modules' paths when they try to physically dock with each other. This scenario is elaborated in Section 3.4. To address this problem, w propose to use a mathematical framework called a Markov Decision Process (MDP) that provide techniques to determine a prescribed action for the agents in the presence of uncertainty in the actions of the agents.

2.4 Markov Decision Process (MDP)

A large amount of research has been devoted to the area of decision making in complex environments. While simple problems can be modeled as a single episodic problem, most real world problems need to be modeled as a sequential decision problem. In a sequential decision problem, the utility an agent receives is based on a series of decisions.

To accurately model real-world problems, a model that accounts for uncertainty was described in the Russell and Norvig Artificial Intelligence text (Russel & Norvig, 2010). A decision process model is defined by a set of states and the transitions that are allowed between states. Each state *s* has a defined set of states that it is allowed to transition to which is called a transition model. If we always take an action *a* and it would always take us from state *s* to state *s'*, we would consider our environment to be deterministic. In other words, we have 100% certainty that performing action *a* at state *s* leads us to state *s'*; however, the real world is not deterministic. In the real world actions are unreliable and we can never be absolutely certain of the outcome of performing an action. In a non-deterministic model, we define a transition model $T(s,a,s')$ as the probability of reaching

state *s'* by performing action *a* at state *s*. In a stochastic environment, if we take action *a* there is a probability that we reach state *s'*, but there is also a probability that we end up at state *s''*. This uncertainty in the outcome of our actions comes from things in the environment such a sensor noise and communication problems. When we say that the probability of reaching state *s'* from state *s* by taking action *a* does not depend on any previous actions that we have taken, we can say that the transition model is Markovian.

An MDP is defined as a sequential decision process where the environment is fully observable and the transition model is Markovian. An MDP is formally defined by the following:

Initial State: S_0
Transition Model: $T(s, a, s')$
Reward Function: $R(s)$

The reward function $R(s)$ is a function that determines the immediate reward an agent receives for reaching state *s*, which could be positive or negative, and does not depend on the series of steps that were taken to reach the state. The idea of an MDP is to traverse through a set of states to reach a goal state which is the end of a sequence of steps. Goal states can be good or bad depending on how the environment is modeled. We also define a utility value for each state which is based on the reward function for the state, but is also based on the reward values of the states around the state. The final result of an MDP is known as a *policy*, where a *policy* is defined as the action that an agent should take when the agent is in any given state. In essence, when an agent is at a state *s*, the *policy* defines what action the agent should take. An *optimal policy* is the policy that leads an agent towards the state with the optimal utility value.

2.4.1 Factored MDP

In some cases where MDPs have a viable option of being used to solve a problem, it is possible that the number of actions in the action space as well as the number of states in the model can become exponential. Factored MDPs are a representation language that is used to represent the cases where an MDP grows into an exponentially large state space (Guestrin, Koller, Parr, & Venkataraman, 2003).

In a factored MDP, a state is defined by a set of variables where each variable is assigned a value from its domain. The transition model of the factored MDP is modeled by a Dynamic Bayesian Network (DBN). If we assume variable x_i is a variable at the current time step, then x_i' is the same variable at the next time step. All arcs in the DBN represent connections between variables in consecutive time slices. Each state has a conditional probability distribution that defines the transitions between a state and its parents, which are simply the states that are reachable from a given state. This transition graph which defines how states are affected based on a previous state is a two-layered graph, which means it only shows a generic graph for time step i and time step $i + 1$. All states can be represented by this graph.

MDPs define a standard way to model an environment that must incorporate an amount of uncertainty. The goal of an MDP is to define that at any given state in an environment, we know what action should be taken to help lead us towards the optimal goal state. In a dynamic environment, MDPs help an agent make the best decision based on the given information at the time the MDP was created.

Complimentary to our research, Chalkiadakis (Chalkiadakis, 2007) has proposed a technique to model uncertainty in coalition games using agent types based on research by (Suijs & Borm, 1999). That work is most theoretical and has some challenges when applied to our setting, such as giving each agent (or module) the distribution of types for all agents, and assuming that communication between agents is free of cost.

Chapter 3

ModRED: A Chain-Type MSR

In this chapter, we will discuss the development of a novel chain-type MSR that uses four degrees of freedom to increase the possible range of motions compared to most MSRs. To advance the development of the robot, we have a created a model of the robot in the Webots software suite. We will begin by introducing the robot simulation software Webots. We will also discuss the design and capabilities of the ModRED robot. Finally, we will introduce the different movements and activities that we have developed for ModRED in the Webots simulator as well as explain the dynamic reconfiguration problem.

3.1 Webots

Webots is a robot simulation software package that allows for the rapid prototyping and simulation of mobile robots (Cyberbotics Ltd., 2011). The goal of Webots is to provide a platform that allows researchers to spend less time developing the physical robots and more time working on the software that will control the robots. As stated on their website, the development of mobile robots combines many different disciplines and Webots strives to help eliminate the time hurdle of building the robots to do experimentation.

The software system has built-in robots that can be used or users can build their own robots from an included set of actuators and sensors. Not only can you build a robot inside of Webots, it also has the capability of importing models with the use of the VRML97 standard. Webots uses the Open Dynamics Engine (ODE) to simulate the physics of factors such as gravity and friction.

Webots also allows for the compilation of the code to run the robots in addition to having the capability of creating a simulated model of a robot. The software is capable of using several different languages to control the running of the robot simulation, such as C, C++, Java, and Python. There are APIs for controlling all of the different actuators and sensors of the robot in each language. Another key feature of Webots is the ability of the software to simulate the robot movements in accelerated time, up to 300 times faster than real time. The accelerated time allows for the quick simulation of experiments that could take very long to perform in real time.

3.2 ModRED

The ModRED (**Mod**ular Self-Reconfigurable **R**obot for **E**xploration and **D**iscovery) robot is a prototype robot that is currently being developed by Dr. Carl Nelson, Khoa Chu, and Mamur Hossain of the University of Nebraska at Lincoln Mechanical Engineering Department (Chu & Nelson, 2011). ModRED is being developed as a chain-type MSR that allows the modules to connect end-to-end to create long chains of modules that allow for increased movements and range of motion. What makes ModRED such a unique design is its incorporation of four degrees of freedom into each module. Most

MSRs being developed are limited to one or two degrees of freedom, with some recent work being done on modules with three degrees of freedom.

Table 1 lists a current set of chain and hybrid MSRs along with their degrees of freedom for each module and the motion space of each module.

System	MSR Type	DOF	Motion Space
YaMor (Moeckel, Faquier, Drapel, Dittrich, Upegui, & Ijspeert, 2006)	Chain	1	2-D
Tetrobot (Lee & Sanderson, 1998)	Chain	1	3-D
PolyBot (Yim, Zhang, Roufas, Duff, & Eldershaw, 2003)	Chain	1	3-D
Molecube (Suh, Homans, & Yim, 2002)	Chain	1	3-D
CONRO (Castano, Behar, & Will, 2002)	Chain	2	3-D
Polypod (Yim, Locomotion Gaits with Polypod, 1994)	Chain	2	3-D
MTRAN III (Kamimura, Yoshida, Murata, Kurokawa, Tomita, & Kokaji, 2008)	Hybrid	2	3-D
Superbot (Salemi, Moll, & Shen, 2006)	Hybrid	3	3-D
iMobot (iMobot - an Intelligent Reconfigurable Mobile Robot)	Hybrid	4	3-D

Table 1. This table lists MSR chain and hybrid robots that are currently being developed. For each robot, we show the MSR type, the degrees of freedom in each module, and the action space for each module (Chu & Nelson, 2011).

The goal of adding a fourth degree of freedom into ModRED was to increase the range of motions that each module can perform, which therefore also increase the range of motions that a chain a modules can perform. Having increased range of motion will help lead to increase the activities that the robot can perform during space exploration missions. The main design of ModRED can be seen in Figure 4. The figure shows a single module with its four degrees of freedom. The main module has a hinge on each end that is a rotational degree of freedom. This hinge is used to lift one end of a single module off the ground, or to change the angle that two modules make while connected.

On this hinge is also a mechanical connecting apparatus which allows the modules to dynamically connect and disconnect from one another. Each of these hinges has a 180 degree range of rotation. In the middle of the module is a translational degree of freedom which allows the module to expand and contract itself. The final degree of freedom comes from one of the ends that is capable of rotating infinitely in either a positive or negative orientation.

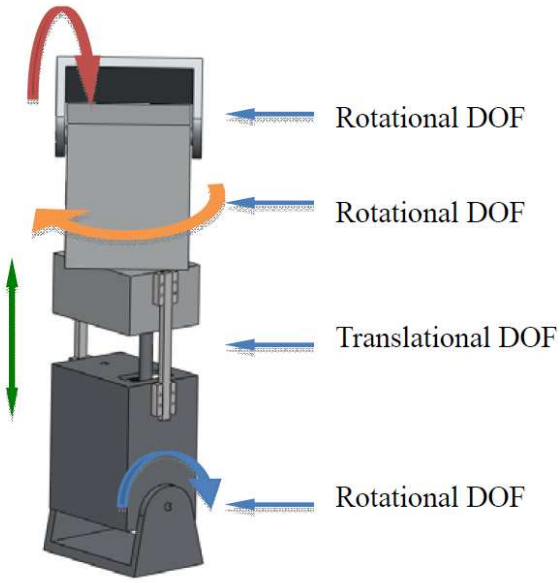

Figure 4. A CAD drawing of the ModRED robot showing its novel four degrees of freedom design (Chu & Nelson, 2011).

Included inside each ModRED module is a set of actuators and sensors that will move the robot as well as allow for information gathering about the environment around it. Each module is equipped with a CPU that will complete all of the necessary calculations and control the motors of the robot. Each robot will also contain motors to control the degrees of freedom. For gathering information, each module will be equipped with

infrared (IR) distance sensors (two in the front, two in the rear, and one on each side). The use of the six distance sensors allows for adequate processing to determine nearby obstacles or other modules. Each module will also be fitted with a compass to detect orientation as well as a tilt sensor to allow determination of which direction is up. For communication, ModRED uses a short-range wireless communication device. The final sensing equipment for each module will be some form of localization device, with that equipment to be determined at a later step in the prototyping phase.

3.3 ModRED Simulation in Webots

While Dr. Nelson's team from the University of Nebraska at Lincoln was developing a physical prototype for ModRED, we developed a simulation model of ModRED using Webots. Webots allowed us to create a very accurate model of the robot that can be seen below in Figure 5. All of the dimensions and weights from the ModRED prototype were incorporated into our simulation. In Webots, we were able to use built-in actuators and sensors to recreate all of the motors and sensors that would be available on the real ModRED modules. As in the prototype, our simulated model has six IR distance sensors, a compass for directional information, an accelerometer to detect which direction is up, wireless radios for communication between modules, and GPS for localization among the modules.

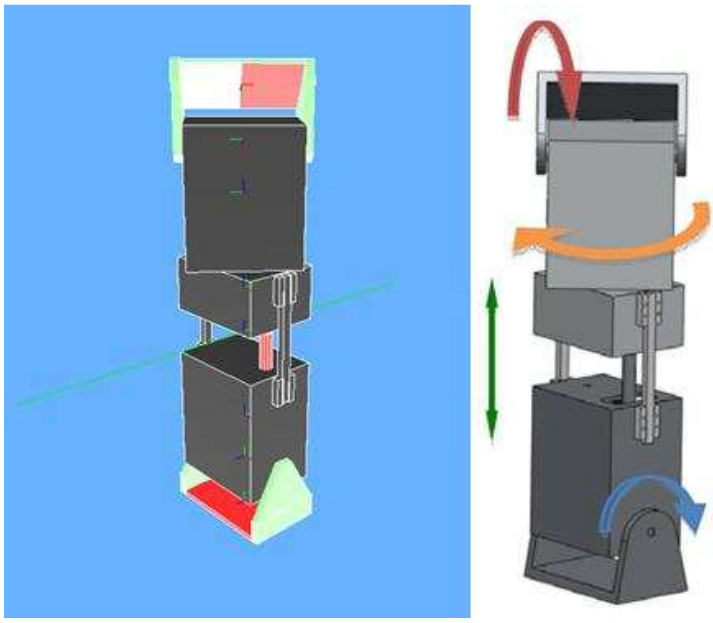

Figure 5. A side-by-side comparison of the CAD rendering of ModRED with the model of ModRED. The Webots ModRED model is on the left, and the original CAD drawing is on the right.

3.3.1 ModRED Movements

During the prototyping of ModRED, Dr. Nelson's team developed a series of steps that modules in different configurations can perform in order to achieve motion (Chu & Nelson, 2011). The basic motions they described were for a single module and for a chain of two modules. Following the main series of steps they created, we were able to successfully create locomotion for a single module as well as a chain of modules. In addition to the simple motions outlined by Dr. Nelson, we were also able to experiment with chains of multiple modules and achieve complex motions of modules in a loop.

3.3.1.1 Single Module Inchworm Motion

In order for the modules to be useful when they are not connected to any other module, they need to be able to move on their own to do both exploration and join together with other modules to form coalitions. To perform the forward movement, the module begins by rotating its front arm down and its back arm up. The module can then expand its translational DOF to move the front of the module forward. Next, the module rotates its front arm up and rotates it rear arm down. Finally, the module contracts its translational DOF to move the back of the module forward. Following these steps continuously allows a module to move forward. The sequence of images in Figure 6 shows a single module following these steps and moving forward. For a module to turn, it can simply rotate its DOF that is able to rotate infinitely in either direction. Rotating that DOF in different directions allows the module to turn left or right.

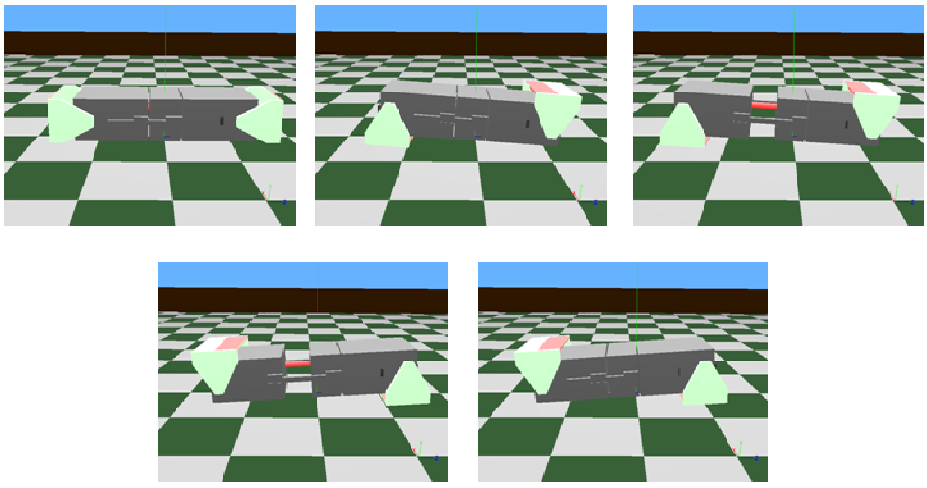

Figure 6. A single ModRED module performing a forward inchworm motion. The pictures are ordered from left to right, and top to bottom. In the pictures, the module is moving from the right to the left across the image.

3.3.1.2 Two Module Chain Inchworm Motion

Once two modules have moved toward each other and connected together, they need to be able to continue exploration or connect again with more modules. To achieve forward motion, the modules can use another inchworm motion. In this chain configuration we have two modules, m_1 we will call the front module and m_2 will be the back module. To begin, we have both modules with their translational DOF contracted. We begin by having m_1 raise its body into the air by rotating its rear arm up. While in the air, m_1 extends its translational DOF and then lowers its body back to the ground. Next, m_1 lowers its front arm and m_2 lowers its back arm to lift the middle of the chain in the air. While raised, m_1 contracts its translational DOF and at the same time m_2 extends its translational DOF to shift the body of the chain forward. After this, m_2 raises its body in the air by rotating its front arm up, and while in the air it contracts its translational DOF before return the body back to the ground. These series of steps are performed by two modules in Figure 7. Again, a module can turn by having m_2 rotate its DOF that has the unlimited rotational capability.

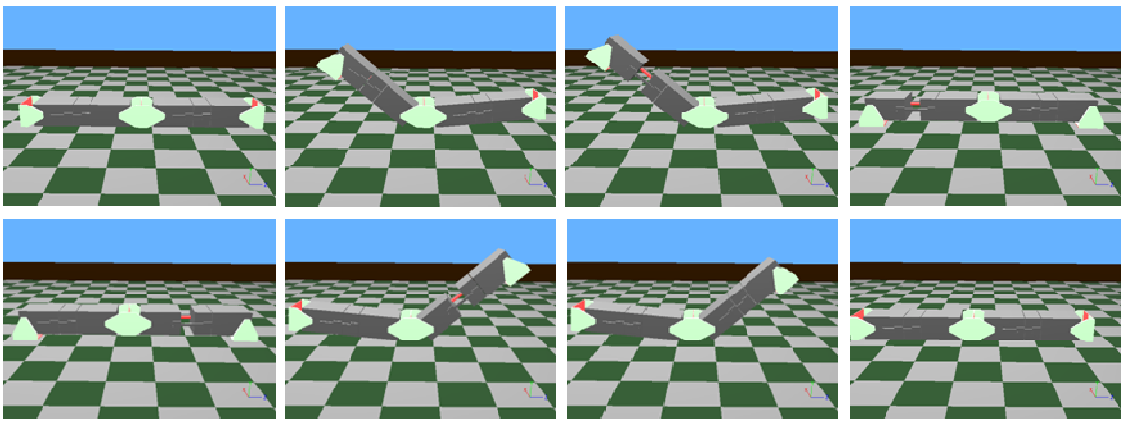

Figure 7. Two ModRED modules performing a forward inchworm motion. The pictures are organized from left to right, and top to bottom. The chain in the pictures is moving from right to left.

3.3.1.3 Two Module Chain Sideways Rolling Motion

A unique motion that ModRED is able to perform compared to other chain robots comes from its unique fourth degree of freedom. The degree of freedom we are referring to is the rotational degree of freedom on each module that is capable of rotation infinitely in either direction. Due to this degree of freedom, we are able to perform what we call a sideways rolling motion with two modules. The advantage of this motion is the capability of the robot to move directly sideways without having to turn itself. This motion is achieved by having two connected modules rotate this fourth DOF at the same time. The two modules begin in a chain configuration, with m_1 being at the front of the chain and m_2 being at the back of the chain. Module m_1 lowers it front arm and slightly raises its back arm, while module m_2 lowers its back arm and slightly raises its front arm. Module m_1 then begins to rotate its fourth DOF in one direction while m_2 rotates its fourth DOF in the opposite direction. This allows the module to roll directly sideways. Figure 8 shows two modules performing the series of steps we have described.

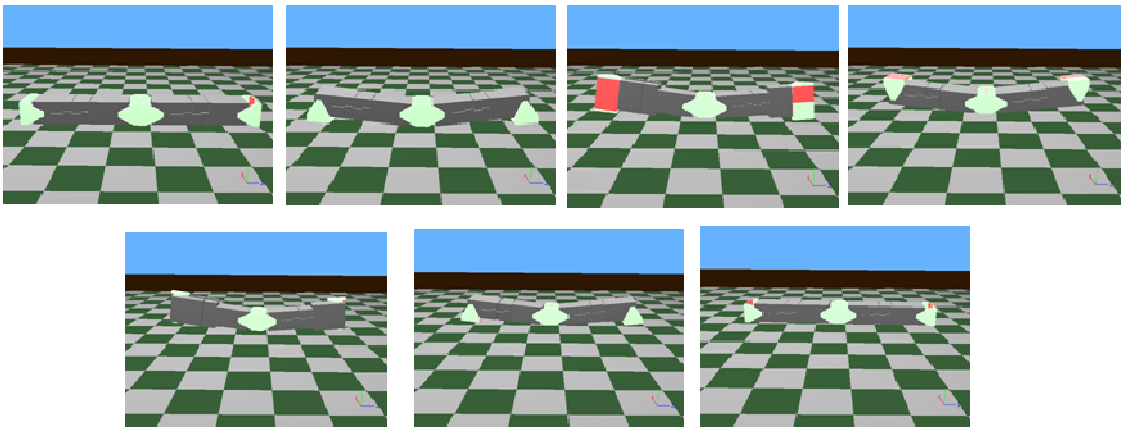

Figure 8. Two ModRED modules performing a sideways rolling motion. The pictures are organized from left to right, and top to bottom. The chain in the pictures is towards the background.

3.3.1.4 Six Module Chain Rolling Motion

The final motion of ModRED we will discuss uses six modules in a chain to perform a rolling motion. One of the limitations of ModRED is that during its single module inchworm and two module chain inchworm motions, the movements are very slow and covering a large distance takes the module a long time. However, with enough modules we can form a chain that allows the modules to perform a forward rolling motion that moves very quickly. This motion begins by having a chain of six modules that are connected together. To form a loop configuration, the front module and the back module are rotated up so that they are at a 90 degree angle with the ground. Next, the second module and fifth module are rotated up so they are at a 90 degree angle with the ground. While doing this, the first and sixth modules again become parallel with the ground and are close enough to connect. To move forward, the modules at the corners bend their front and back arms in unison to move the entire chain forward. Figure 9 shows six modules in a chain, forming the loop, and then moving the loop forward.

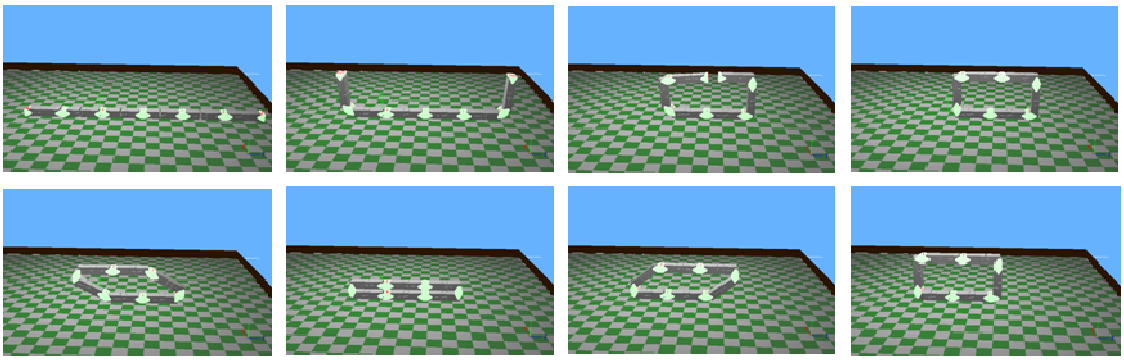

Figure 9. Six ModRED modules performing a forward rolling motion. The six modules begin by forming a chain which then transforms into a loop. Once in a loop, the modules perform simple movements in unison to create the forward motion. The series of pictures goes from left to right and top to bottom.

3.4 Dynamic Reconfiguration

ModRED is designed to be a self-reconfigurable robot that is capable of performing a wide range of tasks. The main capability of the robot is the exploration of an unknown region. When these robots are being used for the exploration of an unknown environment, a large amount of them will be placed in the terrain and they will work together to map the region. The goal of the robots will be to explore the maximum amount of space in the least amount of time while also minimizing the amount of area that is covered by multiple modules; so they must communicate with one another to make sure they do not overlap their exploration activities. The modules must also be capable of handling any type of terrain with different types of obstacles that they encounter, such as valleys, cliffs, and rocks.

Chains of different sizes will be good at achieving these different goals. While a single module will be good at exploring a space that is small with lots of turns, it will be very slow at traveling to a spot far away. However, a chain of modules that form into a loop can cover a long distance very quickly, but will be unable to fit into tight spaces. Another problem that could arise is a situation where a short chain of modules needs to move to a certain location, but there is a trench in the path that a single module cannot cross on its own. While this short chain might not be able to cross it, a chain of modules in a loop configuration is long enough to easily cross the gap. A simulated representation of this situation can be seen in Figure 10. In this case, a chain of two modules is unable to climb a ridge in its path, but a chain of six modules is capable of crossing the ridge.

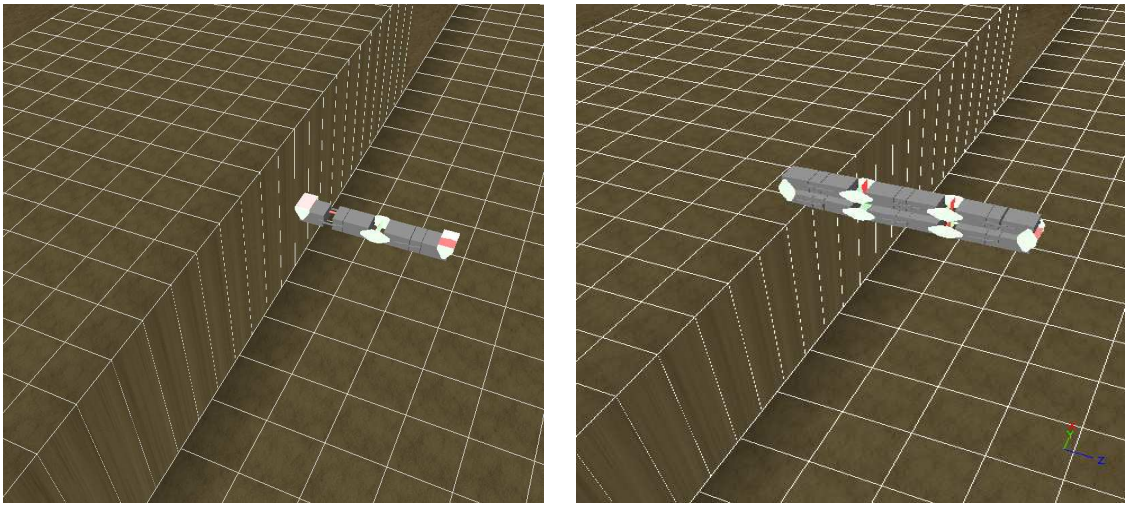

Figure 10. In the image on the left, a two-module chain attempts to cross a simulated ravine and is not long enough to reach across. If the two module chain is joined by four other modules, the chain is then long enough to easily roll across the gap and reach the other side.

All of these situations provide valid reasons for having chains of different lengths; however, before we explore an area we will be unsure of the terrain that the modules will encounter. We will have no way of knowing if we should have a large amount of single modules or lots of chain configurations with multiple modules in each chain. To account for this, dynamic reconfiguration of the modules is necessary to be able to handle whatever terrain the MSR encounters.

The problem of coalition formation has been studied for many years and there are many different approaches to solving the problem (Ray, 2008). The dynamic reconfiguration problem can be modeled as a coalition formation problem, where each coalition represents a chain of modules. The modules will be working on their own in the environment, so there will be no supervising entity that tells the modules which robots should form into coalitions. The modules will need to determine on their own how to form coalitions and what structure or configuration those coalitions should have.

Because each of the modules is very limited in computational abilities, the process of determining the coalitions needs to be computationally simple.

In exploring research done in the field of MSR reconfigurations, most of the work has been done in the area of searching, where the initial and final configurations of the modules is known and they need to determine what movements need to be made to get from the initial configuration to the final configuration. In our case, we know the initial configuration, but what we want to find is the final configuration. Since we do not know the final configuration we are unable to fit our problem into the search model. We also looked at research that has been done in the multi-agent coalition formation area. In the research done by Sandholm, they were able to quickly select a near optimal solution to the coalition structure generation problem; however, their optimal coalition structure always contained either a single coalition of all the agents or the agents split into only two coalitions (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999). In the research conducted by Rahwan, they were able to construct anytime solution to the coalition structure generation problem by grouping modules together and using a form of pruning (Rahwan, 2007). Yet their solution was only optimal as long as the upper bounds and lower bounds of the different groups they created did not overlap. In the case that they did overlap, it was possible to have to search through the entire state space to find the optimal coalition structure.

Another problem we encountered when trying to find a solution to our problem is the fact that we are unsure of the utility value we would give to a coalition structure. In the research we examined for coalition structure generation, the order of the agents in the

coalitions did not matter. However, in our case the order of the modules in the coalition would change the utility value we give to the coalition, so this uncertainty would need to be accounted for.

To solve this problem we introduce a new solution to the coalition structure generation problem. We model our problem in the form of a Markov Decision Process to traverse through the state space of possible coalitions and find the optimal coalition structure. We describe the algorithm we developed in Chapter 4 and discuss the results we obtained by using the algorithm in Chapter 5.

Chapter 4

Approximate Solution to Coalition Structure Generation Using a Modified MDP

In this chapter we introduce our solution to the coalition structure generation problem using a modified MDP. To the best of our knowledge, our work is the first in the direction of using the coalition structure generation (CSG) problem from coalition game theory to solve the self-reconfiguration problem in MSRs. The CSG problem is more complicated in MSRs than in conventional game theory. Because in conventional coalition formation, only the identity of the agents that are together in a coalition determines the coalition's value and the order in which the agents are placed within a coalition is inconsequential. In contrast, in MSRs the order in which the modules in an MSR are connected is vital to being able to determine the value of the coalition. Moreover, when we introduce uncertainty into the value a coalition might get, the current models and solutions for the coalition structure generation problem do not provide any means to find a solution. While Rahwan's algorithm (Rahwan, 2007) does provide an efficient manner to prune the space of possible coalition structures by splitting the coalition structures into distinct groups that each have a maximum and minimum limit, the model fails when the maxima and minima from different groups overlap with one

another. In these cases, the algorithm from Rahwan fails to provide the ability to consistently prune the number of coalition structures.

Unlike most problems that fit into the MDP model, our uncertainty comes from the values that we assign to a state, as compared to the normal uncertainty that arises from the action space. For this fact, we needed to develop a new model that allowed us to incorporate this uncertainty while also providing the ability to prune the space of possible coalition structures.

4.1 MDP Model Representation

We propose using a modified MDP to solve the coalition structure generation problem. As stated in section 2.1, the concept of an MDP is that there is uncertainty in traversing a state space to find a goal state. In a normal MDP, the uncertainty comes from not knowing with 100 percent accuracy what state you will end up in after taking an action. In the model of coalition structure generation we propose, the uncertainty comes from not knowing a true value for a coalition, and therefore a coalition structure.

For coalition formation situations, the value of coalitions and therefore coalition structures is based on predetermined function. The value of a coalition is decided based on what agents will form the team for any given coalition. In the case of the ModRED robot, we know what modules or agents will get together to form a coalition, but we are uncertain of the order they will be in. The value we assign to a coalition will depend on the time the agents take to get into their chain configuration, and the amount of time it takes them will depend on the order of the agents. In order to save computation time, we

cannot compute exactly what order the agents will be in for each coalition, we want to be able to approximate the value efficiently. This is where the uncertainty in the problem arises. We are unsure of the value a coalition receives because we do not want to take the computation time to determine the order of the agents in the chain. In order to model the uncertainty in our problem, we will use a modified MDP to model our problem and find a solution.

The state space in our model will be all of the possible coalition structures that can be generated for a set of n agents. In keeping with the MDP model, our action state will be moving towards any state that can be generated by combining two current coalitions or splitting a single coalition from the current state. There will be a reward value for each state in the model, and a policy will be calculated to determine the best move for any given state.

The coalition structure graph we are using for the state space of our MDP is described in detail in the paper by Sandholm (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999). Each level of the graph represents the number of coalitions in the coalition structure of each state in that level. For example, graph level 1 contains all states that have 1 coalition in the coalition structure; graph level 2 contains all coalition structures that contain two coalitions up to n.

The possible number of coalition structures in the coalition structure graph is n^n , which is too large a state space to explore. In order to prune the state space, we pick only certain states to explore further as we generate the coalition structure graph. We begin by starting at the level n of the coalition structure graph, which is the state where each agent

is in its own coalition. We call the states that can be generated by combining any two coalitions of the current coalition structure the children of a state. So for level 4 of a coalition structure graph with 4 agents, the only state in level 4 would be the state containing the coalitions {1}{2}{3}{4}, and the children of this state would be {{1,2}{3}{4}}, {{1,3}{2}{4}}, {{1,4}{2}{3}}, {{2,3},{1}{4}}, {{2,4}{1}{3}}, {{3,4}{1}{2}}. As we generate each possible child coalition structure, we also generate three possible values for each coalition structure, which we call sub-additive, additive, and super-additive. Each of these states represents the possible values for each of the coalitions. As stated previously, our uncertainty comes from not knowing with certainty the value of a coalition. The sub-additive state represents the case where the coalitions take a long time to get into a chain and therefore would receive a lower value for each coalition. The additive state represents a case where on "average", the agents take a normal time to get into a chain. Finally, the super-additive state represents a case where the agents are already close to being aligned and do not require much work to get into a chain, and therefore would receive a higher than normal value for their coalitions.

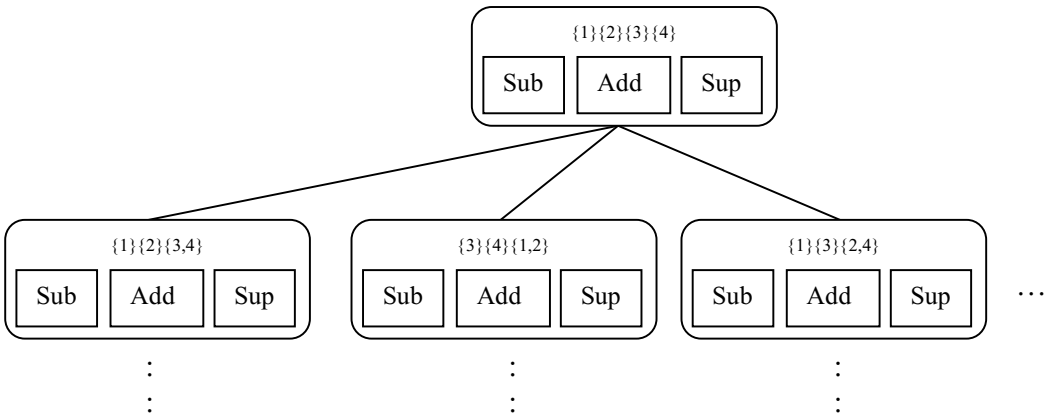

Figure 11. Coalition Structure Graph representation with sub-states of Sub-Additive, Additive, and Super-Additive

We will define a formal model for each node in the coalition structure graph. We will call a node in the coalition structure graph g_n. Each g_n contains a set of variables that define the node. The node has a unique ID, gn_{id}, that is used to distinguish nodes from one another. There are three values inside the node that keep track of the values for the sub-additive state, gn_{sub}, the additive state, gn_{add}, and the super-additive state, gn_{sup}. These are the variables used to store the reward values for the different sub-states. Along with each sub-state value, we also assign a percentage to each sub-state. These percentages are used during the value iteration and MDP traversal to calculate utility values. The sub-additive percentage is noted as $gn_{sub\text{-}p}$, the additive percentage as $gn_{add\text{-}p}$, and the super-additive percentage as $gn_{sup\text{-}p}$, where $gn_{sub\text{-}p} + gn_{add\text{-}p} + gn_{sup\text{-}p} = 1$. To keep the information for the policy of the MDP, each node also has variable gn_p that tells the node which node it should move to during the MDP traversal. Finally, each node also contains an array that keeps track of the neighbors of the node in the graph, gn_n.

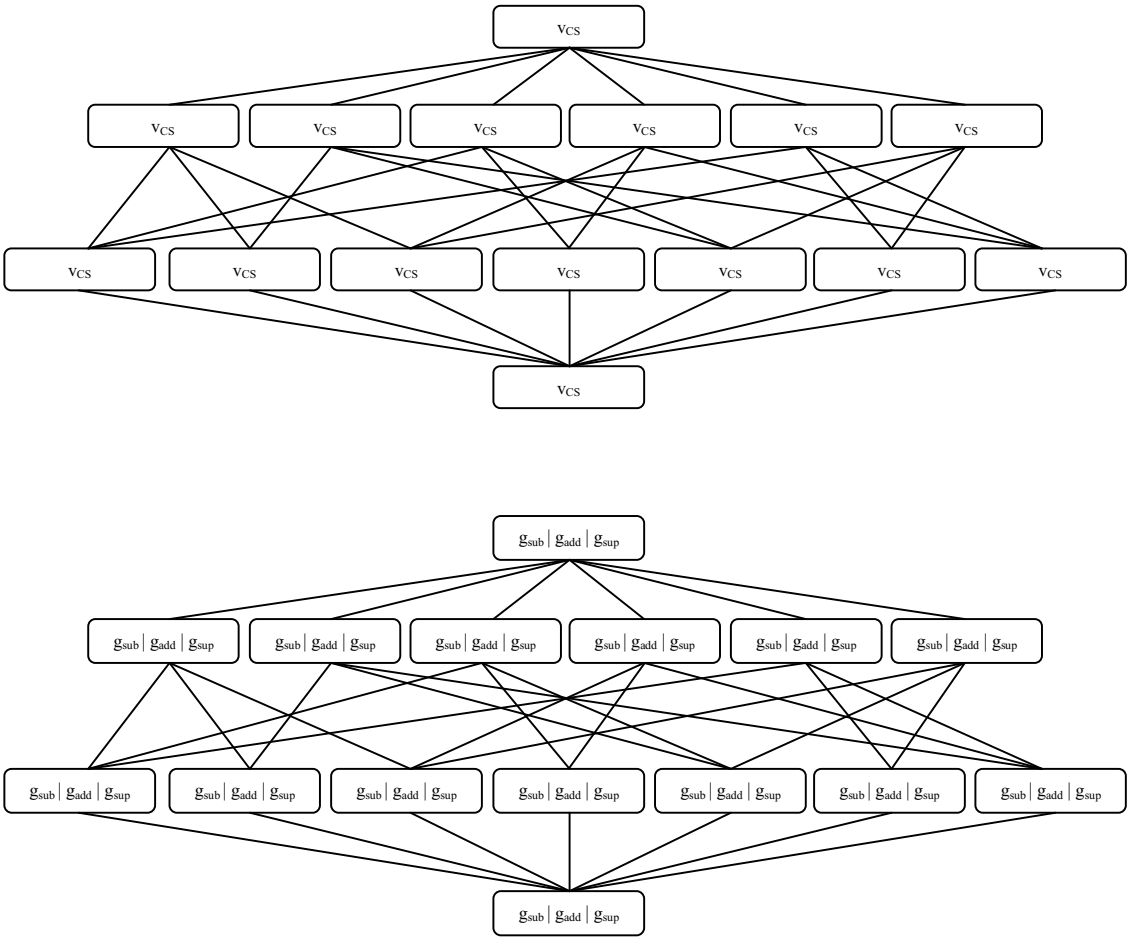

Figure 12. A visual representation of the coalition structure graph shown in Figure 2. The top image shows how the original graph would have one v_{CS} for each node. However, in our modified coalition structure graph shown in the bottom image, we have a sub-additive, additive, and super-additive value for each node represented as g_{sub}, g_{add}, and g_{sup}.

4.2 CS Graph Generation Algorithm

We begin our algorithm by gathering input from all of the available agents. The data comes into the algorithm as a set for each agent, where each set contains the ID of the agent, its current utility value, and finally its x and y positions according to some

universal map[1]. A representation of the data model is visualized below in Figure 13. To begin, we generate all possible coalitions from the set of agents. As we determine each coalition *S,* we also calculate an approximate value for *S,* v_S. We say that v_S is approximate because there are several unknown factors which could contribute to the v_S being higher or lower, such as obstacles between modules when they are trying to navigate to one another.

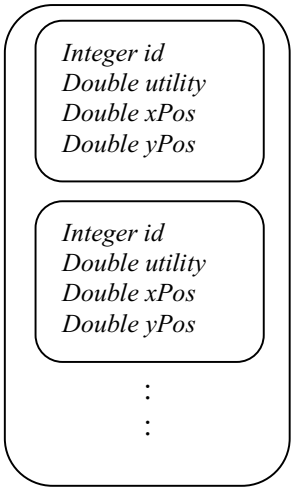

Figure 13. A model representation of the array of data sets that are sent into the coalition structure graph generation algorithm.

To calculate the v_S for each coalition, we first calculate the centroid of the locations of all the agents in *S*. We then calculate the average distance of all the agents in *S* from the centroid and combine that with an average of the utility values for each agent in *S* to get v_S.

[1] In comparison to the algorithms presented by Rahwan and Sandholm, we are generating our v_s from the physical location data of the modules, whereas in Rahwan's and Sandholm's algorithms they were generating random values for their v_s.

$$v_S = p_{dist} * avg_{a \in S}\left(dist(pos_a, pos_{cent})\right) + p_{util} * avg_{a \in S}\left(abs(u_a, u_S)\right)$$

Figure 14. The formula to determine the value of a coalition. p_{dist} represents the weight assigned to the distance portion, p_{util} represents the weight assigned to the utility portion, and $p_{dist} + p_{util} = 1$. pos_{cent} represents the centroid location of all the agents in S, and u_S represents the average utility value of all the agents in S.

After we have determined the v_S for all possible S, we move on to creating the coalition structure graph. In reference to the coalition structure graph (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999), we begin by creating the coalition structure node at level n in the graph, where each agent is in its own coalition. However, when we create a node in our model, the node has three sub-nodes as stated previously, a sub-additive node, an additive node, and a super-additive node. We assume that the v_S we created for each coalition S represents the additive case, so for the additive sub-node, we factor together all of the v_S for each coalition in the current node's coalition structure. For the sub-additive sub-node, we decrease each coalitions' v_S by a percentage when combining them, and for the super-additive sub-node, we increase each coalitions v_S by a percentage when combining them.

$$gn_{sub} = avg_{a \in S}\left(v_S - sub_p * v_S\right)$$

$$gn_{add} = avg_{a \in S}(v_S)$$

$$gn_{sup} = avg_{a \in S}\left(v_S + sup_p * v_S\right)$$

Figure 15. The formulas to calculate the sub-additive, additive, and super-additive values for a node. sub_p represents the percentage used to decrease the values in the sub-additive case, and sup_p represents the percentage used to increase the values in the super-additive case.

As we generate the utility values for each *gn*, we also create a unique ID for each *gn*. The ID allows us to quickly compare nodes together to guarantee that we do not create the same node twice. The ID we generate is based on the ID of the agents in each coalition as well as the number of agents in each coalition. Once the id of the node has been generated, we set gn_{id} equal to the ID.

To keep track of all of the nodes in the graph, we create two data structures to help with the organization of the nodes. The first data structure is an array that holds all of the nodes that we create, and the second is a key-map that maps the IDs of each *CS* node to its index in the array. After we generate the top level node, we place it into the array at index 0, and we place into the key-map the ID from the top node and map it to 0.

We then begin a looping process to loop over each CS node in the array to generate the children of each node (described in section 4.2.1). When we generate a child node, we again determine the sub-additive, additive, and super-additive sub-nodes. Following the creation of the children nodes, we check to see if any of the IDs of the newly created children nodes already exist in the key-map. If a child does not already exist in the key-map, then we know it is a new node and add it to the array of nodes as well as add its ID to the key-map. When a child node is created, the index of the array location for the node is added to the parents' list of neighbors, gn_n, and the parents' array index is added to the neighbors list of the child node as well. This lets us know that from the child node we can traverse to the parent node, as well as traverse from the parent node to the child node when we are navigating through the MDP later. If when checking the ID of the child node against the key-map and we discover, based on the ID, that the node has already

been created, we simply update the list of neighbors in both the parent and child nodes to include each other. Figure 16 provides the pseudo-code for the generation of the coalition structure graph.

```
function generateAllNodes returns node[] CS
inputs: map V, set S
variables: map CV, int num, int total, node[] children

CS[0] = createNode(S,V)
CV.add(CS[0].id, 0);
num = 0
total = 1

for num < CS.size
   children = CS[num].createChildren(V)
   for i = 0 to (|children|-1)
      if CV.contains(children[i].id)
         CS[num].addNeighbor(CV.get(children[i].id))
         CS[CV.get(children[i].id)].addNeighbor(num)
      else
         CS[total] = children[i]
         CS[num].addNeighbor(total)
         CS[total].addNeighbor(num)
         CV.add(CS[total].id, total)
         total++
      num++

return CS
```

Figure 16. The pseudo-code for the algorithm to generate all the nodes of the CS graph. V is a map that contains the IDs of each coalition for its key, and the coalitions' value for the key-value. S is the set of agent IDs.

4.2.1 Children Node Generation

The child of a node in the CS graph is defined as a node that can be generated by combining any two coalitions from the current node into a single coalition. The algorithm begins by looping over each coalition in the current node's coalition structure. The current coalition is then combined with every other coalition in the coalition

structure, and the rest of the coalitions are left as they are to create a new coalition structure. For example, a CS node with the coalitions {{1,2}{3}{4}} can generate the following children nodes: {{1,2,3}{4}}, {{1,2,4}{3}}, {{1,2}{3,4}}. The children nodes are generated in the same way as the parent nodes, while determining the values for each of the sub-nodes.

4.2.2 Pruning the Coalition Structure Graph

In order to keep the number of nodes in the tree from becoming exponential, we propose a form of pruning the tree while we are generating the children from any given node. As we are generating the children, the algorithm keeps only a small number of all possible children by using one of the following three strategies:

1. The algorithm keeps the child with the highest additive node and the two children with the two lowest utility values.
2. The algorithm keeps the child with the highest additive node, the median additive node, and the lowest additive node.
3. The algorithm keeps three randomly chosen children nodes.

For the first two pruning methods, we keep the child with the best utility value because that child node will be the node that the MDP policy will choose as the next node. In determining which other nodes to keep in the pruned model, we keep the nodes with bad utility values because those nodes with a current bad coalition structure utility value should generate a node with a better utility value by combining two coalitions with bad utility values. We keep these immediately bad children because of the uncertainty we must account for. In the algorithms presented by Rahwan and Sandholm, they

immediately prune nodes with bad utility values because they do not incorporate uncertainty in their model, and therefore, there is no possibility of visiting these pruned nodes again while traversing the coalition structure tree. Our third model uses randomly selected children to test against the first two models. A full pseudo-code representation of the children generation algorithm can be found in Figure 17.

```
function generateChildNodes returns node[] children
inputs: node n, map V, set S
variables: int coalitions, int childCnt, coalition tmpCoal, set coals

coalitions = number of coalitions in n
childCnt = 0

for i = 1 to coalitions
   for j = i+1 to coalitions
      coals.empty()
      tmpCoal = combine( n.getCoalition( i ), n.getCoalition( j ) )
      coals.add( tmpCoal )
      for k = 1 to coalitions
         if k != i and k != j
            coals.add( n.getCoalition( k )
      children[childCnt] = createNode(tmpCoal,V)
      childCnt++

children = pruneChildren( children )

return children
```

Figure 17. The pseudo-code for the algorithm that is used to generate the children nodes of a given node. The function returns back an array of nodes that contains the nodes that were not pruned away.

4.3 Value Iteration

After we have constructed the pruned coalition structure graph, we use the value iteration algorithm to determine the optimal policy. In the value iteration algorithm, we continue to update the reward value for each node to take into account the neighboring nodes reward values. For each node, we calculate a utility value for each sub-node. As we

perform more iterations, each node takes more into account the utility values from its neighboring nodes. The following function is used for the value iteration algorithm, where *i* is a sub-node in a node:

$$U'_i = R_i + \gamma(\max_{n \in m}(U_n))$$

where $R_i = gn_{sub}, gn_{add}$, *or* gn_{sup} depending on if *i* is sub-additive, additive, or super-additive for the current node, γ is a discount factor between 0 and 1, and *m* is the set of all neighbors for node *gn*, and the initial value of *U* for a state is set to 0. We continue generating *U'* for all states until the maximum difference between any *U* and *U'* in a given loop is less than some value, ε. When we conclude the algorithm, we store the final U' value for each node in the node, and we have a final utility value for each state. The pseudo-code for the value iteration algorithm is shown in Figure 18.

```
function valueIteration returns node[]nodes
inputs: node[] nodes, double discount
variables: double[][] u,double[][] newU, double maxDiff, double epsilon, double
maxVal, node curNeighbor, double curVal

u = double[nodes.size][3]

for i = 1 to nodes.size
   u[i][0] = 0
   u[i][1] = 0
   u[i][2] = 0

maxDiff = 0
while maxDiff > epsilon
   newU = u
   for i = 1 to nodes.size
      maxVal = 0
      for j = 1 to nodes[i].getNeighbors().size
         curNeighbor = nodes[ nodes[i].getNeighbor( j ) ]
         curVal = u[ curNeighbor ][0] + u[ curNeighbor ][1] + u[ curNeighbor ][2]
         if curVal > maxVal
            maxVal = curVal

      newU[i][0] = nodes[i].getSubVal() + discount * maxVal
      newU[i][1] = nodes[i].getAddVal() + discount * maxVal
      newU[i][2] = nodes[i].getSupVal() + discount * maxVal

   u = newU

for i = 1 to nodes.size
   nodes[i].setSubVal( u[i][0] )
   nodes[i].setAddVal( u[i][1] )
   nodes[i].setSupVal( u[i][2] )

return nodes
```

Figure 18. The pseudo-code for the value iteration algorithm. The function performs the value iteration algorithm on the set of nodes that is passed to it and returns back the same set of nodes with the updated utility values.

4.4 Determining the Optimal Policy

The next step is to determine the optimal policy, where a policy is defined as a solution which tells us which node we should move towards from any node that we could reach,

and the optimal policy is the policy that leads us towards the node with the optimal utility value. The goal of the entire process is to find the node with highest utility value, and the optimal policy tells us that at any given node what node we should move to next that gets us closer to the maximum utility value. To find the optimal policy, we loop over each node in the CS graph and examine the utility values of the neighbors of the node. The neighbor with the highest utility value is the optimal node we can move to during the MDP traversal, and the policy for the current node is set to that neighbor node. A diagram explaining the optimal policy is shown in Figure 19.

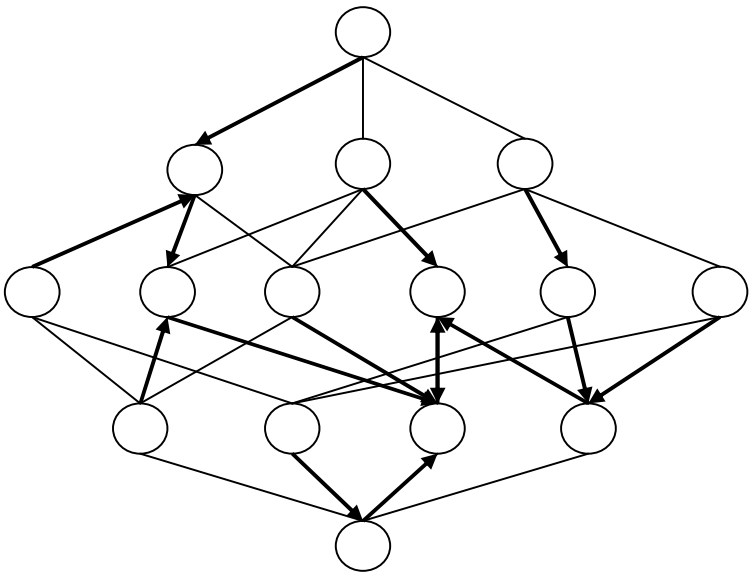

Figure 19. A visual representation of a CS graph and its optimal policy. In the CS Graph, the lines connecting different nodes represent neighboring nodes. The bold lines with arrows represent the optimal policy for this CS graph. Each node can have multiple neighbors, but each node only has a single bold arrow coming from it shows what node to move to next from the current node.

4.4 MDP Traversal

The MDP traversal is the final step in determining the optimal coalition structure based on the final data we have calculated from the value iteration algorithm. Once we have completed the value iteration algorithm and have an optimal policy which guides us from a node to its neighbor with the highest utility value, we are able to traverse through the coalition structure graph to try to find an optimal coalition structure with the highest utility value. Unlike a traditional MDP, we do not have any defined goal states, so the algorithm we use accounts for that.

The policy at a given node tells us which neighboring node should give us the best utility value, yet these utility values were based on the utility values and probabilities of the three sub-nodes at the neighboring node, so we will not know which node gives us the best value until we visit it. We begin the MDP traversal by starting at any given node in the CS graph. We start at the node that represents the current configuration of the modules (i.e., if we currently have modules 1 and 2 in a coalition and modules 3 and 4 are in a coalition, we start at the node that represents the coalition structure of {{1,2}{3,4}). To begin with, we set the max_u to $-\infty$ where max_u represents the best utility value we have seen for any node. When we arrive at a node, we check to see if the node has already been visited. If we have not visited the node previously, we must probabilistically determine if the node will be viewed as having a sub-additive, additive, or super-additive utility value based on the $gn_{sub\text{-}p}$, $gn_{add\text{-}p}$, and $gn_{sup\text{-}p}$ percentages. As stated previously, $gn_{sub\text{-}p} + gn_{add\text{-}p} + gn_{sup\text{-}p} = 1$, and these percentages represent the likelihood of the current node being sub-additive, additive, or super-additive. The

percentages are calculated based on any previous information we have about the terrain between the modules in the node's coalitions (i.e., if we know there is a large cliff between some of the modules of a coalition in the node's coalition structure, the percentage for the sub-additive state would increase and the super-additive and additive states would decrease). Once we determine if the node is viewed as sub-additive, additive, or super-additive, we will set $final_u$ to gn_{sub} if the node is sub-additive, gn_{add} if the node is additive, or gn_{sup} if the node is super-additive. We only calculate $final_u$ the first time we visit a node, and if we visit the same node again, we return $final_u$ instead of calculating it again. The node will return the same $final_u$ value every time it is visited. Once $final_u$ has been determined for the node, we return that value and we check to see if the current $final_u$ is greater than max_u. If so, we set max_u to the value of $final_u$ from the current node.

After we have checked the current node's $final_u$, we follow the policy of the current node to tell us which node to move to next. If the policy tells us to go back to the node we just came from and we followed the policy, we would be in a loop at a local maximum and would never be able to explore any more of the MDP state space. To avoid this, if the policy tells us to return to the node we visited before the current node, we probabilistically determine a neighboring node from the current node's set of neighbors. This is why once we determine the optimal policy we still keep information about all neighbors of the current node. Avoiding local maxima allows us to explore more of the coalition structures in the MDP.

We continue visiting nodes in the graph until we have visited every node in the graph, or we have not seen a node with a utility value better than the max_u in the last m nodes. When the algorithm concludes, we return the coalition structure from the node that gave us the max_u. This coalition structure represents the coalition structure with the best utility value that of any nodes that we have visited. The pseudo-code for the MDP traversal is shown in Figure 20.

```
function mdpTraversal returns node bestNode
inputs: node[] nodes, int cutoff, int startingNode
variables: double maxVal, int bestNode, int lastBest, int nodesVisited, int curNode, int
nextNode, int lastNode

maxVal = -∞
bestNode = -1
lastBest = 0
nodesVisited = 0
curNode = startingNode
lastNode = -1

while nodesVisited < nodes.length && lastBest < cutoff
curVal = nodes[curNode].getFinalVal()
if curVal > maxVal
maxVal = curVal
bestNode = nodesVisited
lastBest = -1

nextNode = nodes[curNode].getPolicy()
if nextNode == lastNode
nextNode = nodes[curNode].getRandomNeighbor()

lastNode = curNode
curNode = nextNode

nodesVisited++
lastBest++

return nodes[bestNode]
```

Figure 20. The pseudo-code for the MDP Traversal algorithm. The function takes a set of nodes as an input and returns back the best node that was found during the traversal of the MDP.

4.5 Complete CS Generation Algorithm

Putting together all of the steps from the coalition structure graph generation, the value iteration, and the MDP traversal, we are able to go from a set of input that contains each module's ID, location, and utility value to determining the optimal coalition structure based on that data. There are four main steps in being able to determine the optimal

coalition structure (a visualization of the process can be seen in Figure 21). The first step is calculating the utility value for each possible coalition based on the input data. Once we have a utility value for each coalition, we can begin the process of generating the coalition structure graph. The CS graph is the state space that defines all of our possible coalition structures and is used for both the value iteration and the MDP traversal. Once we have the CS graph generated, we run the value iteration algorithm on the graph which determines the optimal policy for the given CS graph. Once we have the graph constructed and the optimal policy, we run the MDP traversal. The MDP traversal searches through the state space to find the node in the graph with the maximum utility value. When we know the node that produces the maximum utility value, we have found the optimal coalition structure in the graph. The optimal coalition structure tells us which modules should form into coalitions.

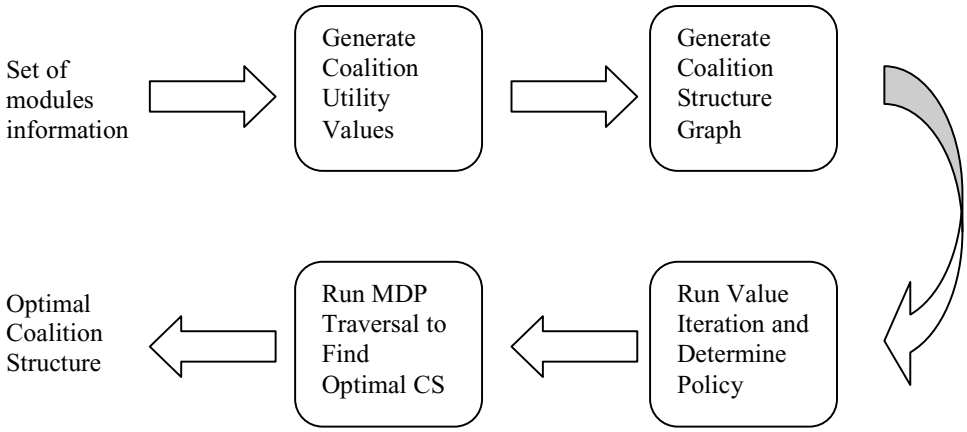

Figure 21. A flowchart the shows the entire set of steps required to go from the input of modules information to the optimal coalition structure.

Chapter 5

Experimental Results

To evaluate the effectiveness of the approach we have proposed to the coalition structure generation problem, we have implemented the algorithm in C++. The goal of our algorithm was to create a quick and simple approach to solve the coalition structure generation problem for MSRs that would take into account the uncertainty that arises from the complexity of creating coalitions. Our algorithm uses a modified Markov Decision Process to traverse a state space while pruning away nodes in the coalition structure graph. To analyze the results of our approach, we will compare the data from the pruned coalition structure graph to the full coalition structure graph.

5.1 Size of the State Space

The driving factor for the necessity to prune the state space of the coalition structure graph came from the fact that the number of possible states in the coalition structure graph was $O(n^n)$. Not only does it take a large amount of onboard memory space to store all of the possible states, it took an extremely long time to generate all possible nodes that would exist in the coalition structure graph. The graph in Figure 22 compares the number of nodes in the full coalition structure graph compared to the average number of nodes generated from the three pruning methods described in section 4.2.1. The y-axis in the graph is a logarithmic scale in base 10. As can be seen from the graph, the number of

nodes in the full CS graph grows large quickly, to over 10^9 possible coalition structures with only 16 agents. When we generate a maximum of three children from each node as is done in each of our three pruning methods, we reduce the possible number of states from $O(n^n)$ to $O(3^n)$ in the worst-case scenario. After experimenting with our data, since multiple nodes can actually produce the same children nodes, we have found that the number of nodes generated in the coalition structure graph ends up being close to $O(2^{n-2})$ in our data collected for up to 16 agents.

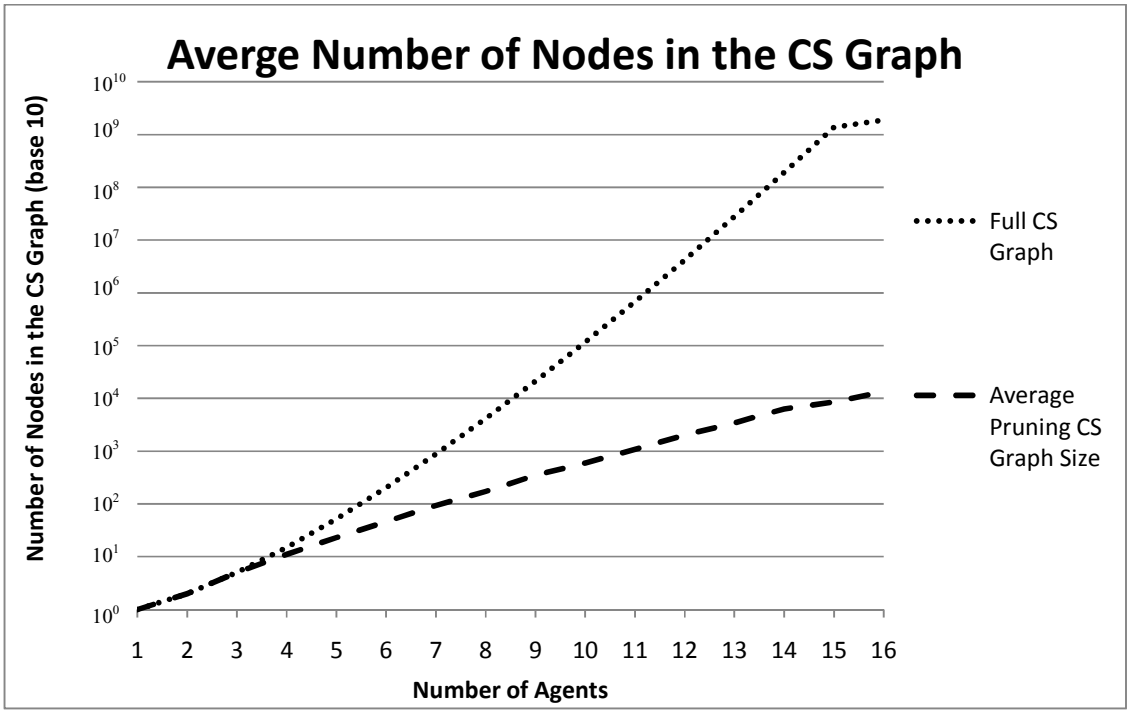

Figure 22. A graph showing the number of nodes generated in the full coalition structure graph compared the average number of nodes generated in the coalition structure graphs while pruning nodes.

Not only does pruning reduce the number of nodes in the coalition structure graph, it also reduces the amount of time it takes to generate the coalition structure graph. Even with

only 11 individual agents, the time to generate the full coalition structure graph was more than 20 minutes (on a laptop running Linux with a dual-core Intel i5 processor). This amount of time is too large for practical applications of finding the optimal coalition structure. In Figure 23, the graph shows how the time complexity for generating the coalition structure graph is reduced using our pruning method. Even with only 10 agents, the average running time to create the full coalition structure graph was around five minutes; while using the pruning methods we cut the average running time down to less than two seconds. This reduction in the amount of time to create the coalition structure graph shows that our pruning methods are capable of performing significantly quicker compared to generating the full coalition structure graph.

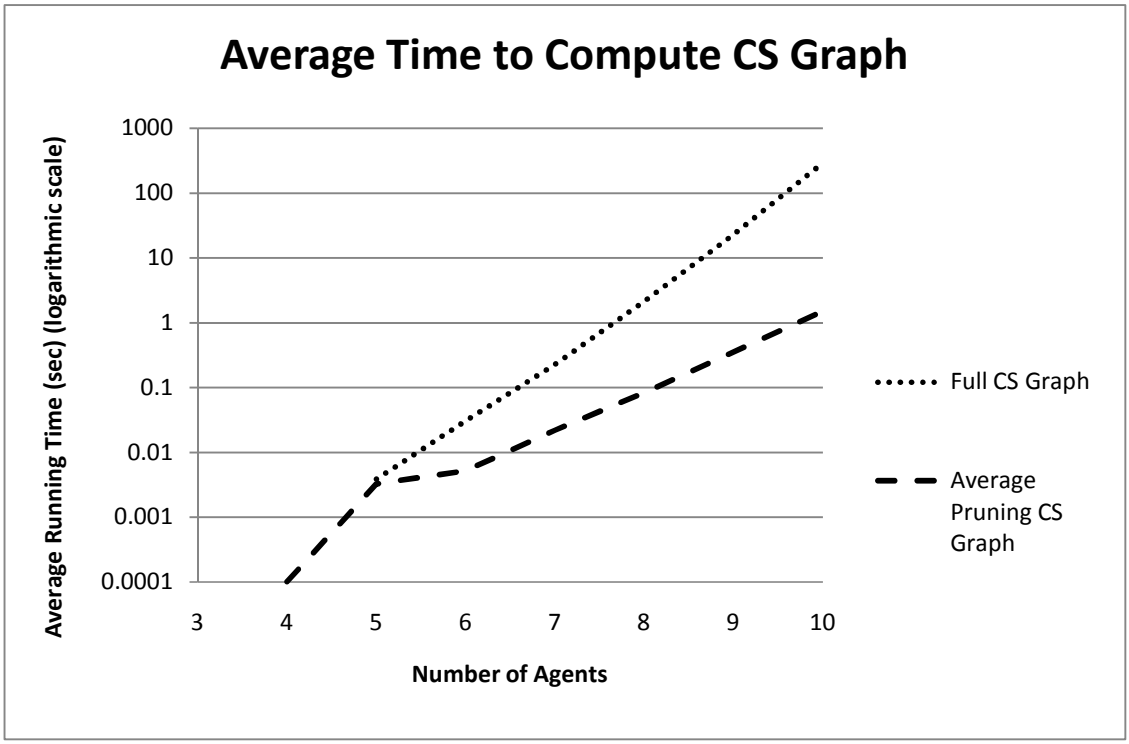

Figure 23. A graph showing the average amount of time it takes to generate the coalition structure graph for the full graph compared to the graph using the pruning methods.

5.2 Maximum CS Reward Value

While it was important to reduce the number of nodes in the coalition structure graph as well as the amount of time to generate the graph, these were not the most important factors. We needed to guarantee that the limited number of nodes we were generating in the pruned coalition structure graph were still giving is high reward values. If we are pruning away the nodes that contain the highest reward values, then our pruning method becomes counterproductive.

In order to verify that we are in fact obtaining nodes during our pruning methods that are near optimal, we compared the nodes in the following way. We first generated the full coalition structure graph for a set of agents, including the reward values for the sub-additive, additive, and super-additive sub-nodes. We have stated that we view the additive sub-node as the average case reward value for a particular node, so for our comparison method we used the reward value of the additive sub-node. After creating the entire coalition structure graph with no pruning, we found the highest additive sub-node reward value in the coalition structure graph and called this our max_{CS}. To compare our pruning methods, we then created a coalition structure graph for each of our three pruning methods using the same set of data that was used to create the full coalition structure graph. We then traversed through each of the pruned graphs to determine the highest reward value of any additive sub-node for each pruning method. Finally, we found the difference between the max_{CS} and the maximum reward value found in each of the pruned graphs. The graph in Figure 24 shows the average maximum reward values found in the full coalition structure graph compared to each of the pruned graphs based

on the number of agents. Figure 25 shows a graph that displays the average percentage away from max_{CS} for each of the three pruning methods. After examining the data presented in Figure 24 and Figure 25, we see that the second pruning method we used (selecting the child with the optimal reward value, the child with the median reward value, and the child with the lowest reward value) results in average maximum reward values that are closest to the max_{CS}. The first pruning method is close to the second pruning method, but the third pruning method (randomly selecting three children to keep) results in average reward values that are the farthest from the max_{CS}.

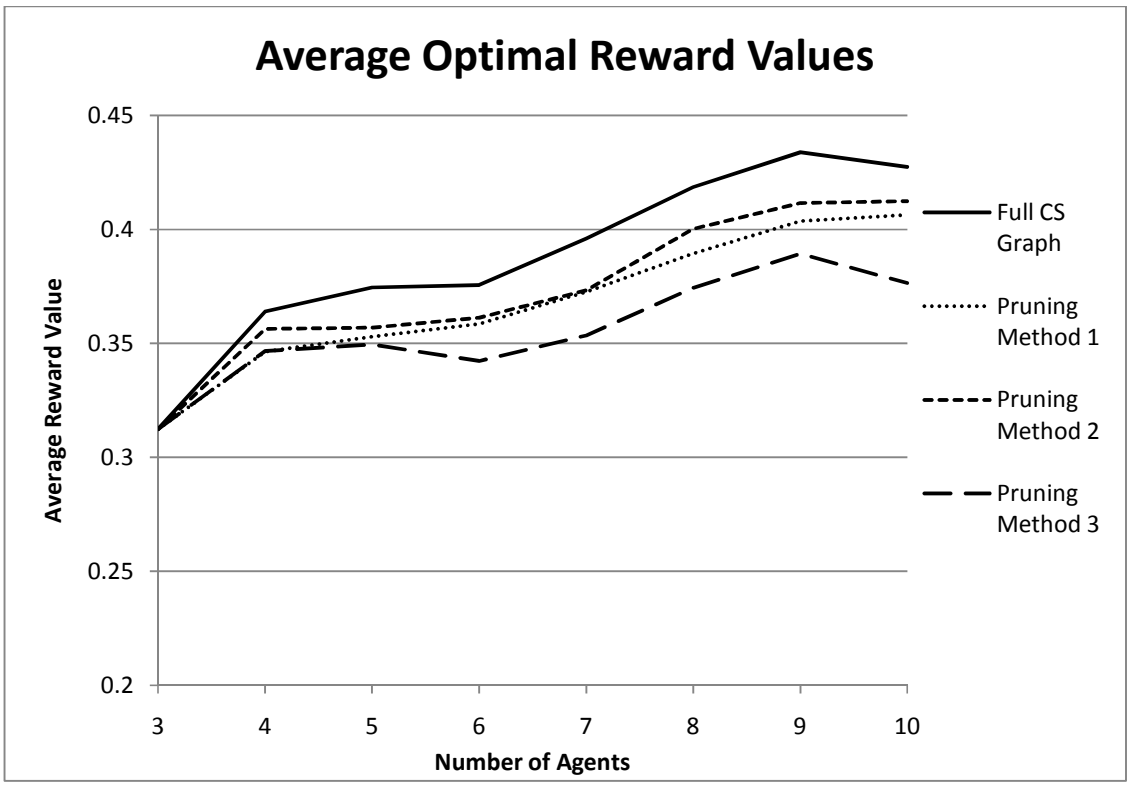

Figure 24. A graph comparing the average maximum reward values found in each of the three pruning methods compared to the average maximum reward value found in the entire CS graph.

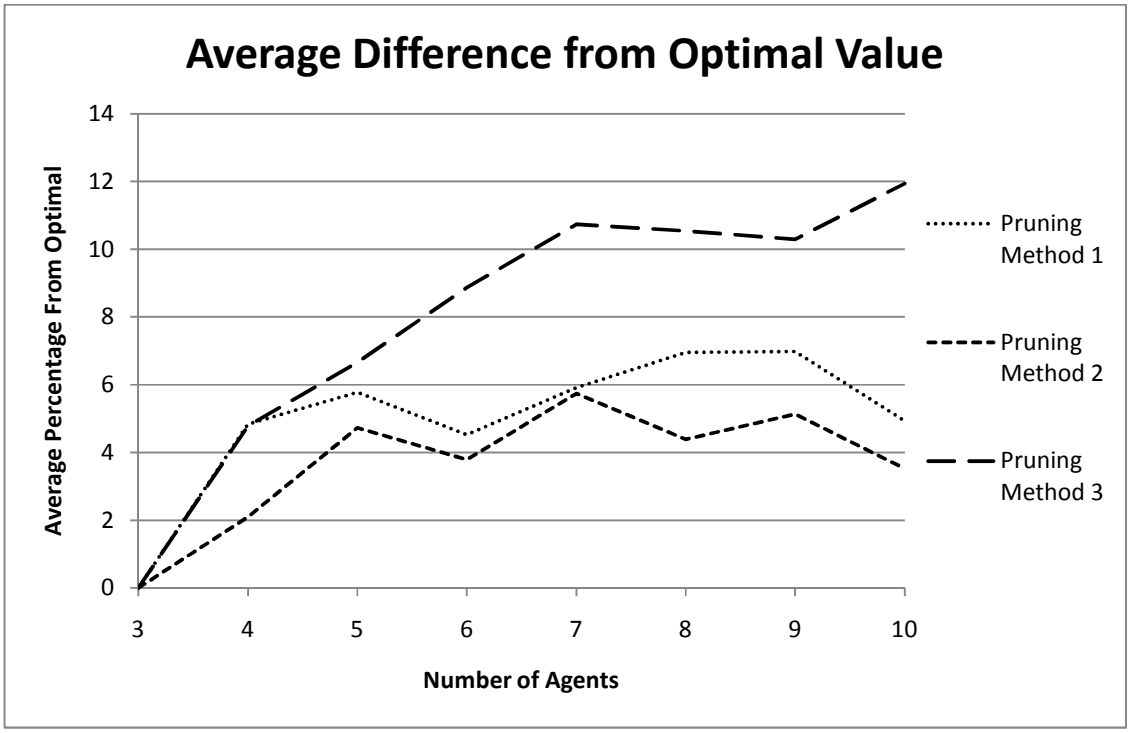

Figure 25. A graph showing the average percentage difference between the maximum reward value found in the entire CS graph compared to each of the three pruning methods (the lower the percentage, the closer the average maximum pruned value is to the optimal value).

5.3 Optimal Node Path Generation

For our final method of comparison we examined the nodes on the different paths from level *n* of the coalition structure graph to the optimal node. In testing for this method, we first generated a full coalition structure graph with no pruning and again found the node with the additive sub-node that had the highest reward value. After finding the optimal node, we then found all of the nodes on all of the paths from the single node at level n to the optimal node. For comparison methods, we then generated pruned coalition structure graphs using our three pruning methods. We then checked what percentages of the nodes

on the paths to the optimal node in the full coalition structure graph were also generated in our pruned graphs. We were experimenting to find out which pruning method generated the greatest amount of the nodes on the paths to the optimal node. If we have a larger amount of nodes on the path to the optimal node being generated, then we will have a better chance of generating the optimal node. Figure 26 shows the results we obtained from our experiments.

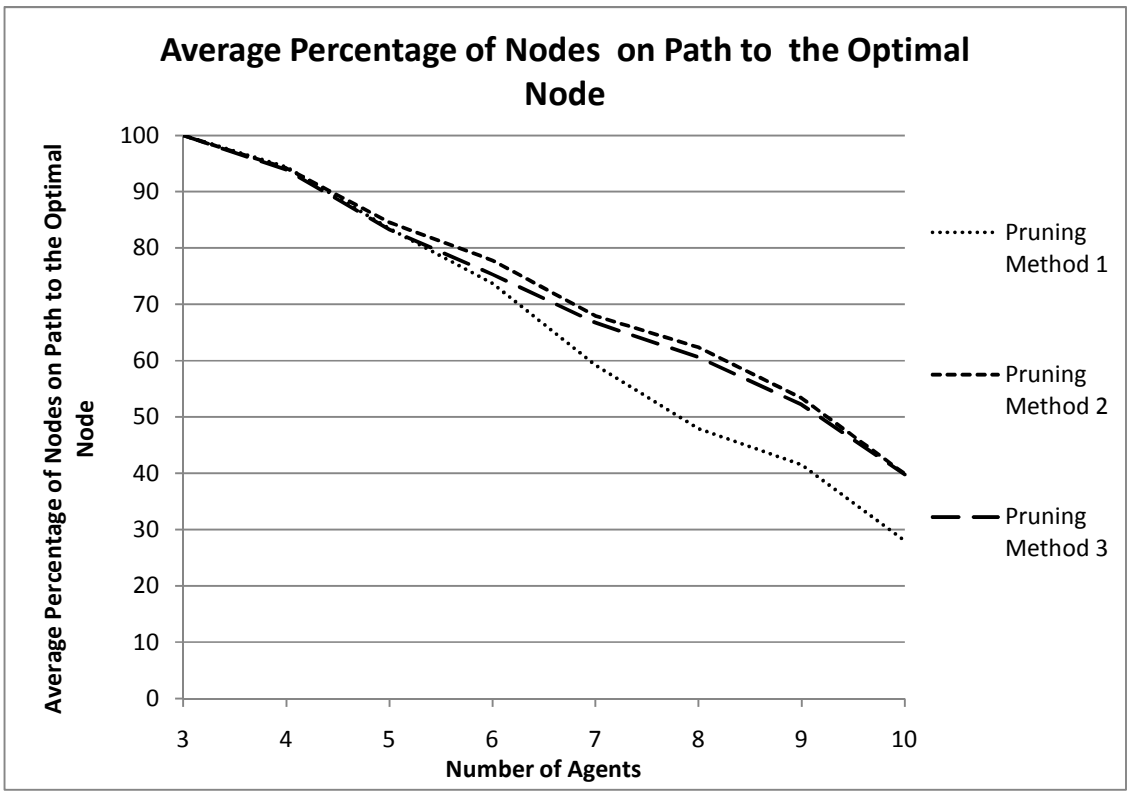

Figure 26. A graph showing the percentage of the nodes in the full coalition structure on the path to the optimal node that are generated from each of our three methods of pruning.

As can be seen from the graph, the second and third pruning methods actually generated more nodes on the paths to the optimal node than did the first pruning method. Using

data from all three of the graphs in Figure 24, Figure 25, and Figure 26, although the random selection of children nodes does result in around the same percentage of nodes on the path to the optimal node as does our second pruning method, we can see that this does not guarantee that we are generating near-optimal nodes. The second pruning method does a good job of generating nodes on the path to the optimal node and also results in much higher average maximum reward values.

5.4 Comparison to Sandholm

For comparison of the algorithm we have developed to find the near-optimal coalition structure, we look at comparing the results of our algorithm to the results of the algorithm proposed by Sandholm (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999). In their algorithm, they look at only the bottom two levels of the coalition structure and find the node that has the highest v_{CS} and they are able to say it is within a bound of the optimal coalition structure of the entire graph. By only looking at the bottom two levels of the graph, they are able to look at only $O(2^{n-1})$ nodes to find a near-optimal coalition structure. In the graph shown in Figure 27, we have compared the number of nodes examined by Sandholm's algorithm and compared it to the number of nodes generated by our pruning algorithm. As we have stated, Sandholm's algorithm looks at $O(2^{n-1})$ nodes, and our algorithm during experimentation generated a little better than $O(2^{n-2})$ nodes. Also in Figure 28, we have compared the amount of time it takes to generate the coalition structure graph using our algorithm in comparison with the time taken to generate the same graph using Sandholm's algorithm (Sandholm, Larson, Andersson, Shehory, & Tohme, 1999). As can be seen, Sandholm's algorithm did take less time to determine to

generate the need coalition structures, but as stated previously the coalition structures are limited to only coalition structures with a single coalition or two coalitions. This approach is not very applicable to situations where we might want to generate any number of coalitions, and possibly even have all modules in their own coalition.

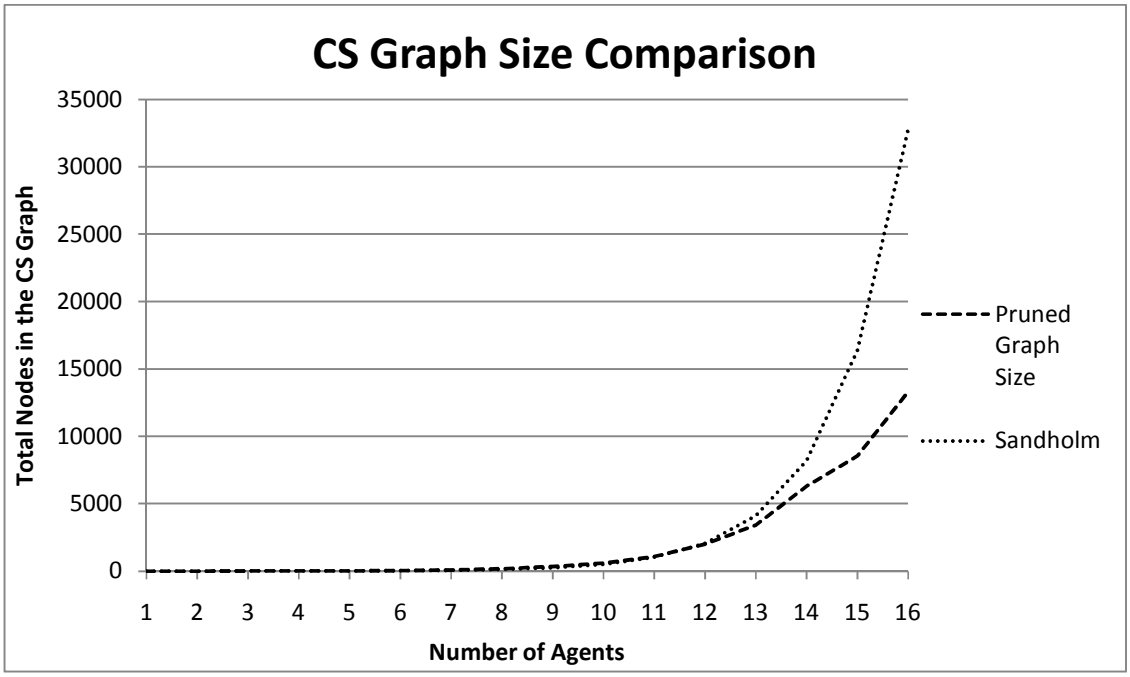

Figure 27. A graph comparing the number of nodes generated by using the algorithm proposed by Sandholm compared the number of nodes generated using our pruning method to reduce the size of the coalition structure graph.

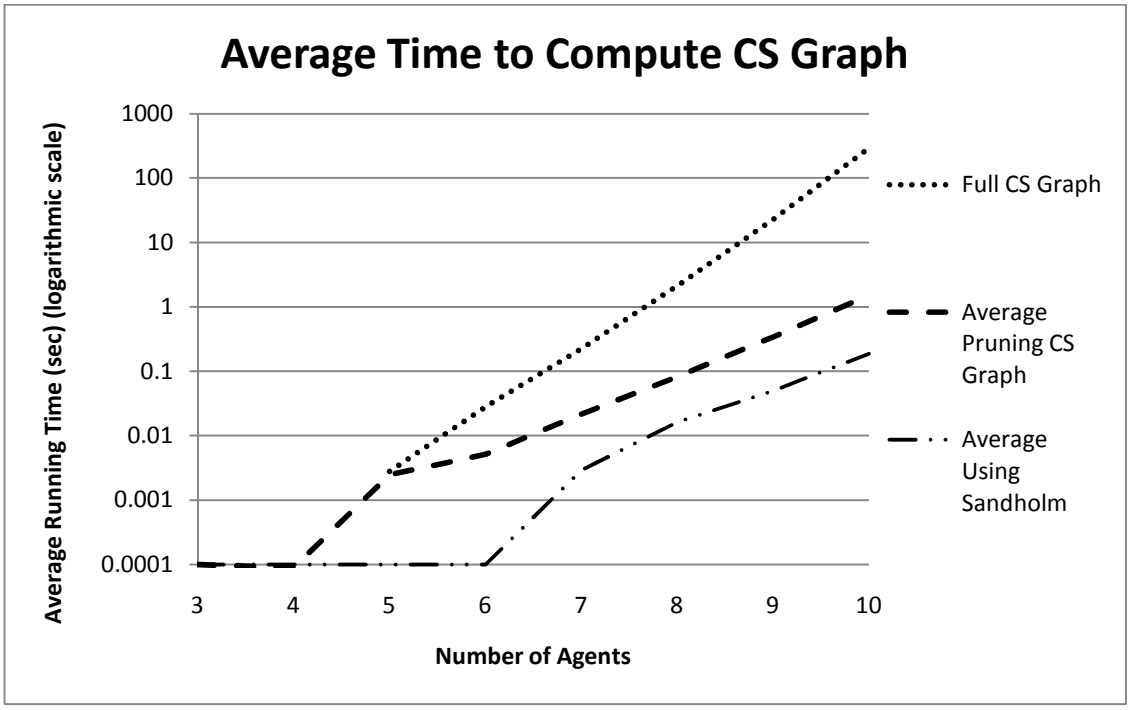

Figure 28. A graph comparing the time to generate the coalition structure graph using the full coalition structure graph, using our pruning method, and using Sandholm's algorithm.

We also implemented Sandholm's algorithm to look at the bottom two levels of coalition structure graph to find the best v_{CS} in those two levels. We again modified the algorithm to allow each node to have sub-additive, additive, and super-additive sub-nodes that each have their own values. For comparison, we found the highest additive value of any node in the bottom two levels and compared that to the highest additive value found by using each of our three pruning methods. The graphs in Figure 29 and Figure 30 compare the additive reward values we found using Sandholm's algorithm and the additive reward values we found using our pruning methods. Figure 29 shows the optimal additive reward values found plotted with the optimal additive reward value of the entire graph. Figure 30 shows the average percentage difference between the optimal additive value of the full

coalition structure graph compared to each pruning method and the values found using Sandholm's algorithm.

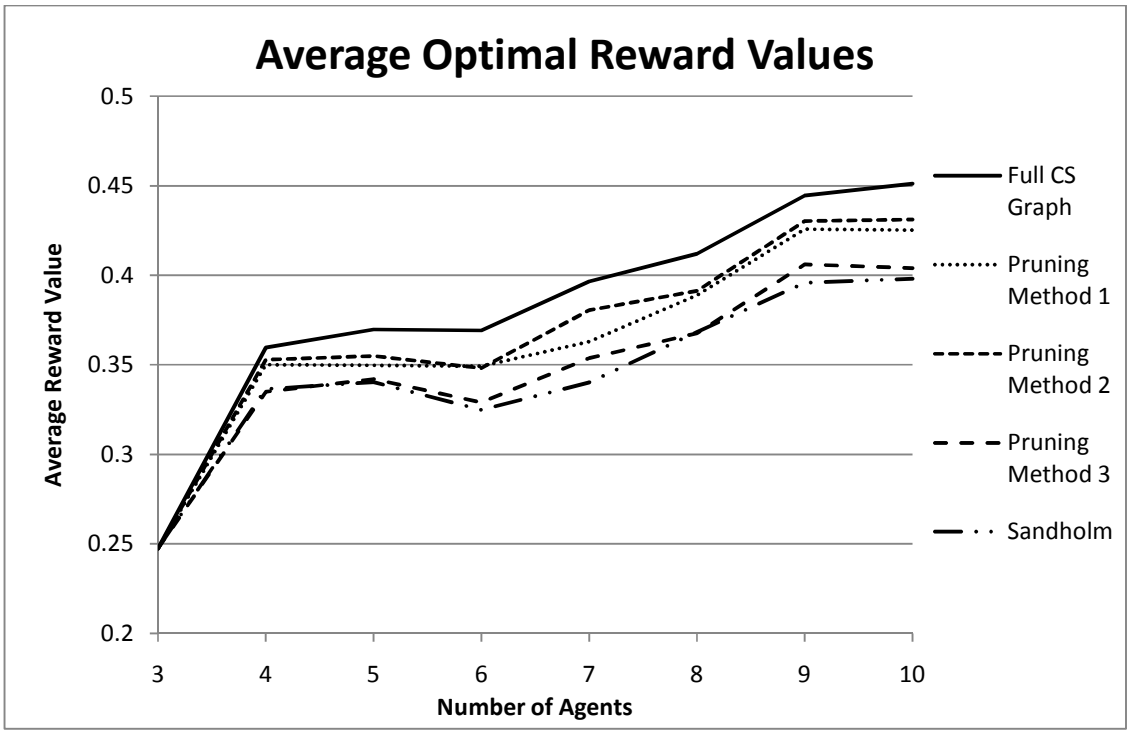

Figure 29. A graph comparing the optimal additive reward values found using the full coalition structure graph compared to our pruning methods and Sandholm's algorithm.

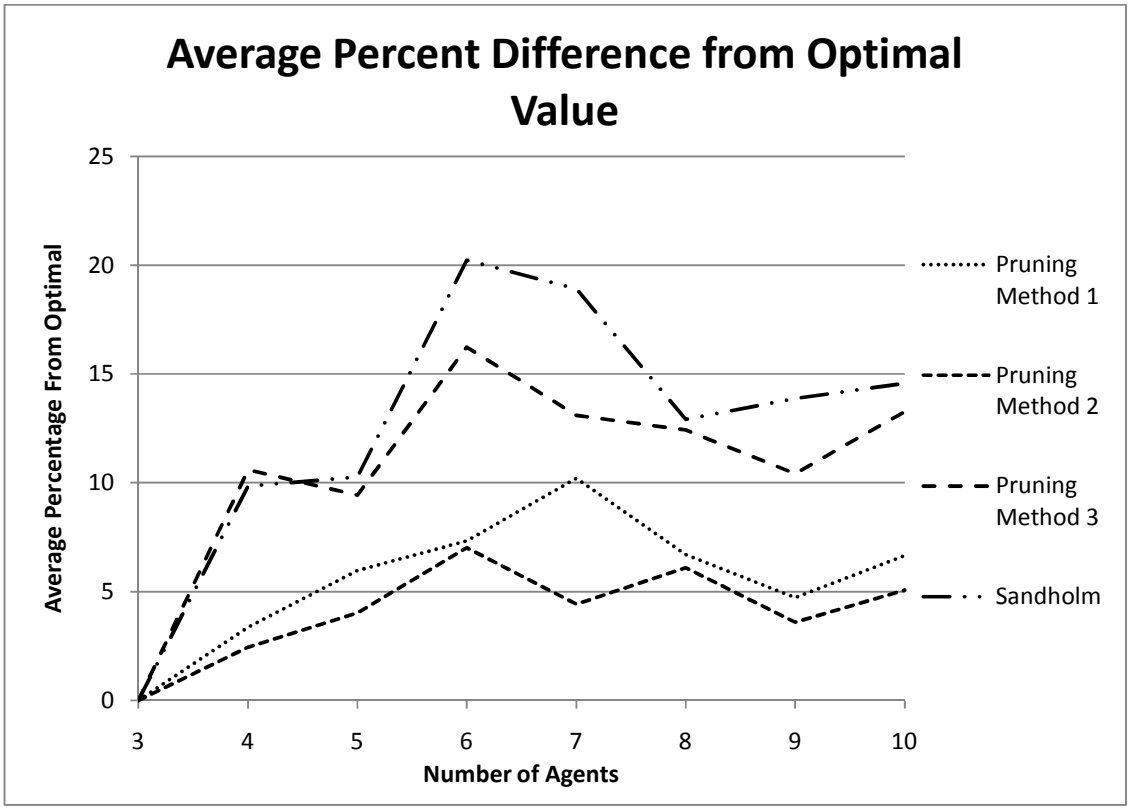

Figure 30. A graph comparing the average percentage difference between our three pruning methods and Sandholm's algorithm compared to the optimal additive reward value found in the entire coalition structure graph.

As can be seen from the two previous graphs, the method for searching for the optimal coalition structure using Sandholm's algorithm gives us less optimal values than using our three pruning methods. On average, the optimal reward value found by using Sandholm's algorithm was five to fifteen percent farther away from the optimal reward value found by using our pruning methods. Using our pruning methods, we are able to find a more optimal additive node while generating fewer nodes in the coalition structure graph. We have been able to show that our approach to the finding the optimal coalition structure reduces the number of nodes explored and also increase the additive reward value found.

5.5 Implementation in Webots

We have been able to successfully incorporate our algorithm for coalition structure generation into our Webots model of the ModRED robot. We begin by placing a group of modules in a simulated environment. We randomly assign each module a utility value that represents how well it is performing in its current configuration. A random module is then assigned to be in charge of calculating the optimal coalition structure of all the modules that responded to the request. Once all of the data has been collected, the module in charge then runs the algorithm for optimal coalition structure generation that we described in Chapter 4. When the algorithm is complete, the module in charge has a set of all the coalitions that should be formed. In our current implementation, the in-charge module then randomly selects a module from each coalition to be the leader of that coalition. The coalition's leaders are then notified that they are leaders and are told which modules in are in their coalition.

Each coalition leader is then in charge of getting the modules in their coalition into a chain. The leader begins by selecting the module in its coalition it is closest to and calculates the angle between them. The leader then rotates to that bearing and tells the other module to move to its location. Once close enough, the modules lock their connectors and form a chain. The process then repeats for the next closest module until all modules in the coalition have joined into a single chain.

A series of screenshots from the implementation of the algorithm in Webots can be seen in Figure 31. In this situation, we have put four ModRED modules randomly in an

environment and oriented them in different directions. We then assigned a random module to begin the coalition structure generation process. As can be seen from the images, in this instance the best coalition structure consisted of two chains of modules each containing two modules. The modules orientated themselves properly and were able to form two chains.

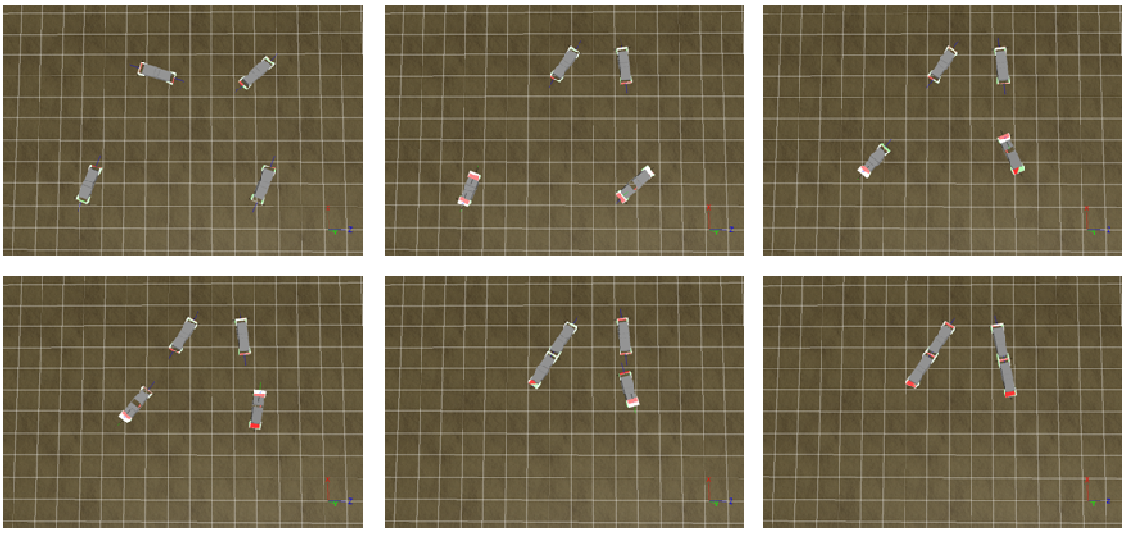

Figure 31. A set of four ModRED modules performing dynamic reconfiguration using our modified MDP algorithm. The modules begin as four individual coalitions but then form two coalitions each containing two modules.

5.6 Summary

As we have shown, our method for creating a pruned coalition structure graph is capable of generating near-optimal coalition structures. We are able to significantly reduce the time complexity to create the graph as well as reduce the number of nodes in the graph from $O(n^n)$ to $O(3^n)$. In looking at the results from our experimentation, the pruning method that keeps the child with the best reward value, the child with the median reward

value, and the child with the lowest reward value produces the best results in generating near-optimal nodes. In combination with our modified MDP approach, our algorithm provides an effective solution to the optimal coalition structure generation problem.

Chapter 6

Future Work

We have shown how our approach to solving the optimal coalition structure generation problem can be solved using a modified Markov Decision Process and a pruned coalition structure graph. From an implementation standpoint, we were able to successfully implement our optimal coalition structure generation algorithm for experimentation as well as within our Webots implementation of the ModRED robot.

Within the generation of the pruned coalition structure graph, there is room for advancement of our pruning methods. The three methods of pruning we introduced can be compared to other methods of pruning. In our pruning methods, we always kept three children nodes, yet it is possible that a better pruning solution could be found by keeping fewer children or by introducing a method of keeping a variable number of children nodes.

In our current implementation of the algorithm within Webots, our process for coalition creation is very simple. In the current process that is described in section 5.5, we discussed how the leaders of the coalitions are picked and how one module at a time move towards the leader in order to form the chain of modules. For future enhancements, not only could the leader and the other module move towards each other, but not all modules have to necessarily join with the leader right away. For example, if we have

four modules that joining into a single coalition, it might be more time effective to have module 1 and module 2 form a chain and at the same time have module 3 and module 4 form a chain, and then have the two chains move towards each other.

Chapter 7

Conclusion

The optimal coalition structure formation is a problem that has been studied and researched from many different angles. The goal of forming the optimal coalition structure is to create coalitions that help lead to the optimal performance of all of the modules in the environment. Coalitions of different sizes are needed to perform different tasks, as some coalitions might be too large or too small to perform the given task.

In this thesis, we have proposed an alternative solution to the optimal coalition structure generation problem and applied it to a novel chain-type MSR, ModRED. To solve the problem of the optimal coalition structure, we outlined an algorithm that uses a modified Markov Decision Process to account for uncertainty in the valuation a coalition and therefore a coalition structure receives. We have also introduced a method for generating and pruning the coalition structure graph which still leads to a near-optimal solution to the problem.

There are several future directions that can be researched starting from the topics explored in this thesis. Compact and distributed representations of MDPs such as decentralized MDPs (dec-MDP) (Becker, Zilberstein, Lesser, & Goldman, 2004), multi-MDPs (MMDPs) (Kumar & Zilberstein, 2009) and sparsely interactive MDPs (SIMDPs) (Spaan & Melo, 2008) are some of the proposed models of MDPs that can be investigated

for a more succinct representation of the MDP proposed in Section 4 of this thesis. Similarly, research can be done to extend distributed algorithms for solving coalition games such as DCVC (Rahwan, Michalak, Sroka, & Wooldridge, 2010), to include uncertainty in the coalition structure graph space along the lines of the work in this thesis. Yet another research direction is to develop realistic representations of terrains, closely resembling extra-terrestrial environments within the Webots simulator for more accurate testing of ModRED's operation. Finally, implementing and testing the algorithm proposed in this thesis on the physical ModRED robot is a challenging and exciting research direction.

Modular self-reconfigurable robots provide a viable option for the future of space exploration. Not only are they capable of handling a wide variety of tasks, they are capable of performing tasks that they were never designed to perform due to their ability toself-reconfigure and form new coalitions. MSR's ability to adapt to new environments and figure out how to accomplish a given task makes them a promising option in space as well as here on earth.

Bibliography

Aziz, H., & de Keijzer, B. (2011). Complexity of Coalition Structure Generation.

Becker, R., Zilberstein, S., Lesser, V., & Goldman, C. (2004). Solving Transition Independent Decentralized Markov Decision Processes. *Journal of Artificial Intelligence Research* , 22:423-455.

Castano, A., Behar, A., & Will, P. (2002). The Conro modules for reconfigurable robots. *IEEE/ASME Transactions on Mechatronics , 7* (4), 403-409.

Chalkiadakis, G. (2007). *A Bayesian Approach to Multiagent Reinforcement Learning and Coalition Formation under Uncertainty.* Ph.D. Thesis, University of Toronto, Toronto, Ontario.

Chu, K. D., & Nelson, C. A. (2011). Design of a Four-DOF Modular Self-Reconfigurable Robot with Novel Gaits. *Proceedings of the ASME 2011 International Design Engineering Techniques Conferences (In Review).* Washington, D.C.

Curtis, S; Brandt, M; Bowers, G; Brown, G; NASA Goddard Space Flight Center. (2007). Tetrahedra Robotcs for Space Exploration. *Aerospace and Electronic Systems Magazine* , 22-30.

Cyberbotics Ltd. (2011). *Webots Home*. Retrieved February 28, 2011, from Cyberbotics Website: http://www.cyberbotics.com

Guestrin, C., Koller, D., Parr, R., & Venkataraman, S. (2003). Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research , 19*, 399-468.

Horling, B., & Lesser, V. (2005). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review , 19* (4), 281-316.

iMobot - an Intelligent Reconfigurable Mobile Robot. (n.d.). Retrieved April 11, 2011, from UC Davis Integratio nEngineering Laboratory: http://iel.ucdavis.edu/projects/imobot/home.html

Kamimura, A., Yoshida, E., Murata, S., Kurokawa, H., Tomita, K., & Kokaji, S. (2008). Distributed Self-Reconfiguration of M-TRAN III Modular Robotic System. *The International Journal of Robotics* , 373-386.

Kuman, A., & Zilberstein, S. (2009). Event-Detecting Multi-Agent MDPs: Complexity and Constant-Factor Approximation. *Proceedings of the 21st International Joint Conference on Artificial Intelligence.*

Lee, W., & Sanderson, A. (1998). Dynamic simulation of tetrahedron-based Tetrobot. *Intelligent Robots and Systems , 1*, 630-635.

Moeckel, R., Faquier, C., Drapel, K., Dittrich, E., Upegui, A., & Ijspeert, A. (2006). Exploring adaptive locomotion with YaMor: a novel autonomous modular robot with Bluetooth interface. *Industrial Robot: An International Journal , 33* (4), 285-290.

Multi-Agent Systems. (2010, October 30). Retrieved February 14, 2011, from AAAI Web Site: http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/MultiAgentSystems

Myerson, R. B. (1997). *Game Theory: Analysis of Conflict.* Cambridge, Massachusetts: Harvard University Press.

Rahwan, T. (2007). *Algorithms for Coalition Formation in Multi-Agent Systems.* Ph.D. Thesis, University of Southampton, Southampton, UK.

Rahwan, T., Michalak, T., Sroka, J., & Wooldridge, M. (2010). A Distributed Algorithm for Anytime Coalition Structure Generation. *Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent Systems*, (pp. 1007-1014). Toronto, Canada.

Ray, D. (2008). *A Game-Theoretic Perspective on Coalition Formation* (1st ed.). Oxford University Press, USA.

Russel, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Salemi, B., Moll, M., & Shen, W. (2006). Superbot: A deployable, multi-functional, and modular self-reconfigurable robotic system. *Intelligent Robots and Systems, IEEE/RSJ International Conferrence*, (pp. 3636-3641).

Sandholm, T., Larson, K., Andersson, M., Shehory, O., & Tohme, F. (1999). Coalition structure generation with worst case guarantees. *Artificial Intelligence , 111*, 209-238.

Shoham, Y., & Leyton-Brown, K. (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* New York, NY: Cambridge University Press.

Siegwart, R., & Nourbaksh, A. (2004). *An Introduction to Autonomous Mobile Robotics.* Cambridge, MA: The MIT Press.

Spaan, M., & Melo, F. (2008). Interaction-drivern Markov Games for Decentralized Multiagent Planning under Uncertainty. *Proceedings of the International Joint Conference on Autonomous Agents and Multi Agent Systems*, (pp. 525-532).

Stoy, K., Brandt, D., & Christensen, D. J. (2010). *Self-Reconfigurable Robots: An Introduction.* Cambridge, MA: The MIT Press.

Suh, J., Homans, S., & Yim, M. (2002). Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics. *IEEE Robotics & Automation Magazine , 4*, 4095-4101.

Suijs, J., & Borm, P. (1999). Stochastic Cooperative Games: Superadditivity, Convexity and Certainty Equivalents. *Journal of Games and Economic Behavior* , 27:331-345.

Townsend, J; Biesiadecki J; Collins, C; Jet Propulsion Lab., California Inst. of Technology. (2010). ATHLETE mobility performance with active terrain compliance. *Aerospace Conference* (pp. 1-7). Big Sky, MT: 2010 IEEE.

Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* Cambridge, Massachusetts: The MIT Press.

Yim, M. (1994). Locomotion Gaits with Polypod. *Video Proc. of the IEEE Intl. Conf. on Robotics and Automation.* San Diego, CA.

Yim, M., Zhang, Y., Roufas, K., Duff, D., & Eldershaw, C. (2003). Connecting and disconnecting for chain self-reconfiguration with PolyBot. *IEEE/ASME Transactions on Mechatronics , 7* (4), 442-451.