The Dissertation Committee for Patrick Madeira MacAlpine
certifies that this is the approved version of the following dissertation:

Multilayered Skill Learning and Movement Coordination for Autonomous Robotic Agents

Committee:

Peter Stone, Supervisor

Dana Ballard

Magnus Egerstedt

Risto Miikkulainen

Scott Niekum

Multilayered Skill Learning and Movement Coordination for Autonomous Robotic Agents

by

Patrick Madeira MacAlpine

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2017

Acknowledgments

First and foremost I want to thank my advisor Peter Stone for his guidance during the long and sometimes arduous process of completing a Ph.D. I am thankful for the freedom Peter allowed me in pursing multiple research problems that I was interested in working on, and I have enjoyed working with Peter both in academia as well as on the soccer field. I also want to thank my other dissertation committee members for their valuable advice and feedback: Dana Ballard, Magnus Egerstedt, Risto Miikkulainen, and Scott Niekum.

I feel fortunate to have been a member of the Learning Agents Research Group (LARG), and want to thank other members of the group who have helped me to flesh out research ideas during my time as a graduate student. This includes Noa Agmon, Michael Albert, Stefano Albrecht, Tsz-Chiu Au, Samuel Barrett, Doran Chakraborty, Katie Genter, Josiah Hanna, Justin Hart, Matthew Hausknecht, Todd Hester, Shivaram Kalyanakrishnan, Brad Knox, Juhyun Lee, Matteo Leonetti, Elad Liebman, Jake Menashe, Sanmit Narvekar, Michael Quinlan, Guni Sharon, Jivko Sinapov, Daniel Urieli, Garrett Warnell, Ruohan Zhang, and Shiqi Zhang among others. Also a special thanks to Stacy Miller for handling the many complicated logistics of travel and reimbursements for the lab.

A highlight of my graduate studies has been working on and attending RoboCup, and I am appreciative of the other members of the RoboCup community whom I have had the opportunity to work with and share research ideas. In

particular I want to thank both past and present UT Austin Villa RoboCup 3D simulation team members, listed in Appendix E.8, whom I have had the pleasure of working with—their contributions have aided in the success of the team as well as have helped to enable some of my research.

I would like to thank the UTCS IT department, and in particular Amy Bush and Kay Nettle, for helping to run and maintain the department's distributed computing cluster. Without their support my research would be nearly impossible to carry out. I also want to thank the department's graduate coordinators, Lydia Griffith and Katie Dahm, for their help in ensuring that I completed all the necessary departmental requirements of the Ph.D. program.

I want to thank Devika Subramanian who helped spark my interest in artificial intelligence while an undergraduate at Rice University. Additionally, I want to thank Shay Harnoy, Mark Lai, Juan Navarro, and Gary Printy whom I spent many hours and long nights with while working on problem sets at Rice—working with them helped instill in me an appreciation for academia and a desire to work on research.

Last but not least I want to thank my family for encouraging me to both start and ultimately complete a Ph.D. Without their love and support I would not be where I am today.

Patrick Madeira MacAlpine

The University of Texas at Austin

August 2017

Multilayered Skill Learning and Movement Coordination for Autonomous Robotic Agents

Publication No. _____

Patrick Madeira MacAlpine, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Peter Stone

With advances in technology expanding the capabilities of robots, while at the same time making robots cheaper to manufacture, robots are rapidly becoming more prevalent in both industrial and domestic settings. An increase in the number of robots, and the likely subsequent decrease in the ratio of people currently trained to directly control the robots, engenders a need for robots to be able to act autonomously. Larger numbers of robots present together provide new challenges and opportunities for developing complex autonomous robot behaviors capable of multirobot collaboration and coordination.

The focus of this thesis is twofold. The first part explores applying machine

learning techniques to teach simulated humanoid robots skills such as how to move or walk and manipulate objects in their environment. Learning is performed using reinforcement learning policy search methods, and layered learning methodologies are employed during the learning process in which multiple lower level skills are incrementally learned and combined with each other to develop richer higher level skills. By incrementally learning skills in layers such that new skills are learned in the presence of previously learned skills, as opposed to individually in isolation, we ensure that the learned skills will work well together and can be combined to perform complex behaviors (e.g. playing soccer). The second part of the thesis centers on developing algorithms to coordinate the movement and efforts of multiple robots working together to quickly complete tasks. These algorithms prioritize minimizing the makespan, or time for all robots to complete a task, while also attempting to avoid interference and collisions among the robots. An underlying objective of this research is to develop techniques and methodologies that allow autonomous robots to robustly interact with their environment (through skill learning) and with each other (through movement coordination) in order to perform tasks and accomplish goals asked of them.

The work in this thesis is implemented and evaluated in the RoboCup 3D simulation soccer domain, and has been a key component of the UT Austin Villa team winning the RoboCup 3D simulation league world championship six out of the past seven years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With advances in technology expanding the capabilities of robots, while at the same time making robots cheaper to manufacture, robots are rapidly becoming more prevalent in both industrial and domestic settings. It is estimated that by 2025 about 1.2 million advanced robots will be added to and deployed in these settings in the U.S. alone [1]. Such an increase in the number of robots, and the likely subsequent decrease in the ratio of people currently trained to directly control the robots, will necessitate more robots to be able to act autonomously. In addition to the heightened importance of robot autonomy, larger numbers of robots present together in the same environment will provide new challenges and opportunities for multirobot collaboration and coordination. More robots, coupled with the advent of improved robot capabilities, motivate and provide a fertile ground for developing autonomous robot and multirobot behaviors capable of completing complex tasks.

The focus of this thesis is twofold. The first part explores using machine learning techniques—automated computing algorithms and methods that adapt or learn from data—to teach humanoid robots skills (e.g. how to move or walk and

manipulate objects in their environment) within a realistic physics-based simulator. Multiple lower level skills are incrementally learned and combined with each other to develop richer higher level skills for use in complex behaviors. After robots have developed higher level skills, and can complete single robot complex behaviors, the second part of the thesis segues to the development of algorithms to coordinate the movement and efforts of multiple robots working together to quickly complete tasks. These algorithms prioritize minimizing the makespan, or time for all robots to complete a task, while also attempting to avoid interference and collisions among the robots. An underlying objective of this research is to develop techniques and methodologies that allow autonomous robots to robustly interact with their environment (through *skill learning*) and with each other (though *movement coordination*) in order to perform tasks and accomplish goals asked of them.

The work in this thesis is implemented and evaluated in simulation using physically realistic simulated humanoid robots in the RoboCup 3D simulation soccer domain. Working in simulation enables the use of large-scale machine learning techniques where learning can be performed on hundreds of simulated robots in parallel running as fast as computationally possible (faster than real-time). Learning in simulation avoids many of the difficulties and limitations of learning directly on physical robots, including time and cost constraints as well as wear and tear on robots. There is a drawback to learning in simulation, however, as policies learned in simulation are often not directly transferable to physical robots due to differences between a simulator and the real world. Although not a focus of this thesis, there is work to bridge the gap between learning in simulation and on physical robots [46, 54, 36, 85].

The remainder of this chapter is organized as follows. Section 1.1 presents

the research question that this thesis focuses on answering. Section 1.2 enumerates and describes each of the contributions of this thesis, and Section 1.3 provides an organizational overview and road map of the dissertation.

1.1 Research Question

Given the preceding problem description and motivation, the key question to be addressed in this thesis is the following:

How can autonomous simulated mobile robots leverage extensive computational resources to learn complex multilayered skills, and then apply these learned skills to perform coordinated behavior in spatial, time-pressured environments?

To answer this question this thesis focuses on two topics we believe are most pertinent for skill creation and coordinated behavior of autonomous robots: skill learning and movement coordination. First, robots must acquire the necessary skills to perform tasks in their environment. Then, using these acquired skills, autonomous robots need to coordinate their behavior and movement to efficiently complete tasks asked of them.

Skill learning for robots is an active area of research [139]. Recent work in this space has focused on learning directly on physical robots, and employing techniques such as learning from demonstration, for which sample complexity is an important consideration [18, 134]. Other work has focused on learning parameterized skills that generalize to different tasks presented to a robot [37]. The work in this thesis differs in than it is concerned with learning multiple skills, through reinforcement learning policy search methods, that can be combined to perform complex behaviors. The focus of the work is less on learning individual skills in isolation, but instead on

developing learning methodologies and designing optimization tasks to produce skills that *work well together*. As learning is carried out in simulation, sample complexity is not a primary concern; rather learning is targeted to best take advantage of parallel computation resources provided by distributed computing clusters.

Problems in coordination and role assignment of multiple robots have also been explored. Previous work on assigning agents to target positions has often focused on minimizing the sum of distances all agents must travel which is the well known *assignment problem* [138]. The work in this thesis differs as it minimizes the makespan (time for all robots to reach target goal positions) instead of the sum of distances traveled. Minimizing the makespan is a decisive factor in performance when robots are moving to target positions to complete a shared task where all robots must be in place before the task can be completed and/or started.

1.2 Contributions

This thesis provides the following contributions:

1. **Methodologies for learning complex robot skills in simulation.** This thesis presents new paradigms for constructing and learning complex robot behaviors through the introduction and use of *overlapping layered learning*, an extension of the hierarchical layered learning paradigm [155]. Layered learning enables learning of complex behaviors by incrementally learning a series of sub-behaviors where learning of subsequent layers depend on previously learned behavior layers. Overlapping layered learning presents ways of combining learning of different behavior layers that extend the traditional sequential layered learning methodology through the use of overlapping or shared parameter sets across behavior layers.

2. **Development and analysis of multirobot role assignment functions.** In multirobot spatial settings robots may need to be assigned to move to different locations or role positions in order to complete a task. This thesis presents and analyzes different role assignment functions for assigning robots to role positions. Properties of role assignment functions analyzed include total distance traveled by all robots, makespan completion time, dynamic consistency, collision avoidance, and standard deviation of the distances traveled by robots.

3. **Novel algorithms for multirobot movement coordination.** Often robots need to be able to move around and interact with their environment in the presence of other robots. While doing so, it is important that robots do not interfere with each other and/or collide with other robots or obstacles in the environment. This thesis surveys existing approaches for robot movement coordination and introduces new algorithms that focus on important considerations of formation completion time, collision avoidance, and scalability. Specifically scalable polynomial time role assignment algorithms known as *SCRAM* that avoid collisions among robots and minimize the makespan, or time for robots to complete a formation, are introduced.

4. **Complete autonomous robot soccer playing agent.** Another contribution of this thesis is that of the UT Austin Villa RoboCup 3D simulation league team, a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years. This agent incorporates the ideas and algorithms presented in this thesis, thus serving as a proof of concept of them, and a public base code release of the agent provides a testbed for future research in multirobot systems.

5. **Detailed empirical evaluation of presented learning methodologies and coordination algorithms.** This thesis provides detailed empirical evaluations of the presented learning methodologies and movement coordination algorithms. Results from within the RoboCup 3D simulation competition are analyzed as well as controlled experiments external to the competition.

1.3 Dissertation Overview

While this dissertation is intended to be read sequentially, it is not necessary to do so to understand all parts of it. Background information on direct policy search in reinforcement learning (Section 2.1) is only applicable to those chapters covering overlapping layered learning (Chapters 3 and 4), and details of the RoboCup 3D simulation domain (Section 2.2) are pertinent to just the chapters on robot soccer (Chapters 4, 7, and 8). Additionally, the chapters on overlapping layered learning can be read independently of the chapters on role assignment (Chapters 5, 6, and 7). For ease of navigation, a flow chart of the dependencies of the parts of this thesis is provided in Figure 1.1.

The remainder of this dissertation is organized as follows.

Chapter 2 – Background: This chapter provides background information useful for understanding subsequent chapters. Specifically it provides background information on direct policy search in reinforcement learning and the CMA-ES algorithm needed for Chapters 3 and 4, and also gives an overview of the RoboCup 3D simulation domain used in Chapters 4, 7, and 8.

Chapter 3 – Overlapping Layered Learning: This chapter presents the overlapping layered learning paradigm—one of the primary contributions of this

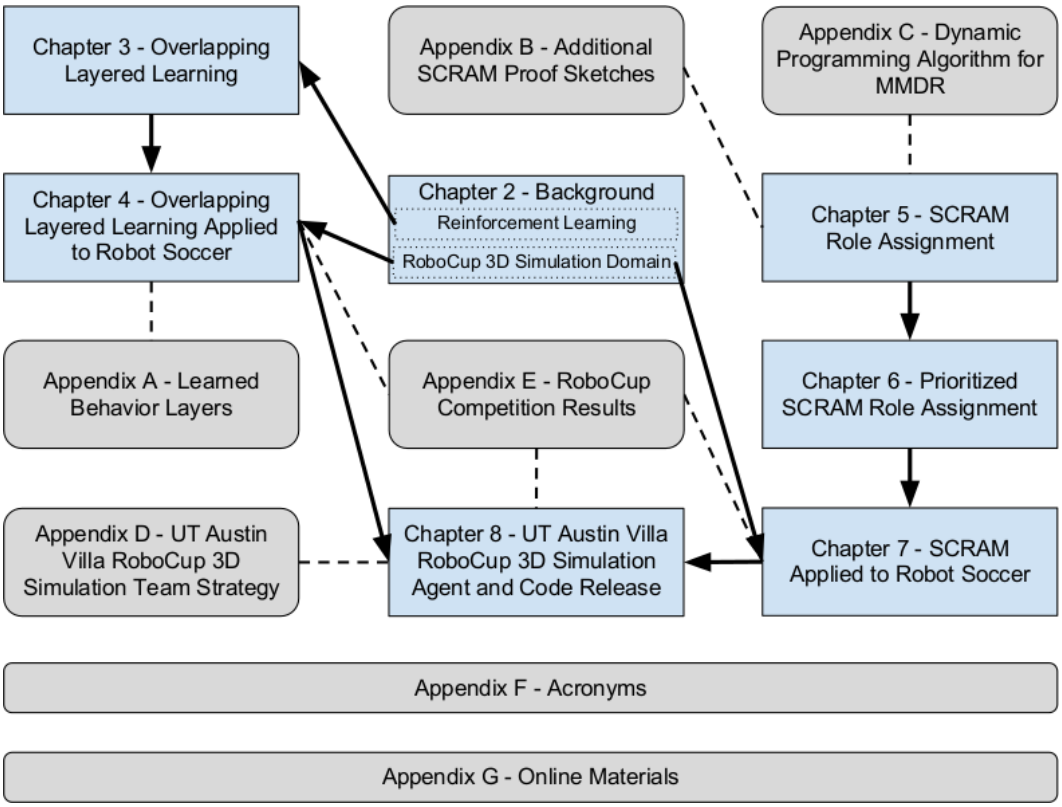


Figure 1.1: Block diagram showing dependencies between parts of this thesis. Solid arrows represent dependencies between chapters, with the solid arrows pointing in the direction of the order in which chapters should be read. The dashed lines represent associations between chapters and appendices.

dissertation (contribution 1 in Section 1.2). Overlapping layered learning is a major extension to the layered learning [155] hierarchical machine learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors. Overlapping layered learning allows learning certain behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap.

Chapter 4 – Overlapping Layered Learning Applied to Robot Soccer:

This chapter presents a case study of overlapping layered learning (presented in Chapter 3) applied to robot soccer which showcases overlapping layered learning as a paradigm for efficient behavior learning. The provided analysis of the overlapping layered learning methodologies serves as part of contribution 5 in Section 1.2.

Chapter 5 – Scalable Collision-avoiding Role Assignment with Minimal-makespan (SCRAM): This chapter introduces SCRAM role assignment algorithms for formational positioning of mobile agents, and presents theoretical and empirical analysis of the role assignment problem—how to assign agents to target positions in a one-to-one mapping such that the time for all agents to reach their assigned target positions (makespan) is minimized while avoiding collisions among the agents. Specifically this chapter addresses primary thesis contributions 2 (role assignment function analysis) and 3 (role assignment algorithms) in Section 1.2. SCRAM role assignment algorithms run in polynomial time and can scale to thousands of agents.

Chapter 6 – Prioritized SCRAM Role Assignment: This chapter introduces an extension to SCRAM role assignment presented in Chapter 5 allowing for subsets of role positions to be given different priorities, and in doing so further addresses thesis contributions 2 and 3 in Section 1.2.

Chapter 7 – SCRAM Applied to Robot Soccer: This chapter presents case studies and analysis of SCRAM role assignment—introduced in Chapter 5—applied to robot soccer. The chapter's empirical evaluation of SCRAM fulfills part of thesis contribution 5 in Section 1.2. SCRAM role assignment is used to assign robots to team formation target positions on the soccer field, and prioritized

SCRAM role assignment—introduced in Chapter 6—is utilized in a marking system to defend and cover members of the opposing soccer team.

Chapter 8 – UT Austin Villa RoboCup 3D Simulation Agent and Code Release: This chapter presents the University of Texas at Austin's RoboCup 3D simulation team UT Austin Villa—a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years—and in doing so addresses thesis contribution 4 in Section 1.2. The UT Austin Villa agent incorporates the ideas and algorithms presented in this thesis, thus serving as a proof of concept of them as detailed in Chapters 4 and 7. Furthermore, a public base code release of the UT Austin Villa agent introduced in this chapter provides a testbed for future research in multirobot systems.

Chapter 9 – Related Work: This chapter discusses work related to this thesis.

Chapter 10 – Conclusion and Future Work: This chapter presents ideas for future work and concludes.

Appendix A – Learned Behavior Layers: This appendix provides a detailed description of the different behavior layers for robot soccer learned through an extensive layered learning approach incorporating overlapping layered learning layered learning as discussed in Chapter 4.

Appendix B – Additional SCRAM Proof Sketches: This appendix provides proof sketches for properties of role assignment functions discussed in Chapter 5: minimizing the makespan, avoiding collisions, and dynamic consistency.

Appendix C – Dynamic Programming Algorithm for MMDR: This appendix details the dynamic programming algorithm for computing the Mini-

mum Maximal Distance Recursive (MMDR) role assignment function that is compared against SCRAM role assignment algorithms in Section 5.4.

Appendix D – UT Austin Villa RoboCup 3D Simulation Team Strategy: This appendix provides details of some of the strategy components used by the UT Austin Villa agent team presented in Chapter 8.

Appendix E – RoboCup Competition Results: This appendix provides RoboCup competition results of the UT Austin Villa RoboCup 3D simulation team from 2011–2017 referenced in Chapters 4, 7, and 8.

Appendix F – Acronyms: This appendix provides a lookup table of acronyms used in this dissertation.

Appendix G – Online Materials: This appendix provides links to some of the online content referenced in this dissertation.

Chapter 2

Background

This chapter provides background information useful for understanding further chapters. Section 2.1 provides information on direct policy search in reinforcement learning and the CMA-ES algorithm needed for chapters covering overlapping layered learning: Chapters 3 and 4. Section 2.2 gives an overview of the RoboCup 3D simulation domain used in Chapters 4, 7, and 8.

2.1 Reinforcement Learning

Reinforcement learning (RL) [159] is a type of of machine learning inspired by behavioral psychology and used for learning in sequential decision making problems. In RL an agent exists in an environment and, given the current state of the world, chooses an action to take. After taking each action, the agent typically receives a reward from the environment. Broadly speaking, the goal of RL is to learn a behavior—a policy mapping the current state of the world to actions—that maximizes the (possibly discounted) cumulative reward over time that the agent receives. Figure 2.1 presents a high level canonical view and description of RL.

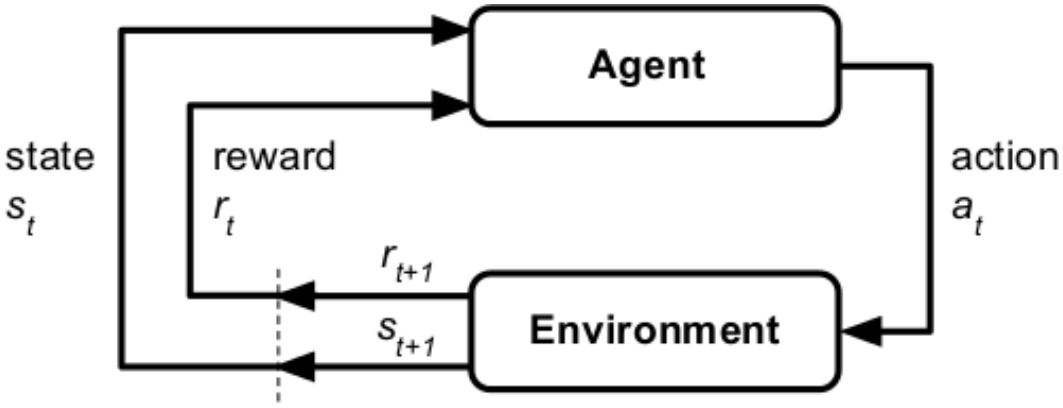


Figure 2.1: A high level view and block diagram of RL. At each time step (t) an agent observes a state (s) of the environment and chooses an action (a) to take. The agent receives a reward (r) after taking an action. Typically the goal of RL is to learn a policy mapping states to actions ($s \mapsto a$) that maximizes the sum of (possibly discounted by γ) rewards over time: $\max_{\pi:s\mapsto a} \sum_{t=1}^{n} \gamma^t r(s_t, a_t)$.

2.1.1 Direct Policy Search

A variant of RL is direct policy search. Rather than learning a specific mapping of states to actions, direct policy search focuses on learning or optimizing parameter values for a parameterized policy, where the parameter values of the policy determine what action the policy selects given the current state of the environment. During learning an agent performs some optimization task, and instead of an agent receiving a reward from the environment after every action that the agent takes, an agent receives an overall return or fitness value at the conclusion of the optimization task as a measure of how well the agent performed on the task. Given the parameter values of a policy, and the fitness value received by an agent when performing an optimization task using that policy, an optimization algorithm attempts to adjust the parameter values of the policy such that the agent will receive higher fitness values on the optimization task. Figure 2.2 provides a high level view and description

of this optimization process used in direct policy search.

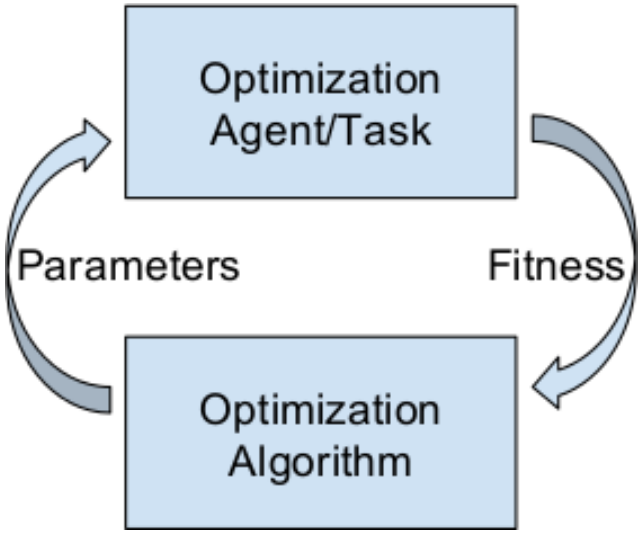


Figure 2.2: The optimization process used in direct policy search. An optimization algorithm produces candidate set of parameter values for a parameterized policy used by an agent while performing an optimization task. At the conclusion of the optimization task, the agent receives a fitness value as a measure of how well it performed on the optimization task. Given the fitness values achieved by an agent using different sets of policy parameter values, the optimization algorithm attempts to produce new sets of policy parameter values to try on the optimization task that will improve an agent's fitness measure on the optimization task.

In Chapters 3 and 4 we use direct policy search, and the CMA-ES optimization algorithm (described next in Section 2.1.2), for robot skill learning.

2.1.2 Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES)

We use the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [57] algorithm for optimizing policy parameter values as described in Section 2.1.1. A complete understanding of CMA-ES is not required for following the work in this thesis, and for comprehension purposes it may suffice to view CMA-ES as a generic

optimization algorithm in Figure 2.2. In this section we provide a brief description of CMA-ES with all details necessary for understanding our use of the algorithm. A full explanation of CMA-ES, including all details necessary for implementation[1] of the algorithm, can be found in the tutorial at [56].

CMA-ES is a derivative-free stochastic optimization algorithm that successively generates and evaluates sets of candidate parameter values for a policy. Once CMA-ES generates a group of candidate parameter value sets (a population), each candidate parameter value set is evaluated with respect to a fitness measure. When all the candidates in the group are evaluated, the next group (or generation) of candidate parameter value sets is generated by sampling from a probability that is biased towards directions of previously successful search steps.

At the start of optimization, CMA-ES is given an initial multivariate normal distribution—with each dimension of the distribution representing a different parameter—as a seed to sample the first group of candidate parameter value sets from. After this first set of candidates have been evaluated, and each member of the population has been assigned a fitness value, the means of this distribution are updated as a weighted by rank average of the top half highest fitness members of the current generation. Weighting by rank, as opposed to relative fitness values, makes CMA-ES invariant under monotonic transformations of the fitness function. Additionally the covariance matrix of the distribution is updated to control the step sizes in each dimension and maximize the likelihood of previously successful search steps. This way of updating both the means and variances of the distribution, which is used to sample candidate parameter set values from for the next generation, can be thought of as a form of natural gradient descent [10, 135]. Figure 2.3 shows

[1] In our work we use an implementation of the CMA-ES algorithm available at https://www.lri.fr/~hansen/cmaes_inmatlab.html#java

an example of how the distribution from which parameter values are sampled is updated across three iterations of CMA-ES.

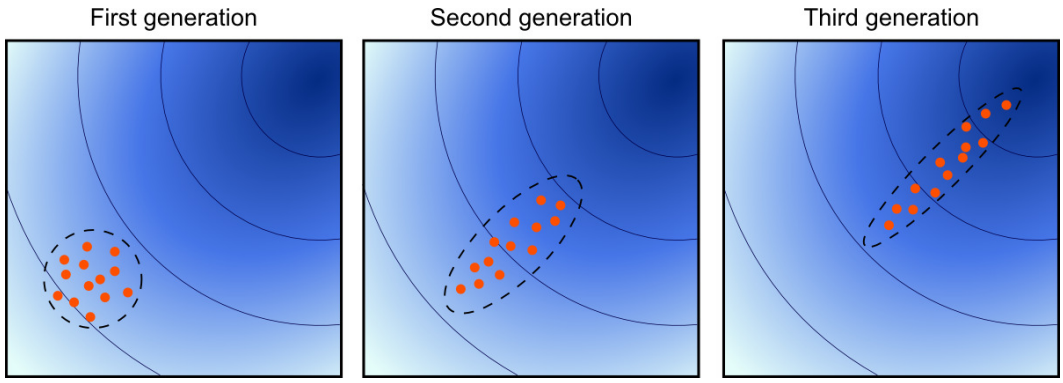


Figure 2.3: CMA-ES searching in a two-dimensional fitness landscape with higher fitness shown in darker blue. Members of a population for each generation are shown as red dots and the dashed lines show the distributions that they are being sampled from. [Image from https://en.wikipedia.org/wiki/CMA-ES#/media/File:Concept_of_directional_optimization_in_CMA-ES_algorithm.png accessed 2015-08-15.]

2.2 RoboCup 3D Simulation Domain Description

Robot soccer has served as an excellent testbed for learning scenarios in which multiple skills, decisions, and controls have to be learned by a single agent, and agents themselves have to cooperate or compete. There is a rich literature based on this domain addressing a wide spectrum of topics from low-level concerns, such as perception and motor control [20, 145], to high-level decision-making [77].

In the RoboCup 3D simulation league, teams of simulated humanoid robots play soccer against each other. Programming humanoid agents in simulation, rather than in reality, brings with it several advantages, such as making simplifying assumptions about the world, low installation and operating costs, and the ability to

automate experimental procedures. All these factors make the RoboCup 3D simulation environment an ideal domain for conducting research in robotics, multiagent systems, and machine learning. For a discussion of some of the research efforts within the RoboCup 3D simulation league see [11].

The RoboCup 3D simulation environment is based on SimSpark [24, 180],[2] a generic physical multiagent systems simulator. SimSpark uses the Open Dynamics Engine[3] (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints.

The robot agents in the simulation are modeled after the Nao robot, which has a height of about 57 cm and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. In order to monitor and control its hinge joints, an agent is equipped with joint perceptors and effectors. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time. Figure 2.4 provides a box model of the robot, and Figure 2.5 shows the robot model's joints.

In addition to the standard Nao robot model, four additional variations of the standard model, known as heterogeneous types, are available for use. The variations from the standard model include two models with changes in leg and arm length

[2] http://simspark.sourceforge.net/
[3] http://www.ode.org/

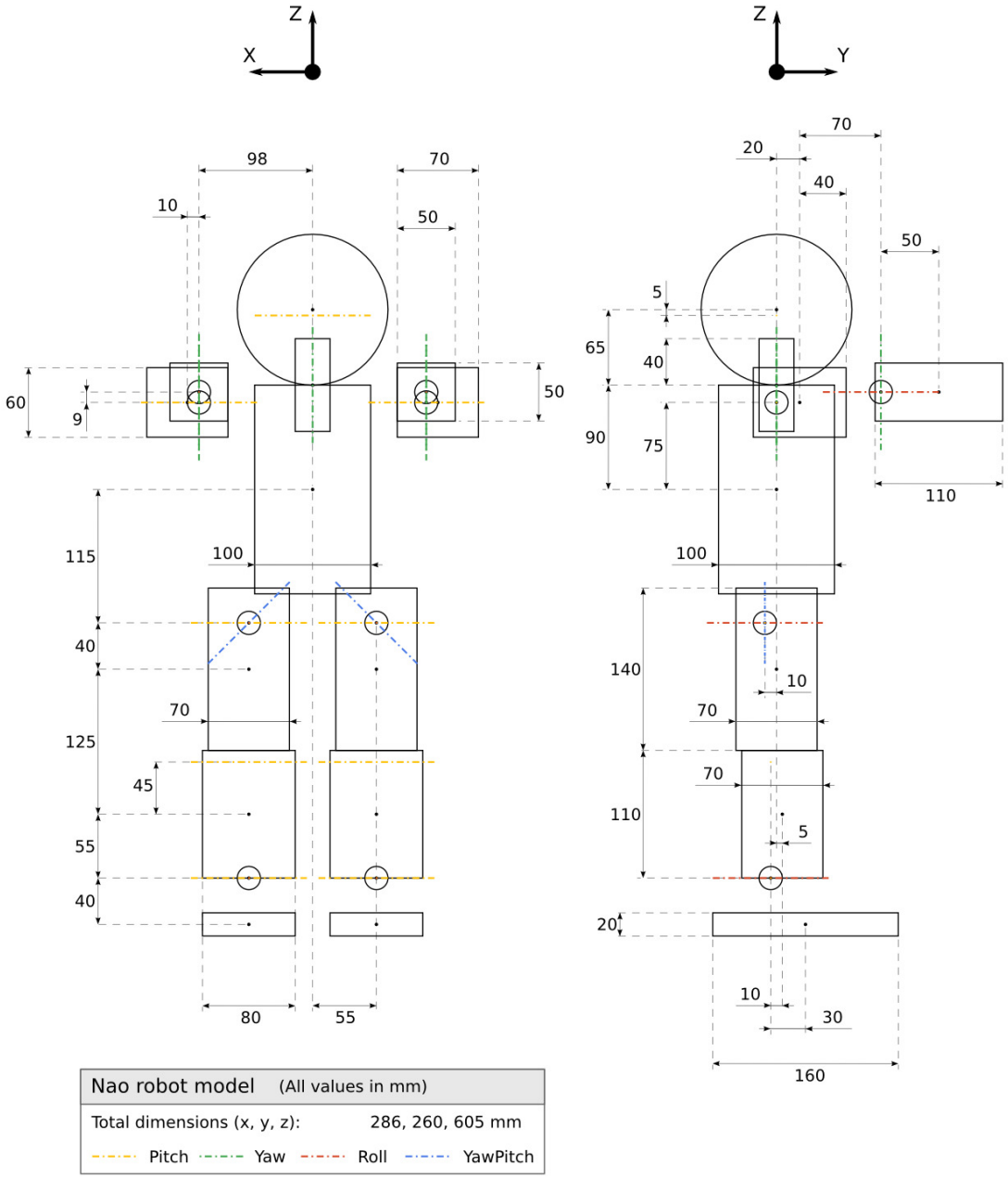


Figure 2.4: Box model diagram of the robot model. [Image from http://simspark.sourceforge.net/wiki/images/4/42/Models_NaoBoxModel.png accessed 2017-07-02.]

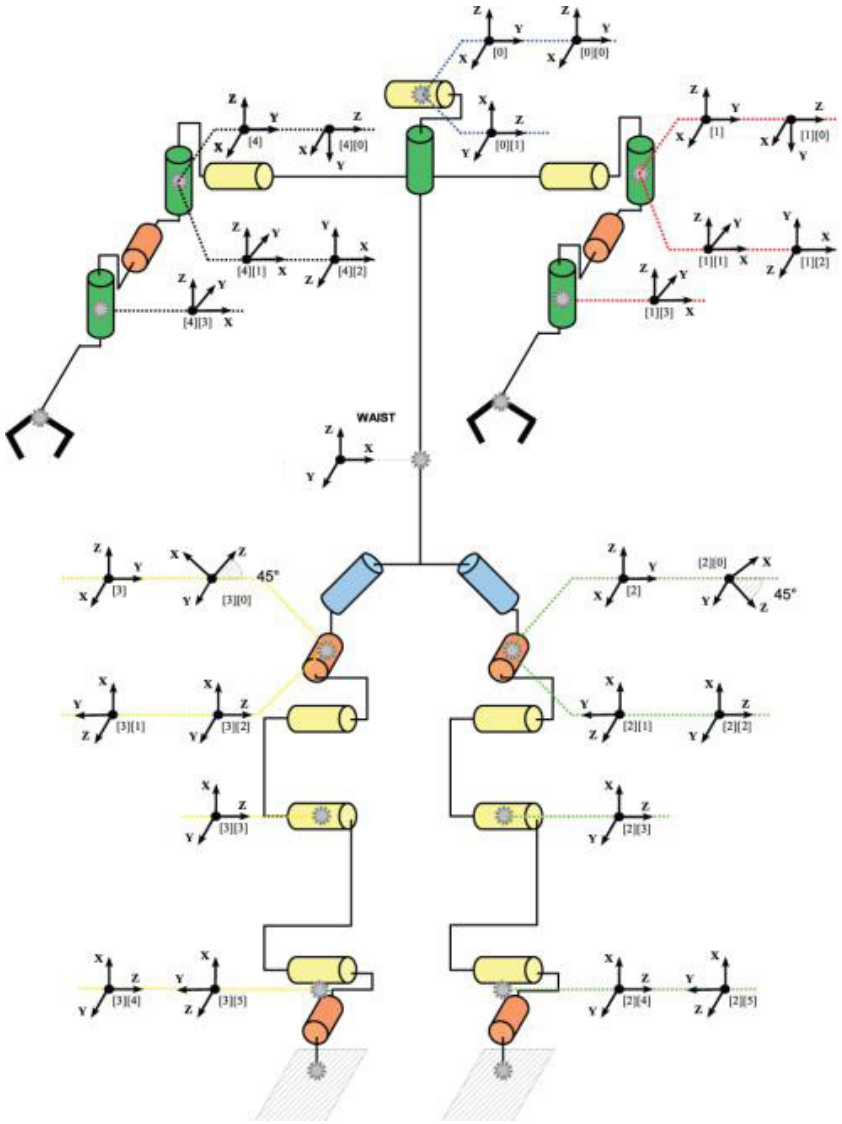


Figure 2.5: Joints of the robot model. [Image from http://simspark.sourceforge.net/wiki/images/d/d0/Models_NaoAnatomy.png accessed 2017-07-02.]

as well as hip width, one model with different joint speeds for its ankle joints, and also a model with the addition of toes to the robot's feet. During 11 versus 11 games, which consist of two five minute halves, teams are required to use at least three different robot models, with no more than seven agents of any single robot type, and no more than nine agents of any two robot types. Figure 2.6 shows a visualization of the different Nao robot types, and a view of the soccer field during a game is shown in Figure 2.7.

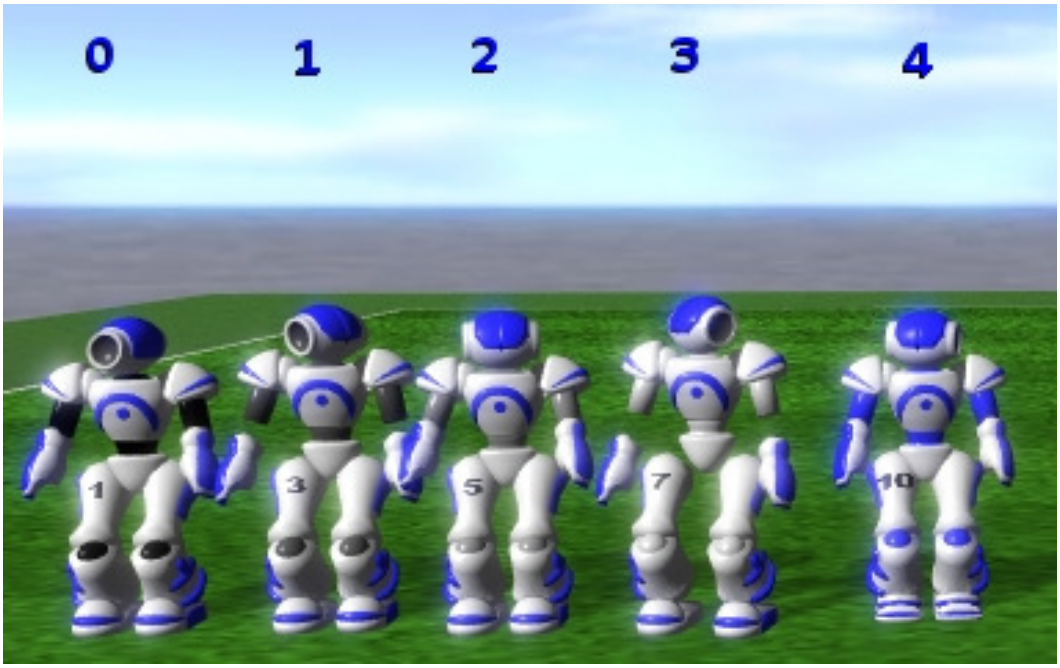


Figure 2.6: Different robot body model types, with the number above an agent corresponding to its robot body type: 0 = standard model, 1 = longer legs and arms, 2 = faster ankle pitch and slower ankle roll, 3 = longest arms and legs as well as wider hips, 4 = toe model.

Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). These objects include landmarks on

Figure 2.7: A view of the soccer field during a 11 versus 11 game.

the field (goal posts and corner flags), field lines, robots (with separate objects for each robot's head, arms, and feet) and the ball. Figure 2.8 shows the locations of landmarks and lines on the 30 m X 20 m in length and width field. There is no occlusion—an agent can see all objects within its vision cone. Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, a single agent can communicate with the other agents every other simulation cycle (40 ms) by sending messages limited to 20 bytes.

During RoboCup 3D simulation league soccer games each team is a provided a single computer to run all of their 11 agents. The server is run on a separate third computer, and a fourth computer is used to run a monitor for visualizing games. It is important that agents are not too computationally intensive such that all 11

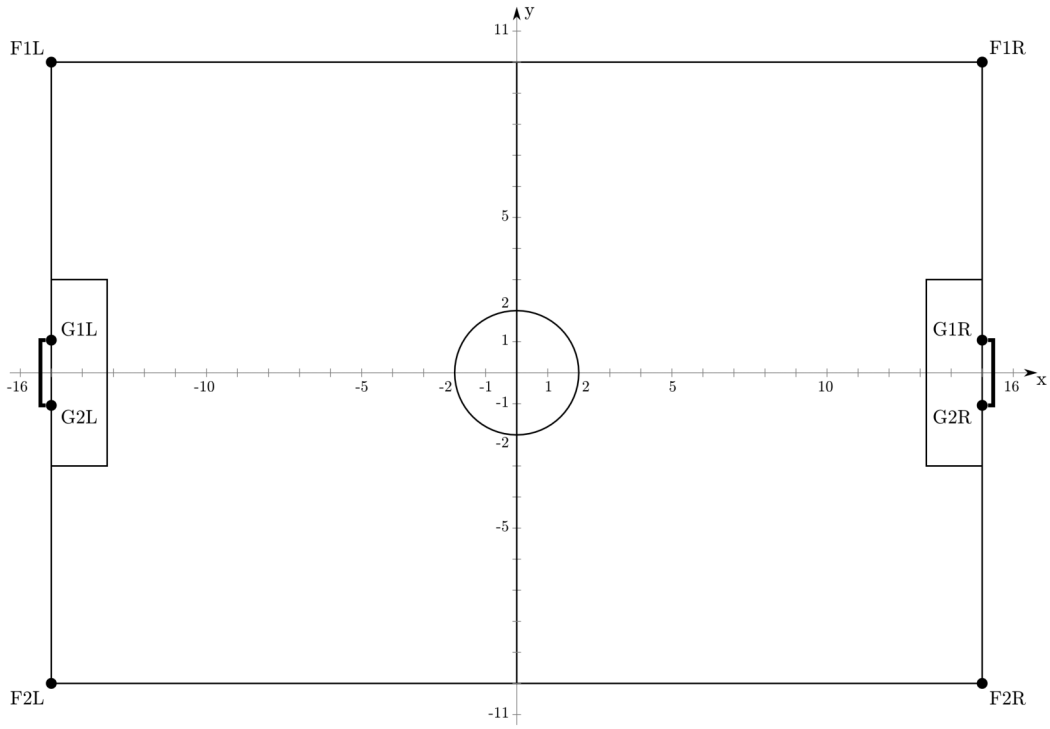


Figure 2.8: Landmarks (F1L, F1R, F2L, F2R, G1L, G1R, G2L, G2R) and lines (in black) on the simulated soccer field. [Image from http://simspark.sourceforge.net/wiki/images/thumb/3/31/SoccerSimulation_FieldPlan.png/600px-SoccerSimulation_FieldPlan.png accessed 2017-07-02.]

agents on a team can respond to the server every 20 ms simulation cycle—if an agent fails to respond to the server during a simulation cycle it may become unstable and fall over due to not updating what torques to put on its joints.

Most rules of 3D simulation league soccer games are modeled after human soccer, however there are a few rules put in a place to help make simulated soccer games run smoothly. Instead of a team being awarded a free kick when a member of the other team commits a foul, the player who committed the foul is immediately moved or "teleported" to a position just outside the sideline near the middle of the field. An automated referee determines when fouls have been committed, and robots

are penalized for charging into each other, having more than three players in their defensive penalty area (to prevent teams from purposely blocking their own goal and making it impossible for the other team to score), as well as having more than two players touching each other at a time (robot collisions can slow down and destabilize the server). Additionally, when the ball goes out of bounds, robots perform a kick-in instead of a throw-in as agents have yet to develop the ability to pick up the ball and perform a throw-in. Rules within the 3D simulation league are changed from year to year in an effort to move closer towards realism and increase the technical challenges from a scientific perspective (keep things scientifically relevant). Some of the rule changes that have occurred from 2011–2017 are summarized in Appendix E.

Chapter 3

Overlapping Layered Learning

This chapter[4] presents the overlapping layered learning paradigm—one of the primary contributions of this dissertation (contribution 1 in Section 1.2). Overlapping layered learning is a major extension to the layered learning [155] hierarchical machine learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors. A key feature of layered learning is that higher layers directly depend on the previously learned lower layers. In layered learning's original formulation, lower layers are frozen prior to learning higher layers. Overlapping layered learning, on the other hand, allows learning certain behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap. An application of overlapping layered learning to robot soccer is discussed in Chapter 4.

The remainder of this chapter is organized as follows. Section 3.1 gives a high level overview of layered learning paradigms including overlapping layered learning. Section 3.2 provides background information on the original layered learning paradigm which is the basis for this work. Section 3.3 specifies and gives examples

[4]This chapter contains material from previously published work in [106].

of the overlapping layered learning paradigm while contrasting it with other layered learning paradigms. Section 3.4 summarizes and mentions directions for future work.

3.1 Overview

Task decomposition is a popular approach for learning complex control tasks when monolithic learning—trying to learn the complete task all at once—is difficult or intractable [152, 174, 175]. Layered learning [155] is a hierarchical task decomposition machine learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors. A key feature of layered learning is that higher layers directly depend on the learned lower layers. In its original formulation, lower layers were frozen prior to learning higher layers. Freezing lower layers can be restrictive, however, as doing so limits the combined behavior search space over all layers. Concurrent layered learning [176] reduced this restriction in the search space by introducing the possibility of learning some of the behaviors simultaneously by "reopening" learning at the lower layers while learning the higher layers. A potential drawback of increasing the size of the search space, however, is an increase in the dimensionality and thus possibly the difficulty of what is being learned.

This chapter considers an extension to the layered learning paradigm, known as *overlapping layered learning*, that allows learning certain parameterized behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap. Overlapping layered learning aims to provide a middle ground between reductions in the search space caused by freezing previously learned layers and the increased dimensionality of concurrent layered learning. Ad-

ditionally, for complex tasks where it is difficult to learn one subtask in the presence of another, it reduces the dimensionality of the parameter search space by focusing only on parts responsible for subtasks working together.

3.2 Layered Learning Paradigm

Table 3.1 summarizes the principles of the original layered learning paradigm which are described in detail in this section.[5]

1. A mapping directly from inputs to outputs is not tractably learnable.
2. A bottom-up, hierarchical task decomposition is given.
3. Machine learning exploits data to train and/or adapt. Learning occurs separately at each level.
4. The output of learning in one layer feeds into the next layer.

Table 3.1: The key principles of layered learning.

Principle 1

Layered learning is designed for domains that are too complex for learning a mapping directly from the input to the output representation. Instead, the layered learning approach consists of breaking a problem down into several task layers. At each layer, a concept needs to be acquired. A machine learning (ML) algorithm abstracts and solves the local concept-learning task.

[5]This section is adapted from [155].

Principle 2

Layered learning uses a bottom-up incremental approach to hierarchical task decomposition. Starting with low-level subtasks, the process of creating new ML subtasks continues until reaching the high-level task that deal with the full domain complexity. The appropriate learning granularity and subtasks to be learned are determined as a function of the specific domain. The task decomposition in layered learning is not automated. Instead, the layers are defined by the ML opportunities in the domain.

Principle 3

Machine learning is used as a central part of layered learning to exploit data in order to train and/or adapt the overall system. ML is useful for training functions that are difficult to fine-tune manually. It is useful for adaptation when the task details are not completely known in advance or when they may change dynamically. In the former case, learning can be done off-line and frozen for future use. In the latter, on-line learning is necessary: since the learner needs to adapt to unexpected situations, it must be able to alter its behavior even while executing its task. Like the task decomposition itself, the choice of machine learning method depends on the subtask.

Principle 4

The key defining characteristic of layered learning is that each learned layer directly affects the learning at the next layer. A learned subtask can affect the subsequent layer by:

- constructing the set of training examples;

- providing the features used for learning; and/or
- pruning the output set.

Formalism

Consider the learning task of identifying a hypothesis h from among a class of hypotheses H which map a set of state feature variables S to a set of outputs O such that, based on a set of training examples, h is most likely (of the hypotheses in H) to represent unseen examples. When using the layered learning paradigm, the complete learning task is decomposed into hierarchical subtask layers $\{L_1, L_2, \ldots, L_n\}$ with each layer defined as

$$L_i = (\vec{F}_i, O_i, T_i, M_i, H_i, h_i)$$

where:

$\vec{\boldsymbol{F_i}}$ is the input vector of state features relevant for learning subtask L_i. $\vec{F}_i =< F_i^1, F_i^2, \ldots >$. $\forall j, F_1^j \in S$.

$\boldsymbol{O_i}$ is the set of outputs from among which to choose for subtask L_i. $O_n = O$.

$\boldsymbol{T_i}$ is the set of training examples used for learning subtask L_i. Each element of T_i consists of a correspondence between an input feature vector $\vec{f} \in \vec{F}_i$ and $o \in O_i$.

$\boldsymbol{M_i}$ is the ML algorithm used at layer L_i to select a hypothesis mapping $\vec{F}_i \mapsto O_i$ based on T_i.

$\boldsymbol{H_i}$ is the policy representation mapping $\vec{F}_i$ to O_i.[6]

[6] Previous work [155, 176] included H_i within M_i, however we separate out H_i for ease of notation.

$\boldsymbol{h_i}$ is the result of running M_i on T_i. h_i is a specific instantiation of H_i and is a function from $\vec{F_i}$ to O_i.

As set out in Principle 2 of layered learning, the definitions of the layers L_i are given a priori. Principle 4 is addressed via the following stipulation. $\forall i < n$, h_i directly affects L_{i+1} in at least one of three ways:

- h_i is used to construct one or more features F_{i+1}^k.
- h_i is used to construct elements of T_{i+1}; and/or
- h_i is used to prune the output set O_{i+1}.

It is noted above in the definition of $\vec{F_i}$ that $\forall j$, $F_1^j \in S$. Since $\vec{F_{i+1}}$ can consist of new features constructed using h_i, the more general version of the above special case is that $\forall i, j$, $F_i^j \in S \cup_{k=1}^{i-1} O_k$.

In the context of this work all learned behaviors—or hypotheses—are represented by parameterized policies, where a parameterized policy with k parameters for layer L_i is the set of parameters $H_i = \{H_i^1, \ldots, H_i^k\}$, and the hypothesis is the corresponding set of the parameters' learned values $h_i = \{h_i^1, \ldots, h_i^k\}$.

3.3 Overlapping Layered Learning Paradigm

As explained in Section 3.2, layered learning is a hierarchical learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors—each learned sub-behavior is a layer in the learning progression. Higher layers depend on lower layers for learning. This dependence can include providing features for learning, such as initial seed values for parameters when H is a parameterized policy, as well as a previous learned layer's behavior being incorporated

into the learning task for the next layer to be learned. In layered learning's original sequential formulation, layers are learned in a sequential bottom-up fashion and, after a layer is learned, it—the learned hypothesis h of the layer—is frozen before beginning learning of the next layer.

Concurrent layered learning, on the other hand, purposely does not freeze newly learned layers, but instead keeps them open during learning of subsequent layers: h_i is not frozen before the training of L_{i+1} begins. Thus, the effect that h_i has on T_{i+1} is not fixed during learning of L_{i+1}, and in fact changes as h_i continues to be learned. We denote the continued learning of h_i as $H_i \subseteq H_{i+1}$, meaning that H_i—the set of of parameters that the values of h_i are assigned to—is a subset of H_{i+1}, and thus the parameters H_i are fully included in what is learned for the hypothesis h_{i+1}—the parameter values h_i are re-learned—in the subsequent layer of learning L_{i+1}. Previously learned layers' behaviors are left open so that learning may enter areas of the behavior search space that are closer to the combined layers' optimum behavior as opposed to being confined to areas of the joint layer search space where the behaviors of previously learned layers are fixed. While concurrent layered learning does not restrict the search space in the way that freezing learned layers does, the increase in the search space's dimensionality can make learning slower and more difficult.

Overlapping layered learning seeks to find a tradeoff between freezing each layer once learning is complete and leaving previously learned layers open. It does so by *keeping some, but not necessarily all, parts of previously learned layers open during learning of subsequent layers.* The part of previously learned layers left open is the "overlap" with the next layer being learned. In this regard concurrent layered learning can be thought of as an extreme of overlapping layered learning

with a "full overlap" between layers. Additionally, overlapping layered learning allows for behaviors to be *learned independently* in parallel—not just sequentially in series—and then *later stitched together* by learning at the "seams" where their influences overlap.

> **Overlapping Layered Learning:** Layered learning both in series and parallel in which some, but not necessarily all, parts of previously learned layers are left open during learning of subsequent layers.

A layer of learning can be partially left open by freezing only a subset of its learned behavior's policy's parameters during subsequent layers of learning. We denote a set of parameters H' as being open and learned in a layer of learning L_i if the set of parameters are included in the layer's learned policy representation H_i (i.e. $H' \subseteq H_i$), and conversely a set of previously learned values h' for parameters H' are frozen if they are still part of the layer of learning—in the context of this work previously learned and frozen parameter values serve to define behavior used in the training task T_i which we denote as $h' \prec T_i$—but are not included in the layer's learned policy representation (i.e. $H' \cap H_i = \emptyset$). A previously learned set of parameter values h' may also be used to initialize or seed values of H_i. We denote such a relationship as $h' \dashrightarrow H_i$

The following are several general scenarios, depicted in the bottom row of Figure 3.1, for overlapping layered learning that help to clarify the learning paradigm and identify situations in which it is useful:

Combining Independently Learned Behaviors (CILB): Two or more behaviors are learned independently in the same or different layers, and then are combined together for a joint behavior at a subsequent layer by relearning

some subset of the behaviors' parameters or "seam" between the behaviors. Let $L_{i,b}$ represent learning of the bth independent behavior in the ith layer of learning. In the case of two independent behaviors learned in parallel in the same layer, which we notate as $h_{i,1}$ learned in $L_{i,1}$ and $h_{i,2}$ learned in $L_{i,2}$, they are combined in a subsequent layer of learning $L_{j,b}$, where $j > i$, to learn a joint behavior $h_{j,b}$ containing parts or all of either or both of $H_{i,1}$ and $H_{i,2}$ (i.e. $H_{i,1} \in L_{j,b}$, $H_{i,2} \in L_{j,b}$, and $\exists H' \subseteq \{H_{i,1} \cup H_{i,2}\}$ s.t. both $H' \neq \emptyset$ and $H' \subseteq H_{j,b}$). This scenario is best when subtask behaviors are too complex and/or potentially interfere with each other during learning, such that they must be learned independently, but ultimately need to work together for a combined task. *Example: A basketball playing robot that must be able to dribble the ball across the court and shoot it in the basket. The tasks of dribbling and shooting are too complex to attempt to learn them together, but after the tasks are learned independently they can be combined by re-optimizing parameters that control the point on the court at which the robot stops dribbling and the angle at which the robot shoots the ball.*

Partial Concurrent Layered Learning (PCLL): Only part, but not all, of a previously learned layer's behavior parameters are left open when learning a subsequent layer with new parameters. Thus there exists a non-empty proper subset of parameters H'_i of a learned behavior h_i that are re-learned in the behavior h_{i+1} as part of the subsequent layer of learning L_{i+1} (i.e. $\exists H'_i \subset H_i$ s.t. $H'_i \neq \emptyset$ and $H'_i \subset H_{i+1}$). The part of the previously learned layer's parameters left open is the "seam" between the layers. Partial concurrent learning is beneficial if full concurrent learning unnecessarily increases the dimensionality of the search space to the point that it hinders learning, and

completely freezing the previous layer diminishes the potential behavior of the layers working together. *Example: Teaching one robot to pick up and hand an object to another robot. First a robot is taught to pick up an an object and then reach out its arm and release the object. The second robot is then taught to reach out its arm and catch the object released by the first robot. During learning by the second robot to catch the object, the part of the previously learned behavior of the first robot to hand over the object is left open so that the first robot can adjust its release point of the object to a place that the second robot can be sure to reach.*

Previous Learned Layer Refinement (PLLR): After a layer is learned and frozen, and then a subsequent layer is learned, part or all of the previously learned layer is then unfrozen and relearned to better work with the newly learned layer that is now fully or partially frozen. We consider re-optimizing a previously frozen layer under new conditions as a new learned layer behavior with the "seam" between behaviors being the unfrozen part of the previous learned layer. Thus there exists a non-empty subset of parameters H_i' of a learned behavior h_i that is frozen in a subsequent layer of learning L_j, where $j > i$, but then is eventually unfrozen and re-learned in the behavior h_k as part of a later layer of learning L_k, where $k > j$ (i.e. $\exists H_i' \subseteq H_i$ s.t. $H_i' \neq \emptyset$, $h_i' \prec T_j$ but $H_i' \cap H_j = \emptyset$ *(learned values h_i' for parameters H_i' are frozen in L_j)*, and $H_i' \subseteq H_k$ *(learned values h_i' for parameters H_i' are unfrozen in L_k)*). This scenario is useful when a subtask is required to be learned before the next subsequent task layer can be learned, but then refining or relearning the initial learned subtask layer's behavior to better work with the newly learned subsequent task layer's behavior provides a benefit. *Example: Teaching a robot*

to walk. First the robot needs to learn how to stand up so that if it falls over it can get back up and continue trying to walk. Eventually the robot learns to walk so well that it barely if ever falls over during training. Later, when the robot does eventually fall over, it is found that the walking motion learned by the robot is not stable if the robot tries to walk right after standing up. The robot needs to relearn the standing up behavior layer such that after doing so it is in a stable position to start walking with the learned walking behavior layer.

3.4 Summary and Discussion

This chapter introduced the overlapping layered learning paradigm, and presented general scenarios where its use is beneficial. Chapter 4 presents a case study of overlapping layered learning applied to robot soccer which showcases overlapping layered learning as a paradigm for efficient behavior learning.

Currently the overlapping layered learning methodologies require a person to select which parameters to freeze and leave open during each successive layer of learning. Additionally, learning of complex skills is manually segmented into different layers of learning. Future work in the area of layered learning includes the automated determination of appropriate subtasks for layered learning, as well as automated identification and selection of useful layer overlap or "seams" to use with overlapping layered learning methodologies. If these selection processes can be automated it would lessen the burden and potential need for someone with expert domain knowledge when performing optimizations.

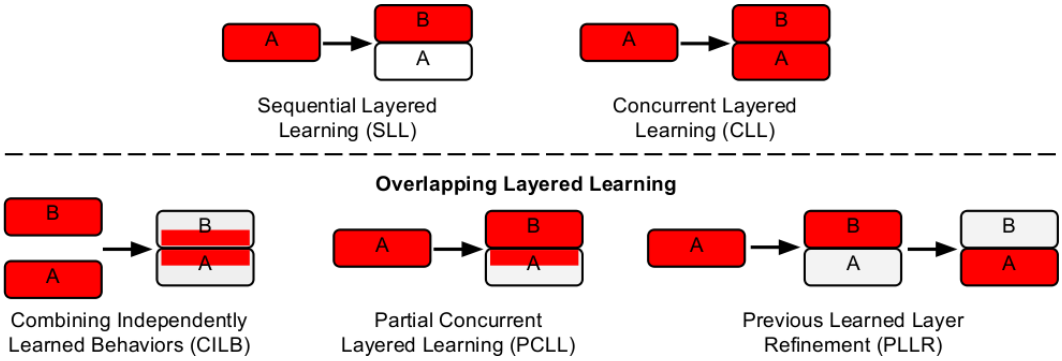


Figure 3.1: Paradigms for layered learning. Boxes represent different behaviors with the behaviors or parts of behaviors being learned shown in red. The arrows represent the transition from one layer of learning to the next.

Chapter 4

Overlapping Layered Learning Applied to Robot Soccer

This chapter[7] presents a case study and analysis of overlapping layered learning paradigms—introduced in Chapter 3—applied to robot soccer. The chapter's empirical evaluation of overlapping layered learning methodologies fulfills part of thesis contribution 5 in Section 1.2.

The University of Texas at Austin has been a perennial participant in the annual RoboCup 3D simulation soccer competitions—described in Section 2.2—and has won the championship six times between 2011 and 2017 due in large part to the main contributions of this dissertation.[8] UT Austin's RoboCup team, known as UT Austin Villa, began using sequential layered learning in 2011 [103], but introduced overlapping layered learning in 2014, and has used it since. In this chapter, we therefore focus on that year's team.

The remainder of this chapter is organized as follows. Section 4.1 details the

[7] This chapter contains material from previously published work in [106].
[8] RoboCup competition results from these years are provided in Appendix E.

overlapping layered learning approach of the 2014 UT Austin Villa team, and in Section 4.2 we provide detailed analysis of its performance. Section 4.3 summarizes.

4.1 Overlapping Layered Learning Approach

The 2014 UT Austin Villa team introduced an extensive layered learning approach to learn skills for the robot such as getting up, walking, and kicking. This approach includes sequential layered learning where a newly learned layer is frozen before learning of subsequent layers, as well as overlapping layers where parts of previously learned layers are re-optimized as part of the current layer being learned.

In total over 500 parameters were optimized during the course of layered learning. All parameters were optimized using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm [57]—described in Section 2.1.2—which has been successfully applied previously to learning skills in the RoboCup 3D simulation domain [172].

A total of 705,000 learning trials were performed during the process of optimizing 19 behaviors. As CMA-ES is a parallel search algorithm, optimization was performed on a Condor [165] distributed computing cluster allowing for many jobs to be run in parallel. Running the complete optimization process took about 5 days, and we calculated it could theoretically be completed in as little as 49 hours assuming no job queuing delays on the computing cluster, and all possible parallelism during the optimization process is exploited. Note that this same amount of computation, when performed sequentially on a single computer,[9] would take approximately 561 days, or a little over 1.5 years, to finish.

The following subsections document the overlapping layered learning parts of

[9] As measured on an Intel(R) Xeon(R) CPU E31270 @ 3.40GHz.

the approach used by the team. Full details of all of the learned behavior layers are provided in Appendix A,[10] and a diagram of how all the different layered learning behaviors fit together during the course of learning can be seen in Figure 4.1.

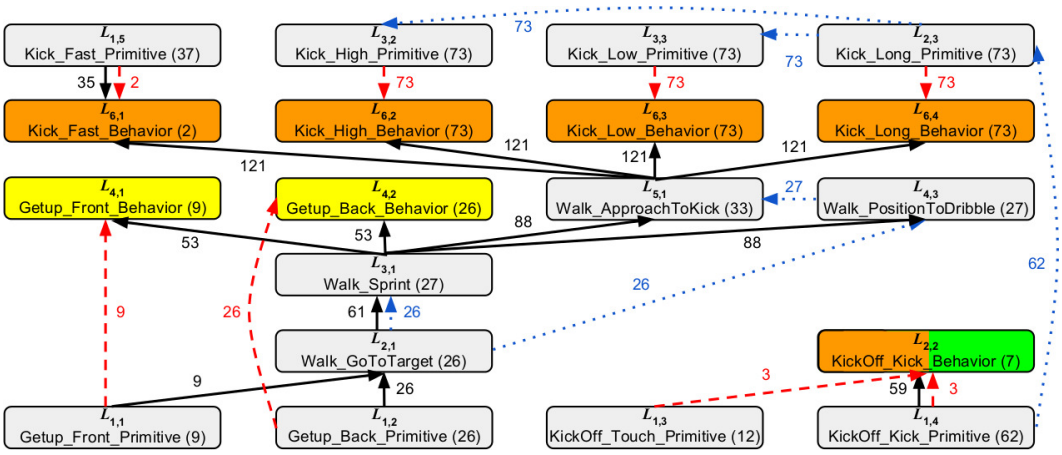


Figure 4.1: Different layered learning behaviors with the number of parameters optimized for each behavior shown in parentheses (best viewed in color). Solid black arrows show number of learned and frozen parameters passed from previously learned layer behaviors, dashed red arrows show the number of overlapping parameters being passed and relearned from one behavior to another, and the dotted blue arrows show the number of parameter values being passed as seed values to be used in new parameters at the next layer of learning. Overlapping layers are colored with CILB layers in orange, PCLL in green, and PLLR in yellow. Descriptions of the layers are provided in Appendix A.

4.1.1 Getup and Walking using PLLR

The UT Austin Villa team employs an omnidirectional walk engine using a double inverted pendulum model to control walking. The walk engine has many parameters that need to be optimized in order to create a stable and fast walk including the length and frequency of steps as well as center of mass offsets. Full details of the

[10] Videos of some of the behaviors being learned are available at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/overlappingLayeredLearning.html

walk engine and its parameters are given in Section 8.2.5. Instead of having a single set of parameters for the walk engine, which in previous work we found to limit performance [103], walking is broken up into different subtasks for each of which a set of walk engine parameters is learned.

Before optimizing parameters for the walk engine, "getup" behaviors are learned so that if the robot falls over it is able to stand back up and start walking again. Getup behaviors are necessary for faster learning during walk optimizations, as without the ability to get up after falling, a walk optimization task would have to be terminated as soon as the robot fell over. There are two such behaviors for getting up: *GetUp_Front_Primitive* for standing up from lying face down and *Getup_Back_Primitive* for standing up from lying face up. Each getup behavior consists of different joint angle movements that form a fixed series of poses. The poses themselves are specified in a parameterized skill description language. Information about the skill description language is provided in Section 8.2.6. During learning, getup behaviors are evaluated based on how quickly the robot is able to stand up [104].

After the getup primitive behaviors are learned, we start optimizing the first walk engine parameter set in the next layer of learning. This is the *Walk_GoToTarget* behavior which is used for walking to different target locations on the soccer field. Learning this walk is accomplished by having the robot walk to a series of target points on the field in the form of an obstacle course, and the robot is rewarded for how quickly it can complete the obstacle course while being penalized for every time it falls over. After the *Walk_GoToTarget* parameter set is learned and fixed, the *Walk_Sprint* walk engine parameter set used to quickly walk straight forward—to targets within 15° of the robot's current heading—is then optimized in the third

layer of learning using the same obstacle course optimization task. Full details of how these walk parameter sets are optimized can be found in Appendix A.1. Note that the optimization tasks used for learning different walk parameter sets purposely transition between all previously learned walk parameter sets and the current one being learned to ensure that the robot can smoothly transition between them without losing stability.

After learning both the *Walk_GoToTarget* and *Walk_Sprint* walk engine parameter sets, we re-optimize the getups by learning the *GetUp_Front_Behavior* and *GetUp_Back_Behavior* behaviors in the fourth layer of learning. *GetUp_Front_Behavior* and *GetUp_Back_Behavior* are overlapping layered learning behaviors as they contain the same parameters as the previously learned *GetUp_Front_Primitive* and *GetUp_Back_Primitive* behaviors respectively. The getup behavior parameters are re-optimized from their primitive behavior values through the same optimization as the getup primitives, but with the addition that right after completing a getup behavior the robot is asked to walk in different directions and is penalized if it falls over while trying to do so. Unlike the getup primitive behaviors, which were learned in isolation, the relearned getup behaviors are stable transitioning from standing up and then almost immediately walking. One might think that the walk parameter sets learned would be stable transitioning from the original learned getups due to the getup primitive behaviors being used in the walk parameter optimization tasks, however this is not always the case. During learning, walks become stable such that toward the end of optimizing a walk parameter set the robot almost never falls, and thus rarely uses the getup primitive behaviors. Relearning the getup behaviors is an example of previous learned layer refinement (PLLR).

4.1.2 Kicking using CILB

Four primitive kick behaviors were learned by the 2014 UT Austin Villa team (*Kick_Long_Primitive*, *Kick_Low_Primitive*, *Kick_High_Primitive*, and *Kick_Fast_Primitive*). Each kick primitive, or kicking motion, was learned by placing the robot at a fixed position behind the ball and having it optimize joint angles for a fixed set of key motion frames defined by a skill description language. Information about the skill description language is provided in Section 8.2.6. Note that initial attempts at learning kicks directly with the walk, instead of learning kick primitives independently, proved to be too difficult due to the variance in stopping positions of the walk as the robot approached to kick the ball.

While the kick primitive behaviors work quite well when the robot is placed in a standing position behind the ball, they are very hard to execute when the robot tries to walk up to the ball and kick it. One reason for this difficulty is that when the robot approaches the ball to kick it using the *Walk_ApproachToKick* walk parameter set—used for approaching and stopping at a precise position behind the ball before executing a kick—the precise offset position from the ball that the kick primitives were optimized to work with do not match that of the position the robot stops at after walking up to the ball. In order to allow the robot to transition from walking to kicking, full kick behaviors for all the kicks are optimized (*Kick_Long_Behavior*, *Kick_Low_Behavior*, *Kick_High_Behavior*, *Kick_Fast_Behavior*). Each full kick behavior is learned by having the robot walk up to the ball and attempt to kick it from different starting positions—as opposed to having the robot just standing behind the ball as was done when optimizing the kick primitive behaviors.

The full kick behaviors are overlapping layered learning behaviors because they re-optimize previous learned parameters. In the case of *Kick_Fast_Behavior*,

only the x and y kick primitive offset position parameters from the ball, which is the target position for the walk to reach for the kick to be executed, are re-optimized. The fast kick is quick enough that it almost immediately kicks the ball after transitioning from walking, and thus just needs to be in the correct position near the ball to do so. A comparison of overlapping layered learning paradigms for learning *Kick_Fast_Behavior* is shown in Figure 4.2. The above overlapping layered approach of first independently learning the walk approach and kick, and then learning the two position parameters, does better than both the sequential layered learning approach where all kick parameters are learned after freezing the approach, and the concurrent layered learning approach where both approach and kick parameters are learned simultaneously.

For the other full kick behaviors, all kick parameters from their respective kick primitive behaviors are re-optimized. Unlike the fast kick, there is at least a one second delay between stopping, walking, and kicking the ball, during which the robot can easily become destabilized and fall over. By opening up all kicking parameters the robot has the necessary freedom to learn kick motions that maintain its stability between stopping after walking and making contact with the ball. Learning the kick behaviors by combining them with the *Walk_ApproachToKick* behavior for walking up to the ball are all examples of combining independently learned behaviors (CILB).

4.1.3 KickOff using both CILB and PCLL

For kickoffs the robot is allowed to "teleport" itself to a starting position next to the ball before kicking it, and thus does not need to worry about walking up to the ball. Scoring directly off a kickoff is not allowed, however, as another robot must first touch the ball before it goes into the opponent's goal. In order to score on a kickoff

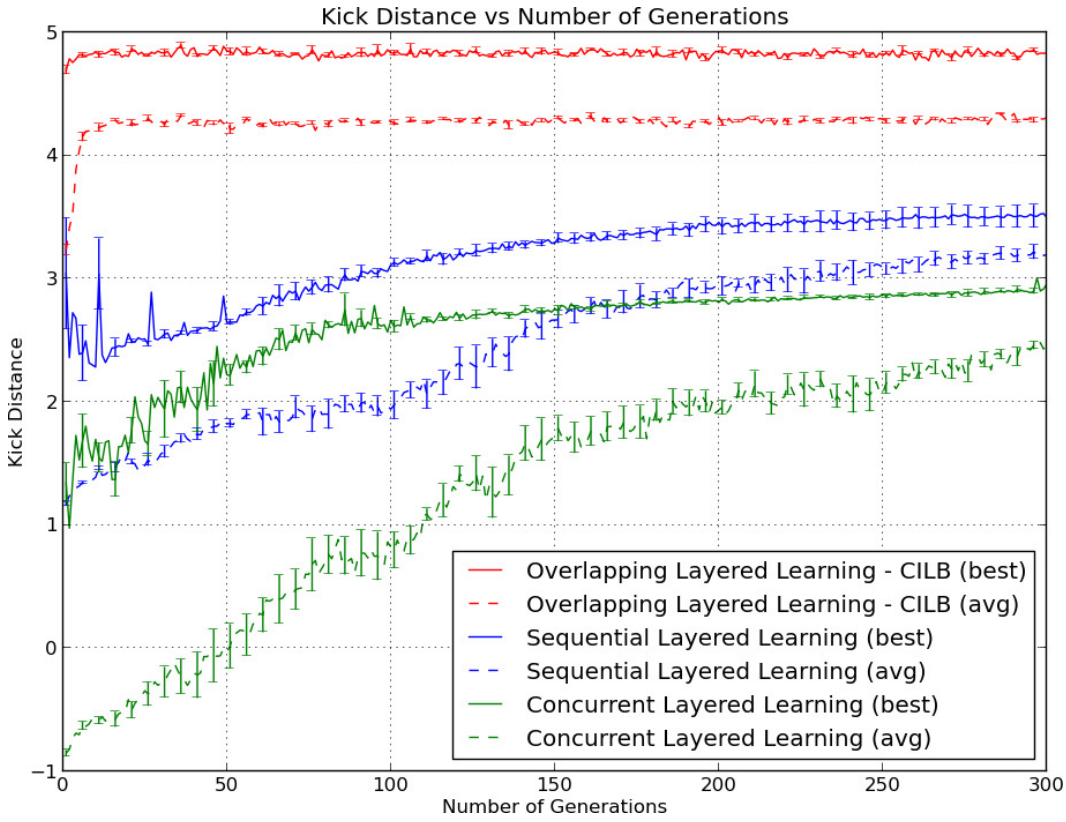


Figure 4.2: Performance of different layered learning paradigms across generations of CMA-ES when optimizing *Kick_Fast_Behavior*. Results are averaged across five optimization runs and error bars show the standard error.

we perform a multiagent task where one robot touches the ball before another kicks it.

The first behavior optimized for scoring off the kickoff is *KickOff_Kick_Primitive* in which a robot kicks the ball from the middle of the field. The robot is rewarded for kicking the ball as high and as far as possible as long as the ball crosses the goal line below the height of the goal. In parallel a behavior for another robot is learned to lightly touch the ball called *KickOff_Touch_Primitive*. Here a robot is rewarded for touching the ball lightly and, after ensuring that the robot has made

contact with the ball, that the ball moves as little as possible. Finally an overlapping layered behavior called *KickOff_Kick_Behavior* is learned which re-optimizes x, y, and θ angle offset positions from the ball from both the *KickOff_Kick_Primitive* and *KickOff_Touch_Primitive* behaviors. Re-optimizing these positioning parameters together is important so that the robots do not accidentally collide with each other and also so that the kicking robot is at a good position to kick the ball after the first agent touches it. Learning *KickOff_Kick_Behavior* is another example of combining independently learned behaviors (CILB).

In addition to the positioning parameters of both robots being re-optimized for *KickOff_Kick_Behavior*, a new parameter that determines the time at which the first robot touches the ball is optimized. This synchronized timing parameter is necessary so that the robots are synced with each other and the kicking robot does not accidentally try to kick the ball before the first robot has touched it. As a new parameter is optimized along with a subset of previously learned parameters, learning *KickOff_Kick_Behavior* is also an example of partial concurrent layered learning (PCLL).

Further information about the kickoff, including how a seed for the kick was learned through observation, can be found in [41].

4.2 Results and Analysis

At the 2014 RoboCup 3D simulation competition—the first year the UT Austin Villa team used overlapping layered learning—UT Austin Villa finished first among 12 teams while scoring 52 goals and conceding none across 15 games.[11] Considering that most of the team's strategy layer—including team formations using a dynamic

[11] Full game results are provided in Appendix E.

role assignment and formation positioning system [111]—remained unchanged from that of the previous year's second place finishing team, a key component to the 2014 team's improvement and success at the competition was the new approach incorporating overlapping layered learning used to learn the team's low level behaviors.

After every RoboCup competition teams are required to release the binaries that they used during the competition. In order to analyze the performance of the different components of our overlapping layered learning approach before the 2014 competition, we played 1000 games with different versions of the UT Austin Villa team against each of the top three teams from the RoboCup 2013 competition. The following subsections provide analysis of game results when turning on and off the kickoff and kicking components learned though an overlapping layered learning approach. Additionally, to demonstrate the generality of our overlapping layered learning approach, we provide data that isolates the performance of our complete overlapping layered learning approach applied to different robot models.

4.2.1 Overall Team Performance

Table 4.1 shows the average goal difference across all games against each opponent achieved by the complete 2014 UT Austin Villa team. Against all opponents the team had a significantly positive goal difference, and in fact out of the 3000 games played the team only lost one game (to AustinVilla2013). These game results show the effectiveness of the team's overlapping layered learning approach in dramatically improving the performance of the team from the previous year in which the team achieved second place at the competition—the 2014 team is able to beat the previous year's team by an average of 1.525 goals.

Data provided in Appendix E, showing the overall team's performance when

Table 4.1: Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by the 2014 UT Austin Villa team versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2852, 1, and 147 respectively.

Opponent	Average Goal Difference
Apollo3D	2.726 (0.036)
AustinVilla2013	1.525 (0.032)
FCPortugal	3.951 (0.049)

playing against the released 2014 teams' binaries, corroborates the overall team's strong performance. When playing 1000 games against each of the eleven 2014 opponents UT Austin Villa did not lose a single game out of the 11,000 played, and had at least an average goal difference of 2 against every opponent. Additionally, bolstered by the team's strong set of skills developed through overlapping layered learning techniques, UT Austin Villa won all games it played at the RoboCup 2015 [108], 2016 [114], and 2017 competitions.[12]

4.2.2 KickOff Performance

To isolate the performance of the learned multiagent behavior to score off the kickoff, we disabled this feature and instead just had the robot taking the kickoff kick the ball toward the opponent's goal to a position as close as possible to one of the goal posts without scoring. Table 4.2 shows results from playing against the top three teams at RoboCup 2013 without attempting to score on the kickoff.

By comparing results in Table 4.2 to that of Table 4.1 we see a significant drop in performance when not attempting to score on kickoffs. This result is not surprising as we found that the kickoff was able to score around 90% of the time against Apollo3D and FCPortugal, and over 60% of the time against the 2013 version

[12]Competition results are provided in Appendix E.

Table 4.2: Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *not attempting to score on a kickoff* versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2644, 5, and 351 respectively.

Opponent	**Average Goal Difference**
Apollo3D	2.059 (0.038)
AustinVilla2013	1.232 (0.032)
FCPortugal	3.154 (0.046)

of UT Austin Villa. The combination of using both CILB and PCLL overlapping layered learning led to a large boost to the team's performance.

4.2.3 Kicking Performance

To isolate the performance of kicking learned through an overlapping layered learning approach we disable all kicking (except for on kickoffs where we once again have a robot kick the ball as far as possible toward the opponent's goal without scoring) and used an "always dribble" behavior. Data from playing against the top three teams at the RoboCup 2013 competition when only dribbling is shown in Table 4.3.

Table 4.3: Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *using a dribble only strategy* versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2480, 15, and 505 respectively.

Opponent	**Average Goal Difference**
Apollo3D	1.790 (0.033)
AustinVilla2013	0.831 (0.023)
FCPortugal	1.593 (0.028)

Here we see another significant drop in performance when comparing Ta-

ble 4.3 to Table 4.2. Kicking provided a large gain in performance, nearly doubling the average goal difference against FCPortugal, compared to only dribbling. This result is in stark contrast to when UT Austin Villa won the 2011 RoboCup competition, in which the team tried to incorporate kicking skills without using an overlapping layered learning approach, and found that kicking actually hurt the performance of the team [117].

4.2.4 Different Robot Models

At the 2014 RoboCup competition teams were given the option of using five different robot types with the requirement that at least three different types of robots must be used on a team and no more than seven of any one type. The five types of robots available were the following:

Type 0: Standard Nao model

Type 1: Longer legs and arms

Type 2: Quicker moving feet

Type 3: Wider hips and longest legs and arms

Type 4: Added toes to foot

We applied our overlapping layered learning approach for learning behaviors to each of the available robot types. Game data from playing against the top three teams at RoboCup 2013 is provided in Table 4.4 for each robot type.

While there are some differences in performance between the different robot types, likely due to the differences in their body models, all of the robot types are able to reliably beat the top teams from the 2013 RoboCup competition. This shows the efficacy of our overlapping layered learning approach and its ability to generalize

Table 4.4: Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *using different heterogeneous robot types* versus the given opponent team.

	Avg. Goal Difference per Robot Type				
Opponent	**Type 0**	**Type 1**	**Type 2**	**Type 3**	**Type 4**
Apollo3D	1.788	1.907	1.892	1.524	2.681
AustinVilla2013	0.950	0.858	1.152	0.613	1.104
FCPortugal	2.381	2.975	3.331	2.716	3.897

to different robot models. During the 2014 competition the UT Austin Villa team used seven type 4 robot models as they showed the best performance, two type 0 robot models as they displayed the best performance on kickoffs, and one each of the type 1 and type 3 robot models as they were the fastest at walking [105].

4.2.5 Summary of Results

Results from the data we collected in Section 4.2.1 and Appendix E provide evidence of the 2014 team's overall improvement and success gained by incorporating overlapping layered learning into the approach used to learn the team's low level behaviors. In particular, kicks learned through overlapping layered learning techniques boosted the performance of the team as shown by the data isolating the kicks' contributions to game results in Sections 4.2.2 and 4.2.3. Finally, the generality of our learning approach is demonstrated by its success when applied to different robot models in Section 4.2.4.

4.3 Summary

This chapter provided a detailed description and experimental analysis of the extensive overlapping layered learning approach used by the UT Austin Villa team in

winning the 2014 RoboCup 3D simulation competition.[13] Furthermore, the complete learning process is repeated on four different robot body types, showcasing its generality as a paradigm for efficient behavior learning. While first introduced in 2014, overlapping layered learning has continued to be used successfully by the UT Austin Villa team in all years since then as well.

A base code release of the 2015 UT Austin Villa RoboCup 3D simulation agent [115], which includes hooks for optimizing the skills and behaviors presented in this work, is discussed in Chapter 8.

[13] Team video highlights from the competition can be found on the team's homepage at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/#2014

Chapter 5

Scalable Collision-avoiding Role Assignment with Minimal-makespan (SCRAM)

This chapter[14] presents Scalable Collision-avoiding Role Assignment with Minimal-makespan (SCRAM) role assignment functions and algorithms—another primary contribution of this dissertation. Specifically this chapter addresses thesis contributions 2 and 3 in Section 1.2.

Coordinated movement among mobile agents is an important research area with many applications such as search and rescue [81] and warehouse operations [178]. A topic within this space is role assignment—deciding which agent moves to which position or role. Work in this chapter focuses on the problem of assigning mobile agents to move to a set of fixed target positions such that the makespan—time for all agents to reach their targets positions—is minimized. Minimizing the makespan

[14] This chapter contains material from previously published work in [111].

is a decisive factor in performance when agents are moving to target positions to complete a shared task where all agents must be in place before the task can be completed and/or started.

The remainder of this chapter is organized as follows. Section 5.1 gives a high level overview of SCRAM role assignment. Section 5.2 provides a formulation of the role assignment problem we are solving. Two role assignment functions, as well as algorithms implementing them, are presented in Section 5.3. An empirical evaluation of role assignment functions and algorithms is given in Section 5.4. Section 5.5 summarizes.

An extension to SCRAM role assignment for the prioritization of target positions is presented in Chapter 6, and Chapter 7 provides applications and case studies of positioning systems incorporating SCRAM role assignment used within RoboCup robot soccer domains.

5.1 Overview

Movement coordination of mobile agents spans multiple research topics including role assignment [29, 123, 70], path planning [121, 151, 90], and collision avoidance [64, 144, 173]. The work in this chapter focuses on role assignment—specifically tackling the problem of assigning interchangeable homogeneous mobile agents to move to a set of fixed target positions such that an agent is present at every target position in as little time as possible. Path planning and collision avoidance issues are addressed during role assignment, as mappings of agents to target positions operate under the constraint that no agents collide.

Previous work on assigning agents to target positions has focused on minimizing the sum of distances all agents must travel which is the well known *assignment*

problem [138]. Our work differs as we minimize the makespan—time for all agents to reach goal positions—instead of the sum of distances traveled. Minimizing the makespan is a decisive factor in performance when agents are moving to target positions to complete a shared task where all agents must be in place before the task can be completed and/or started. Such tasks include those requiring agents be synchronized when they start jobs at their target positions, e.g., mobile robots assuming necessary positions on an assembly line. Other such tasks involve scenarios for which the bottleneck is the time it takes for the last agent to get to its target position, e.g., warehouse robots delivering items for an order to be shipped, and mobile robots used as pixels to display images [15].

We refer to our role assignment as SCRAM, for Scalable Collision-avoiding Role Assignment with Minimal-makespan. It provides a collision free mapping of agents to target positions, minimizes the makespan, and scales to thousands of agents as role assignment algorithms run in polynomial time. Primary contributions of this chapter include a complete specification of SCRAM, the presentation of role assignment functions for assigning agents to target positions, algorithms (both new and existing) for computing the role assignment functions,[15] as well as a thorough theoretical and empirical analysis of the role assignment problem.

5.2 Role Assignment Problem

Let there be n homogeneous mobile agents with current positions $A := \{a_1, ..., a_n\}$, and we want to assign the agents to move to n specified target goal positions or roles $P := \{p_1, ..., p_n\}$ such that the time for agents to have reached every goal

[15] Videos of SCRAM role assignment in action, as well as C++ implementations of the role assignment algorithms, can be found at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/scram.html

position is minimized under the constraint that no agents collide with each other. Figure 5.1 illustrates an example problem with six agents and target positions. This problem can be thought of as finding a perfect matching M^* within the set of perfect matchings $\mathbb{M}$ of a weighted bipartite graph $G := (A, P, E)$ that meets the above criteria with the weight for each edge in E being the Euclidean distance between associated agent and target positions.

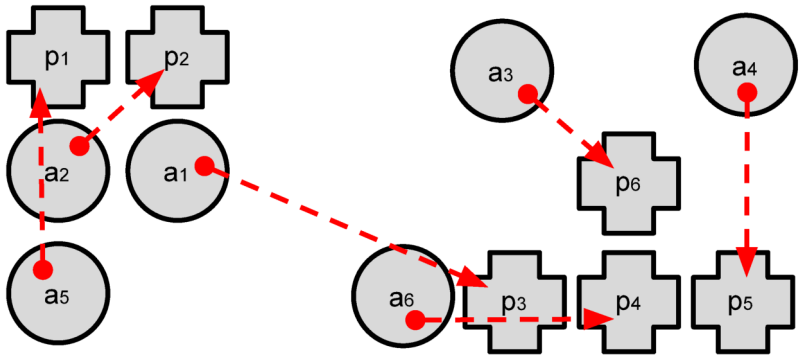


Figure 5.1: Role assignment problem where we want to assign agents (circles) $\{a_1,...,a_6\}$ to target positions (crosses) $\{p_1,...,p_6\}$. Dashed arrows show solution with minimal makespan.

Similar to path planning work by Broucke [25, 26], we model agents as point masses with zero width. Additionally, we make two more assumptions. First, no two agents and no two target positions occupy the same position. Second, we assume that all agents move toward fixed target positions along a straight line at the same constant speed. While the assumptions may not hold in practice, they are necessary for theoretical analysis of the role assignment problem and are often good enough approximations, as corroborated by our successful empirical results in RoboCup domains presented in Chapter 7. We mention a potential extension of our work in Section 10.2.2 that includes allowing for non-point masses as well as obstacles in the environment.

We consider a role assignment function f to be *CM valid* (Collision-avoiding with Minimal-makespan) if and only if it always returns a perfect mapping M^* that satisfies two properties:

1. *Minimizing longest distance* - M^* minimizes the longest distance from an agent to target, with respect to all possible mappings. Such a mapping for the problem shown in Figure 5.1 would not include $a_2 \rightarrow p_5$ as that is the longest distance between an agent and target. Instead M^* includes $a_1 \rightarrow p_3$ which is the minimal longest distance any agent travels across all possible assignments of agents to targets.

2. *Avoiding collisions* - agents do not collide with each other as they move to their assigned positions. In Figure 5.1 a mapping including both $a_1 \rightarrow p_1$ and $a_2 \rightarrow p_2$ would not have this property as it would cause agents a_1 and a_2 to collide.

A third desirable property, although not necessary for a role assignment function f to be *CM valid*, is the following:

3. *Dynamically consistent* - Given a *fixed* set of target positions, if f outputs a mapping M of agents to targets at time T, then f continues to output M for every time $t > T$ as the agents move to the targets specified by M.

The first two properties come directly from the definition of the role assignment problem. The third property guarantees that once a role assignment function f outputs a mapping, f will always output that same mapping as long as there is no change in the target positions. This guarantee is desirable as otherwise agents might thrash between roles thus impeding progress. In the following section we construct *CM valid* role assignment functions.

5.3 Role Assignment Functions

The following subsections present two *CM valid* role assignment functions for the role assignment problem detailed in Section 5.2. Algorithmic implementations of the functions and analysis of their time and space complexities are also given.

5.3.1 Minimum Maximal Distance Recursive (MMDR) Function

One potential role assignment function is to find a mapping of agents to target positions which recursively minimizes the maximum distance that any agent travels. We refer to this as this the Minimum Maximal Distance Recursive (MMDR) function. It is also known as the *lexicographic bottleneck assignment problem* [138]. In this section we first analyze properties of MMDR, and then identify efficient polynomial time algorithms to compute MMDR.

Let $\mathbb{M}$ be the set of all one-to-one mappings between agents and roles. If there are n agents and n target role positions, then there are $n!$ possible mappings $M \in \mathbb{M}$. Let the *cost* of a mapping M be the n-tuple of distances from each agent to its target, sorted in decreasing order. We can then sort all the $n!$ possible mappings based on their costs, where comparing two costs is done lexicographically. Sorted costs of mappings for a small example are shown in Figure 5.2.

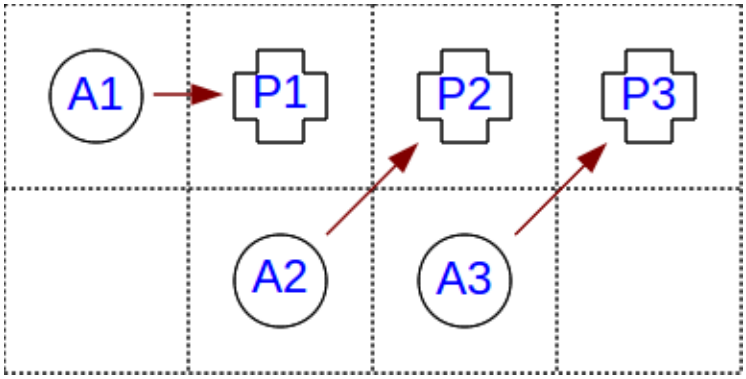


Figure 5.2: Lowest lexicographical cost (shown with arrows) to highest cost ordering of mappings from agents (A1,A2,A3) to role positions (P1,P2,P3). Each row represents the cost of a single mapping.

1:	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P3),	1 (A1→P1)
2:	2 (A1→P2),	$\sqrt{2}$ (A3→P3),	1 (A2→P1)
3:	$\sqrt{5}$ (A2→P3),	1 (A1→P1),	1 (A3→P2)
4:	$\sqrt{5}$ (A2→P3),	2 (A1→P2),	$\sqrt{2}$ (A3→P1)
5:	3 (A1→P3),	1 (A2→P1),	1 (A3→P2)
6:	3 (A1→P3),	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P1)

Denote the role assignment function that always outputs the lexicographically smallest cost mapping as MMDR. Here we provide an informal proof sketch that MMDR is *CM valid* and is also dynamically consistent; we provide a longer, more thorough derivation showing MMDR as *CM valid* in Appendix B.1, and being dynamically consistent in Appendix B.2.

Theorem 1. *MMDR is CM valid and dynamically consistent.*

Proof Sketch: MMDR minimizes the longest distance (Property 1) as the lexicographical ordering of distance tuples sorted in descending order ensures this. If two agents in a mapping are to collide (Property 2) it can be shown, through the triangle

inequality, that MMDR will find a lower cost mapping as switching the two agents' targets reduces the maximum distance either must travel. Finally, as we assume all agents move toward their targets at the same constant rate, the distance between an agent and any target will not decrease any faster than the distance between any agent and the target that agent is assigned to. This observation provides dynamic consistency (Property 3) by preserving the lowest lexicographical cost ordering of a MMDR mapping across all timesteps. □

$O(n^5)$ Polynomial Time Algorithm for MMDR

We can compute the MMDR role assignment function in polynomial time by transforming MMDR into the *assignment problem*—finding a perfect matching in a bipartite graph that minimizes the sum of edge weights—which is solvable by the Hungarian algorithm [86] in $O(n^3)$ time.

In order to transform MMDR into the *assignment problem* we modify the weights of the edges of our bipartite graph to be a set of values such that the weight of any edge e is greater than the sum of weights of all edges with weight values less than that of e. A key insight into this transformation is expressed in Lemma 1.

Lemma 1. *Denote* $W_n := \{w_0, ..., w_n\}$ *where* $w_i := 2^i$. *Then for all* $W \subseteq W_{n-1}$: $w_n > \sum_{w \in W} w$.

By sorting all edges in ascending order by distance, and then relabeling edge weights to be the value 2^i where i is the index of an edge in this sorted list, the sum of all edge weights of shorter distance edges will be less than any sum of edge weights with a longer edge. Solutions to the *assignment problem* return lowest cost MMDR mappings as the sum of modified weights of any mapping with a higher cost is greater than that of a lower cost mapping.

Algorithm 1 gives a polynomial time solution for computing MMDR. First, weights are sorted in ascending order of distance (line 1). Next, edge weights are transformed into appropriate values for the assignment problem as expressed in Lemma 1 (line 8). Finally, the re-weighted edges are given as input into the Hungarian algorithm which returns the lowest cost MMDR mapping (line 10). Time complexity is dominated by the $O(n^3)$ Hungarian algorithm. Note that our transformed edge weights, represented as bit vectors with the *ith* bit of a 2^i value turned on, are of size n^2. The Hungarian algorithm must do comparisons of these weights and thus the time complexity of Algorithm 1 is $O(n^5)$. As our implementation of the Hungarian algorithm requires us to store length n lists of size n^2 transformed weights, Algorithm 1 has a space complexity of $O(n^3)$. Algorithm 1 meets the criterion for being SCRAM as it computes the *CM valid* MMDR role assignment function with polynomial time and space complexities.

Algorithm 1 MMDR $O(n^5)$ Polynomial Time Implementation

Input:

$Agents := \{a_1, ..., a_n\}$; $Positions := \{p_1, ..., p_n\}$
$Edges := \{\overline{a_1p_1}, \overline{a_1p_2}, ..., \overline{a_np_n}\}$; $|\overline{a_ip_j}| :=$ `euclideanDist`(a_i, p_j)

1: $edgesSorted :=$ `sortAscendingDist`$(Edges)$
2: $lastDistance := -1$
3: $rank, currentIndex := 0$
4: **for each** $e \in edgesSorted$ **do**
5: **if** $|e| > lastDistance$ **then**
6: $rank := currentIndex$
7: $lastDistance := |e|$
8: $|e| := 2^{rank}$
9: $currentIndex := currentIndex + 1$
10: **return** `hungarianAlg`$(edgesSorted)$

There exists previous work in modifying edge weights to transform the *lexicographic bottleneck assignment problem* into the *assignment problem*. For cases

in which there are n^2 edges, with each having a unique cost, a higher complexity $O(n^5 \log n)$ algorithm exists [27]. Work by Croce et al. [35] changes edge weights into weight vectors of length n before solving the *assignment problem* and has the same time complexity as our method of $O(n^5)$. However, on modern computer architectures Algorithm 1 is more efficient as we represent edge weights as bit vectors instead of vectors of integers. The compact format of bit vectors allows for integer operations to be performed on w bits in parallel where w is the size of a processor's maximum word length. This parallelism reduces the running time by a factor of w, e.g., a factor of 64 on a 64-bit architecture.

$O(n^4)$ Polynomial Time Algorithm for MMDR

Another approach to compute MMDR, introduced by Sokkalingam and Aneja [153], and detailed in Algorithm 2, alternates between solving the *bottleneck assignment problem* [138]—finding the smallest maximum edge in a perfect matching—and a 0-1 cost version of the *assignment problem*.

At every iteration of Algorithm 2 solving the *bottleneck assignment problem* (line 5 implemented by Algorithm 3 discussed later in this section) returns the current largest edge weight value in the MMDR mapping. Next solving the *assignment problem* using the Hungarian algorithm (line 7), with 0-1 edge costs as specified in line 6, returns a mapping whose sum of costs (line 8) reveals the number of edges of this weight in the MMDR mapping.

At the same time the Hungarian algorithm naturally computes a potential function *poten* over the set of vertices in the bipartite graph such that $\forall e_{a,p} \in Edges$: $poten(a) + poten(p) <= cost(e_{a,p})$. It is revealed by Sokkalingam and Aneja [153] that all perfect matchings of the subset of tight edges—defined as edges for which

Algorithm 2 MMDR $O(n^4)$ Polynomial Time Implementation

Input:

$Agents := \{a_1, ..., a_n\}$; $Positions := \{p_1, ..., p_n\}$
$Edges := \{\overrightarrow{a_1p_1}, \overrightarrow{a_1p_2}, ..., \overrightarrow{a_np_n}\}$; $|\overrightarrow{a_ip_j}| :=$ `euclideanDist`(a_i, p_j)

1: **function** GETTIGHTEDGES($poten$)
2: **return** $e_{a,p} \in Edges$, s.t. $poten(a) + poten(p) = cost(e_{a,p})$

3: $numEdgesLeft := n$
4: **loop**
5: $minLongestEdge :=$ `getMinimalMaxEdgeInPerfectMatching`$(Edges)$
6: $\forall e \in Edges \begin{cases} |e| < |minLongestEdge| : & cost(e) := 0 \\ |e| = |minLongestEdge| : & cost(e) := 1 \\ |e| > |minLongestEdge| : & cost(e) := \infty \end{cases}$
7: $\{matching, poten\} :=$ `hungarianAlgWithEdgeCosts`$(Edges)$
8: $numLongestEdges := \sum_{e \in matching} cost(e)$
9: $numEdgesLeft := numEdgesLeft - numLongestEdges$
10: **if** $numEdgesLeft = 0$ **then**
11: **return** $matching$
12: $Edges :=$ `getTightEdges`$(poten)$
13: $\forall e \in Edges$, s.t. $|e| = |minLongestEdge| : |e| := -1$

$poten(a) + poten(p) = cost(e_{a,p})$—contain exactly $numLongestEdges$ edges of length $|minLongestEdge|$. Given this knowledge we remove all non-tight edges from consideration in the MMDR mapping (line 12). The reduction to tight edges, and reducing the weight of edges of length $|minLongestEdge|$ (line 13), results in subsequent solutions of the *bottleneck assignment problem* revealing the next largest edge weight value in the MMDR mapping as every perfect matching will have exactly $numLongestEdges$ edges of length $|minLongestEdge|$. We learn the weight of $numLongestEdges$ edges in the MMDR mapping during every iteration of Algorithm 2, and after determining the weights for n edges, the solution returned by the Hungarian algorithm is the MMDR mapping (line 11).

Algorithm 3 finds the minimal maximum edge in a perfect matching by in-

crementally adding edges to the graph in order of increasing distance from the list of edges sorted in ascending order of weight (line 23). It interleaves adding edges (line 30) with running the Ford-Fulkerson algorithm [47] for finding a maximum cardinality, i.e., maximum number of edges, matching. Ford-Fulkerson—implemented with the `flood`, `resetFlood`, and `reversePath` functions—works by using a breath-first search to find augmenting paths from an agent to a target. Augmenting paths are alternating paths between agents and targets, along directed edges, whose start and end points have a degree of one.

Algorithm 3 starts with a graph with the empty set of edges *allowedEdges* (line 1), and whenever the breadth-first search of the Ford-Fulkerson algorithm is unable to find a path from an agent to a target, Algorithm 3 adds an edge to the graph (line 30) and continues the breadth-first search. At the point when the algorithm finds n paths from agents to target, the last edge added is the minimal maximum edge for a perfect matching.

An important factor for performance in Algorithm 3 is that it can pick up the Ford-Fulkerson breadth-first search where it left off after adding an edge as any nodes previously reachable in the graph remain reachable. Because the algorithm does not lose state in each breadth-first search, each breadth-first search takes $O(E)$ time. Thus the total time for running Algorithm 3 to find a perfect matching with the minimum maximal edge length is $O(nE)$ which is less than the $O(n^3)$ time complexity of the Hungarian algorithm.

Note that one can compute a maximum cardinality matching using the Hopcroft-Karp algorithm [65] in $O(E\sqrt{n})$ time, and thus can perform a binary search across all n^2 edges to find the minimal maximum edge in a perfect matching in $O(n^{2.5} \log n^2)$ time. Empirically we found this approach to be slower than the

Algorithm 3 Minimal-maximum Edge Perfect Matching

Input:

$Agents := \{a_1, ..., a_n\}$; $Positions := \{p_1, ..., p_n\}$
$Edges := \{\overrightarrow{a_1p_1}, \overrightarrow{a_1p_2}, ..., \overrightarrow{a_np_n}\}$; $|\overrightarrow{a_ip_j}| :=$ `euclideanDist`(a_i, p_j)

1: $matchedAgents, allowedEdges := \{\}$
2: **function** FLOOD($curNode$, $prevNode$)
3: $curNode.visited :=$ **true**
4: $curNode.previous := prevNode$
5: **if** $curNode \in Positions$ **and** $\nexists\ e \in allowedEdges$, s.t. $e.start = curNode$ **then**
6: **return** $currentNode$
7: **for each** $e \in allowedEdges$, s.t. ($e.start = curNode$ **and not** $e.end.visited$) **do**
8: $val :=$ `flood`$(e.end, e.start)$
9: **if** $val \neq \varnothing$ **then**
10: **return** val
11: **return** $\varnothing$
12: **function** RESETFLOOD
13: **for each** $node \in \{Agents \cup Positions\}$ **do**
14: $node.visited :=$ **false**
15: $node.previous := \varnothing$
16: **for each** $a \in \{Agents \setminus matchedAgents\}$ **do**
17: `flood`$(a, \varnothing)$
18: **function** REVERSEPATH($node$)
19: **while** $node.previous \neq \varnothing$ **do**
20: `reverseEdgeDirection`$(\overrightarrow{node, node.previous})$
21: $node := node.previous$
22: **return** $node$

23: $edgeQ :=$ `sortAscendingDist`$(Edges)$
24: $longestEdge := \varnothing$
25: **for** $match := 1$ **to** n **do**
26: `resetFlood`()
27: $matchedPosition := \varnothing$
28: **while** $matchedPosition = \varnothing$ **do**
29: $longestEdge :=$ $edgeQ$.`pop`()
30: $allowedEdges \leftarrow longestEdge$
31: $matchedPosition :=$ `flood`$(longestEdge.end, longestEdge.start)$
32: $matchedAgent :=$ `reversePath`$(matchedPosition)$
33: $matchedAgents \leftarrow matchedAgent$
34: **return** $longestEdge$

breadth-first search approach of Algorithm 3, possibly due to the overhead of constructing different graphs, which is not that surprising as other have found similar results [150]. For dense graphs there exists an algorithm with a better time complexity than Hopcroft-Karp of $O(n^{1.5}\sqrt{E/\log n})$ for computing a maximum cardinality matching [16], however we have not tried using this algorithm with a binary search to find the minimal maximum edge in a perfect matching.

At least one new minimal maximum edge in a perfect matching is determined during every iteration of the loop in Algorithm 2. Thus no more than n instances of both the Hungarian algorithm and Algorithm 3 need to be computed. As the $O(n^3)$ time complexity of the Hungarian algorithm dominates Algorithm 2's loop, the time complexity of Algorithm 2 is $O(n^4)$. The breadth-first search of Ford-Fulkerson in Algorithm 3 gives a space complexity of $O(n^2)$. Algorithm 2 meets the criterion for being SCRAM as it computes the *CM valid* MMDR role assignment function with polynomial time and space complexities.

5.3.2 Minimum Maximal Distance + Minimum Sum Distance^2 (MMD+MSD^2) Function

Another role assignment function to map agents to target goal positions is one which minimizes the maximum distance any agent has to travel—but not recursively as done by MMDR in Section 5.3.1—after which it minimizes the sum of distances squared that all agents travel. We call this the Minimum Maximal Distance + Minimum Sum Distance^2 (MMD+MSD^2) role assignment function. Specifically we

want to find a perfect matching M^* such that

$$\mathbb{M}'' := \{X \in \mathbb{M} \mid \|X\|_\infty = \min_{M \in \mathbb{M}}(\|M\|_\infty)\} \tag{5.1}$$

$$M^* := \operatorname*{argmin}_{M \in \mathbb{M}''}(\|M\|_2^2) \tag{5.2}$$

Here we provide an informal proof sketch that MMD+MSD^2 is a *CM valid* role assignment; we provide a longer, more thorough derivation in Appendix B.1.

Theorem 2. *MMD+MSD^2 is CM valid but not necessarily dynamically consistent.*

Proof Sketch: By only considering the set of perfect matchings $\mathbb{M}''$ with minimal longest edges (equation 5.1) we are minimizing the longest distance any agent must travel (Property 1). If two agents in a mapping are to collide (Property 2), it can be shown, through the triangle inequality, that MMD+MSD^2 will find a lower cost mapping as switching the two agents' targets reduces, but never increases, the distance that one or both must travel thereby reducing the sum of distances squared (equation 5.2) and the longest distance (equation 5.1).

Unlike MMDR, MMD+MSD^2 is not dynamically consistent because distances squared do not decrease at a constant rate, but in fact decrease at faster rates for larger distances, as agents move toward targets. As an example, the difference in distance squared as an agent moves from 5 meters to 4 meters from a target ($5^2 - 4^2 = 9$) is greater than the difference moving from 4 meters to 3 meters ($4^2 - 3^2 = 5$). This lack of a constant rate of decrease for distances squared allows for squared distances between an agent and targets it is not assigned to travel toward to decrease faster than the squared distance between an agent and the target it is assigned to. The sum of distances squared for non-MMD+MSD^2 mappings can thus become less than the current MMD+MSD^2 mapping as agents travel to

their targets. An example of where MMD+MSD^2 is shown to not be dynamically consistent is referred to in Appendix B.2. □

A potentially beneficial byproduct of minimizing the sum of distances squared is that, as alluded to in [25], subsets of edges in a mapping that hold this property will never cross. This is useful in preventing collisions if the assumption that agents all move at the same constant speed does not hold true. Note that finding a smallest maximum edge perfect matching—one that minimizes the makespan—with no path crossings has been shown to be NP-hard [7, 28].

Polynomial Time Algorithm for MMD+MSD^2

Algorithm 4 implements MMD+MSD^2 by first finding a perfect matching with the smallest maximum edge (line 1) which is computed by Algorithm 3 presented earlier in Section 5.3.1. Algorithm 4 then creates a set of *minimalEdges* consisting of all edges with length less than or equal to the longest edge in the perfect matching (line 2) and uses it as input to the Hungarian algorithm (line 3). Note that edge weights are their distances squared and thus the Hungarian algorithm minimizes the sum of distances squared. As all edges greater in length than the minimal maximum edge in a perfect matching are removed before running the Hungarian algorithm, the maximum distance any agent travels is also minimized.

Algorithm 4 MMD+MSD^2 $O(n^3)$ Polynomial Time Implementation

Input:

$Agents := \{a_1, ..., a_n\}$; $Positions := \{p_1, ..., p_n\}$
$Edges := \{\overrightarrow{a_1p_1}, \overrightarrow{a_1p_2}, ..., \overrightarrow{a_np_n}\}$; $|\overrightarrow{a_ip_j}| :=$ `euclideanDist`$(a_i,p_j)^2$

1: $longestEdge :=$ `getMinimalMaxEdgeInPerfectMatching`$(Edges)$
2: $minimalEdges := e \in Edges$, s.t. $|e| \leq |longestEdge|$
3: **return** `hungarianAlg`$(minimalEdges)$

The $O(n^3)$ time complexity of the Hungarian algorithm dominates Algorithm 3 and thus the time complexity of Algorithm 4 is $O(n^3)$. The breadth-first search of Ford-Fulkerson in Algorithm 3 gives a space complexity of $O(n^2)$. Algorithm 4 meets the criterion for being SCRAM as it computes the *CM valid* MMD+MSD^2 role assignment function with polynomial time and space complexities.

5.4 Assignment Function and Algorithm Analysis

To evaluate role assignment algorithms, we generated mapping scenarios for n agents and targets. Both agents and targets were assigned random integer value positions on a two dimensional square grid with sides of length n^2. Table 5.2 shows the average run-time of the SCRAM algorithms presented in Section 5.3 for different values of n. For comparison purposes Table 5.2 also includes data from a couple of non-SCRAM algorithms:[16] an exponential time dynamic programming algorithm for MMDR presented in Appendix C, and a brute force algorithm that evaluates all possible mappings. The slowest was the brute force algorithm evaluating all $n!$ possible mappings. The fastest was MMD+MSD^2 which has the lowest time complexity and a relatively low space complexity as shown in Table 5.1. MMD+MSD^2 took less than half a second for 1000 agents and less than two minutes for 10,000 agents. The polynomial time implementations of MMDR scale well to 100s of agents and are much faster than the dynamic programming implementation of MMDR. The $O(n^4)$ implementation of MMDR scales to 1000 agents and is faster than the $O(n^5)$ implementation except for smaller ($n \leq 20$) inputs—our use case for RoboCup discussed later in Chapter 7—where it takes longer due to the extra computations

[16] Algorithms do not run in polynomial time and thus do not meet the scalable criterion to be considered SCRAM.

needed in its main loop.

Table 5.1: Time and space complexities of algorithms. SCRAM algorithms are shown in bold.

Algorithm	Time Complexity	Space Complexity
MMD+MSD^2	$O(n^3)$	$O(n^2)$
MMDR $O(n^4)$	$O(n^4)$	$O(n^2)$
MMDR $O(n^5)$	$O(n^5)$	$O(n^3)$
MMDR dyn. prog.	$O(n^2 2^{(n-1)})$	$O(n\binom{n}{n/2})$
brute force	$O(n!n)$	$O(n)$

Table 5.2: Average running time (ms) of algorithms for values of n on an Intel(R) Xeon(R) CPU E31270 @ 3.40GHz. Dashes indicate inputs to algorithms that did not complete within five minutes. SCRAM algorithms are shown in bold.

Algorithm	$n = 10$	$n = 20$	$n = 100$	$n = 300$	$n = 10^3$	$n = 10^4$
MMD+MSD^2	0.016	0.062	1.82	21.2	351.3	115006
MMDR $O(n^4)$	0.049	0.262	17.95	403.0	14483	—
MMDR $O(n^5)$	0.022	0.214	306.4	40502	—	—
MMDR dyn. prog.	0.555	2040	—	—	—	—
brute force	317.5	—	—	—	—	—

In Table 5.4 we compare MMDR and MMD+MSD^2 against the following role assignment functions when assigning 10 agents to targets on a 100 X 100 grid.

MSD Minimize sum of distances between agents and targets.

MSD^2 Minimize sum of distances^2 between agents and targets.

Greedy Assign agents to targets in order of shortest distances.

Random Random assignment of agents to targets.

Both MMDR and MMD+MSD^2 have the same lowest average makespan for they are defined so as to minimize the makespan. As can be seen in Table 5.3 none of the other functions are *CM valid* as they fail to minimize the makespan—further analysis of how other functions fail to hold properties is provided in Appendix B.3.

MMDR is the only dynamically consistent function of the ones we compare.

Table 5.3: Role assignment function properties from Section 5.2. *CM valid* functions are shown in bold.

Function	Minimum Makespan	No Collisions	Dynam. Consistent
MMD+MSD^2	Yes	Yes	No
MMDR	Yes	Yes	Yes
MSD^2	No	Yes	No
MSD	No	No	No
Random	No	No	No
Greedy	No	No	No

Table 5.4: Average makespan, average distance, and distance standard deviation over 10^6 assignments of 10 agents to targets on a 100^2 grid. *CM valid* role assignment functions are shown in bold.

Function	Average Makespan	Average Distance	Distance StdDev
MMD+MSD^2	45.79	27.38	10.00
MMDR	45.79	28.02	9.30
MSD^2	48.42	26.33	10.38
MSD	55.63	25.86	12.67
Random	90.78	52.14	19.38
Greedy	81.73	28.66	18.95

Average distance is not something *CM valid* role assignment functions explicitly attempt to minimize. However, this metric can be useful if agents exhaust a shared resource such as fuel when moving. MSD by definition minimizes the average distance and thus represents the best possible value for this metric. MMDR and MMD+MSD^2 both have average distance values close to that of MSD. A third metric is distance standard deviation which is useful if there is a preference for having agents travel similar distances, e.g., wanting to have equal wear and tear across robots. MMDR has the best value for this metric with MMD+MSD^2 being second.

MMDR and MMD+MSD^2 do well across all metrics in Table 5.4.

5.5 Summary

This chapter introduced SCRAM role assignment algorithms for formational positioning of mobile agents, and presented theoretical and empirical analysis of the role assignment problem. SCRAM minimizes the makespan for agents to reach target goal positions while also avoiding collisions among agents. As role assignment algorithms run in polynomial time SCRAM scales to thousands of agents.

An extension to SCRAM role assignment for the prioritization of target positions is presented in Chapter 6, and Chapter 7 provides applications and case studies of positioning systems incorporating SCRAM role assignment used within RoboCup robot soccer domains.

Chapter 6

Prioritized SCRAM Role Assignment

This chapter[17] introduces an extension to SCRAM role assignment presented in Chapter 5 allowing for subsets of role positions to be given different priorities, and in doing so further addresses thesis contributions 2 and 3 in Section 1.2.

It is not always the case that minimizing the makespan—completing a formation as fast as possible—is what is best for a team of robots. There are cases where it is preferable to have a subset of high priority role positions be reached by agents as soon as possible. One example of this is soccer where it is often desirable for players to arrive as fast as possible at positions for marking, i.e., covering/defending, opponents in dangerous offensive locations. An application of a marking system for robot soccer using prioritized role assignment is presented in Section 7.3.

The remainder of this chapter is organized as follows. Section 6.1 presents an extension to SCRAM for prioritized role assignment. An analysis of this extension

[17] This chapter contains material from previously published work in [113].

is provided in Section 6.2, and Section 6.3 summarizes.

6.1 Prioritized Role Assignment

This section presents an extension to SCRAM for prioritized role assignment.

Figure 6.1 shows an example of two agents being assigned to both high priority (H) and low priority (L) target positions using SCRAM role assignment. As SCRAM does not take into account priorities of different positions, the high priority position H will not be reached by an agent until time = 3 despite agent A2 starting only a distance of 1 from H.

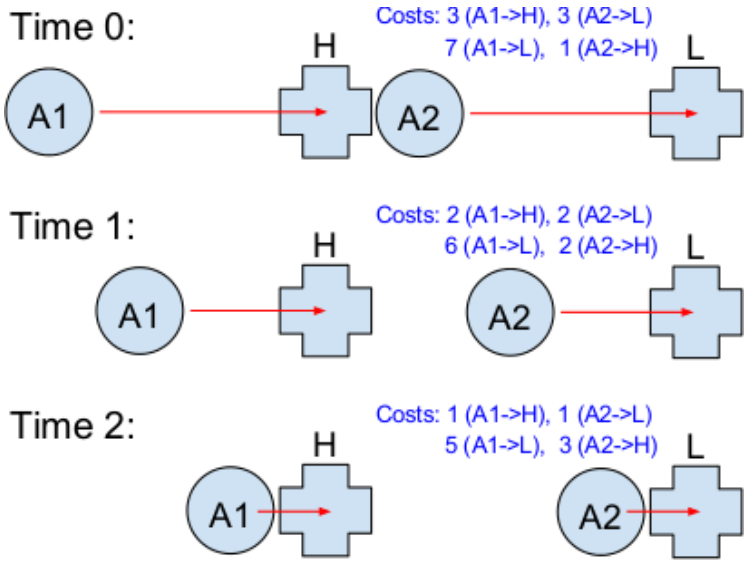


Figure 6.1: Agents A1 and A2 being assigned and moving to the high priority (H) and low priority (L) target positions using SCRAM role assignment.

To bias SCRAM into producing an assignment that has agents reach all high priority positions as fast as possible we can add a large priority value P to costs for reaching all high priority positions. As long as P is greater than all possible distances

to lower priority positions, SCRAM will assign the closest agents to high priority positions before considering the assignment of agents to lower priority positions. This bias of SCRAM assigning closer agents to higher priority positions is due to all costs to higher priority positions being greater and thus needing to be minimized before that of costs to lower priority positions.

Figure 6.2 shows an example of two agents being assigned to both high priority (H) and low priority (L) target positions using the MMDR SCRAM role assignment algorithm, but with a large priority value P added to the costs of reaching H. This results in H being reached at time = 1 by agent A2, but unfortunately later agent A1, on its way to its assigned position L, collides with A2. Assigning the closest agents to high priority target positions, and thereby no longer necessarily recursively minimizing the maximum distance that any agent must travel to reach its assigned target, breaks the collision avoidance property of SCRAM.

To preserve collision avoidance, but still prioritize a subset of targets being reached as fast as possible, we can define a priority distance D around high priority targets for which agents within D distance of a target will not have the priority value P added to the cost of that target.

$$\texttt{cost}(agent,target) = \begin{cases} |\overline{agent,target}| + \mathrm{P} & \text{if } |\overline{agent,target}| > \mathrm{D} \\ |\overline{agent,target}| & \text{otherwise} \end{cases}$$

Figure 6.3 shows an example of two agents being assigned to both high priority (H) and low priority (L) target positions using SCRAM role assignment, but with a large priority value P added to the costs of reaching H when agents are outside a priority distance D of H. This results in H being reached at time = 1

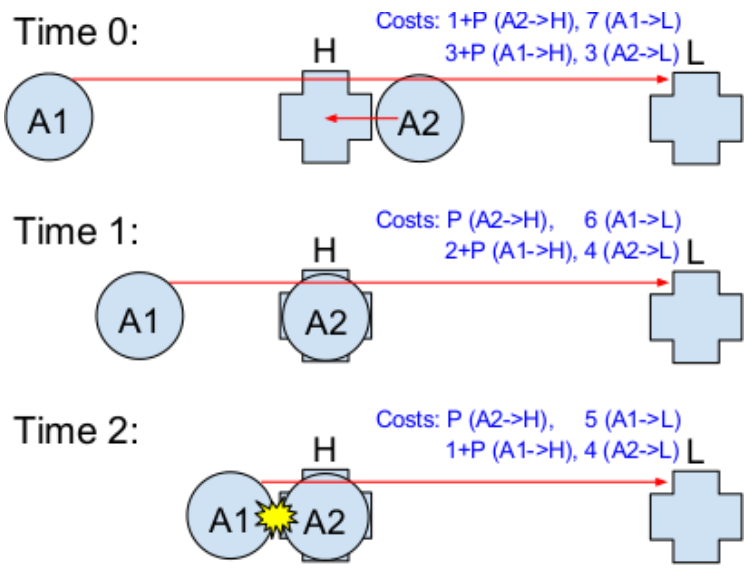


Figure 6.2: Agents A1 and A2 being assigned and moving to the high priority (H) and low priority (L) target positions using SCRAM role assignment, but with a large priority value P added to the costs of reaching H. At time = 2 agents A1 and A2 collide with each other.

by agent A2, and then later when agent A1 gets within D of H agents A1 and A2 switch targets and avoid colliding.

Defining a priority distance D causes agents to arrive within a distance D of all high priority targets as fast as possible. Although agents might not arrive exactly at the high priority targets in as little time as possible, this is often fine for many applications including marking in soccer. When marking a player does not need to be right next to an opponent, but just within a close enough distance to the opponent to be able to react quickly and prevent the opponent from receiving the ball. Assuming D is not too large, should a player come within D distance of an opponent who is already being marked by a teammate, it should then be safe for the players to switch who is marking the opponent.

Augmenting costs with a large P priority value and D priority distance for

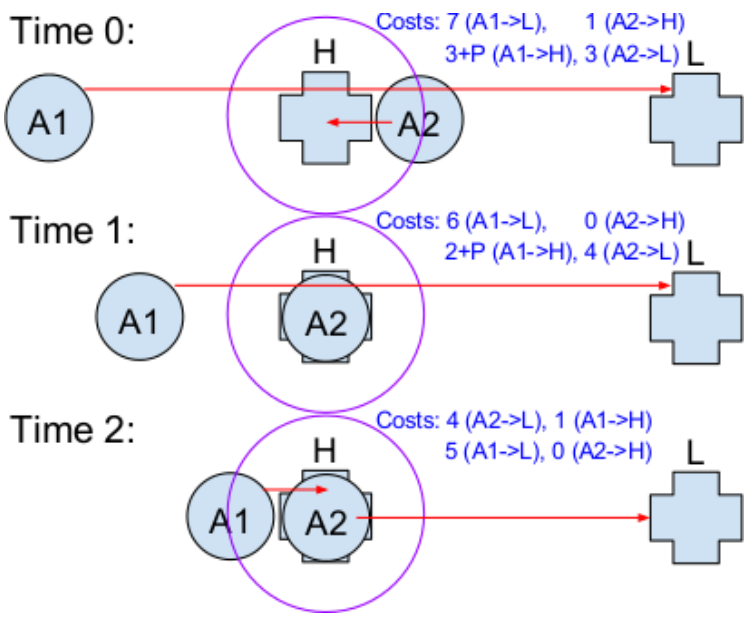


Figure 6.3: Agents A1 and A2 being assigned and moving to the high priority (H) and low priority (L) target positions using SCRAM role assignment, but with a large priority value P added to the costs of reaching H for any agents outside the priority distance of H (the purple circle). At time = 2 agents A1 and A2 switch targets due to agent A1 being within the priority distance of H.

high priority positions extends SCRAM role assignment to allow for prioritization of targets. The collision avoidance property of SCRAM, based on the triangle inequality and fully explained in Appendix B.1, is still preserved with prioritization as any agents within D distance of a high priority target will switch targets before they collide.

It is possible to have multiple subsets of targets with different priorities, or a hierarchy of prioritization, by assigning different P values to different subsets of targets. An example of a hierarchy of prioritization is discussed in [113] for which a highest priority target is given a priority value of P_s, and other high priority targets are given a priority value of P_m, where $P_s >> P_m$.

6.2 Prioritized Role Assignment Function Analysis

Similar to the process used to generate the data in Table 5.4, Table 6.1 compares the results of using SCRAM with and without prioritized role assignment, discussed in Section 6.1, when assigning 10 agents to targets on a 100 X 100 grid, but this time with half of the targets (5) being labeled as high priority. Both MMDR and MMD+MSD^2 *CM valid* role assignment functions are evaluated with P = 99,999 and D = 1 when adjusting costs for prioritization. The values listed under the High Priority Targets columns in Table 6.1 are measured from when an agent is within D distance of the high priority target it is assigned to. As expected prioritization minimizes the makespan and reduces the average distance to reach high priority targets, although in doing so both the makespan and average distance for having reached all targets increase.

Table 6.1: Average makespan and average distance over 10^6 assignments of 10 agents to targets on a 100^2 grid for both high priority (5 of the targets) and all targets. Role assignment functions using prioritization are shown in bold.

	High Priority Targets		All Targets	
Function	Avg. Makespan	Avg. Distance	Avg. Makespan	Avg. Distance
MMD+MSD^2 prior.	31.71	9.61	59.72	28.58
MMDR prior.	31.71	9.68	57.76	29.50
MMDR	40.86	13.52	45.86	28.03
MMD+MSD^2	41.03	13.21	45.86	27.41

6.3 Summary

This chapter introduced an extension to SCRAM role assignment presented in Chapter 5 allowing for subsets of role positions to be given different priorities, and presented theoretical and empirical analysis of prioritized role assignment. Prioritized

SCRAM role assignment minimizes the makespan for agents to reach higher priority target goal positions while also avoiding collisions among agents. As role assignment algorithms run in polynomial time prioritized SCRAM scales to thousands of agents.

Combining SCRAM with target prioritization makes it well suited for use in general patrol and coverage tasks. An application of a marking system for robot soccer using prioritized role assignment is presented in Section 7.3.

Chapter 7

SCRAM Applied to Robot Soccer

This chapter[18] presents case studies and analysis of SCRAM role assignment—introduced in Chapter 5—applied to robot soccer. The chapter's empirical evaluation of SCRAM fulfills part of thesis contribution 5 in Section 1.2.

The University of Texas at Austin has been a perennial participant in the annual RoboCup 3D simulation soccer competitions—described in Section 2.2—and has won the championship six times between 2011 and 2017 due in large part to the main contributions of this dissertation.[19] In 2010 UT Austin's RoboCup team, known as UT Austin Villa, first began using the MMDR role assignment function—detailed in Section 5.3.1—to assign players to move to target formation positions on the soccer field. In 2010 games were only 6 versus 6, and so it was possible to compute MMDR using a brute force method that checked all 6! possible assignments of players to target formation positions. In 2011 games were expanded to

[18] This chapter contains material from previously published work in [111, 113].
[19] RoboCup competition results from these years are provided in Appendix E.

9 versus 9, at which point the team switched to using the dynamic programming algorithm in Appendix C to compute MMDR as a brute force method of doing so was no longer tractable. However, when games were further expanded to 11 versus 11 in 2012, the dynamic programming algorithm for computing MMDR was not always fast enough to compute an assignment of players to target formation positions within a single simulation cycle time step (20 ms) [104].

Beginning in 2013, UT Austin Villa started using polynomial time SCRAM algorithms for role assignment that easily scale to 11 versus 11 matches. As an extension and improvement to the team's positioning system, UT Austin Villa developed and employed a marking system for the 2016 RoboCup competition incorporating prioritized role assignment—introduced in Chapter 6—to cover and defend against opponents in dangerous offensive positions from receiving passes.

The remainder of this chapter is organized as follows. Section 7.1 provides an overview of UT Austin Villa's positioning system for players using SCRAM role assignment, while Section 7.2 analyses the positioning system's performance. Section 7.3 presents a marking system using prioritized role assignment, and Section 7.4 analyses the marking system's performance. Finally, Section 7.5 evaluates the use of SCRAM in the RoboCup 2D simulation domain—and environment focusing more on high level strategy and coordination that abstracts away many of the low-level behaviors required for humanoid robot soccer in the RoboCup 3D simulation league, after which Section 7.6 summarizes.

7.1 Positioning using SCRAM

UT Austin Villa uses SCRAM role assignment when determining which positions on the field players should move to during games. In UT Austin Villa's positioning

system players' positions are determined in three steps. First, a full team formation is computed using Delaunay triangulation [14] based on set offset positions from the ball. An example formation is shown in Figure 7.1.

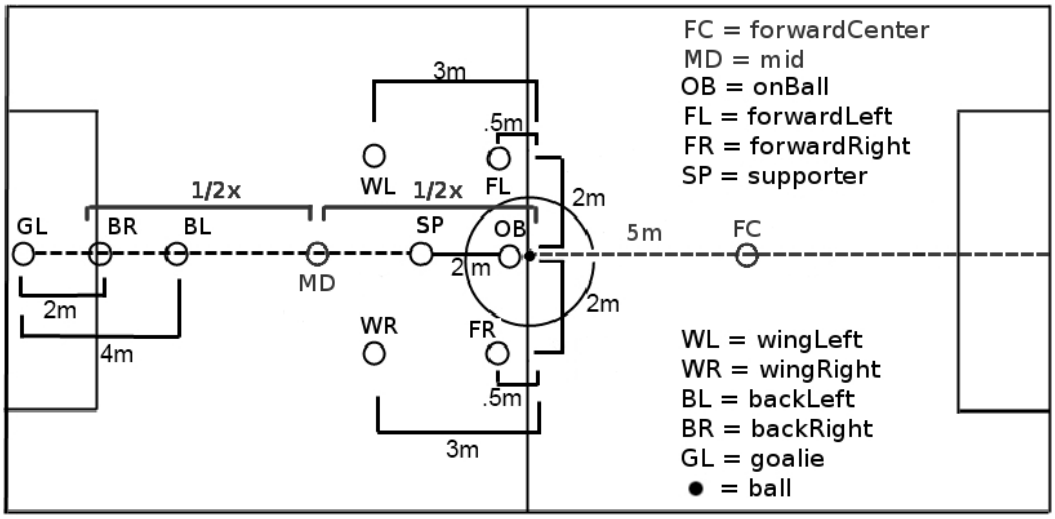


Figure 7.1: UT Austin Villa base soccer formation.

Second, each player computes an assignment of players to positions in the formation in Figure 7.1 according to its own view of the world. The player closest to the ball is automatically assigned the *onBall* role and is told to go to the ball to try and dribble or kick it toward the opponent's goal. The *supporter* role is also assigned to the agent closest to it as it was found in [102] that doing so provides a small boost in team performance due to having a player in a key position to recover the ball should the player assigned to the *onBall* role lose it. All other field players are assigned to roles using SCRAM role assignment.

For the third and final step of the positioning system a voting coordination mechanism synchronizes players' computed assignments. In order for agents on a team to assume correct positions on the field they all must coordinate and agree on

which mapping of agents to roles to use. If every agent had perfect information of the locations of the ball and its teammates this would not be a problem as each could independently calculate the optimal mapping to use. Agents do not have perfect information, however, and are limited to noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Fortunately agents can share information with each other every other simulation cycle (40 ms). The bandwidth of this communication channel is very limited, however, as only one agent may send a message at a time and messages are limited to 20 bytes.

The positioning system utilizes the agents' limited communication bandwidth in order to coordinate role mappings as follows. Each agent is given a rotating time slice to communicate information, as in [158], which is based on the uniform number of an agent. When it is an agent's turn to send a message it broadcasts to its teammates its current position, the position of the ball, and also what it believes the optimal mapping should be using the communication system discussed in Section 8.2.3. By sending its own position and the position of the ball, the agent provides necessary information for computing the optimal mapping to those of its teammates for which these objects are outside of their view cones. Sharing the optimal mapping of agents to role positions enables synchronization between the agents, as follows.

First note that just using the last mapping received is dangerous, as it is possible for an agent to report inconsistent mappings due to its noisy view of the world. This can easily occur when an agent falls over and accumulates error in its own localization. Additionally, messages from the server are occasionally dropped or received at different times by the agents preventing accurate synchronization. To help account for inconsistent information, a sliding window of received mappings

from the last n time-slots is kept by each agent where n is the total number of agents on a team. Each of these kept messages represents a single vote by each of the agents as to which mapping to use. The mapping chosen is the one with the most votes or, in the case of a tie, the mapping tied for the most votes with the most recent vote cast for it. By using a voting system, the agents on a team are able to synchronize the mapping of agents to role positions in the presence of occasional dropped messages or an agent reporting erroneous data.

As a test of the voting system the number of cycles all agents shared a synchronized mapping of agents to roles was measured during five minutes of gameplay (15,000 cycles). The agents were synchronized 100% of the time when using the voting system compared to only 36% of the time when not using it.

7.2 Positioning System Analysis

A SCRAM positioning system using the MMDR role assignment function—presented in Section 5.3.1—was a key component in UT Austin Villa winning the RoboCup 3D simulation world championship in 2011 [117], 2012 [104], 2014 [106], 2015 [108], 2016 [114], and 2017. SCRAM using the MMD+MSD^2 role assignment function —presented in Section 5.3.2—was also an important factor in the team achieving 2nd place in 2013. In Table 7.1 we show UT Austin Villa's performance against the top three teams[20] at the 2013 RoboCup competition using both SCRAM and the following alternative assignment functions with all else being the same.

Static Role assignments fixed based on player's uniform number.

Greedy Assign agents to targets in order of shortest distances.

[20] Descriptions of RoboCup 2013 teams at
http://chaosscripting.net/files/competitions/RoboCup/WorldCup/2013/3DSim/tdps

Greedy Offense Similar to previously reported work in the RoboCup 3D simulation domain [30], assign closest agents to roles in order of most offensive positions.

Table 7.1: Average goal difference (standard error shown in parentheses) over 1000 games when playing against the top three teams at RoboCup 2013. A positive goal difference means a team is winning. *CM valid* role assignment functions are shown in bold.

Function	1. Apollo3D	2. UTAustinVilla	3. FCPortugal
MMDR	0.710 (0.027)	0.007 (0.013)	0.469 (0.024)
MMD+MSD^2	0.698 (0.027)	0 (self)[a]	0.499 (0.024)
Static	0.604 (0.027)	-0.012 (0.016)	0.356 (0.024)
Greedy	0.530 (0.028)	-0.044 (0.016)	0.315 (0.024)
Greedy Offense	0.670 (0.027)	-0.039 (0.016)	0.435 (0.024)

[a]Games were not played, but assumed to be an average goal difference of 0 in expectation with self play.

The *CM valid* role assignment functions are superior to the other functions as they perform better against all opponents. In an earlier study [102] MMDR was also compared to and did better than Static role assignment when playing against the best three 2011 RoboCup teams.

7.3 Marking using Prioritized Role Assignment

As an extension and improvement to the team's positioning system, UT Austin Villa developed and employed a marking system [113][21] for the 2016 RoboCup competition to cover and defend against opponents in dangerous offensive positions from receiving passes. The marking system first identifies target positions that agents need to move to in order to cover opponents. Then, after labeling these positions

[21]Videos of SCRAM prioritized role assignment and the marking system in action can be found at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/marking.html

needed for marking as high priority so that they will be reached as soon as possible, the marking system assigns agents to target positions using SCRAM prioritized role assignment introduced in Chapter 6.

The marking system implemented by the UT Austin Villa team is a sequential process encompassing the following steps:

1. Decide which players to mark
2. Select which roles to use for marking purposes
3. Use prioritized role assignment to assign players to positions

Each of these steps, as well as additional details of the marking system, are described in the following subsections.

7.3.1 Deciding Whom to Mark

The first step in the marking system is to decide which if any opponents should be marked (those opponents considered to be in dangerous offensive positions). The decision on whether or not to mark an opponent is heuristic-based and uses the following rules all of which must be true for an opponent to be marked:

1. Opponent is close enough to take a shot on goal
2. Opponent is not the closest opponent to the ball
3. Opponent is not too close to the ball
4. Opponent is not too far behind the ball

The first rule suggests that an opponent is in a dangerous scoring position. As the team always sends one player to the ball (the *onBall* role in Figure 7.1), the

second and third rules prevent marking of opponents when the team should already have a player moving toward their positions. The fourth rule is due to very few teams passing the ball backwards. Figure 7.2 shows opponent agents selected to be marked.

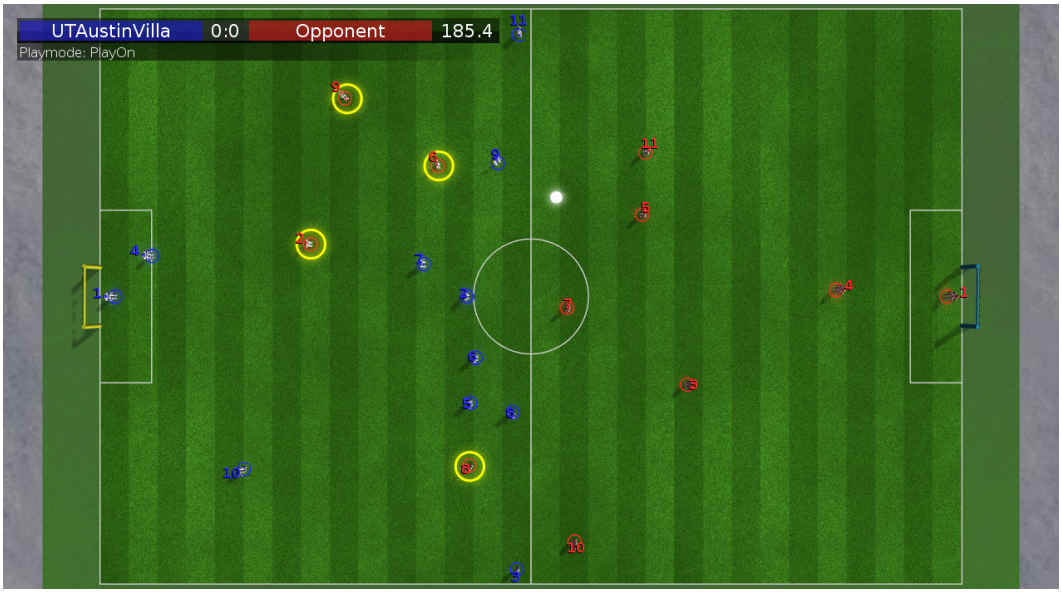


Figure 7.2: Deciding Whom to Mark: Opponent agents selected to be marked are circled in yellow. The white dot is the ball.

7.3.2 Selecting Roles for Marking

Once it is determined which opponents should be marked, the next step is to select which formation role positions should be given up in favor of having agents who would otherwise be assigned to those roles instead move to marking role positions. Marking role positions are calculated as the position 1.5 meters from a marked opponent along the line from that opponent to the center of our goal (shown in orange in Figure 7.3). The selection of formation positions to replace with marking positions

is determined by using the Hungarian algorithm [86] to compute the minimum sum of distances matching between all formation and marking positions in a bipartite graph. This matching results in the closest formation positions to marking positions being replaced by the marking positions that they are nearest. If there are more marking positions than available formation positions then some marking positions will be matched to dummy nodes in the bipartite graph and not be assigned to an agent.

Figure 7.3 shows the result after selecting formation positions to be used as marking role positions with the formation positions selected drawn in purple, and those not selected and still being used drawn in green.

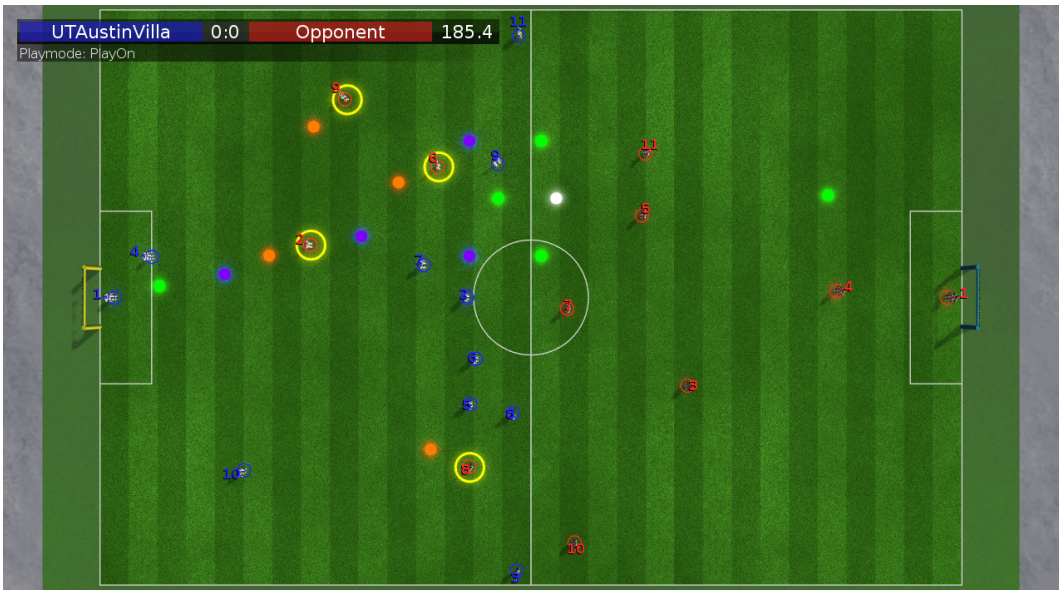


Figure 7.3: Selecting Roles for Marking: Green dots represent target formation positions with purple dots representing target formation positions that have been selected to be replaced by the orange dot marking positions.

7.3.3 Assigning Roles

For the third and final step of the marking system, agents are assigned to marking positions and formation positions (Figure 7.4 shows these assignments in orange and light blue respectively) using prioritized SCRAM role assignment discussed in Section 6.1. Marking positions are considered higher priority than formation positions, and use a priority value $P_m = 100$ and a priority distance $D_m = 3$. Additionally, when a teammate is kicking the ball, a couple of players are assigned to high priority "kick anticipation" position roles near the location where the ball is being kicked to [105]. Kick anticipation roles are given the same priority value and distance as marking roles.

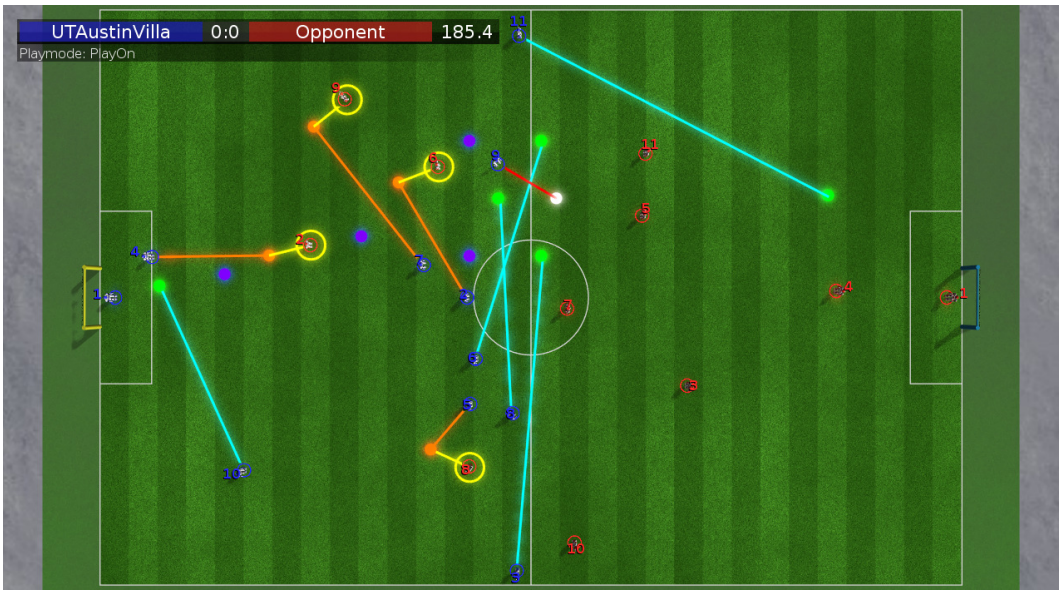


Figure 7.4: Assigning Roles: Orange lines represent agents assigned to marking positions, light blue lines represent agents assigned to target formation positions, and the red line shows the agent assigned to go to the ball.

There are several roles in Figure 7.1 that are never reassigned to be marking

roles. The *goalie* role is always assigned to a single agent designated as the goalie who is allowed to dive and block a ball when an opponent takes a shot on goal. The *onBall* role is always assigned to the agent closest to the ball as it is that agent's job to gain possession of the ball. The *supporter* role is also very important as the *supporter* is in a critical position right behind the ball should the *onBall* role agent lose possession of the ball. The *supporter* is considered a higher priority role than marking roles, and thus uses a priority value $P_s = 10000$ along with a priority distance $D_s = 1.5$. All P and D values were chosen experimentally by sampling several different priority values and then selecting those which produced the most desired behavior based on visual inspection.

7.3.4 Coordination

The marking system uses the same voting coordination system described in Section 7.1 to synchronize the mapping of agents to roles used by the team. Coordination can become more difficult, however, if an opponent is standing in a position right on the borderline of whether or not the opponent should be marked. To prevent thrashing between different role assignments in such a situation, opponents who are currently being marked must move at least .25 meters outside a mark-able position on the field before they will stop being marked. Also, to prevent thrashing between different selections of formation positions to use for marking, a selection is never changed—assuming the cardinality of matchings are the same—unless the new selection's matching's sum of distances is at least one meter less than the previous selection's matching's sum of distances.

7.4 Marking System Analysis

After the 2015 RoboCup competition was over, we played 1000 games of our team's released binary against all teams' released binaries (this includes playing against ourselves) and found that only the UTAustinVilla and FCPortugal teams' binaries were able to score over 100 goals against our released binary [108]. Both the UTAustinVilla and FCPortugal teams created set plays allowing them to score quickly off kickoffs which empirically we found to be the source of the majority of the goals that they scored against our released binary (74.5% of goals for UTAustinVilla and 78.2% of goals for FCPortugal)

To test the effectiveness of our marking system using prioritized role assignment we played 1000 games against both the UTAustinVilla and FCPortugal teams' released binaries using the marking system (`Prioritized Marking`). We also played 1000 games against both teams without using marking (`No Marking`) as well as with marking but using normal non-prioritized SCRAM role assignment (`Marking No Prioritization`). Results of the number of goals against scored by opponents can be seen in Table 7.2, and an analysis of the scoring percentage of opponents' set plays is shown in Table 7.3.

Table 7.2 shows a dramatic drop in the number of goals scored by opponents when using marking. There is also a small decrease in the number of goals against when using prioritized SCRAM instead of non-prioritized SCRAM for marking.

Table 7.2: Number of goals against when playing 1000 games against the released binaries of UTAustinVilla and FCPortugal from RoboCup 2015.

Opponent	No Marking	Marking No Prioritization	Prioritized Marking
UTAustinVilla	1525	336	319
FCPortugal	230	40	37

Table 7.3: Scoring percentage of opponents' set plays when playing 1000 games against the released binaries of UTAustinVilla and FCPortugal from RoboCup 2015.

Set Play	No Marking	Marking No Prior.	Prior. Marking
UTAustinVilla Kickoff	48.31	0.16	0.16
FCPortugal Kickoff	6.22	0.06	0.06
UTAustinVilla Corner Kick	15.97	12.31	7.59

Table 7.3 reveals the source of the reduction in goals against as using marking almost completely eliminates the opponents' abilities to score on kickoff set plays. FCPortugal's kickoff (shown in Figures 7.5, 7.6, and 7.7) consists of a player first passing the ball backwards on the kickoff to a waiting player who then passes the ball forward to a teammate running forward to a dangerous offensive position on the side of the field. UTAustinVilla's kickoff and corner kick set plays are described in [108]. Although the numbers in Table 7.3 do not show an advantage in using prioritized SCRAM over non-prioritized SCRAM against kickoffs, we have seen some instances such as in Figure 7.6 where not using prioritization is harmful. Prioritized SCRAM does however show an advantage against UTAustinVilla's corner kick set plays. Videos of set plays and marking are available online.[22]

Overall using marking, and to a greater extent using marking with prioritized SCRAM role assignment, provides a considerable defensive advantage when playing soccer against opponents who use set plays. The average goal difference across 1000 games when playing against UTAustinVilla with prioritized marking improved to 0.657 (+/-0.028) from ∼0 without marking, and this same number against FCPortugal improved to 2.530 (+/-0.040) with prioritized marking from 2.476 (+/-0.043) without marking. We also played 1000 games against the other ten teams' released binaries from RoboCup 2015, none of which we knew to use passing for set plays,

[22] http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2016/html/marking.html

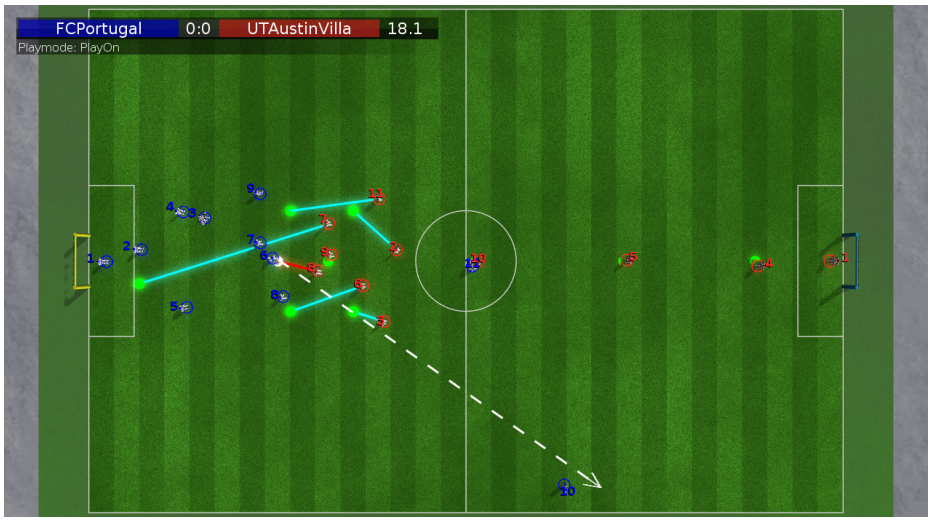


Figure 7.5: Not marking against FCPortugal kickoff. Dashed white line shows trajectory of ball during pass. Not marking allows for an opponent to run forward and receive a pass in an open position to score a goal (blue 10 is not marked).

and found no measurable difference in goals against or game performance when using marking versus those teams.

To further test our marking system using prioritized role assignment against set plays, we evaluated it against the top three teams from the 2016 RoboCup competition. A new rule change for the 2016 competition made free kicks indirect—another player other than the player who took the kick must touch the ball before a goal can be scored—thus encouraging teams to develop set plays for free kicks.

Table 7.4 displays the average number of goals scored against our team—both with and without the use of prioritized role assignment during marking, as well as without any marking—when playing 1000 games against each of the top three teams[23] at the 2016 RoboCup competition. Table 7.4 also provides the percentage

[23]Descriptions of RoboCup 2016 teams are at
http://chaosscripting.net/files/competitions/RoboCup/WorldCup/2016/3DSim/tdps

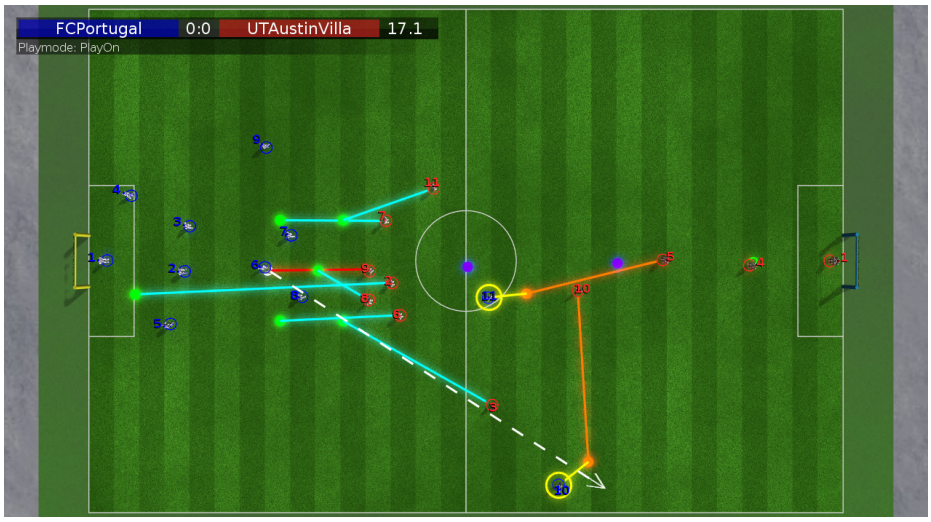


Figure 7.6: Marking, but not prioritized, against FCPortugal kickoff. Dashed white line shows trajectory of ball during pass. Not using prioritization with marking results in a player assigned to mark an opponent being too far away from that assigned opponent to prevent the opponent from scoring a goal (red 10 instead of red 3 assigned to mark blue 10).

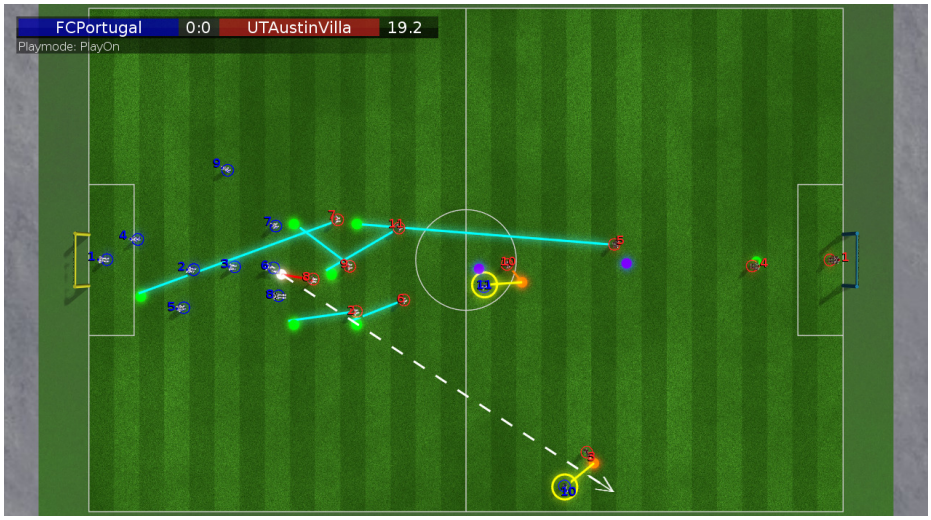


Figure 7.7: Prioritized marking against FCPortugal kickoff. Dashed white line shows trajectory of ball during pass. Prioritized marking prevents opponents from receiving a pass in an open position to score a goal (red 3 marking blue 10).

of goals against that occurred during set play situations[24] when defending against open opponents receiving passes is especially important.

Table 7.4: Average goals against (standard error in parentheses), and percentage of goals against scored off set plays, achieved by versions of the UT Austin Villa team without marking, and with and without the use of prioritized role assignment during marking, when playing 1000 games against the top three teams at RoboCup 2016.

	Goals Against; Set Play %		
Opponent	No Marking	Marking No Prioritized Role Assignment	Marking Prioritized Role Assignment
1. UTAustinVilla	0.667 (0.024); 61.67%	0.307 (0.017); 19.87%	0.290 (0.016); 18.62%
2. FUT-K	0.288 (0.016); 66.67%	0.085 (0.009); 34.12%	0.063 (0.008); 25.40%
3. FCPortugal	0.160 (0.012); 81.88%	0.033 (0.006); 48.48%	0.021 (0.005); 23.81%

The data in Table 7.4 shows a large drop in both the average goals against and percentage of goals from set plays when using marking. Additionally, as prioritized role assignment results in an assignment of agents to targets such that high priority marking positions are reached as soon as possible, both the total number and percentage of goals against from set plays are lowest against all opponents when using prioritized role assignment with marking.

[24]Set play goals are recorded as goals scored within a certain time window of kicks awarded to a team. Times were chosen after measuring how long it took for a team to score a goal without an opponent on the field. When playing against the FUT-K and FCPortugal opponents, time windows for counting goals as being scored off set plays were the following: 20 seconds after corner kicks, 25 seconds after kick-ins and indirect free kicks, and 35 seconds from the start of kickoffs. These time windows were reduced by 5 seconds when playing against the UTAustinVilla opponent.

7.5 RoboCup 2D Simulation

As one of the oldest RoboCup leagues, 2D simulation soccer has been well explored, both in competition and in research. The domain consists of two teams of eleven autonomous agents playing soccer on a simulated 2D soccer field shown in Figure 7.8. Agents receive sensory information, including the position of the ball and other agents, from a central game server. After processing this information, agents then tell the server what actions they want to take such as dashing, kicking, and turning. 2D soccer abstracts away many of the low-level behaviors required for humanoid robot soccer in the 3D simulation league, including walking, and thus affords the chance to focus on higher-level aspects of playing soccer such as multiagent coordination and strategic play.

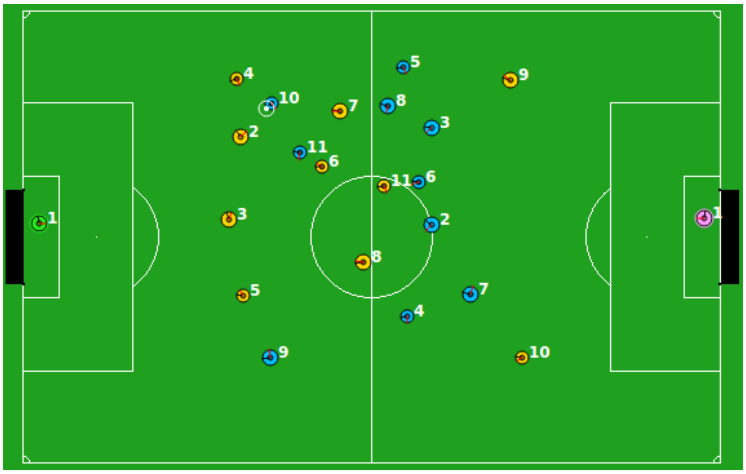


Figure 7.8: A screenshot of a 2D soccer simulation league game.

To test SCRAM in the RoboCup 2D simulation league we used the Agent2D [13] base code release which provides a fully functional soccer playing agent team. Agent2D includes default formation files using Delaunay triangulation [14] to spec-

ify agent role positions. In the Agent2D base code, agents are statically assigned to roles based on their uniform numbers. Agent2D teams only modified to use the MMDR and MMD+MSD^2 role assignment functions beat the default Agent2D team by an average goal difference of 0.118 (+/- 0.025) and 0.105 (+/- 0.024) respectively across 10,000 games.

New at RoboCup 2013 was the addition of a drop-in player challenge [107][25] where agent teams consisting of different players randomly chosen from participants in the competition play against each other. This event is also known as an ad hoc teamwork challenge in which agents must work together as a team without pre-coordination [156]. Performance in the challenge was measured by an agent's average goal difference across all games played. An important aspect of the challenge is for an agent to be able to adapt to the behaviors of its teammates: for instance if most of an agent's teammates are assuming offensive roles, that agent might better serve the team by taking on a defensive role. SCRAM implicitly allows for this adaptation to occur as it naturally chooses roles for an agent that do not currently have another agent nearby.

Using agents from the drop-in player challenge, we played 2800 drop-in player matches with both the default version of Agent2D and a version of Agent2D with SCRAM (MMD+MSD^2). Empirically we found most agents used static role assignment, thus underscoring the need for adapting to teammates' fixed roles, as it was unlikely that teammates would adapt to roles assumed by you. Adding SCRAM to Agent2D improved performance in the challenge from an average goal difference of 1.473 (+/-0.157) with static role assignments to 1.659 (+/-0.153) with SCRAM. This result shows promise for SCRAM not only a way to coordinate motion among

[25] Full rules of the challenge can be found at
http://www.cs.utexas.edu/~AustinVilla/sim/2dsimulation/2013_dropin_challenge/2D_DropInPlayerChallenge.pdf

one's own teammates, but also as as a way to adapt to unknown teammates' behaviors in an ad hoc teamwork setting.

7.6 Summary

This chapter provided a detailed description and experimental analysis of SCRAM role assignment applied to positioning and marking systems in the RoboCup 3D simulation league. These applications of SCRAM role assignment have been a key component of the UT Austin Villa team winning the RoboCup 3D simulation league six out of the past seven years. Additionally the chapter presented an application of SCRAM to the RoboCup 2D simulation league showing SCRAM's versatility and potential use for ad hoc teamwork.

Chapter 8

UT Austin Villa RoboCup 3D Simulation Agent and Code Release

This chapter[26] presents the University of Texas at Austin's RoboCup 3D simulation team UT Austin Villa—a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years—and in doing so addresses thesis contribution 4 in Section 1.2. The UT Austin Villa agent incorporates the ideas and algorithms presented in this thesis, thus serving as a proof of concept of them as detailed in Chapters 4 and 7. Furthermore, a public base code release of the UT Austin Villa agent provides a testbed for future research in multirobot systems.

The remainder of this chapter is organized as follows. Section 8.1 gives a high level overview of the UT Austin Villa agent while Section 8.2 details different components of the agent. Section 8.3 presents a base code release of the UT Austin

[26]This chapter contains material from previously published work in [116, 104, 105, 103, 115].

Villa agent, and Section 8.4 summarizes.

Further details about the UT Austin Villa agent pertaining to the team's strategy are provided in Appendix D, with details of how the team's strategy incorporates formations and role assignment discussed in Chapter 7. The optimization process used to develop the agent's skills is detailed in Chapter 4.

8.1 Agent Overview and Architecture

The RoboCup 3D simulation environment—described in Section 2.2—is a 3-dimensional world that models realistic physical forces such as friction and gravity, in which teams of autonomous soccer playing humanoid robot agents compete with each other. The UT Austin Villa team, from the University of Texas at Austin, first began competing in the RoboCup 3D simulation league in 2007. Over the course of nearly a decade the team has built up a strong state of the art code base enabling the team to win the RoboCup 3D simulation league six out of the past seven years (2011 [117], 2012 [104], 2014 [105], 2015 [108], 2016 [114]), and 2017 while finishing second in 2013. During those six years of competitions the UT Austin Villa team scored a total of 469 goals while only conceding 7.[27]

The UT Austin Villa agent works as follows. At intervals of 0.02 seconds, the agent receives sensory information from the environment. Every third cycle a visual sensor provides distances and angles to different objects on the field from the agent's camera, which is located in its head. It is relatively straightforward to build a world model by converting this information about the objects into Cartesian coordinates. Building a world model also requires the robot to be able to localize itself for which the agent uses a particle filter incorporating both landmark and field

[27] RoboCup competition results are provided in Appendix E.

line observations as described in Section 8.2.2. In addition to the vision perceptor, the agent also uses its accelerometer readings to determine if it has fallen as explained in Section 8.2.4, and employs its auditory channels for communication which is discussed in Section 8.2.3.

Once a world model is built the agent's control module is invoked. Figure 8.1 provides a schematic view of the UT Austin Villa agent's control architecture.

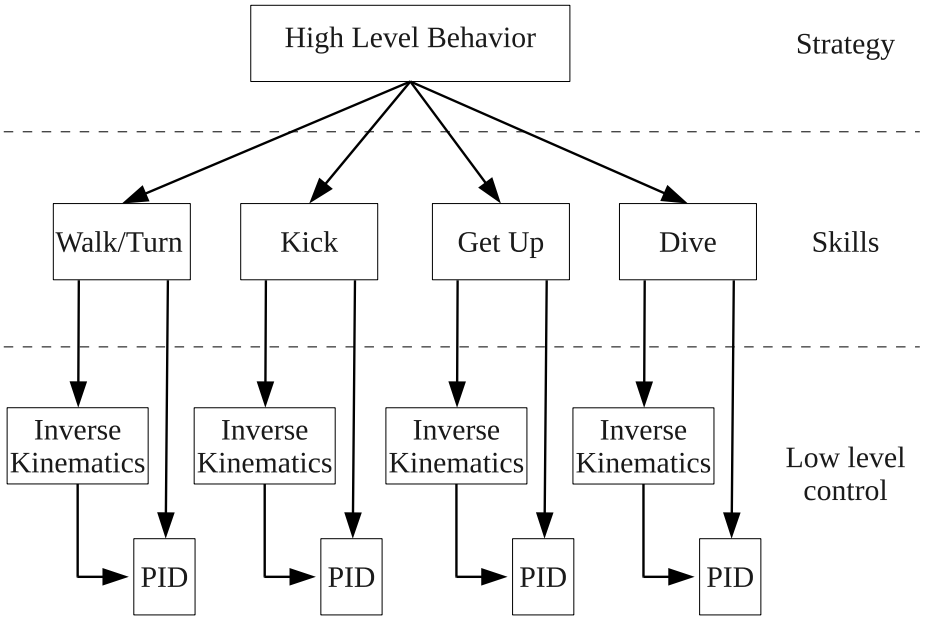


Figure 8.1: Schematic view of UT Austin Villa agent control architecture.

At the lowest level, the humanoid is controlled by specifying torques to each of its joints. The amount of torque to apply is determined by PID controllers for each joint, which take as input the desired angle of the joint and compute the appropriate torque. Further, the agent uses routines describing inverse kinematics for the arms and legs. Given a target position and pose for the hand or the foot, the inverse kinematics routine uses trigonometry to calculate the angles for the different joints

along the arm or the leg to achieve the specified target, if at all possible.

The PID control and inverse kinematics routines are used as primitives to describe the agent's skills. In order to determine the appropriate joint angle sequences for walking and turning, the agent utilizes an omnidirectional walk engine which is described in Section 8.2.5. Other provided useful skills for the robot are kicking discussed in Section 8.2.7, and getting up from a fallen position described in Section 8.2.4. These skills are accomplished through a programmed sequence of poses and specified joint angles as discussed in Section 8.2.6.

Section 8.2 goes into more detail about different components of the UT Austin Villa agent.

8.2 Agent Components

The following subsections detail different components of the UT Austin Villa agent. These components include a perception system in Section 8.2.1, localization in Section 8.2.2, a communication system in Section 8.2.3, fall recovery in Section 8.2.4, walk engine in Section 8.2.5, skill description language in Section 8.2.6, and kicking in Section 8.2.7.

8.2.1 Perception System

The agent receives noisy vision percepts from the server every 0.06 seconds. Vision percepts in polar coordinates are received for all objects that are within a 120° view cone with the view cone's origin at the robot's head. In this section we describe how the agent moves its head to monitor objects, how the agent maintains objects' positions in memory in a *WorldObjects* array, and how the agent uses visual memory to enhance its perception system to handle unseen objects whenever possible.

Head Movement

The agent continually pans its head from side to side in order to monitor all objects on the field. This panning consists of a repeating cycle, eight seconds in duration, during which the agent adjusts the pan of its head every two seconds by starting out looking straight ahead, turning its head 120° to the left, returning to looking straight ahead, turning its head 120° to the right, and then returning back to looking straight ahead again. As the agent has a 120° view cone this movement allows for a full 360° vision sweep of the field. The only time the agent does not spin its head around for 360° coverage of the field is when it is focusing on the ball. When focusing on the ball the agent still pans its head left and right, but centers its view on the ball while moving its view 30° to the left and right of the ball. The goalie, described in Appendix D.3, always stays focused on the ball in order to accurately track it. Field player agents focus on the ball when they are within a meter of it as it is necessary to do so for dribbling (Appendix D.1.6) and kicking (Section 8.2.7). The agent also keeps its head tilted at a 45° angle downward providing continual vision both 15° above the agent's camera and 15° behind the agent's feet.

World Objects

The basic structure for holding an object is:

```
struct WorldObject
  id;              // unique id for all objects that can ever be seen
  visionInformation;        // polar coordinates from robot to object
  position;                              // global, Cartesian position
  isCurrentlySeen;         // is the object seen in the current cycle
  isValid;      // false if looking at position but do not see object
```

;

In the RoboCup 3D simulation domain the set of objects is fixed: teammates, opponents, ball, goal posts, and field corners. Therefore, objects are maintained in an array of WorldObjects, indexed by the object id. Whenever vision percepts are sent, the polar coordinates are stored in the *visionInformation* field, and are translated to global position after the agent updates its localization. In addition, the vision information of fixed objects, like goal posts and field corners, is used by the localization system to update the agent's belief about its location as is described in Section 8.2.2.

One point that is noteworthy is that the vision percepts arrive in polar coordinates with respect to the robot's head, which continually pans left and right, to increase the robot's field of view. This head movement has the potential to create a problem: it is likely that at the times of recording vision information of two different objects, the head is in two different positions with respect to the torso. In addition, the robot's head is slightly bent downwards, so that even if the head is looking straight ahead, the recorded angles to objects are still different than their angles with respect to the torso. Therefore, as a preprocessing step, vision data is transformed using several matrix multiplications to be with respect to the robot's torso. This transformation is done to create a common frame of reference during the translation of vision information to global field coordinates.

Visual Memory

When an object, such as another agent or ball, is no longer in the field of view of the robot it is assumed that the object remains at the same position it was last seen—as moving objects frequently change direction potential velocity of an object

outside the robot's field of view is not used to estimate the location of the unseen object. The location of the object is updated once it is seen again or if another agent communicates the location of the object (see Section 8.2.3). If the stored position of the object is in the agent's current field of vision, but the agent no longer sees the object there, the location of the object is marked as being no longer valid—the *isValid* field of WorldObject struct is set to `false`.

8.2.2 Localization

Localization is performed using a particle filter also known as Monte Carlo localization [48]. In the agent's particle filter, 500 particles are updated every cycle, where a particle is an (x, y, θ) estimate of an agent's pose, and a probability assigned to this estimate. The agent estimates its (x, y) position and θ orientation as weighted averages of the particles' positions and orientations, respectively, weighted by the particles' weights. Each time the agent receives a set of vision percepts, the particle filter performs the following three steps: (1) update particles from odometry, (2) update particles from landmark and field line observations, and (3) resample particles. Agents also use communication to help localize each other.

Update Particles from Odometry

In this step, odometry information in the form of $(\delta x, \delta y, \delta\theta)$ is extracted from the walk engine described in Section 8.2.5. Then, for each particle, the following update is performed:

$$(x, y, \theta) := (x, y, \theta) + (a \cdot \Delta x, b \cdot \Delta y, c \cdot \Delta\theta)$$

where a, b, and c are factors we tuned by hand, using manual measurements.

Update Particles from Landmark and Line Observations

In this step, particles' weights are updated from landmark—four corner flags and two goal posts at each end of the field—and field line observations. Figure 2.8 shows the locations of landmarks and lines on the field.

While in general at least two landmarks are needed for position estimation, when the agent is roughly localized, it is possible to decrease its estimation error even with one landmark or line. The particle filter is able to take advantage of this estimation ability by performing this step to update its particles whenever at least one landmark or line is seen. At the beginning of this step all particles are initialized to have the same weights, and then these weights are multiplied by the particles' probabilities they assign to the currently seen landmarks and field lines: the difference between the expected measurement and the actual measurement is input to a Gaussian distribution, which in turn determines the probability of this difference. The particle filter has two such Gaussian distributions, one for the distance error, and one for the orientation error. Both of the Gaussians' standard deviations were hand tuned using a trial and error process. Note that as the multiplied probabilities can get small, log-probabilities are used in the numerical calculations.

Line information is processed by the particle filter based on previous work by Hester and Stone [60]. In particular, the longest K observed lines are each compared to known positions of all the lines that exist on the field. Metrics such as the distance between endpoints, acute angle between the lines, and line length ratio are used to determine the similarity of an observed line with each actual line. For each observed line, the highest similarity value is expressed as a probability and used to update particles. Figure 8.2 shows how line information improves localization accuracy for various values of K.

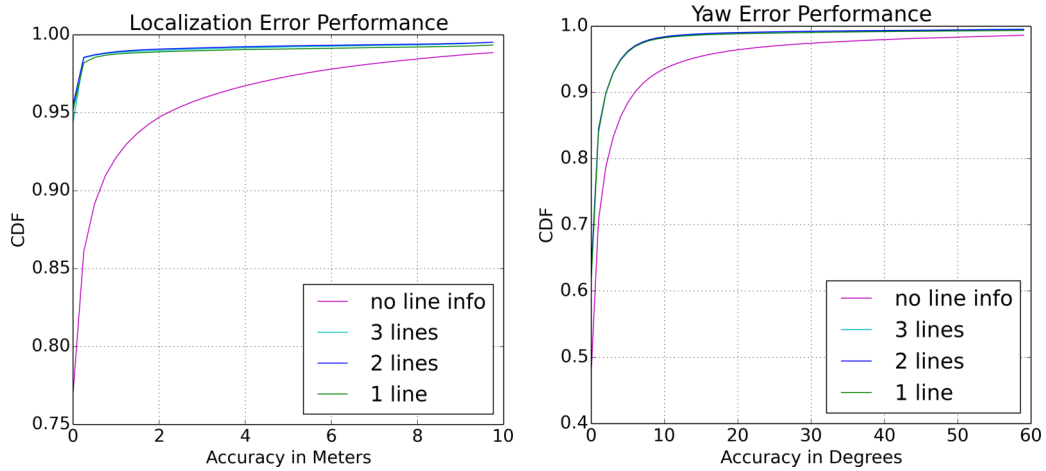


Figure 8.2: CDF of localization error (left) and yaw error (right) for using $K = 1, 2, 3$ lines when incorporating line information. For comparison, not using line information (purple line) is shown as well.

Resample Particles

In this step the new weights computed in the previous step are used as a new distribution over the particles, and then 500 new particles are resampled from this distribution. More accurately, 5% of the particles are sampled completely randomly in the field's area, to handle a "kidnapped" agent scenario—a scenario where the agent is moved to a new location by a third party without the agent being informed of the move. This step is done only when at least one landmark or line is seen, such that particles' weights were updated in the previous step. If no landmarks or lines are seen, the agent just uses its odometry updates and no resampling is done, to avoid inserting additional noise. After particles are resampled, a small amount of noise to each one of them is added by slightly moving their position and orientation according to a random walk.

Localization from Teammates' Observations

Additionally, if a robot does not see any landmarks or lines, it broadcasts to its teammates that it is not localized using the communication system described in Section 8.2.3. If any teammates see a robot that reports itself as not being localized they will broadcast the current (x,y) position and angle of orientation of the unlocalized robot so that the unlocalized robot may use other robots' observations to localize itself.

8.2.3 Communication System

As described in Section 2.2, soccer agents only receive noisy and restricted perceptual information. Consequently none of the agents possesses complete and perfect state information about the world. In such a scenario, inter-agent communication can significantly add to each agent's knowledge about the world and improve decision making.

The 3D simulator provides an "audio" channel for agents to communicate. An agent may broadcast a SAY message once every two simulation cycles (40 milliseconds); agents receive HEAR messages corresponding to SAY messages sent in the previous cycle, and only one HEAR message may be heard from a teammate every two simulation cycles with additional messages not being transmitted. As agents can only communicate one at a time, each agent is given a rotating time slice to communicate information, as in [158], which is based on the uniform number of an agent. The HEAR messages do not come tagged with information identifying the sender, and so we find it necessary to have agents send identifying information within their messages.

The 3D simulation server allows for communication messages of size 20 ASCII

characters with only a set of 94 different ASCII characters allowed to be transmitted (ASCII 0x21 to 0x7E excluding 0x28 and 0x29). The UT Austin Villa agent conservatively uses an alphabet of only 85 ASCII characters, and a Base85 encoding allowing for 128 out of 160 bits of a message to be used. Below we describe the rationing of the 128 bits at the agent's disposal for communicating different types of information. Table 8.1 breaks down the number of bits allocated to each piece of information communicated.

Every message sent by an agent includes the agent's uniform number—to identify the agent—as well as the agent's current location, if the agent sees the ball, and if the agent is reporting a vote for a role assignment as described in Section 7.1. If necessary agents will also report the location of the ball assuming they currently see the ball, if they have fallen over or are not localized, a role assignment vector mapping players to role positions—only reported if an agent does not agree with the most recent vote heard, and if they are kicking the ball their intended target location of the kick. If there are enough remaining unused bits in a message, agents may additionally communicate the locations of teammates and opponents they currently see as well as their own robot body type.

8.2.4 Fall Detection and Recovery

Factors such as slippage on the ground and collision with objects on the field could precipitate the fall of a humanoid robot, and indeed the success of a soccer-playing robot depends crucially on the robustness of its fall management strategy.

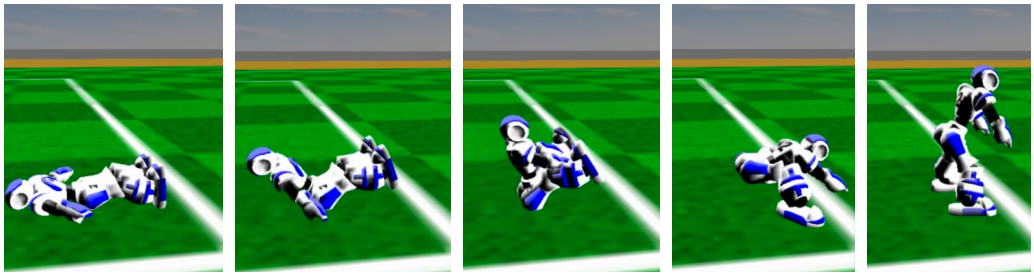
Despite our best efforts in having robots avoid falls, falling is an inevitable eventuality that needs to be dealt with efficiently. To detect that a fall has occurred—or that it is impending—a simple rule that thresholds the X and Y compo-

Table 8.1: Number of bits allocated to each piece of information communicated.

Field	Number of Bits
Always Transmitted	
Sender's uniform number	4
Is sender seeing ball?	1
Is sender reporting role assignment?	1
Sender's perceived self X coordinate	10
Sender's perceived self Y coordinate	10
Sender's perceived orientation angle	6
Always Transmitted if Necessary	
Is the sender fallen flag?	5
Is the sender not localized flag?	5
Sender's perceived ball X coordinate	10
Sender's perceived ball Y coordinate	10
Role assignment vector	50 (10 * 5)
Is the sender planning to kick the ball flag?	5
Kick target location X	8
Kick target location Y	7
Is the sender already behind the ball before kicking?	1
Optional (from higher to lower priority)	
Reporting teammate location flag	5
Reported teammate X coordinate	10 or 8
Reported teammate Y coordinate	10 or 8
Reported teammate orientation angle	6 or 4
Reporting opponent location flag	5
Reported opponent X coordinate	10 or 8
Reported opponent Y coordinate	10 or 8
Reported opponent orientation angle	6 or 4
Reported opponent is fallen?	1
Sender's agent type flag	5
Sender's agent type	3
End of message flag	5

nents of the robot's accelerometer reading is used. If indeed a fall is detected based on this thresholding rule, the robot proceeds to execute a sequence of commands as part of a "getup" routine. We observe that if the robot's arms are stretched out at 90° to the torso, along the frontal plane, the robot necessarily falls to the ground either face-up or face-down (that is, not sideways). Using the accelerometer again to detect whether it has fallen face-up or face-down, the robot proceeds through an appropriate sequence of keyframes—described by the skill description language in Section 8.2.6—in each case to return to an upright position.

The robot's getup routines were manually designed and are divided into stages. If fallen face down, the robot bends at the hips and stretches out its arms until it transfers weight to its feet, at which point it can stand up by straightening the hip angle. If fallen face up, the robot uses its arms to push its torso up, and then rocks its weight back to its feet before straightening its legs to stand. The getup routine used by a robot lying on its back iterates through the series of poses shown in Figure 8.3.



(a) (b) (c) (d) (e)

Figure 8.3: Routine for getting up after falling backwards. The robot begins lying on its back (a) and then propels itself up with its arms (b). Next the robot throws its arms forward and contracts its legs to get its center of mass in front of its feet (c). Using momentum from the initial push the robot manages to roll into a squatting position (d) after which the robot can get up by extending its knees and hips (e).

These getup sequences are executed entirely in an open-loop fashion. There is a small probability the getup routine is not successful—for example, if the robot is in contact with some other object while getting up—and if so the routine is repeated. Parameters controlling the poses the robot assumes during the getup motions were optimized as described in Chapter 4.

8.2.5 Walk Engine

The UT Austin Villa team uses an omnidirectional walk engine[28] based on one that was originally designed for the real Nao robot [51]. The omnidirectional walk is crucial for allowing the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach continually changing destinations (often the ball) more smoothly and quickly than the team's previous set of unidirectional walks [172].

The walk engine, though based closely on that of Graf et al. [51], differs in some of the details. Specifically, unlike Graf et al., the walk engine uses a sigmoid function for the forward component and uses proportional control to adjust the desired step sizes. The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. The work flow of how joint commands are generated for the walk is shown in Figure 8.4. The walk engine processes desired walk velocities chosen by the behavior, chooses destinations for the feet and torso, and then uses inverse kinematics to determine the joint positions required. Finally, PID controllers for each joint convert these positions into torque commands that are sent to the simulator.

The walk engine selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. The walk uses x

[28]Thanks to Samuel Barrett for originally writing the team's omnidirectional walk engine.

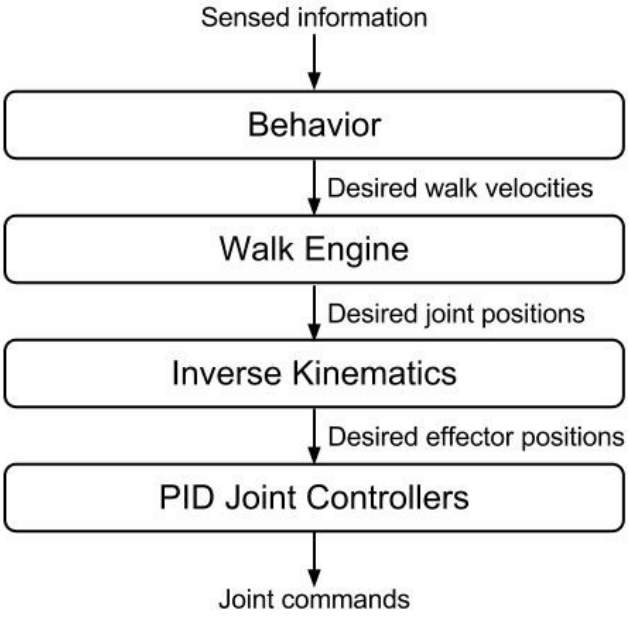


Figure 8.4: Workflow for generating joint commands from the walk engine.

as the forwards dimension, y as the sideways dimension, z as the vertical dimension, and θ as rotating about the z axis. The trajectory is chosen using a double linear inverted pendulum, where the center of mass is swinging over the stance foot. In addition, as in Graf et al.'s work [51], the walk engine uses the simplifying assumption that there is no double support phase, so that the velocities and positions of the center of mass must match when switching between the inverted pendulums formed by the respective stance feet.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 walk engine parameters were used, but only the ones we optimize—as described in Chapter 4—are listed in Table 8.2.

To smooth changes in the velocities, the walk engine uses a simple proportional controller to filter the requested velocities coming from the behavior mod-

Notation	Description
$\text{maxStep}_{\{x,y,\theta\}}$	Maximum step sizes allowed for x, y, and θ
y_{shift}	Side to side shift amount with no side velocity
z_{torso}	Height of the torso from the ground
z_{step}	Maximum height of the foot from the ground
f_{g}	Fraction of phase swing foot spends on the ground before lifting
f_{a}	Fraction that the swing foot spends in the air
f_{s}	Fraction before the swing foot starts moving
f_{m}	Fraction that the swing foot spends moving
ϕ_{length}	Duration of a single step
$\delta_{\text{step}\{x,y,\theta\}}$	Factor of how fast the step sizes change for x, y, and θ
x_{offset}	Constant offset between the torso and feet
x_{factor}	Factor of the step size applied to the forwards position of the torso
$\delta_{\text{target}\{\text{tilt,roll}\}}$	Factors of how fast tilt and roll adjusts occur for balance control
$\text{ankle}_{\text{offset}}$	Angle offset of the swing leg foot to prevent landing on toe
err_{norm}	Maximum COM error before the steps are slowed
err_{max}	Maximum COM error before all velocity reach 0
$\text{COM}_{\text{offset}}$	Default COM forward offset
$\delta_{\text{COM}\{x,y,\theta\}}$	Factors of how fast COM changes x, y, θ values for balance control
$\delta_{\text{arm}\{x,y\}}$	Factors of how fast arm x and y offsets change for balance control

Table 8.2: Optimized parameters of the walk engine.

ule. Specifically, the walk engine calculates $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta_{\text{step}}(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$. In addition, the value is cropped within the maximum step sizes so that $-\text{maxStep}_i \leq \text{step}_{i,t+1} \leq \text{maxStep}_i$.

The phase is given by $\phi_{\text{start}} \leq \phi \leq \phi_{\text{end}}$, and $t = \dfrac{\phi - \phi_{\text{start}}}{\phi_{\text{end}} - \phi_{\text{start}}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{\text{length}}$, until $\phi \geq \phi_{\text{end}}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{\text{start}} = -0.5$ and $\phi_{\text{end}} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

$$
\begin{aligned}
k &= \sqrt{9806.65/z_{\text{torso}}} \\
\alpha &= 6 - \cosh(k - 0.5\phi) \\
\phi_{\text{start}} &= \begin{cases} \dfrac{\cosh^{-1}(\alpha)}{0.5k} & \text{if } \alpha \geq 1.0 \\ -0.5 & \text{otherwise} \end{cases} \\
\phi_{\text{end}} &= 0.5(\phi_{\text{end}} - \phi_{\text{start}})
\end{aligned}
$$

The stance foot remains fixed on the ground, and the swing foot is smoothly lifted and placed down, based on a cosine function. The current distance of the feet from the torso is given by

$$
\begin{aligned}
z_{\text{frac}} &= \begin{cases} 0.5(1 - \cos(2\pi \dfrac{t - f_{\text{g}}}{f_{\text{a}}})) & \text{if } f_{\text{g}} \leq t \leq f_{\text{a}} \\ 0 & \text{otherwise} \end{cases} \\
z_{\text{stance}} &= z_{\text{torso}} \\
z_{\text{swing}} &= z_{\text{torso}} - z_{\text{step}} * z_{\text{frac}}
\end{aligned}
$$

It is desirable for the robot's center of mass to steadily shift side to side, allowing it to stably lift its feet. The side to side component when no side velocity is requested is given by

$$
\begin{aligned}
y_{\text{stance}} &= 0.5y_{\text{sep}} + y_{\text{shift}}(-1.5 + 0.5\cosh(0.5k\phi)) \\
y_{\text{swing}} &= y_{\text{sep}} - y_{\text{stance}}
\end{aligned}
$$

where y_{sep} is the distance between the feet. If a side velocity is requested, y_{stance} is

augmented by

$$y_{\text{frac}} = \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ 0.5(1 + \cos(\pi \frac{t - f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ 1 & \text{otherwise} \end{cases}$$

$$\Delta y_{\text{stance}} = \text{step}_y * y_{\text{frac}}$$

These equations allow the y component of the feet to smoothly incorporate the desired sideways velocity while still shifting enough to remain dynamically stable over the stance foot.

Next, the forwards component is given by

$$\begin{aligned} s &= \text{sigmoid}(10(-0.5 + \frac{t - f_{\text{s}}}{f_{\text{m}}})) \\ x_{\text{frac}} &= \begin{cases} (-0.5 - t + f_{\text{s}}) & \text{if } t < f_{\text{s}} \\ (-0.5 + s) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ (0.5 - t + f_{\text{s}} + f_{\text{m}}) & \text{otherwise} \end{cases} \\ x_{\text{stance}} &= 0.5 - t + f_{\text{s}} \\ x_{\text{swing}} &= \text{step}_x * x_{\text{frac}} \end{aligned}$$

These functions are designed to keep the robot's center of mass moving forwards steadily, while the feet quickly, but smoothly approach their destinations. Furthermore, to keep the robot's center of mass centered between the feet, there is an additional offset to the forward component of both the stance and swing feet, given by

$$\Delta x = x_{\text{offset}} - \text{step}_x x_{\text{factor}}$$

After these calculations, all of the x and y targets are corrected for the current

position of the center of mass. Finally, the requested rotation is handled by opening and closing the groin joints of the robot, rotating the foot targets. The desired angle of the groin joint is calculated by

$$\text{groin} = \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ \frac{1}{2}\text{step}_\theta(1 - \cos(\pi\frac{t - f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \leq t < f_{\text{s}} + f_{\text{m}} \\ \text{step}_\theta & \text{otherwise} \end{cases}$$

After these targets are calculated for both the swing and stance feet with respect to the robot's torso, the inverse kinematics module calculates the joint angles necessary to place the feet at these targets. Further description of the inverse kinematic calculations is given in [51].

To improve the stability of the walk, the walk engine tracks the desired center of mass as calculated from the expected commands. Then, the walk engine compares this value to the sensed center of mass after handling the delay between sending commands and sensing center of mass changes of approximately 20ms. If this error is too large, it is expected that the robot is unstable, and action must be taken to prevent falling. As the robot is more stable when walking in place, the walk engine immediately reduces the step sizes by a factor of the error. In the extreme case, the robot will attempt to walk in place until it is stable. The exact calculations are given by

$$\begin{aligned} \text{err} &= \max_i(\text{abs}(\text{com}_{\text{expected},i} - \text{com}_{\text{sensed},i})) \\ \text{stepFactor} &= \max(0, \min(1, \frac{\text{err} - \text{err}_{\text{norm}}}{\text{err}_{\text{max}} - \text{err}_{\text{norm}}})) \\ \text{step}_i &= \text{stepFactor} * \text{step}_i \, \forall i \in \{x, y, \theta\} \end{aligned}$$

This solution is less than ideal, but performs effectively enough to stabilize the robot in many situations.

There is one robot body type—type 4 as described in Section 2.2—that has as added toe joint on each foot. The only modification made to the walk engine to take advantage of this toe joint is to add an offset to both the ankle pitch and toe joint—the ankle pitch is altered in addition to the toe joint as the ankle pitch can counteract the toe joint's effect on the robot's center of mass. This correction allows the remainder of the walk engine to perform as designed, resulting in a well-tuned walk. The offset to both joints takes the form of

$$\text{offset} = a\cos(t\pi + p) + c$$

where a is the amplitude of the movement, p controls the phase, and c is a constant offset. A sinusoidal curve was chosen to maintain smooth movement that repeats once per step. The parameters for the ankle pitch and toe joint are not linked, resulting in an additional six parameters in the walk engine (three for each foot).

Walk Engine Inputs

Once an agent has decided what direction it wants to walk in (`walk_direction`) and what direction it wants to face (relative to its current orientation), it must give the correct inputs to the walk engine, which accepts as inputs three real numbers in the range [-1, 1]. These numbers are the desired speed—as a percentage of the engine's maximum speed—to walk in the X and Y directions and to rotate. The sign of the number determines the direction of the movement (e.g. positive X for forward and negative X for backward). Converting the desired orientation into rotation speed is simple: divide by the (admittedly arbitrary) number 180.

Converting walk direction into the X/Y speeds is more tricky. An agent cannot simply use `sin(walk_direction)` and `cos(walk_direction)` as the Y and X speed, respectively. There are two reasons for why an agent cannot directly use these values:

1. An agent generally wants to walk in `walk_direction` as fast as possible. Walking in a direction as fast as possible means that either the X speed or the Y speed should equal 1.

2. The maximum X and Y speeds do not have to be the same, so an agent must scale the X and Y speeds accordingly.

The following formula addresses both concerns:

```
if tan(walk_direction) < (max_y_speed / max_x_speed), then
     x_speed = 1
     y_speed = tan(walk_direction) * max_x_speed / max_y_speed
otherwise,
     x_speed = tan(walk_direction) * max_y_speed / max_x_speed
     y_speed = 1
```

Under certain conditions, an agent may also want to change the walk engine parameter set it is using. An agent can change the walk parameter set by simply specifying the name of the desired parameter set along with the X/Y/Rotational speeds. If the walk engine is not already using the specified parameter set, it will switch its parameters values accordingly to that of the new set. This capability allows the agent to switch between different walks, optimized for different purposes as described in Chapter 4, as it sees fit, instead of relying on some type of one-size-fits-all walk.

Stopping and Jogging In Place

An important decision on how a robot should move occurs when a robot wants to stop after reaching a desired target position on the field. We found that if a robot immediately stops and stands still after moving quickly, then stability becomes a concern with the robot often falling over due to the sudden change in motion. One way to preserve stability is to request the walk engine to have the robot jog in place instead of standing still so that the change in motion is more gradual. The drawback of jogging in place is that the added movement adds noise to the robot's localization and perception of objects around it. This added noise is of particular concern for the goalie (discussed in Appendix D.3) who needs very accurate measurements of the position of the ball relative to itself so that it can determine when to dive to stop the ball if an opponent attempts a shot on goal. The walk engine compromises between standing and jogging in place by having the robot jog in place when stopping for half a second, after which the robot enters a motionless standing pose.

8.2.6 Skill Description Language

The UT Austin Villa agent has skills for getting up after falling and kicking, each of which is implemented as a periodic state machine with multiple *key frames*, where a key frame is a static pose of fixed joint positions. Key frames are separated by a waiting time that lets the joints reach their target angles. To provide flexibility in designing and parameterizing skills, we designed an intuitive skill description language that facilitates the specification of key frames and the waiting times between them. Below is an illustrative example describing a kick skill.

```
SKILL KICK_LEFT_LEG


KEYFRAME 1
```

```
setTarget JOINT1 $jointvalue1 JOINT2 $jointvalue2 ...
setTarget JOINT3 4.3 JOINT4 52.5
wait 0.08

KEYFRAME 2
increaseTarget JOINT1 -2 JOINT2 7 ...
setTarget JOINT3 $jointvalue3 JOINT4 (2 * $jointvalue3)
wait 0.08
.
.
.
```

As seen above, joint angle values can either be numbers or be parameterized as `$<varname>`, where `<varname>` is a variable value that can be loaded after being learned (how parameters are learned is described in Chapter 4). Values for skills and other configurable variables are read in and loaded at runtime from parameter files.

8.2.7 Kicking

The UT Austin Villa agent's kick engine works as follows. First, a kick and target to kick the ball toward is selected, and the agent approaches the ball. Once close enough to the ball, the agent shifts its weight onto the support foot and executes one of two types of kicks: fixed posed keyframe based or inverse kinematics based. The following subsections provide further details of the kick engine's components.

Ball Approach

When approaching the ball before a kick, the agent uses the *Walk_ApproachToKick* walk parameter set (mentioned in Section 4.1.2) of the walk engine (discussed in Section 8.2.5) to stop within a small bounding box of a target point while guaranteeing that the agent does not overshoot that target. With this walk approach, the agent is able to successfully approach and kick a ball without thrashing around or running into the ball.

As the agent approaches the ball target walk velocities in the X and Y directions are updated based on the following equation:

$$
\begin{aligned}
& \text{sqrt}(2 * \textsc{maxDecel}[\text{X},\text{Y}] \\
desired[X,Y]Vel = \; & *(distToBall[X,Y] > 2 * \textsc{buffer} \; ? \\
& distToBall[X,Y] : distToBall[X,Y] - \textsc{buffer}))
\end{aligned}
$$

The values for MAXDECEL[X,Y] and BUFFER are optimized using the CMA-ES [57] algorithm over a task where the robot walks up to the ball to a position from which it can kick the ball as described in Appendix A.

Kick Choice

As the agent approaches the ball, it must decide which type of kick to attempt—usually a function of whether or not the agent thinks it has time to execute the kick based on the positions of opponents as described in Appendix D.2.1, where to kick the ball (discussed in Appendix D.2.2), and whether to use the left or right foot. For each kick there is a target offset position relative to the ball that the agent wants to execute the kick from, and choosing between kicking with the left or right foot reduces to choosing the kick with the target offset position that has the lowest cost

for the agent to move to. The walk engine calculates the cost of moving to each target offset position from the ball through the following variables and formula:

$$
\begin{aligned}
distCost &= |\text{agentPosition} - \text{targetOffsetPosition}| \,/m \\
turnCost &= \frac{|\text{agentOrientation} - \text{targetOrientation}|}{360^\circ} \\
ballPenalty &= \begin{cases} .5 & \text{if ball is in path to target offset} \\ 0 & \text{otherwise} \end{cases} \\
kickCost &= \text{distCost} + \text{turnCost} + \text{ballPenalty}
\end{aligned}
$$

Kick Skills

Kicking motions for each kick are specified by the the skill description language presented in Section 8.2.6. There are two different types of kick skills: fixed pose keyframe based and inverse kinematics based.

Fixed pose keyframe kicks consist of a sequence of body positions, defined by different joint angle positions, which the agent proceeds through in order to kick the ball. For each of these the agent first places its support (non-kicking) leg near the ball and shift its weight to the support leg. Next it lifts its kicking leg, and pulls it backward, before finally swinging its kicking leg forward to strike the ball.

A weakness of the fixed pose keyframe kicks is that they require very precise positioning relative to the ball in order for them to be executed. An alternative to fixed pose keyframes is to define a path relative to the ball that the robot's foot should follow during a kick, and then use inverse kinematics to move the foot along this path. The main advantage gained through such an approach is that a kick is able to adapt to the position of the ball and thus does not require as precise positioning by an agent to line up the kick.

For inverse kinematics based kicks[29] the skill description language is extended to allow the specification of Cartesian coordinate waypoints for the kicking foot relative to the ball. These control points are used to compute a smooth 3D curve for the foot to move through. A Cubic Hermite Spline formulation is used to interpolate the control points because Hermite Splines yield curves with C^1 continuity which pass through all control points [17]. The time offset from the start of the kick is normalized to the range $[0-1]$ (0 is the start of the kick; 1 is the end), and the normalized offset is used to sample the Hermite Spline. Inverse kinematics calculations—computed through OpenRAVE's [42] analytic inverse kinematics solver—are used to compute the necessary joint angles for the foot to follow the trajectory of the curve. The skill description language also defines the Euler angles (roll, pitch, and yaw) of the foot at each control point. These angles are linearly interpolated.

Figure 8.5 shows the relative waypoints for an example inverse kinematics based kick, while Figure 8.6 provides a flow diagram of the necessary steps taken by an agent during the process of approaching the ball and executing an inverse kinematics based kick.

In addition to forward kicks, the UT Austin Villa team has developed inverse kinematics based kicks that allow for kicking the ball at 45° and 90° angles either outward or inward, depending on which leg is used. The team has also created directional kicks which assume that the ball is to the side of or behind one of the legs. See Figure 8.7 for a diagram of these kicks.

[29] Thanks to Adrian Lopez-Mobilia and Nicolae Ştiurcă for originally implementing inverse kinematics based kicks.

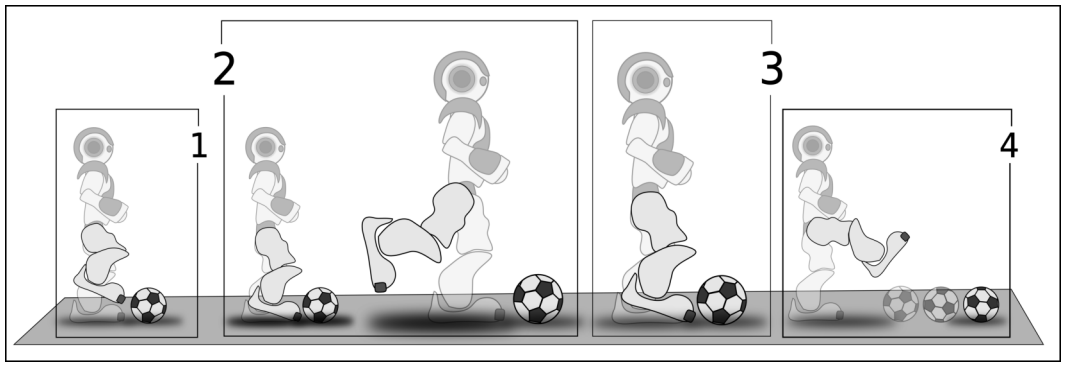


Figure 8.5: Waypoints relative to the ball that define the path of the foot for an inverse kinematics based kick. (1) Lift leg to center behind ball. (2) Pull leg back from ball. (3) Bring leg back to position of ball. (4) Kick through ball.

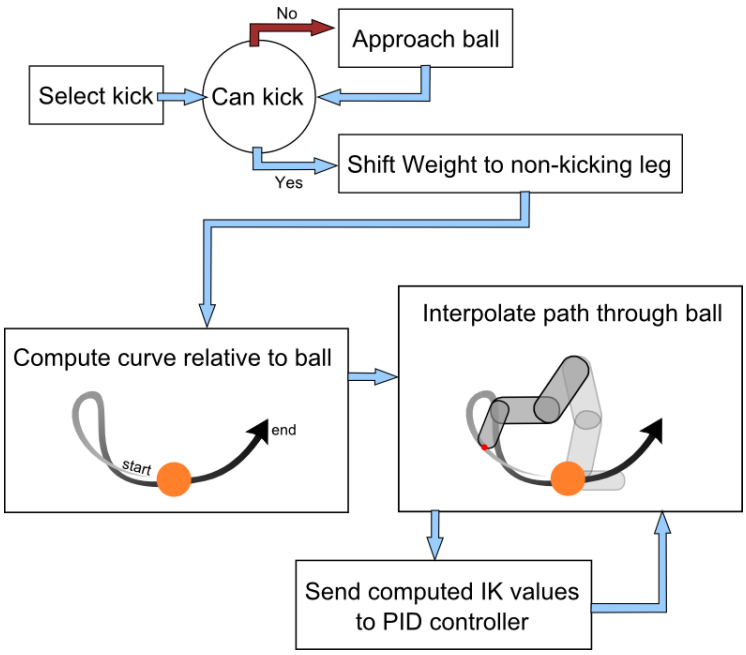


Figure 8.6: Flow diagram of the agent deciding when to kick the ball and how to interpolate the curve created relative to the ball when executing an inverse kinematics based kick. At each time step during the kick, the kick engine interpolates the control (way-) points defined in skill description language to produce a target pose for the foot in Cartesian space. Finally, an IK solver computes the necessary joint angles of the kicking leg, and these angles are fed to the joint PID controllers.

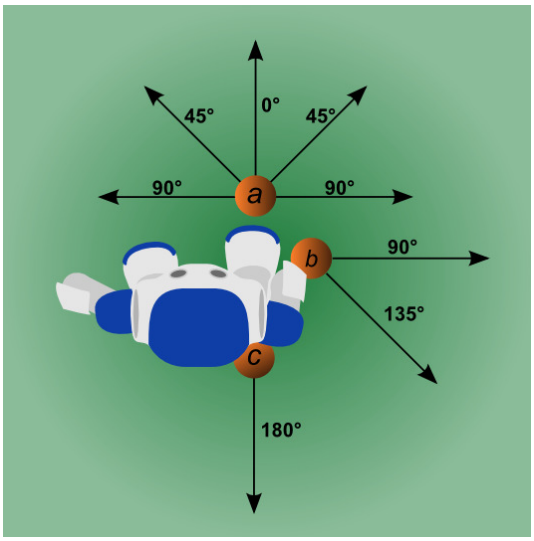


Figure 8.7: Using different directional inverse kinematics based kicks, the agent can dynamically kick the ball in varied directions with respect to the placement of the ball at *a*, *b*, and *c*.

8.3 Code Release

It is difficult for new RoboCup 3D simulation teams to be competitive with veteran teams as the complexity of the the RoboCup 3D simulation environment results in an often higher than expected barrier of entry for new teams wishing to join the league. With the desire of providing new teams to the league a good starting point, as well as offering a foundational platform for conducting research in the RoboCup 3D simulation domain, UT Austin Villa has released the base code for its agent team.

The remainder of this section is organized as follows. Section 8.3.1 provides an overview of the code release and what it includes. Section 8.3.2 highlights the optimization task infrastructure included with the code release, and Section 8.3.3

references other code releases.

8.3.1 Code Release Overview

The UT Austin Villa base code release, written in C++ and hosted on GitHub,[30] is based off of the 2015 UT Austin Villa agent. A key consideration when releasing the team's code is what components should and should not be released. A complete full release of the team's code could be detrimental to the RoboCup 3D simulation community if it performs too strongly. In the RoboCup 2D soccer simulation domain the former champion Helios team released the Agent2D code base [13] that allowed for teams to be competitive by just typing `make` and running the code as is. Close to 90% of the teams in the 2D league now use Agent2D as their base effectively killing off their original code bases and resulting in many similar teams. In order to avoid a similar scenario in the 3D league certain parts of the team's code have been stripped out. Specifically all high level strategy, some optimized long kicks [41], and optimized fast walk parameters for the walk engine [103] have been removed from the code release. Despite the removal of these items, which are described in detail in this document as well as other research publications [116, 117, 103, 102, 41, 111, 106], we believe it should not be too difficult for someone to still use the code release as a base, and develop their own optimized skills—we provide an example of how to optimize skills with the release—and strategy, to produce a competitive team.

The following features are included in the release:

- Omnidirectional walk engine based on a double inverted pendulum model (Section 8.2.5)
- A skill description language for specifying parameterized skills/behaviors (Sec-

[30] UT Austin Villa code release: https://github.com/LARG/utaustinvilla3d

tion 8.2.6)

- Getup (recovering after having fallen over) behaviors for all agent types (Section 8.2.4)
- A couple basic skills for kicking one of which uses inverse kinematics (Section 8.2.7)
- Sample demo dribble and kick behaviors for scoring a goal
- World model (Section 8.2.1) and particle filter for localization (Section 8.2.2)
- Kalman filter for tracking objects (Appendix D.3.2)
- All necessary parsing code for sending/receiving messages from/to the server
- Code for drawing objects in the RoboViz [154] monitor
- Communication system previously provided for drop-in player challenges[31]
- Example behaviors/tasks for optimizing a kick and forward walk (Section 8.3.2)
- Support for Gazebo RoboCup 3D simulation plugin

What is not included in the release:

- The team's complete set of skills such as long kicks [41] and goalie dives (Appendix D.3.3)
- Optimized parameters for behaviors such as the team's fastest walks (slow and stable walk engine parameters are included, as well as optimized parameters for positioning/dribbling [103] and approaching the ball to kick [105])
- High level strategy including formations and role assignment (Chapter 7 and Appendix D)

[31] http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/2015_dropin_challenge/

Figure 8.8 shows an image of the default demo behavior included with the code release in which agents kick the ball back and forth in a circle.

Figure 8.8: Default demo code release behavior where agents kick the ball back and forth in a circle.

8.3.2 Optimization Task Infrastructure

A considerable amount of the UT Austin Villa team's efforts in preparing for RoboCup competitions has been in the area of skill optimization and optimizing parameters for walks and kicks. Example agents for optimizing both a kick and forward walk are provided with the code release. Optimization agents perform some task (such as kicking a ball) and then determine how well they did at the task (such as how far they kicked the ball) which is known as the agent's *fitness* for the task. Optimization agents are able to adjust the values of parameterized skills at runtime by loading in different parameter files as mentioned in Section 8.2.6, thus allowing

the agents to easily try out and evaluate different sets of parameter values for a skill. After evaluating itself on how well it did at a task, an optimization agent writes its *fitness* for the task to an output file.

Optimization agents can be combined with machine learning algorithms to optimize and tune skill parameters for maximum *fitness* on a task. During optimization, agents try out different parameter values from loaded parameter files written by a machine learning algorithm, and then the agents write out their *fitness* values indicating how well they performed with those parameters so that the machine learning algorithm can attempt to adjust the parameters to produce higher *fitness* values. When performing an optimization task, agents are able to change the world as needed—such as move themselves and the ball around—by sending special training command parser commands[32] to the server.

8.3.3 Other Code Releases

There have been several previous agent code releases by members of the RoboCup 3D simulation community. These include releases by magmaOffenburg[33] (Java 2014), libbats[34] (C++ 2013), Nexus[35] (C++ 2011), and TinMan[36] (.NET 2010). The UT Austin Villa code release (C++ 2016) expands on these previous code releases in a number of ways. First the UT Austin Villa code release offers a proven base having won the RoboCup 3D simulation competition six out of the past seven years. Second the release provides an infrastructure for carrying out optimization and machine learning tasks, and third the code is up to date to work with the most recent version

[32] http://simspark.sourceforge.net/wiki/index.php/Network_Protocol#Command_Messages_from_Coach.2FTrainer

[33] http://robocup.hs-offenburg.de/uploads/media/magmaOffenburg3D-2014Release.tar.gz

[34] https://github.com/sgvandijk/libbats

[35] http://nexus.um.ac.ir/index.php/downloads/base-code

[36] https://github.com/drewnoakes/tin-man

of the RoboCup 3D simulator (rcssserver3d 0.7.1).

8.4 Summary and Discussion

This chapter presented the University of Texas at Austin's RoboCup 3D simulation team UT Austin Villa—a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years. The UT Austin Villa RoboCup 3D simulation team's base code release, which was awarded second place for the HARTING Open Source Prize at the 2016 RoboCup competition, provides a fully functioning agent and good starting point for new teams to the RoboCup 3D simulation league. At the 2016 RoboCup competition one team used the code release as the base of their team (KgpKubs), and at the 2017 RoboCup competition six teams used the code release as the base of their teams (AIUT3D, HfutEngine, KgpKubs, Miracle3D, Nexus3D, RIC-AASTMT). Additionally the code release offers a foundational platform for conducting research in multiple areas including robotics, multiagent systems, and machine learning. We hope that the code base may both inspire other researchers to join the RoboCup community, as well as facilitate non-RoboCup competition research activities akin to the reinforcement learning benchmark keepaway task in the RoboCup 2D simulation domain [157].

Recent and ongoing work within the RoboCup community is the development of a plugin[37] for the Gazebo[38] [84] robotics simulator to support agents created for the current RoboCup 3D simulation league simulator (SimSpark). The UT Austin Villa code release has been tested with this plugin and provides an agent that can walk in the Gazebo environment.

A link to the UT Austin Villa 3D simulation code release, as well as additional

[37] https://bitbucket.org/osrf/robocup3d
[38] http://gazebosim.org/

information about the UT Austin Villa agent, can be found on the UT Austin Villa 3D simulation team's homepage.[39]

[39] http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/

Chapter 9

Related Work

This chapter discusses related work in different research areas relevant to the work presented in this document. As there is far too much related work to be comprehensive, and much work has already been cited throughout this dissertation, we primarily focus on additional work that is most closely related to our own.

Related work is organized into different sections as follows. Section 9.1 discusses work related to robot skill learning, Section 9.2 examines other work using and related to layered learning, and Section 9.3 focuses on related work in the area of movement coordination. Section 9.4 summarizes.

9.1 Robot Skill Learning

Skill learning for robots is an active area of research [139]. As described in Section 2.1, our approach to skill learning uses the CMA-ES [57] derivative-free stochastic optimization algorithm algorithm to perform direct policy search (finding good parameters for a parameterized policy) within the context of model-free reinforcement learning.

There are a multitude of other model-free (not modeling the environment) policy search algorithms used within robotics. One class of model-free algorithms is gradient descent methods. Some examples of model-free gradient descent methods include both REINFORCE [177] algorithms that estimate the gradient of a policy to follow, and algorithms that follow the natural gradient such as episodic Natural Actor Critic (eNAC) [141]). Another class of model-free algorithms is expectation maximization based algorithms including the Policy Learning by Weighting Exploration with the Returns (PoWER) [83] algorithm. Other model-free algorithms use information-theoretic approaches when learning such as Relative Entropy Policy Search (REPS) [140]. REPS seeks to bound the loss of policy information between policy updates by limiting the KullbackLeibler (KL) divergence between observed data and the next policy. There are also model-free policy search algorithms, such as Path Policy Improvements with Path Integral (PI^2) [166]), that use statistical inference and a path integral approach to learning.

In addition to model-free policy search algorithms, model-based policy search algorithms that model the environment are also used for robot skill learning. One example of a model-based algorithm is Probabilistic Inference for Learning Control (PILCO) [39] which uses Bayesian inference to evaluate policies. Model-based policy search algorithms are generally more sample efficient than model-free approaches as they can use their learned models to simulate the results of taking actions. Model-based approaches may require much more computation than model-free approaches, however, and model-based approaches can also suffer from model bias.

Within the context of other policy search algorithms CMA-ES can be thought of as a form of natural gradient descent [10, 135]. Additionally, recent work by Abdolmaleki et al. has shown a promising information-theoretic extension to CMA-

ES called Trust-Region Covariance Matrix Adaptation Evolution Strategy (TR-CMA-ES) [4]. For good surveys on the use of reinforcement learning and policy search within robotics see [82] and [40] respectively.

We choose to use CMA-ES over other policy search algorithms as it has provided good results in the RoboCup 3D simulation domain when compared to other learning algorithms [172], and also because CMA-ES allows us to leverage our computing cluster's extensive computational resources by utilizing data from running hundreds of simulations in parallel. Our use of CMA-ES as the optimization algorithm allows for a high degree of parallelization during learning, and recent work by Salimans et al. has corroborated the success and scalability of evolution strategies when applied to reinforcement learning tasks [147].

An important consideration when learning parameterized skills for robots is how to parameterize the policies controlling their movements. A popular representation for learning motions is Dynamic Movement Primitives [148] which represents motion primitives as nonlinear dynamical systems. Kimura et al. have modeled walking motions for quadruped robots with central pattern generators (CPGs) [80]. Johnson and Ballard have explored efficient and sparse codes for representing the inverse dynamics of walking motion [72, 73]. Another policy representation that necessitates mentioning is deep neural networks as deep learning [22] has shown to be very successful in robot skill learning tasks. Some examples of deep learning include Levine et al. using deep convolutional neural networks to learn robot manipulation tasks [93], Levine and Koltun using Guided Policy Search [94] to learn controllers for planar simulated 3D humanoid running, and Schulman et al. using Trust Region Policy Optimization (TRPO) [149] to learn simulated humanoid walking gaits. In our work, we use a skill description language described in Section 8.2.6 for defining

motion that divides movement into a series of fixed poses. Using fixed poses provides us an intuitive representation for creating initial skills to seed optimizations with.

Another consideration when learning skills for robots is what fitness or scoring function should be used when evaluating different candidate sets of parameters. Lehman et al. have found that rewarding novelty or diversity in candidate policies for a bipedal robot locomotion controller can produce better results than directly optimizing for walking speed [91]. In our own work we have experimented with dynamically changing fitness functions to produce walks that can move equally fast in all directions [112]. We have also explored adaptively changing our fitness function when performing the goToTarget obstacle course walk optimization task (detailed in Appendix A.1.3) so as to learn walks that are better for playing soccer [109].

A current focus in the space of skill learning for robots is that of directly learning on physical robots, and employing techniques such as learning by demonstration [18, 134] and real-time online learning algorithms like TEXPLORE [61], for which sample complexity is an important consideration. While there exist learning algorithms that are extremely sample efficient, such as PILCO [39], we are not overly concerned with sample efficiency due to the relatively low cost of samples while learning in simulation. Although not a focus of this thesis, there is work to bridge the gap between learning in simulation and on physical robots [46, 54, 36, 85].

Other work in robot learning has focused on learning parametrized skills that generalize to different tasks presented to a robot [87, 37], and includes work by Abdolmaleki et al. in the RoboCup 3D simulation domain for learning directional walks [2, 3] and variable distance kicks [5]. The work in this thesis differs in than it is concerned with learning multiple skills that can *work well together* and be combined to perform complex behaviors.

9.2 Layered Learning

Within RoboCup soccer domains there has been previous work in using layered learning approaches to learn complex agent behaviors. Stone used layered learning to train three behaviors for agents in the RoboCup 2D simulation domain and specified an additional two that could be learned as well [155]. Gustafson et al. used two layers of learning when applying genetic programming to the keepaway subtask within the RoboCup 2D simulation domain [52]. Whiteson and Stone later introduced concurrent layered learning within the same keepaway domain during which four layers were learned. Cherubini et al. used layered learning for teaching AIBO robots soccer skills that included six behaviors [31]. Leottau et al. explored applying layered learning strategies to learn a ball dribbling task [92]. Layered learning has also been applied to non-RoboCup domains such as Boolean logic [66], non-playable characters in video games [127], and concept synthesis in road traffic simulations [132]. To the best of our knowledge our overlapping layered learning approach applied to robot soccer, containing 19 learned behaviors as discussed in Section 4.1, has more than three times the behaviors of any previous layered learning systems.

Work by Mondesire has discussed the concept of learned layers overlapping, and focuses on a concern of information needed to perform a subtask being lost or forgotten as it is replaced during the learning of a task in a subsequent layer [128]. Our work differs in that we are not concerned with the performance of individual subtasks in isolation, but instead are interested in maximizing the performance of subtasks when they are combined.

While our implementation of overlapping layered learning uses CMA-ES for learning the component skills, its hierarchical nature also bears some resemblance to,

and shares some motivation with, classic approaches to hierarchical reinforcement learning for learning complex behaviors. Most hierarchical reinforcement learning approaches use gated behaviors: a gating function decides which among a collection of behaviors should be executed, with each behavior mapping potential environment states to low-level actions [76]. In these approaches, the behaviors and gating function are all control tasks with similar inputs and actions (sometimes abstracted). Layered learning, on the other hand, allows for conceptually different tasks, such as a soccer player evaluating the probability of a pass being successful and moving to get open for a pass [176], at the different layers.

One widely used approach to hierarchical reinforcement learning is the MAXQ algorithm [43]. The MAXQ algorithm learns at all levels of a hierarchy simultaneously. MAXQ converges to a recursively optimal policy in which learned subtasks are locally optimal as opposed to being hierarchically optimal—the learned subtasks' policies are not necessarily optimal when taking into account transitions to and from other subtasks. Also, unlike MAXQ, layered learning allows for the flexibility of using different machine learning algorithms at each level of the hierarchy.

Another popular approach to hierarchical reinforcement learning is the options framework [160]. Options, or temporally extended actions, can be thought of as learned policies with initiation and termination conditions used to complete subtasks. Layers in our behavior hierarchy can be seen as options in the sense that they are policies that run for limited periods of time within the overall behavior. In the context of robot soccer one could learn two separate options for both walking and kicking a ball, and—after finding out that the robot is unable to transition from the walk option to the kick option without falling—then learn a third option as a bridge to transition between walking and kicking that stops and stabilizes the robot

before it kicks the ball. However, it could be inefficient and slow to need to execute three option behaviors to walk to and kick the ball. Instead, it may be possible to learn a better policy through overlapping layered learning and CILB (described in Section 3.3) that is more efficient and faster to execute: a policy consisting of a new behavior that combines the previously learned walking and kicking subtask behaviors without needing to stop and stabilize the robot before kicking the ball.

Progressive neural networks [146] follow some of the concepts of layered learning. These networks have been successfully used to learn policies for a series of related reinforcement learning tasks. Progressive neural networks are sequentially trained on individual tasks, with only the weights for a single column of the network being learned for the current task the network is being trained on. After each task is learned the weights of the neural network are frozen, and a new column with open weights is added to the network before learning the next task. The outputs from the layers of the previously learned task's network column is used as inputs to the layers of the current task's network column that is being learned. The architecture of progressive neural networks—where the output of a previously learned task is used as input to the next task being learned—is similar in spirit to layered learning. The architecture differs from overlapping layered learning, however, in that learned weights of the network are never unfrozen and relearned. Furthermore, progressive neural networks leverage task similarities to learn successive tasks through transfer learning [163], in which they focus on learning policies for tasks that are performed in isolation from each other, where as overlapping layered learning is better suited for developing subtasks that work well together and can smoothly transition between each other.

9.3 Movement Coordination

Research within the space of movement coordination spans multiple topics including role assignment (deciding which agent moves to which position or role) [29, 123, 70], path planning (paths agents take to assigned positions) [121, 151, 90], and collision avoidance (how to avoid agents colliding) [64, 144, 173]. Our work has primarily focused on using graph theoretic methods to tackle the role assignment problem. For a good review of graph theoretic multiagent coordination methods see [122].

Work very related to ours [169] assigns interchangeable robots to goal positions where robots have non-point masses and exist in environments containing obstacles. Additional work by Turpin et al. [171] suggests using the Hungarian algorithm to minimize the sum of distances raised to large powers as a proxy for MMDR (discussed in Section 5.3.1), however such an approach is not guaranteed to return the same result as MMDR and minimize the makespan. We believe the above work could be augmented with the MMD+MSD^2 role assignment algorithm (discussed in Section 5.3.2) to allow for minimizing the makespan of robots traveling in cluttered environments in $O(n^3)$ time.

As an application of role assignment Chopra and Egerstedt have created a robot music wall, a spatio-temporal constrained version of the vehicle routing problem [168], where robots travel around to target positions on a wall and play musical notes at specific times by plucking strings at the target positions [34]. They approach this task as an assignment problem, with the objective of finding an assignment that minimizes the total distance all robots travel, and also consider robot connectivity [32] and heterogeneity [33].

Alonso-Mora et al. consider the role assignment problem when using mobile robots as pixels to create animated images [15]. They minimize the sum of distances

squared—the MSD^2 role assignment function in Section 5.4—when assigning roles which avoids collisions but does not minimize the makespan.

Macdonald [118] examines a multi-robot assignment and formation control problem in which robots use a decentralized algorithm to decide on a pose (translation and rotation) for a given formation, and then assign themselves to unique positions in the formation. Macdonald also uses the MSD^2 role assignment function when assigning robots to positions which does not minimize the makespan.

Ma and Koenig focus on the TAPF problem, i.e., combined target assignment and path-finding, where there are teams of agents, and only members of specific teams can be assigned to each target [100]. They too have the goals of both minimizing the makespan and avoiding collisions. Their approach uses a min-cost max-flow algorithm [50] on a time-expanded network to assign agents to targets within specific teams, and then uses conflict-based search [151] to resolve collisions among agents in different teams.

Akella explores the problem of assigning interchangeable robots to a goal formation, with the twist that the goal formation can be scaled or translated [9]. He formulates role assignment as a linear bottleneck assignment problem (LBAP) which minimizes the maximum distance—but not recursively—that any robot must travel. Such an assignment could be produced by running Algorithm 4 to compute $MMD+MSD^2$, but first setting all edge weights to the same value before running the Hungarian algorithm on line 3 of Algorithm 4. While such as assignment minimizes the makespan, it neither avoids collisions or is dynamically consistent. Using a SCRAM role assignment function, instead of an assignment function that only minimizes the maximum distance any robot travels, would allow for assignments that avoid collisions.

There exists previous work on role assignment in RoboCup soccer domains. Stone and Veloso define an order of precedence or importance to positions, and a player is only allowed to switch to a new position if that position is more important than the player's current position [158]. Reis et al. allow for two players to switch their assigned positions when doing so improves a global team utility metric [143]. Lau et al. give a ranked priority to each target position on the field, and then iteratively assign the closest robot to each position in order from highest to lowest priority target [88]. Chen and Chen use a similar greedy approach to assigning robots to different priority target positions, and robots bid on positions based on their cost or path distance to reach the targets [30]. Abeyruwan et al. attempt to learn a role assignment function through the use of general value functions [6]. None of this previous work on role assignment in RoboCup soccer domains has focused on collision avoidance or formation completion time.

As an extension to SCRAM role assignment, Jaishy et al. have created the Breakdown Agent Replacement (BAR) algorithm for SCRAM [69, 67, 68]. BAR focuses on situations where a subset of agents may break down and are no longer able to move.

9.4 Summary

This chapter discussed related work in different research areas relevant to the work presented in this document. As there is far too much related work to be comprehensive, we primarily focused on work that is most closely related to our own in the areas of robot skill learning, layered learning, and movement coordination.

Chapter 10

Conclusion and Future Work

This chapter summarizes the ideas and contributions of this thesis. After summarizing, the chapter provides directions for future work.

Robots are rapidly becoming more prevalent in both industrial and domestic settings [1]. Such an increase in the number of robots, and the likely subsequent decrease in the ratio of people currently trained to directly control the robots, will necessitate more robots to be able to act autonomously. In addition to the heightened importance of robot autonomy, larger numbers of robots present together in the same environment will provide new challenges and opportunities for multirobot collaboration and coordination. In a step toward addressing these challenges and opportunities, this thesis focuses on two topics we believe are most pertinent for the development of autonomous robots and multirobot collaborative behavior: skill learning and movement coordination. An underlying objective of this research is to develop techniques and methodologies that allow autonomous robots to robustly interact with their environment (through *skill learning*) and with each other (though *movement coordination*) in order to perform tasks and accomplish goals asked of

them.

First, robots must acquire the necessary skills to autonomously perform tasks in their environment. While previous work in robot skill learning has concentrated on learning individual skills in isolation, we instead focus on developing learning methodologies and designing optimization tasks to produce skills that *work well together*. In our work, multiple lower level skills are incrementally learned and combined with each other to develop richer higher level skills. Overlapping layered learning hierarchical machine learning paradigms (introduced in Chapter 3) are utilized during skill learning. Overlapping layered learning allows for learning certain skill behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap.

After robots have developed higher level skills, and can perform single robot behaviors, autonomous robots present in the same environment together need to coordinate their behavior and movement to efficiently complete (possibly multi-robot) tasks. As such, we develop algorithms to coordinate the movement and efforts of multiple robots working together to quickly complete tasks. These algorithms, known as Scalable Collision-avoiding Role Assignment with Minimal-makespan (SCRAM) role assignment algorithms (presented in Chapters 5 and 6), prioritize *minimizing the makespan*, or time for all robots to complete a task, while also attempting to avoid interference and collisions among the robots. Minimizing the makespan is a decisive factor in performance when robots are moving to target positions to complete a shared task where all robots must be in place before the task can be completed and/or started.

The work in this thesis is implemented and evaluated in the RoboCup 3D simulation soccer domain (applications of overlapping layered learning and SCRAM

role assignment are evaluated in Chapters 4 and 7 respectively), and has been a key component of the University of Texas at Austin's RoboCup 3D simulation team UT Austin Villa winning the RoboCup 3D simulation league world championship six out of the past seven years. As a contribution of this thesis, a public base code release of the UT Austin Villa agent (presented in Chapter 8) provides a testbed for future research in multirobot systems.

The remainder of this chapter is organized as follows. Section 10.1 reviews each contribution of the thesis. Possible directions for future work are discussed in Section 10.2, and Section 10.3 concludes.

10.1 Contributions

This thesis has provided the following contributions first presented in Section 1.2:

1. **Methodologies for learning complex robot skills in simulation.** This thesis presented new paradigms for constructing and learning complex robot behaviors through the introduction and use of *overlapping layered learning* hierarchical machine learning paradigms in Chapter 3. Overlapping layered learning presents ways of combining learning of different behavior layers that extend the traditional sequential layered learning methodology [155] through the use of overlapping or shared parameter sets across behavior layers.

2. **Development and analysis of multirobot role assignment functions.** This thesis presented and analyzed different role assignment functions for assigning robots to role positions in Chapter 5. Properties of role assignment functions analyzed include total distance traveled by all robots, makespan completion time, dynamic consistency, collision avoidance, and standard deviation

of the distances traveled by robots.

3. **Novel algorithms for multirobot movement coordination.** This thesis introduced new robot movement coordination algorithms that focus on important considerations of formation completion time, collision avoidance, and scalability. Specifically scalable polynomial time role assignment algorithms known as *SCRAM* that avoid collisions among robots and minimize the makespan, or time for robots to complete a formation, were presented in Chapters 5 and 6.

4. **Complete autonomous robot soccer playing agent.** The UT Austin Villa RoboCup 3D simulation league team, a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years, was presented in Chapter 8. This agent incorporates the ideas and algorithms presented in this thesis, thus serving as a proof of concept of them, and a public base code release of the agent provides a testbed for future research in multirobot systems.

5. **Detailed empirical evaluation of presented learning methodologies and coordination algorithms.** This thesis provided detailed empirical evaluations of the presented overlapping layered learning methodologies in Chapter 4, and SCRAM role assignment movement coordination algorithms in Chapter 7. Results from within the RoboCup 3D simulation competition were analyzed as well as controlled experiments external to the competition.

10.2 Future Work

This section presents directions for future work. First, future work on skill learning is discussed in Section 10.2.1. Next, Section 10.2.2 presents possible extensions to SCRAM role assignment. Ideas for future research using the UT Austin Villa RoboCup 3D simulation team and code base are provided in Section 10.2.3, and Section 10.2.4 suggests additional domains outside of robot soccer to which the work in this dissertation may be applied.

10.2.1 Skill Learning

This section presents four ideas for future work in the area of skill learning. First, we discuss the idea of reshaping surrogate optimization tasks for target optimization tasks that are intractable to directly optimize over. This topic is useful for learning complex skills and behaviors as it is often the case that increased task complexity decreases the tractability of learning directly on the task. Second, we consider using layered learning methodologies to learn generalizable skills (skills that automatically adapt to different variations or contexts of a task). Learning a generalizable skill for similar tasks is more efficient than learning separate skills for multiple similar tasks. Third, we discuss the idea of automating overlapping layered learning methodologies by algorithmically selecting sets of overlapping parameters and layers to optimize. Finally, we suggest applying layered learning methodologies to additional robot environments in order to test out and validate the methodologies' applicability and effectiveness. Each idea for future work aims to improve on the knowledge and understanding of how to learn complex multilayered skills.

Simultaneous Learning and Reshaping of Surrogate Optimization Tasks

In many learning and optimization tasks, the sample cost of performing the task is prohibitively expensive or time consuming, and attempting to directly employ a learning algorithm on the task could quickly become intractable. For this reason, learning is instead often performed on a less expensive task that is believed to be a reasonable approximation or a surrogate of the actual target task—this approach to learning is known as surrogate-assisted optimization [71].

One way to perform surrogate-assisted optimization is to learn a mapping from the value of some surrogate function, that is more tractable computationally, to the amount of success in the objective task, based on a small empirical sample. This approach, albeit enticing, is dangerous, because a global mapping may not exist — the mapping may change as we transition from one part of the parameter space to another. For example, let us assume we are trying to optimize the parameter set of an autonomous vehicle based on its performance in several racetracks. We can estimate how the performance on the racetracks correlates to the performance on a larger-scale problem, for instance, driving in a small town. However, as parameters change the behavior of the vehicle changes, and we will inevitably see different mappings from racetrack performance to larger-scale performance as the autonomous vehicle changes its policy. For this reason we propose tackling the challenging open problem of *simultaneously performing learning on an approximation or surrogate of the true target task, while at the same time shaping the task used for learning to be a better representation of the true target task.*

Our[40] early initial work on this problem [109] has targeted learning omnidirectional walks that allow simulated robots to playing soccer well in the RoboCup

[40]This is joint work with Elad Liebman.

3D simulation environment. Optimizing the parameter set that governs the walk has been one of the key challenges in this domain [103]. Ideally, we would want to evaluate any parameter set directly on full 11 versus 11 gameplay. However, playing a full game to evaluate each parameter set is not computationally tractable. For this reason, in the past we have trained an agent directly on an obstacle course optimization task (discussed in Appendix A.1.3), comprised of 11 different activities such as walking straight, in curves, and quickly stopping.[41] It has been empirically shown that doing well on the obstacle course is correlated with gameplay success. However, other approaches, such as learning from infant walk trajectories [8], and learning to optimize based on trajectories observed in real gameplay, have proven less successful than the obstacle course.

What is it then about the obstacle course that makes it effective? More exactly, which of the 11 different walking activities comprising the obstacle course are most significant in learning a successful walk, and could it be that weighting the tasks differently would result in a better learned walk? More interestingly, is it possible that in different stages of the learning, different weighting schemes would result in a better learning rate, and a better final walk? Finally, is it possible to adjust or evolve the optimization task (in our case modify the walking trajectories within the obstacle course) during learning to improve performance? While we have some promising preliminary work on using a genetic algorithm to evolve different walking task trajectories during learning [110], these are all still open questions.

One avenue to explore in this line of research is a surrogate-assisted version of CMA-ES, ACM-ES [97], which uses rank-based Support Vector Machines to learn the surrogate model. Closely related to the ides of adapting the surrogate task is a

[41] A video of the optimization task can be found at
http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/walk.html#goToTarget

self-adaptive extension to ACM-ES, ^{s*}ACM-ES [98], which adjusts the time between changing the surrogate model—the number of generations between sampling from the true target optimization task—and the surrogate model's hyper-parameters. Additional work on ^{s*}ACM-ES has focused on using larger population sizes to better utilize the surrogate model [99].

Another possible approach to changing or adapting optimization tasks during the course of learning is that of curriculum learning [130, 131, 161] in reinforcement learning. In curriculum learning the goal is to design a sequence of tasks for an agent to train on such that final performance or learning speed of the agent is improved.

Generalizable Skills

Often when learning a specific robot skill, such as having a robot throw a ball at a fixed target, the learning process will overfit to the exact task being learned and will not be able to generalize well to similar tasks. Rather than training a robot to handle each new task it encounters, it is more efficient if generalizable skills can be learned for completing a set of similar tasks. These generalizable skills take in a context or set of parameters for a task (e.g. a new target position in the ball throwing robot example) and the robot is able to complete these task variations without necessarily having been directly trained on them. Recent work in robot skill learning has focused on learning skills that generalize well to different contexts or variations in tasks [87, 37, 134].

As layered learning focuses on learning skills that work well together to complete tasks, we hypothesize that it is possible to combine layered learning with general skill learning to develop generalizable skills which work well together with a variety of different skills. Specifically, and to validate this idea, we suggest em-

ploying layered learning methodologies during learning of a generalizable skill for kicking the ball at different targets in the RoboCup 3D simulation domain.

A first step for this work is to learn a general kick skill for kicking the ball at different targets. A good starting point for learning such a general kick skill is recent work by Abdolmaleki et al. who used contextual policy search [87] to develop both a general skill for walking in different directions at different speeds [2], and a skill for variable distance kicks [5], within the RoboCup 3D simulation domain. Another method for learning general skills is the approach of learning parameterized skills [38] which was successfully used to train a robot to throw a ball at different targets [37]. Assuming one is able to learn a general kicking skill, the final step would be to utilize overlapping layered learning methodologies to combine this kicking skill with previously learned walking skills in a similar manner to how we combined walking skills with different non-general kicking skills as discussed in Section 4.1.2.

Automated Selection of Overlapping Parameters and Layers

Currently the overlapping layered learning methodologies require a person to select which parameters to freeze and leave open during each successive layer of learning. Additionally learning of complex skills is manually segmented into different layers of learning. We propose finding ways of automating the segmentation of layers and/or the selection of parameters to leave open when using overlapping layered learning methodologies. If these selection processes can be automated it would lessen the burden and potential need for someone with expert domain knowledge when performing optimizations.

A possible starting point for the automated selection of parameters to optimize is to look into techniques from the research area of feature selection [53]. Using

Beta Process Autoregressive Hidden Markov Models (BP-AR-HMMs) might be a potential way to segment complex skills into different layers as Niekum et al. successfully used BP-AR-HMMs to segment demonstrations of skills while performing learning from demonstration with robots [134]. Lioutikov et al. have also segmented robot skills from demonstration using probabilistic inference and the Probabilistic Segmentation (ProbS) algorithm [95]. Another method for delineating skills is work by Mankowitz et al. who learn skills and where to apply them in the RoboCup 2D soccer simulation domain using the Adaptive Skills Adaptive Partitions (ASAP) framework [119]. Methods used in curriculum learning [130, 131, 161]—where the goal is to design a sequence of tasks for an agent to train on such that final performance or learning speed of the agent is improved—might also be utilized for automating the selection of learning layers.

Apply Layered Learning Methodologies to Additional Robot Environments

Layered learning methodologies have performed well in the RoboCup 3D simulation domain, and have been shown to generalize to different robot models within this same domain in Section 4.2.4. We believe that the learning methodologies are very general, and propose testing them out in robot domains outside of the RoboCup 3D simulation environment to verify that they work well there too.

For ease of testing we would like to choose a new domain that will allow us to reuse as much of our current RoboCup 3D simulation code and optimization framework as possible. The best candidate domain for reusing our current code infrastructure is the Gazebo [84][42] robot simulation environment. The company that

[42] http://gazebosim.org/

maintains Gazebo, the Open Source Robotics Foundation (OSRF),[43] is currently working on a Gazebo plugin to support RoboCup 3D simulation league agents with the eventual goal of having Gazebo replace the SimSpark simulator used by the RoboCup 3D simulation league. The Gazebo plugin[44] supports the same message communication protocol as SimSpark so that current RoboCup 3D simulation league agents will not need to be modified to run within Gazebo. Gazebo's physics model is very different from the one used in SimSpark, however, and so all skills such as walking would need to be re-optimized to work in the Gazebo environment. Gazebo's support of the RoboCup 3D simulation communication protocol, coupled with a different physics model, make it an ideal new testing environment for our layered learning methodologies.

Other potential robot simulation environments to test out our layered learning methodologies include simRobot [89],[45] Webots [124],[46] and MuJoCo [167].[47]

10.2.2 Extensions to SCRAM

This section presents four possible extensions to SCRAM role assignment. The first extension considers relaxing SCRAM's point mass approximation and allowing for robots to travel in environments containing obstacles. This extension would increase SCRAM's versatility allowing for its use in additional environments. The second extension suggests transforming SCRAM from a centralized algorithm to a distributed algorithm. Distributed algorithms are often preferable in robotic domains as they typically require less communication. A third extension looks at ways to improve the time complexity of SCRAM algorithms which is an important factor in how

[43] http://www.osrfoundation.org
[44] https://bitbucket.org/osrf/robocup3d
[45] http://www.informatik.uni-bremen.de/simrobot/index_e.htm
[46] https://www.cyberbotics.com/overview
[47] http://www.mujoco.org/

well the algorithms scale to larger numbers of agents. Finally, the fourth extension discusses the Minimum-makespan Multi-vehicle Routing Problem (MMMVRP) in which there are more target positions than agents, and we want all target positions to be visited by an agent in as little time as possible. MMMVRP is an extension of the minimal-makespan role assignment problem that SCRAM solves. Each extension seeks to improve algorithms for coordinating the movement of robots when minimizing the makespan is an objective.

Relaxation of SCRAM Point Mass Approximation

Although in its current form, SCRAM role assignment generalizes well to many realistic and real-world multiagent systems, for theoretical analysis purposes SCRAM approximates agents as being zero width point masses as discussed in Section 5.2. We would like to remove this approximation from SCRAM and also extend SCRAM to work in environments containing obstacles. These proposed extensions to SCRAM would increase its versatility allowing for its use in additional environments.

Our primary starting point for extending SCRAM is work by Turpin et al. called CAPT [169]. CAPT provides a collision-free assignment of interchangeable robots to goal positions where robots have non-point masses and exist in environments containing obstacles. We believe CAPT's role assignment algorithm can be augmented with the MMD+MSD^2 SCRAM role assignment algorithm, thereby maintaining the rest of CAPT's O(n^3) time complexity, to allow for minimizing the makespan of robots traveling in cluttered environments in polynomial time.

Distributed SCRAM Algorithms

The SCRAM algorithms presented in Section 5.3 are centralized algorithms. In robotic domains it can be preferable to have distributed algorithms, however, as

communication between robots to a centralized controller may not always be a viable option. Borrowing ideas from distributed auction [23] and market-based [96] algorithms, which have been used to compute a solution to the related *assignment problem*—minimizing the sum of distances traveled by agents—in polynomial time, could be a promising direction for the development of distributed SCRAM algorithms. It may also be possible to minimize the amount of communication needed to preserve assignments as agents move toward their assigned targets, as Nam et al. [129] have found ways to reduce the amount of global communication needed to preserve solutions to the assignment problem in scenarios with dynamically changing costs.

Better Time Complexity

A bottleneck for the time complexity of the SCRAM algorithms is the $O(n^3)$ Hungarian algorithm. Algorithms reported to be faster than the Hungarian algorithm for solving the *assignment problem*, such as the Jonker-Volgenant algorithm [75], and the dynamic Hungarian algorithm [125] for the special case when most agents have reached their targets and few distances are changing, can be explored to further speed up role assignment algorithms. In the case of the MMDR $O(n^5)$ algorithm (Algorithm 1) in Section 5.3.1, for which all edge weights are integers, scaling algorithms [49, 137] can possibly be used instead of the Hungarian algorithm to reduce the time complexity.

Minimum-makespan Multi-vehicle Routing Problem

Another possible extension to SCRAM includes role assignment problems where there are unequal numbers of agents and targets. To extend SCRAM to the case when there are m agents and n target locations, and $m > n$, is trivial. All that must

done is to add $m-n$ dummy target locations with all agents being assigned a distance of 0 to each of the dummy locations. As the Hungarian algorithm minimizes the sum of edge weights, the excess $m - n$ agents—those not in a minimum makespan matching to real targets—are assigned to dummy locations. Conversely, if there are more target locations than agents ($n > m$), and we desire agents to travel from one target to another such that every target location is eventually visited by an agent, role assignment becomes a vehicle routing problem [168]. What we are interested in computing are the routes—the series of assignments of agents to targets—such that all targets are visited by a vehicle in as little time as possible. Determining what routes to assign agents is an instance of the Minimum-makespan Multi-vehicle Routing Problem (MMMVRP) which, as it can be reduced to the traveling salesman problem, is NP-Hard. Recent initial work on MMMVPR by Turpin et al. can compute a solution in polynomial time that is no more than five times the optimal completion time [170].

10.2.3 UT Austin Villa RoboCup 3D Simulation Agent

A driving force behind the success of the UT Austin Villa RoboCup 3D simulation team has been the team's use of machine learning. In this section we outline three ways in which the team could potentially further improve its performance using machine learning. First, the team could use deep learning for skill learning. Second, team formations could be learned instead of being hand-specified. Third, instead of using a hand-designed scoring function for deciding where to kick the ball, such a scoring function could be learned. Lastly, we discuss ongoing work using our RoboCup 3D simulation code base to apply what is learned in simulation to robots in the physical world.

Deep Learning of Skills

Currently the UT Austin Villa team uses a skill description language for specifying kicking motions as described in Section 8.2.6. The skill description language is implemented as a periodic state machine with multiple key frames, where a key frame is a parameterized static pose of fixed joint positions. Ideally we might like to learn a joint position (parameter value) for each of the agents 22 joints at every simulation cycle (20 ms) so as to learn a policy over the entire range of possible poses. To optimize values for every joint position at every simulation cycle during a two second kicking motion would require learning over 2000 parameter values, and unfortunately CMA-ES does not scale well to thousands of parameters [136].

An alternative to using our skill description language is to represent the policy of a kicking motion as a deep neural network, and then use deep learning [22] to learn kicking motions. More concretely, we could pass as input to a neural network the amount of time since a kicking motion has started, and then have an output for each of the robot's joints specifying the target joint angle position to move the joint to. Before training the neural network, we would seed the network with the policy of our longest kick using supervised learning and backprop. Then, with the network initialized with a policy that mimics our longest kick's motion, we could train the neural network with the Trust Region Policy Optimization (TRPO) algorithm [149].

Formations

The UT Austin Villa team's formations are computed using Delaunay triangulation [14] based on set offset positions from the ball as described in Section 7.1. Currently these formations are just hand-specified and occasionally manually tuned using a GUI formation editor [12].

Rather than manually specifying formations, it may be possible to use machine learning to create and optimize formations. Specifically we could learn and modify formation position offset values with the CMA-ES algorithm based on the results of playing games using different formations. If there are too many formation parameters for CMA-ES to optimize, we could instead represent a formation as a neural network—with the input to the network being the position of the ball as well as potentially the positions of opponents, and output of the network being desired formation positions—and use deep learning to learn formations.

We note that there is existing work in the RoboCup 2D simulation league for learning formations, as Henn et al. used the firefly algorithm to learn formation positions for corner kicks [59].

Where to Kick

When deciding where to kick the ball, the UT Austin Villa agent samples kicking the ball to different target locations as described in Appendix D.2.2, and then assigns each location a score based on hand-designed Equation D.1. The location with the highest score is chosen as the location to kick the ball to.

Instead of using a hand-designed function, we postulate that one could learn a scoring function for kick locations using machine learning. Specifically a scoring function could be represented as a neural network—with the input to the network being the positions of the ball and agents on the field, and the output of the network being the target location to kick the ball to—where the network is trained through deep learning based on the results of kicking the ball to different field locations during games.

We note that there is existing work in the RoboCup 2D simulation league

for learning where to kick the ball, as Xiong et al. used Q-learning to learn where to pass the ball [179].

Applying What is Learned in Simulation to the Physical World

Learning in simulation has several advantages over learning on robots in the physical world. These advantages of learning in simulation include the following:

- In simulation thousands of learning trials can be run in parallel on distributed computing clusters.
- It may be possible to speed up learning trials and run them faster than real-time in simulation.
- No supervision or manual resetting of robots is required in simulation.
- Unlike in the physical world robots never break or wear out in simulation.

Given the preceding advantages of learning in simulation, it would be nice to leverage these advantages so as to be able to apply what is learned in simulation to the physical world. Unfortunately, policies learned in simulation often fail to work in the physical world due to overfitting to inaccuracies in the simulator's model of the physical world.

As a step toward applying what is learned in simulation to the real world, Farchy et al. developed a framework for robot learning in simulation called Grounded Simulation Learning (GSL) [46] using the UT Austin Villa RoboCup 3D simulation code base. GSL works by modifying—or grounding—a simulator with real world data so that the behavior of policies executed in simulation will closer match to how they will function in the real world. This grounding is performed by first learning

a forward kinematics model of a physical robot from state–action (joint position–joint torque) trajectories recorded while the robot executes a fixed policy in the real world. Next, a policy is optimized in simulation. During optimization, a grounding function embedded in the simulator uses the learned forward kinematics model of the robot to modify actions taken in simulation. The grounding function modifies these actions in simulation such that the modified actions produce the state outcomes that are estimated to result from taking the original unmodified actions in the real world.

With GSL, Farchy et al. were able to use the RoboCup 3D simulator and UT Austin Villa code base to learn a walk for a physical Nao robot that was 25% faster than the hand-coded walk used as a seed for learning. Recent work by Hanna and Stone has extended GSL with Grounded Action Transformation (GAT) [54] in which a learned inverse kinematics model of the simulator is added to the grounding function. Learning using GAT applied to GSL produced the fastest known walk on a physical Nao robot. Bridging the gap between learning in simulation and on physical robots is an active area of research, and one for which the UT Austin Villa RoboCup 3D simulation code base may continue to help enable.

10.2.4 Other Domain Applications

The learning methodologies and movement coordination algorithms presented in this dissertation are designed to be general in nature, and as such can be applied in principle to many domains outside of robot soccer. In this section we suggest some additional domains for applying overlapping layered learning and SCRAM role assignment, some of which were touched on when discussing related work in Chapter 9.

Overlapping Layered Learning

One potential area that overlapping layered learning can be applied to is that of video games. Video games often have low level skills or subtasks that can be learned and then later combined together for more complex behaviors. Such a hierarchy of skills make layered learning a natural fit for learning policies for video games, and in turn a good fit for overlapping layered learning as well assuming that skills are represented as parameterized policies.

One video game environment overlapping layered learning may be applied to is the Atari Arcade Learning Environment (ALE) [21][48] which has been a popular testbed for both neuroevolution [58] and deep reinforcement learning [126]. Another video game environment that is conducive to overlapping layered learning is the real-time-strategy game Starcraft and associated learning framework TorchCraft [162].[49] A third video game environment to apply overlapping layered learning to is Minecraft and its associated platform for AI research Malmo [74].[50] Minecraft has been used as a testbed for both curriculum learning [120] and lifelong learning [164], and as such it would be a great fit for layered learning approaches too. One other platform to test overlapping layered learning with is OpenNERO [78],[51] an open-source machine learning video game and platform for AI research and education. OpenNERO has been used to develop complex agent behaviors by learning a sequence of gradually more challenging tasks [79], and thus matches well with both curriculum learning and layered learning approaches to learning.

Another general area that overlapping layered learning may be applied to is Genetic Programming (GP) problems. In Section 9.2 we mentioned that Gustafson

[48] https://github.com/mgbellemare/Arcade-Learning-Environment
[49] https://github.com/TorchCraft/TorchCraft
[50] https://github.com/Microsoft/malmo
[51] https://github.com/nnrg/opennero/wiki

et al. used layered learning when applying GP to the keepaway subtask within the RoboCup 2D simulation domain [52]. Others have used layered learning with GP on tasks outside of robot soccer however, including Nguyen et al. who had success using layered learning GP (GPLL) to solve symbolic regression problems [62, 133].

SCRAM Role Assignment

SCRAM role assignment is useful for general problems in which agents are tasked with assuming different formations. Such problems include formation control of quadrotors [44] and formations for marching bands [63].

Another potential application of SCRAM is that of controlling the positions of robots acting as pixels to form different images. As we previously mentioned in Section 9.3, Alonso-Mora et al. consider the role assignment problem when using mobile robots as pixels to create animated images [15].

SCRAM may also be used during the process of coordinating multiple droplets in light-actuated digital microfluidic systems intended for use as lab-on-a-chip systems [101]. In such systems, droplets of chemicals are actuated on a photosensitive chip by moving projected light patterns. The goal is to move multiple droplets in parallel on a microfluidic platform without having the droplets collide with each other.

Prioritized SCRAM role assignment, presented in Chapter 6, is well suited to coverage and patrol tasks [45] as it allows for specifying a set of high priority targets (areas of importance that need to be quickly visited and/or covered) that agents will minimize the makespan when moving to. Furthermore, an agent will maintain coverage of a high priority target until another agent is near enough to also cover that target.

Vehicle routing is another application where SCRAM role assignment is directly applicable. Hanna et al. have used SCRAM role assignment in a carsharing setting to match autonomous cars to people requesting rides [55].

SCRAM role assignment may also have utility in warehouses where robots retrieve items for orders to be shipped [178]. As SCRAM minimizes the makespan, or time for all items for an order to be retrieved, it allows for faster processing of orders.

Finally, given that all experiments in this dissertation using SCRAM role assignment were carried out in simulation, it would be worthwhile to run some experiments using SCRAM on physical robots as another reference point for how the algorithms perform. A good testbed for running SCRAM on physical robots is the Robotarium [142],[52] a remotely accessible swarm robotics research platform at the Georgia Institute of Technology.

10.3 Concluding Remarks

This thesis introduces the overlapping layered learning paradigm and presents Scalable Collision-avoiding Role Assignment with Minimal-makespan (SCRAM) role assignment algorithms. The thesis also presents the University of Texas at Austin's RoboCup 3D simulation team UT Austin Villa—a successful state of the art agent having won the RoboCup 3D simulation competition six out of the past seven years—along with a public base code release of the UT Austin Villa agent that provides a testbed for future research in machine learning and multirobot systems. While the UT Austin Villa agent incorporates the ideas and algorithms presented in this thesis, thus serving as a proof of concept of them, the learning methodologies

[52] https://www.robotarium.gatech.edu/

and movement coordination algorithms are designed to be general in nature, and ought to have broad applicability to real-world problems well beyond that of robot soccer. We hope that the contributions of this thesis may play a role in the AI community's ongoing effort to develop robust autonomous robots and multirobot systems in the real world.

Appendices

Appendix A

Learned Behavior Layers

The following is a detailed description of the different behavior layers learned by the 2014 UT Austin Villa RoboCup 3D simulation team as discussed in Chapter 4. The team used an extensive layered learning approach incorporating overlapping layered learning—introduced in Chapter 3—to learn skills for the robots such as getting up, walking, and kicking. The notation used to describe the behavior layers is presented in Section 3.2. Additional details about the optimization process and training tasks used during the layered learning approach to develop three walk parameter sets needed for general walking, sprinting, and dribbling the ball are provided in Appendix A.1.

For all layers of learned behaviors the CMA-ES [57] algorithm is used as the ML algorithm (M). All input feature vector ($\vec{F}$) for learned behavior layers include the position of each of the robots' 22 joints, as well as the robots' three dimensional (x, y, z) accelerometer and gyroscope measurements. Any behavior in which a kick is learned also takes in as input the x and y position of the ball relative to the robots. The output (O) for all behavior layers are current target positions for each

of the robots' joints. A diagram of how all the layers connect with each other can be seen in Figure 4.1 within Section 4.1.

$L_{1,1}$: Getup_Front_Primitive: The robot learns a behavior to stand up when starting from lying on its front.

$T_{1,1}$: The robot is forced to fall on its front from a standing position and then attempts to get up. The robot is then given a negative return value equal to the time it remains in a fallen over state as measured over 4 seconds. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_front\_primitive}} = -\text{timeNotStanding}$$

See [104] for more details about the training task.

$M_{1,1}$: Learning was performed across 200 generations of CMA-ES with a population size of 150.

$H_{1,1}$: The learned policy consists of 9 parameters specifying a fixed series of poses for a getup motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Initial parameter values were seeded with those from a hand-coded policy.

$L_{1,2}$: Getup_Back_Primitive: The robot learns a behavior to stand up when starting from lying on its back.

$T_{1,2}$: The robot is forced to fall on its back from a standing position and then attempts to get up. The robot is then given a negative return value equal to the time it remains in a fallen over state as measured over 4 seconds.

The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_back\_primitive}} = -\text{timeNotStanding}$$

See [104] for more details about the training task.

$\boldsymbol{M_{1,2}}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{1,2}}$**:** The learned policy consists of 26 parameters specifying a fixed series of poses for a getup motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Initial parameter values were seeded with those from a hand-coded policy.

$\boldsymbol{L_{1,3}}$ **: KickOff_Touch_Primitive:** Single robot behavior where the robot learns to lightly touch the ball resulting in little ball motion.

$\boldsymbol{T_{1,3}}$**:** The robot attempts to lightly touch the ball and is given lower return values the more the ball moves. If the robot falls over, fails to touch the ball, or touches the ball more than once it is given a negative return value. The objective function used during optimization is the following where distance is in meters:

$$f_{\text{touch}} = \begin{cases} -1 & : \text{Failure} \\ 10 - \text{distBallTraveled} & : \text{Otherwise} \end{cases}$$

See [41] for full details of the training task.

$\boldsymbol{M_{1,3}}$**:** Learning was performed across 100 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{1,3}}$**:** The learned policy consists of 12 parameters specifying a fixed series of poses for a ball touch motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those from a hand-coded policy.

$\boldsymbol{L_{1,4}}$ **: KickOff_Kick_Primitive:** Single robot kick behavior where the robot learns a kick that scores on a kickoff from a motionless ball.

$\boldsymbol{T_{1,4}}$**:** The robot attempts to kick the ball on a kickoff directly into the opponent's goal. The robot is given higher return values for both kicks that travel closer to the opponent's goal, and for kicks that travel farther distances while remaining above the height of opponent robots. If the robot fails to kick the ball it is given a negative return value, and if it kicks the ball out of bounds—misses the goal—it receives a return value of 0. The objective function used during optimization is the following where all distances are in meters:

$$f_{\text{kick\_kickoff}} = \begin{cases} -1 & : \text{Failure} \\ 0 & : \text{Missed goal} \\ 100 + \text{distBallForward} + 2 * \text{distBallInAir} & : \text{Otherwise} \end{cases}$$

See [41] for full details of the training task.

$\boldsymbol{M_{1,4}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{1,4}}$**:** The learned policy consists of 59 parameters specifying a fixed series of poses for a kick motion defined by our skill description language. Infor-

mation about the skill description language is provided in Section 8.2.6. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those sampled from an observed kick as described in [41].

$\boldsymbol{L_{1,5}}$ **Kick_Fast_Primitive:** The robot learns a short but fast to execute kick starting from a standing position behind the ball. The robot is also expected to be stable and still standing after the kick.

$\boldsymbol{T_{1,5}}$**:** The robot attempts to quickly kick the ball as far as possible from a standing position behind the ball. The robot is given higher return values for longer kicks, is penalized if the kick takes too long to execute (greater than 0.25 seconds), and is given a negative return value if it fails to kick the ball or falls over while doing so. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_fast\_primitive}} = \begin{cases} -1 & : \text{Failure} \\ -1 & : \text{Robot fell over} \\ \text{distBallForward} - \max(\text{kickTime} - 0.25, 0) & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{1,5}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{1,5}}$**:** The learned policy consists of 33 parameters specifying a fixed series of poses for a fast kick motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Parameters for the x and y offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those

from a hand-coded policy.

$L_{2,1}$: Walk_GoToTarget: The robot learns a walk for moving to general target positions on the field.

$T_{2,1}$: The training task consists of an obstacle course in which the robot tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the robot is rewarded based on its distance traveled toward the active target. To promote stability, the robot is given a penalty if it falls over. After falling the robot executes one of the getup behaviors learned in $L_{1,1}$ and $L_{1,2}$—*Getup_Front_Primitive* or *Getup_Back_Primitive* respectively—so that it can stand up and continue the walk training task. Full details of the training task, known as the goToTarget task, are provided in Appendix A.1.3.

$M_{2,1}$: Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{2,1}$: The learned policy consists of 27 parameters for a walk engine described in Section 8.2.5. The maxStep_x and maxStep_y parameters in Table 8.2 are learned as a single value, however, as we found this advantageous in ensuring that the learned policy would not overly favor walking in the x direction over the y direction. Initial walk engine parameter values were seeded with those from a hand-coded policy used on physical robots as described in [103].

$L_{2,2}$: KickOff_Kick_Behavior: A two robot behavior is learned for scoring on a kickoff with one robot lightly touching ball before the other robot kicks the

ball in the goal.

$T_{2,2}$: One robot attempts to lightly touch the ball at the beginning of a kick-off, after which the second robot attempts to kick the ball in the goal. During the task the touch and kick robots use the previously learned *KickOff_Touch_Primitive* and *KickOff_Kick_Primitive* respectively. The task is evaluated with the same objective function as is used in the training task $T_{1,4}$, except that a negative return value is also given if the first robot fails to touch the ball before the second robot kicks the ball. The objective function used during optimization is the following where all distances are in meters:

$$f_{\text{kickoff}} = \begin{cases} -1 & : \text{First robot failed touch} \\ -1 & : \text{Second robot failed kick} \\ 0 & : \text{Missed goal} \\ 100 + \text{distBallForward} + 2 * \text{distBallInAir} & : \text{Otherwise} \end{cases}$$

See [41] for full details of the training task.

$M_{2,2}$: Learning was performed across 100 generations of CMA-ES with a population size of 150.

$H_{2,2}$: The learned policy consists of the x, y, and θ offset standing position parameters of both the robots from the ball that are part of $H_{1,3}$ and $H_{1,4}$. This learning is an example of CILB as $H_{1,3} \in L_{2,2}$, $H_{1,4} \in L_{2,2}$, and $\{H'_{1,3} \cup H'_{1,4}\} \subset H_{2,2}$ where $H'_{1,3}$ and $H'_{1,4}$ are the subsets of standing position parameters of $H_{1,3}$ and $H_{1,4}$ respectively. An additional synchronized timing parameter for how long the first robot should wait before touching the ball is also learned so that the second robot does not acci-

dentally try and kick the ball before the first robot has touched it. This new added parameter makes this learned behavior also an example of PCLL as $\{H'_{1,3} \cup H'_{1,4}\} \subset H_{2,2}$ instead of just $\{H'_{1,3} \cup H'_{1,4}\} = H_{2,2}$.

$\boldsymbol{L_{2,3}}$ **: Kick_Long_Primitive:** The robot learns a kicking motion to kick the ball a long distance starting from a standing position behind the ball.

$\boldsymbol{T_{2,3}}$**:** The robot attempts to kick the ball as far as possible from a standing position behind the ball. The robot is given higher return values for longer kicks, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_long\_primitive}} = \begin{cases} -1 & : \text{Failure} \\ \text{distBallForward} - \max(\text{kickTime} - 2, 0) & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{2,3}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{2,3}}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a long kick motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Parameters learned in $h_{1,4}$ for the *KickOff_Kick_Primitive* are used as seed values for an initial policy ($h_{1,4} \dashrightarrow H_{2,3}$), and added parameters for speeding up the kick are also learned. Parameters for the x, y, and θ offset standing position of the robot from the ball are learned as well.

$\boldsymbol{L_{3,1}}$ **: Walk_Sprint:** The robot learns a walk for quickly walking in the forward

direction.

$\boldsymbol{T_{3,1}}$**:** The same goToTarget training task is performed as in $L_{2,1}$, however the robot only uses the walk it is learning when its orientation is within 15° of its target. During the remainder of the training task the robot is using the learned behavior from $L_{2,1}$ thus ensuring that the walk being learned can transition from/to the previously learned *Walk_GoToTarget* behavior's walk. Full details of the training task are provided in Appendix A.1.4.

$\boldsymbol{M_{3,1}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{3,1}}$**:** The learned policy consists of 27 parameters for a walk engine described in Section 8.2.5. Initial values for the policy are seeded with those from $h_{2,1}$ ($h_{2,1}$ --→ $H_{3,1}$) that were optimized for walking to general target positions on the field.

$\boldsymbol{L_{3,2}}$ **: Kick_High_Primitive:** The robot learns a kicking motion to kick the ball over opponents starting from a standing position behind the ball.

$\boldsymbol{T_{3,2}}$**:** The robot attempts to kick the ball as far as possible in the air from a standing position behind the ball. The robot is given higher return values for kicks that travel longer distances at a height above that of opponents, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_high\_primitive}} = \begin{cases} -1 & : \text{Failure} \\ \text{distBallInAir} - \max(\text{kickTime} - 2, 0) & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{3,2}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{3,2}}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a high kick motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Parameters learned in $h_{2,3}$ for the *Kick_Long_Primitive* are used as seed values for an initial policy ($h_{2,3} \dashrightarrow H_{3,2}$).

$\boldsymbol{L_{3,3}}$ **: Kick_Low_Primitive:** The robot learns a kicking motion to kick the ball such that it stays below the height of the goal when starting from a standing position behind the ball.

$\boldsymbol{T_{3,3}}$**:** The robot attempts to kick the ball as far as possible on the ground from a standing position behind the ball. The robot is given higher return values for kicks that travel longer distances, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball or the ball travels above the height of the goal. The objective function used during optimization is the following where all distance is in meters and time is in seconds:

$$f_{\text{kick\_low\_primitive}} = \begin{cases} -1 & : \text{Failure} \\ -1 & : \text{Kick above goal height} \\ \text{distBallForward} - \max(\text{kickTime} - 2, 0) & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{3,3}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{3,3}}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a low kick motion defined by our skill description language. Information about the skill description language is provided in Section 8.2.6. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Parameters learned in $h_{2,3}$ for the *Kick_Long_Primitive* are used as seed values for an initial policy ($h_{2,3}$ --→ $H_{3,3}$).

$\boldsymbol{L_{4,1}}$ **: Getup_Front_Behavior:** The robot learns to stand up when starting from lying on its front and then walks around.

$\boldsymbol{T_{4,1}}$**:** The training task is the same as $T_{1,1}$ except that after standing up the robot is asked to walk in different directions, and its return value is penalized if it falls over while trying to do so. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_front}} = \begin{cases} -\text{timeNotStanding} - 5 & : \text{ Fell over after standing} \\ -\text{timeNotStanding} & : \text{ Otherwise} \end{cases}$$

$\boldsymbol{M_{4,1}}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{4,1}}$**:** The learned policy consists of the same 9 *Getup_Front_Primitive* parameters in $H_{1,1}$ which are now unfrozen and relearned. As parameters are unfrozen and relearned this learned behavior is an example of PLLR.

$\boldsymbol{L_{4,2}}$ **: Getup_Back_Behavior:** The robot learns to stand up when starting from lying on its back and then walks around.

$\boldsymbol{T_{4,2}}$**:** The training task is the same as $T_{1,2}$ except that after standing up

the robot is asked to walk in different directions, and its return value is penalized if it falls over while trying to do so. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_back}} = \begin{cases} -\text{timeNotStanding} - 5 & : \text{Fell over after standing} \\ -\text{timeNotStanding} & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{4,2}}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{4,2}}$**:** The learned policy consists of the same 26 *Getup_Back_Primitive* parameters in $H_{1,2}$ which are now unfrozen and relearned. As parameters are unfrozen and relearned this learned behavior is an example of PLLR.

$\boldsymbol{L_{4,3}}$ **: Walk_PositionToDribble:** The robot learns a walk for dribbling the ball.

$\boldsymbol{T_{4,3}}$**:** The robot starts from different positions relative to the ball and is asked to dribble the ball toward the opponent's goal for 15 seconds. The robot only uses the walk it is learning when positioning around the ball to dribble it. During the remainder of the training task the robot is using the learned behaviors from $L_{2,1}$ and $L_{3,1}$ thus ensuring that the walk being learned can transition from/to the previously learned walk behaviors for walking and sprinting. The robot is given a return value equal to the distance it is able to dribble the ball toward the opponent's goal while being penalized if it falls over. Full details of the training task, known as the driveBallToGoal2 task, are given in Appendix A.1.5.

$\boldsymbol{M_{4,3}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{4,3}}$**:** The learned policy consists of 27 parameters for a walk engine described in Section 8.2.5. Initial values for the policy are seeded with those from $h_{2,1}$ ($h_{2,1}$ $\dashrightarrow$ $H_{4,3}$) that were optimized for walking to general target positions on the field.

$\boldsymbol{L_{5,1}}$ **: Walk_ApproachToKick:** The robot learns a walk for stopping at a precise position behind the ball in preparation to kick the ball.

$\boldsymbol{T_{5,1}}$**:** The robot is asked to walk to a target position near the ball from which a kick might be executed. The robot is rewarded for stopping at this target position in as little time as possible while being penalized if it runs into the ball or falls over. The robot only uses the walk it is learning when close to the ball. During the remainder of the training task the robot is using the learned behaviors from $L_{2,1}$ and $L_{3,1}$ thus ensuring that the walk being learned can transition from/to the previously learned walk behaviors for walking and sprinting. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$
\begin{aligned}
f_{\text{walk\_approach\_to\_kick}} = \; & -\text{timeTaken} \\
& +\text{fFellOver} \; ? \; -1 : 0 \\
& +\text{timeTaken} > 12 \; ? \; -0.7 : 0 \\
& +\text{fRanIntoBall} \; ? \; -0.5 : 0 \\
& +\text{velocityWhenInPositionToKick} > 0.005 \; ? \; -0.5 : 0
\end{aligned}
$$

See [105] for full details of the training task.

$\boldsymbol{M_{5,1}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{5,1}$: The learned policy consists of 27 parameters for a walk engine described in Section 8.2.5. Initial values for the policy are seeded with those from $h_{4,3}$ ($h_{4,3}$ --→ $H_{5,1}$) that were optimized for positioning around the ball when dribbling. Additional parameters described in [105] for determining how quickly the robot should accelerate and decelerate to reach its target without running into the ball are also learned.

$L_{6,1}$: Kick_Fast_Behavior: The robot learns to walk to the ball and perform a short but fast to execute kick. The robot is also expected to be stable and still standing after the kick.

$T_{6,1}$: The same $T_{1,5}$ optimization task for kicking a ball quickly is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_fast}} = \begin{cases} -1 & : \text{Failure} \\ -1 & : \text{Robot fell} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{Otherwise} \end{cases}$$

$M_{6,1}$: Learning was performed across 100 generations of CMA-ES with a population size of 150.

$H_{6,1}$: The learned policy consists of the same two *Kick_Fast_Primitive* x and y offset position parameters from the ball in $H_{1,5}$—the target position

for the walk to reach for the kick to be executed—which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Fast_Behavior* and *Walk_ApproachToKick* ($L_{1,5}$ and $L_{5,1}$) are combined ($H_{1,5} \in L_{6,1}$, $H_{5,1} \in L_{6,1}$) by re-learning a subset of their parameters ($H_{6,1} \subset \{H_{1,5} \cup H_{5,1}\}$).

$L_{6,2}$: Kick_High_Behavior: The robot learns to walk to the ball and perform a high kick to kick the ball over opponents.

$T_{6,2}$: The same $T_{3,2}$ optimization task for kicking a ball high is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_high}} = \begin{cases} -1 & : \text{Failure} \\ \text{distBallInAir} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{Otherwise} \end{cases}$$

$M_{6,2}$: Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{6,2}$: The learned policy consists of the same 73 *Kick_High_Primitive* parameters in $H_{3,2}$ which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_High_Behavior* and *Walk_ApproachToKick* ($L_{3,2}$ and $L_{5,1}$) are combined ($H_{3,2} \in L_{6,2}$, $H_{5,1} \in L_{6,2}$) by re-learning a subset of their parameters ($H_{6,2} \subset \{H_{3,2} \cup$

$H_{5,1}\}$).

$L_{6,3}$: Kick_Low_Behavior: The robot learns to walk to the ball and perform a low kick that stays below the height of the goal.

$T_{6,3}$: The same $T_{3,3}$ optimization task for kicking a ball low is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_low}} = \begin{cases} -1 & : \text{Failure} \\ -1 & : \text{Kick above goal} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{Otherwise} \end{cases}$$

$M_{6,3}$: Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{6,3}$: The learned policy consists of the same 73 *Kick_Low_Primitive* parameters in $h_{3,3}$ which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Low_Behavior* and *Walk_ApproachToKick* ($L_{3,3}$ and $L_{5,1}$) are combined ($H_{3,3} \in L_{6,3}$, $H_{5,1} \in L_{6,3}$) by re-learning a subset of their parameters ($H_{6,3} \subset \{H_{3,3} \cup H_{5,1}\}$).

$L_{6,4}$: Kick_Long_Behavior: The robot learns to walk to the ball and perform a

long kick.

$T_{6,4}$: The same $T_{2,3}$ optimization task for kicking a ball a long distance is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_long}} = \begin{cases} -1 & : \text{Failure} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{Otherwise} \end{cases}$$

$M_{6,4}$: Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{6,4}$: The learned policy consists of the same 73 *Kick_Long_Primitive* parameters in $H_{2,3}$ which are now re-optimized.This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Long_Behavior* and *Walk_ApproachToKick* ($L_{2,3}$ and $L_{5,1}$) are combined ($H_{2,3} \in L_{6,4}$, $H_{5,1} \in L_{6,4}$) by re-learning a subset of their parameters ($H_{6,4} \subset \{H_{2,3} \cup H_{5,1}\}$).

A.1 Optimization Process and Training Tasks for Learning Walk Parameter Sets

The following subsections detail the optimization process and training tasks used during the layered learning approach to develop three walk parameter sets needed for general walking, sprinting, and dribbling the ball. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm [57] was used to learn the walk engine parameters listed in Table 8.2 for each set of walk parameters. Further details about the development of the optimization process—first implemented for the 2011 team—are available in [103], however here we only present the details most relevant to the 2014 team.

A.1.1 Walk Usage Considerations

Before describing the procedure for optimizing the walk parameters, we provide some brief context for how the agent's walk is typically used. These details are important for motivating the optimization procedure's fitness functions.

During gameplay the agent is usually either moving to a set target position on the field or dribbling the ball toward the opponent's goal and away from the opposing team's players. Given that an omnidirectional walk engine can move in any direction as well as turn at the same time, the agent has multiple ways in which it can move toward a target. We chose the approach of both having the agent move and turn toward a target at the same time as this allows for both quick reactions (the agent is immediately moving in the desired direction) and speed (where the bipedal robot model is faster when walking forward as opposed to strafing sideways). We validated this design decision by playing our agent against a version of itself which does not turn to face the target it is moving toward, and found our agent that turns

won by an average of .7 goals across 100 games. Additionally we played our agent against a version of itself that turns in place until its orientation is such that it is able to move toward its target at maximum forward velocity, and found our agent that immediately starts moving toward its target won by an average of .3 goals across 100 games. All agents we compared used walks optimized by the process described in the following subsections.

Dribbling the ball is a little different in that the agent needs to align behind the ball, without first running into the ball, so that it can walk straight through the ball, moving it in the desired dribble direction. When the agent circles around the ball, it always turns to face the ball so that if an opponent approaches, it can quickly walk forward to move the ball and keep it out of reach of the opponent.

A.1.2 Optimization Task Architecture

To ease the optimization process, we build optimization tasks out of a series of independent phases, called OptPhases. Each OptPhase encapsulates a logically distinct action that the agent must take, the utility function for that action, and any of the agent's observations during the execution of that phase that is used as input to the utility function. To guard against the case where the agent cannot complete the action, each OptPhase also has a maximum duration. If the agent does not complete an OptPhase's associated action within the specified duration, it simply moves on to the next OptPhase. Should the agent fall down during a phase, the next phase is not started until the agent gets up again. For optimizing the agent's walk, we primarily used the following phase types—all of which punish for falling down:

- A WaypointOptPhase, during which the agent attempts to walk to a certain coordinate and is rewarded based on how far it can walk before the phase

ends. If the agent arrives at its destination before the time runs out, we try to extrapolate how far the agent would have gone if allowed to walk for the entire phase.

- A StopOptPhase, during which the agent stands still and is punished for moving. This phase type is useful for making sure that the agent is stable, not just fast.

- A MoveOptPhase, during which the agent walks in a certain direction and is rewarded based on the distance it can travel before the phase ends.

At the beginning of an optimization task, the agent is initialized with a list of OptPhases and it simply needs to execute them all in order. Once it has gone through all of the OptPhases, it simply adds up all of the utility values for each of the individual phases and use that as the utility for the entire run. This approach allows us to quickly and easily create and experiment with different optimization strategies. For example, we can optimize for stability: by chaining together a series of short MoveOptPhases sprinkled with a number of StopOptPhases to make the agent quickly change direction, or for navigation speed: by using WaypointOptPhases to create a sort of obstacle course as the goToTarget optimization task described in Appendix A.1.3.

A.1.3 Walk_GoToTarget Parameter Set Optimization

To learn walk parameters for moving to general target positions on the field we created a training task—called the goToTarget optimization task—consisting of an obstacle course in which the robot tries to navigate to a variety of target positions on the field. The targets are represented as WaypointOptPhases described in Appendix A.1.2. Each target is active, one at a time for a fixed period of time,

which varies from one target to the next, and the robot is rewarded based on its distance traveled toward the active target. If the robot reaches an active target, the robot receives an extra reward based on extrapolating the distance it could have traveled given the remaining time on the target. In addition to the target positions, the robot has stop targets—represented as StopOptPhases described in Appendix A.1.2—where it is penalized for any distance it travels. To promote stability, the robot is given a penalty if it falls over during the optimization run.

In the following equations specifying the agent's rewards for targets, *Fall* is 5 if the robot fell and 0 otherwise, d_{target} is the distance traveled toward the target, and d_{moved} is the total distance moved. Let t_{total} be the full duration a target is active and t_{taken} be the time taken to reach the target or t_{total} if the target is not reached.

$$\begin{aligned} \text{reward}_{\text{target}} &= d_{\text{target}} \frac{t_{\text{total}}}{t_{\text{taken}}} - \text{Fall} \\ \text{reward}_{\text{stop}} &= -d_{\text{moved}} - \text{Fall} \end{aligned}$$

The goToTarget optimization includes quick changes of target/direction for focusing on the reaction speed of the agent, as well as targets with longer durations to improve the straight line speed of the agent. The stop targets ensure that the agent is able to stop quickly, while remaining stable. The trajectories that the agent follows during the optimization are described in Figure A.1.

A.1.4 Walk_Sprint Parameter Set Optimization

To further improve the forward speed of the agent, we optimized a parameter set for walking straight forwards for ten seconds starting from a complete stop. Unfortunately, when the robot tried to switch between the forward walk and *Walk_GoToTarget*

- Long walks forward/backwards/left/right
- Walk in a curve
- Quick direction changes
- Stop and go forward/backwards/left/right
- Switch between moving left-to-right and right-to-left
- Quick changes of target to simulate a noisy target
- Weave back and forth at 45 degree angles
- Extreme changes of direction to check for stability
- Quick movements combined with stopping
- Quick alternating between walking left and right
- Spiral walk both clockwise and counter-clockwise

Figure A.1: GoToTarget Optimization walk trajectories

parameter sets it was unstable and usually fell over. This instability is due to the parameter sets being learned in isolation, resulting in them being incompatible.

To overcome this incompatibility, we ran the goToTarget subtask optimization again, but this time we fixed the *Walk_GoToTarget* parameter set and learned a new parameter set. We call these parameters the *Walk_Sprint* parameter set, and the agent uses them when its orientation is within 15° of its target. The *Walk_Sprint* parameter set was seeded with the values from the *Walk_GoToTarget* parameter set. This approach to optimization is an example of sequential layered learning as the output of one learned subtask (the *Walk_GoToTarget* parameter set) is fed in as input to the learning of the next subtask (the learning of the *Walk_Sprint* parameter set). By learning the *Walk_Sprint* parameter set in conjunction with the

Walk_GoToTarget parameter set, the robot was stable switching between the two parameter sets.

A.1.5 Walk_PositionToDribble Parameter Set Optimization

Although adding the *Walk_GoToTarget* and *Walk_Sprint* walk engine parameter sets improved the stability, speed, and game performance of the agent, the agent was still a little slow when positioning to dribble the ball. This slowness is explained by the fact that the goToTarget subtask optimization emphasizes quick turns and forward walking speed while positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learned a third parameter set which we call the *Walk_PositionToDribble* parameter set. To learn this parameter set, we created a new driveBallToGoal2[53] optimization in which the agent is evaluated on how far it is able to dribble the ball over 15 seconds when starting from a variety of positions and orientations from the ball. The *Walk_PositionToDribble* parameter set is used when the agent is .8 meters from the ball and is seeded with the values from the *Walk_GoToTarget* parameter set. Both the *Walk_GoToTarget* and *Walk_Sprint* parameter sets are fixed and the optimization naturally includes transitions between all three parameter sets, which constrained them to be compatible with each other. As learning of the *Walk_PositionToDribble* parameter set takes the two previously learned parameter sets as input, it is a third layer of sequential layered learning.

[53]The '2' at the end of the name driveBallToGoal2 is used to differentiate it from the driveBallToGoal optimization that was used in [103].

Appendix B

Additional SCRAM Proof Sketches

This appendix provides proof sketches for properties of role assignment functions discussed in Chapter 5: minimizing the makespan, avoiding collisions, and dynamic consistency.

B.1 Role Assignment Function CM Validity

The following[54] is a more in depth analysis of the *CM validity* of the role assignment functions MMDR and MMD+MSD^2 described in Section 5.3.

B.1.1 Minimizing Longest Distance

It is trivial to determine that both MMDR and MMD+MSD^2 select a mapping of agents to role positions that minimizes the time for all agents to have reached their target destinations. The total time it takes for all agents to move to their

[54]This appendix contains material from previously published work in [111].

desired positions is determined by the time it takes for the last agent to reach its target position. As the first comparison between mapping costs for both role assignment functions is the maximum distance that any single agent must travel, and it is assumed that all agents move toward their targets at the same constant rate, the property of minimizing the longest distance holds for both MMDR and MMD+MSD^2.

B.1.2 Avoiding Collisions

Given the assumptions that no two agents and no two role positions occupy the same position on the field, and that all agents move toward role positions along a straight line at the same constant speed, if two agents collide it means that they both started moving from positions that are the same distance away from the collision point. Furthermore if either agent were to move to the collision point, and then move to the target of the other agent, its total path distance to reach that target would be the same as the path distance of the other agent to that same target. Considering that we are working in a Euclidean space, by the triangle inequality we know that the straight path from the first agent to the second agent's target will be less than the path distance of the first agent moving to the collision point and then moving on to the second agent's target. The path distance of the first agent moving to the collision point, and then moving on to the second agent's target, is in fact equal to the distance of the second agent moving on a straight line to its target. Thus if the two colliding agents were to switch targets the maximum distance either is traveling will be reduced—along with the sum of the squared distances traveled—thereby reducing the cost of the mapping for both MMDR and MMD+MSD^2, and the collision will be avoided. Figure B.1 illustrates an example

of this scenario.

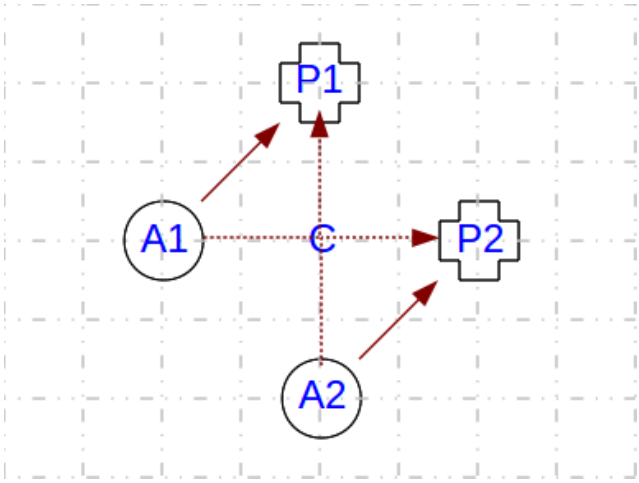


Figure B.1: Example collision scenario. If the mapping (A1→P2,A2→P1) is chosen the agents will follow the dotted paths and collide at the point marked with a C. Instead both MMDR and MMD+MSD^2 will choose the mapping (A1→P1,A2→P2), as this minimizes both maximum path distance and sum of distances squared, and the agents will follow the paths denoted by the solid arrows thereby avoiding the collision.

The following is a proof sketch related to Figure B.1 that no collisions will occur.

Assumption. *Agents A1 and A2 move at constant velocity v on straight line paths to static positions P2 and P1 respectively. $A1 \neq A2$ and $P1 \neq P2$. Agents collide at point C at time t.*

Claim. *A1→P2 and A2→P1 is an optimal mapping returned by MMDR.*

Case 1. *$P1$ and $P2 \neq C$.*

By assumption:

$\overline{A_1C} = \overline{A_2C} = vt$

$\overline{A_1P_2} = \overline{A_1C} + \overline{CP_2} = \overline{A_2C} + \overline{CP_2}$

$\overline{A_2P_1} = \overline{A_2C} + \overline{CP_1} = \overline{A_1C} + \overline{CP_1}$

By triangle inequality:

$\overline{A_1P_1} < \overline{A_1C} + \overline{CP_1} = \overline{A_2P_1}$

$\overline{A_2P_2} < \overline{A_2C} + \overline{CP_2} = \overline{A_1P_2}$

$\max(\overline{A_1P_1}, \overline{A_2P_2}) < \max(\overline{A_1P_2}, \overline{A_2P_1})$

$\overline{A_1P_1}^2 + \overline{A_2P_2}^2 < \overline{A_1P_2}^2 + \overline{A_2P_1}^2$

$\therefore \texttt{cost}(A1 \to P1, A2 \to P2) < \texttt{cost}(A1 \to P2, A2 \to P1)$ *and claim is False.*

Case 2. *P1 = C, P2* $\neq$ *C.*

By assumption:

$\overline{CP_2} > \overline{CP_1} = 0$

$\overline{A_2C} \leq \overline{A_1C} = vt$

$\overline{A_1P_1} = \overline{A_1C} < \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$

By triangle inequality:

if $\overline{A_1C} = \overline{A_2C}$

$$\overline{A_2P_2} < \overline{A_2C} + \overline{CP_2} = \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$$

otherwise $\overline{A_2C} < \overline{A_1C}$

$$\overline{A_2P_2} \leq \overline{A_2C} + \overline{CP_2} < \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$$

$\max(\overline{A_1P_1}, \overline{A_2P_2}) < \max(\overline{A_1P_2}, \overline{A_2P_1})$

$\overline{A_1P_1}^2 + \overline{A_2P_2}^2 < \overline{A_1P_2}^2 + \overline{A_2P_1}^2$

$\therefore \texttt{cost}(A1 \to P1, A2 \to P2) < \texttt{cost}(A1 \to P2, A2 \to P1)$ *and claim is False*

Case 3. *P2 = C, P1* $\neq$ *C.*

Claim False by corollary to Case 2.

Case 4. *P1, P2 = C.*

Claim False by assumption.

As claim is False for all cases MMDR does not return mappings with collisions. □

B.2 Dynamic Consistency

The following is a more in depth analysis of the dynamic consistency of the role assignment functions MMDR and MMD+MSD^2 described in Section 5.3.

Dynamic consistency is important such that as agents move toward fixed target role positions they do not continually switch or thrash between roles thus impeding their progress in reaching target positions. Given the assumption that all agents move toward target positions at the same constant rate, all distances to targets in a MMDR mapping of agents to role positions will decrease at the same constant rate as the agents move until becoming 0 when an agent reaches its destination. Considering that agents move toward their target positions on straight line paths, it is not possible for the distance between any agent and any role position to decrease faster than the distance between an agent and the role position it is assigned to move toward. Given this fact, the cost of any MMDR mapping can not improve over time any faster than the lowest cost MMDR mapping being followed, and thus dynamic consistency is preserved. Note that it is possible for two mappings of agents to role positions to have the same MMDR cost as the case of two agents being equidistant to two role positions. In this case one of the mappings may be arbitrarily selected and followed by the agents. As soon as the agents start moving the selected mapping will acquire and maintain a lower cost than the unselected

mapping. The only way that the mappings could continue to have the same MMDR cost would be if the two role positions occupy the same place on the field, however, as stated in the given assumptions, this is not allowed.

MMD+MSD^2 is not dynamically consistent as minimizing the sum of distances squared (MSD^2) is not dynamically consistent. MSD^2 is shown to be not dynamically consistent in Appendix B.3.

B.3 Other Role Assignment Functions

The following is an analysis of both the *CM validity* and dynamic consistency of the role assignment functions other than MMDR and MMD+MSD^2 evaluated in Section 5.4.

Other potential ordering heuristics for mappings of agents to target positions include minimizing the sum of all distances traveled (MSD), minimizing the sum of all path distances squared (MSD^2), and assigning agents to targets in order of shortest distances (Greedy). None of these heuristics preserve both required properties listed in Section 5.2 for *CM validity* which are true for both MMDR and MMD+MSD^2. Also none of them are dynamically consistent.

As can be seen in the example given in Figure B.2, none of the properties necessarily hold for MSD.

The first property of all agents having reached their target destinations in as little time as possible is not always true for MSD^2 as shown in the example in Figure B.3. MSD^2 does avoid collisions as explained in Appendix B.1.2. The following is an example in which MSD^2 is not dynamically consistent:

At time $t = 0$:

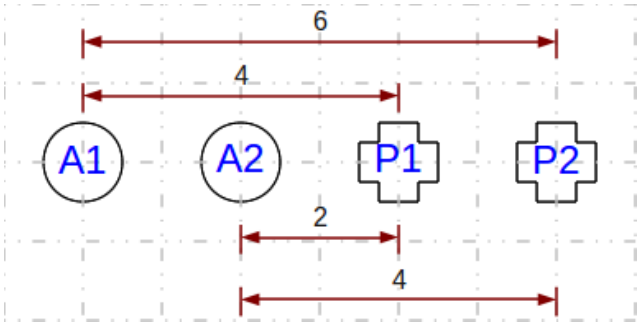


Figure B.2: Example where minimizing the sum of path distances fails to hold desired properties. Both mappings of (A1→P1,A2→P2) and (A1→P2,A2→P1) have a sum of distances value of 8. The mapping (A1→P2,A2→P1) will result in a collision and has a longer maximum distance of 6 than the mapping (A1→P1,A2→P2) whose maximum distance is 4. Once a mapping is chosen and the agents start moving the sum of distances of the two mappings will remain equal which could result in thrashing between the two.

$A_1 = (3, 0)$

$A_2 = (2, 999)$

$P_1 = (0, 0)$

$P_2 = (1, 0)$

$A1 \to P1, A2 \to P2$

$\overline{A_1P_1} = 3$, $\overline{A_2P_2} = \sqrt{998002}$; $\overline{A_1P_1}^2 + \overline{A_2P_2}^2 = 998011$

$A1 \to P2, A2 \to P1$

$\overline{A_1P_2} = 2$, $\overline{A_2P_1} = \sqrt{998005}$; $\overline{A_1P_2}^2 + \overline{A_2P_1}^2 = 998009$

MSD^2 mapping $(A1 \to P2, A2 \to P1) \because 998009 < 998011$

At time $t = 2$:

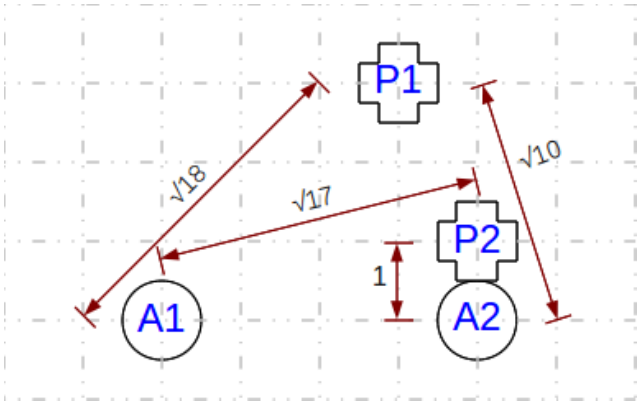


Figure B.3: Example where minimizing the sum of path distances squared fails to hold desired property of minimizing the time for all agents to have reached their target destinations. The mapping (A1→P1,A2→P2) has a path distance squared sum of 19 which is less than the mapping (A1→P2,A2→P1) for which this sum is 27. Both MMDR and MMD+MSD^2 will choose the mapping with the greater sum as its maximum path distance (proportional to the time for all agents to have reached their targets) is $\sqrt{17}$ which is less than the other mapping's maximum path distance of $\sqrt{18}$.

$A_1 = (1, 0)$

$A_2 = (\sim 2, \sim 997)$

$P_1 = (0, 0)$

$P_2 = (1, 0)$

$A1 \to P1, A2 \to P2$

$\overline{A_1P_1} = 1$, $\overline{A_2P_2} = \sqrt{994010}$; $\overline{A_1P_1}^2 + \overline{A_2P_2}^2 = 994011$

$A1 \to P2, A2 \to P1$

$\overline{A_1P_2} = 0$, $\overline{A_2P_1} = \sqrt{994013}$; $\overline{A_1P_2}^2 + \overline{A_2P_1}^2 = 994013$

MSD^2 mapping $(A1 \to P1, A2 \to P2) \because 994011 < 994013$

As the mapping switched MSD^2 is not dynamically consistent.

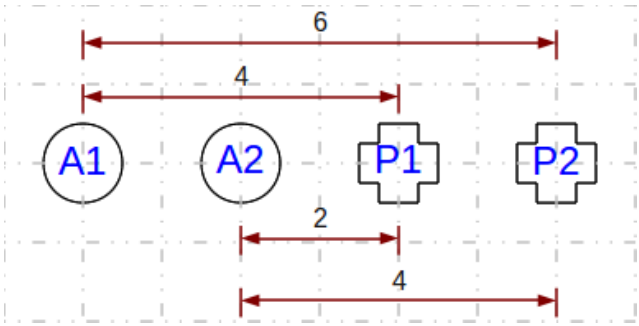


Figure B.4: Example where greedily choosing shortest paths fails to hold desired properties. The shortest distance is from A2→P1 resulting in a mapping of (A2→P1,A1→P2) to be chosen. The mapping (A2→P1,A1→P2) will result in a collision and has a longer maximum distance of 6 than the mapping (A1→P1,A2→P2) whose maximum distance is 4. Once the agents collide it is possible that A1 will move on top of P1 thus pushing A2 off of P1 and towards P2. This displacement of A2 may result in a switch between mappings and potential thrashing.

As can be seen in the example given in Figure B.4, none of the properties necessarily hold for Greedy.

Appendix C

Dynamic Programming Algorithm for MMDR

The following[55] is a description of the dynamic programming algorithm for computing the Minimum Maximal Distance Recursive (MMDR) role assignment function that is compared against SCRAM role assignment algorithms in Section 5.4.

A key recursive property of MMDR that allows us to exploit dynamic programming is expressed in Theorem 3. This property stems from the fact that if within any subset of a mapping a lower cost mapping is found, then the cost of the complete mapping can be reduced by augmenting the complete mapping with that of the subset's lower cost mapping.

Theorem 3. *Let A and P be sets of n agents and positions respectively. Denote the mapping $m := MMDR(A, P)$. Let m_0 be a subset of m that maps a subset of agents $A_0 \subset A$ to a subset of positions $P_0 \subset P$. Then m_0 is also the mapping returned by $MMDR(A_0, P_0)$.*

[55] This appendix contains material from previously published work in [102].

The savings from using dynamic programming comes from only evaluating mappings whose subset mappings are returned by MMDR. This savings is accomplished in Algorithm 5 by iteratively building up optimal mappings for position sets from $\{p_1\}$ to $\{p_1, ..., p_n\}$, and using optimal mappings of $k-1$ agents to positions $\{p_1, ..., p_{k-1}\}$ (line 8) as a base when constructing each new mapping of k agents to positions $\{p_1, ..., p_k\}$ (line 9), before saving the lowest cost mapping for the current set of k agents to positions $\{p_1, ..., p_k\}$ (line 10).

Algorithm 5 Dynamic programming implementation of MMDR

```
1:  HashMap bestRoleMap = ∅
2:  Agents = {a_1, ..., a_n}
3:  Positions = {p_1, ..., p_n}
4:  for k = 1 to n do
5:      for each a in Agents do
6:          S = (n-1 choose k-1) sets of k-1 agents from Agents - {a}
7:          for each s in S do
8:              Mapping m_0 = bestRoleMap[s]
9:              Mapping m = (a → p_k) ∪ m_o
10:             bestRoleMap[{a} ∪ s] = mincost(m, bestRoleMap[{a} ∪ s])
11: return bestRoleMap[Agents]
```

An example of the mapping combinations evaluated in finding the optimal mapping for three agents through the dynamic programming approach of Algorithm 5 can be seen in Table C.1. In this example the algorithm begins by computing the distance of each agent to the first role position. Next the algorithm computes the cost of all possible mappings of agents to both the first and second role positions and saves off the lowest cost mapping of every pair of agents to the the first two positions. The algorithm then proceed by sequentially assigning every agent to the third position and computes the lowest cost mapping of all agents mapped to all three positions. As all subsets of an optimal (lowest cost) mapping will themselves be optimal, the algorithm only needs to evaluate mappings to all three positions

which include the previously calculated optimal mapping agent combinations for the first two positions.

Table C.1: All mappings evaluated during dynamic programming using Algorithm 5 when computing an optimal mapping of agents A1, A2, and A3 to positions P1, P2, and P3. Each column contains the mappings evaluated for the set of positions listed at the top of the column.

{P1}	{P2,P1}	{P3,P2,P1}
A1→P1	A1→P2, MMDR(A2→P1)	A1→P3, MMDR({A2,A3}→{P1,P2})
A2→P1	A1→P2, MMDR(A3→P1)	A2→P3, MMDR({A1,A3}→{P1,P2})
A3→P1	A2→P2, MMDR(A1→P1)	A3→P3, MMDR({A1,A2}→{P1,P2})
	A2→P2, MMDR(A3→P1)	
	A3→P2, MMDR(A1→P1)	
	A3→P2, MMDR(A2→P1)	

Recall that during the kth iteration of the dynamic programming process to find a mapping for n agents, where k is the current number of positions that agents are being mapped to, each agent is sequentially assigned to the kth position and then all possible subsets of the other $n-1$ agents are assigned to positions 1 to $k-1$ based on computed optimal mappings to the first $k-1$ positions from the previous iteration of the algorithm. These assignments result in a total of $\binom{n-1}{k-1}$ agent subset mapping combinations to be evaluated for mappings of each agent assigned to the kth position. The total number of mappings computed for each of the n agents across all n iterations of dynamic programming is thus equivalent to the sum of the $n-1$ binomial coefficients. That is,

$$\sum_{k=1}^{n} \binom{n-1}{k-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}$$

Therefore the total number of mappings that must be evaluated using our dynamic programming approach is $n2^{n-1}$.

Appendix D

UT Austin Villa RoboCup 3D Simulation Team Strategy

This appendix provides details of some of the strategy components used by the UT Austin Villa agent team presented in Chapter 8. This includes general strategy for movement in Appendix D.1, strategy for kicking in Appendix D.2, and the team's goalie in Appendix D.3. Details of how the team's strategy incorporates formations and role assignment are discussed in Chapter 7.

D.1 General Locomotion in the Field

This section covers details regarding how agents behave on the field. Section D.1.1 explains how which agent should go to the ball is determined. A system used to avoid collisions is given in Section D.1.2. Movement and actions around the ball including facing the ball (Section D.1.3), how the ball is approached (Section D.1.4), where to move the ball (Section D.1.5), and how to dribble (Section D.1.6) follow.

D.1.1 Closest to Ball Heuristic

In Section 7.1 we mention that the closest player to the ball is assigned the *onBall* role and is instructed to go to the ball. In order to measure "closeness" we do not purely use Euclidean distance, however, as certain positions such as being behind the ball instead of in front of it are more advantageous. An agent is considered to be in front of the ball if its X coordinate is greater than that of the ball's X coordinate. If an agent is in front of the ball it will typically have to take time to circle and walk around behind the ball in order to dribble the ball forward toward the opponent's goal. For this reason we add 1 meter to the distances of agents in front of the ball when determining the agent closest to the ball.

When the ball is close to either end of the field we modify the definition of being in front of the ball to take into consideration that agents near the opponent's goal want to move the ball toward the center of the field (toward the opponent's goal) and agents near their own goal want to push the ball out to the sides (away from their goal). For this reason whenever the ball is to either side of the goal (its Y coordinate is outside the closest goal post's Y coordinate), and the ball's distance to the nearest endline is less than the distance between a goal post and the closest corner of the field to the goal post (approximately 6 meters), we declare an agent to be in front of the ball if on offense the agent is closer than the ball to the goal post nearest the ball, or on defense if the agent is farther than the ball from the goal post nearest to the ball.

Another situation in which we adjust the measure of the distance an agent is considered to be from the ball is when an agent has fallen. As it takes time after a fall for an agent to get back up, we add an extra 1.5 meters to the distance a fallen agent is considered to be from the ball. The only time we do not add in this extra

distance measure for a fallen agent is when the agent has fallen very near (within .65 meters) of the ball. In this case, when the fallen agent is almost on top of the ball, having another agent assume the *onBall* role will likely force that agent to have to navigate around and possibly trip over the fallen agent when moving toward the ball.

```
// Function for computing the adjusted distance (in meters)
// an agent is to the ball.
function getClosenessToBallMeasure(agent) {
  // Adjustment value to add to distance agent is from ball
  adjust = 0.0;

  // Agent has fallen but not right on top of ball
  if agentIsFallen and agentDistToBall > .65
    adjust += 1.5; // Added distance for having fallen

  // Ball is to the sides of the goals
  if abs(ball_Y) > HALF_GOAL_Y {
    // Ball close to own goal
    if ball_X < -HALF_FIELD_X + (HALF_FIELD_Y-HALF_GOAL_Y) {
      if ball_Y > 0
        nearestPost = Position(-HALF_FIELD_X, HALF_GOAL_Y);
      else
        nearestPost = Position(-HALF_FIELD_X, -HALF_GOAL_Y);

      // Agent is in front of ball
```

```
      if agentDistToNearestPost > ballDistToNearestPost
        adjust += 1.0; // Added distance to walk around ball
    }
    // Ball close to opponent's goal
    else if ball_X > HALF_FIELD_X - (HALF_FIELD_Y-HALF_GOAL_Y) {
      if ball_Y > 0
        nearestPost = Position(HALF_FIELD_X, HALF_GOAL_Y);
      else
        nearestPost = Position(HALF_FIELD_X, -HALF_GOAL_Y);

      // Agent is in front of ball
      if agentDistToNearestPost < ballDistToNearestPost
        adjust += 1.0; // Added distance to walk around ball
    }
  }
  // Agent is in front of ball
  else if agent_X >= ball_X
    adjust += 1.0; // Added distance to walk around ball

  return agentDistToBall + adjust;
}
```

D.1.2 Collision Avoidance

Although the positioning system discussed in Section 7.1 is designed to avoid assigning agents to positions that might cause them to collide, external factors outside of

the system's control, such as falls and the movement of the opposing team's agents, still result in occasional collisions. To minimize the potential for these collisions the agents employ an active collision avoidance system. When an obstacle, such as a teammate, is detected in an agent's path the agent will attempt to adjust its path to its target in order to maneuver around the obstacle. This adjustment is accomplished by defining two thresholds around obstacles: a *proximity* threshold at 1.25 meters and a *collision* threshold at .5 meters from an obstacle. If an agent enters the *proximity* threshold of an obstacle it will adjust its course to be tangent to the obstacle thereby choosing to circle around to the right or left of said obstacle depending on which direction will move the agent closer to its desired target. Should the agent get so close as to enter the *collision* proximity of an obstacle it must take decisive action to prevent an otherwise imminent collision from occurring. In this case the agent combines the corrective movement brought about by being in the *proximity* threshold with an additional movement vector directly away from the obstacle. Figure D.1 illustrates the adjusted movement of an agent when attempting to avoid a collision with an obstacle.

D.1.3 Ball Facing

When an agent is assigned to move to a new role position on the field, as described in Section 7.1, the agent both turns toward and moves to the new target position as described in Appendix A.1.1. Once the agent gets within .5 meters of its target role position it no longer attempts to face in the direction of its target position, however, and instead turns to face the ball as it finishes moving toward its target. This change in the position the agent is facing is done so that slight adjustments to an agent's target, brought about by small movement or noise in the position of the

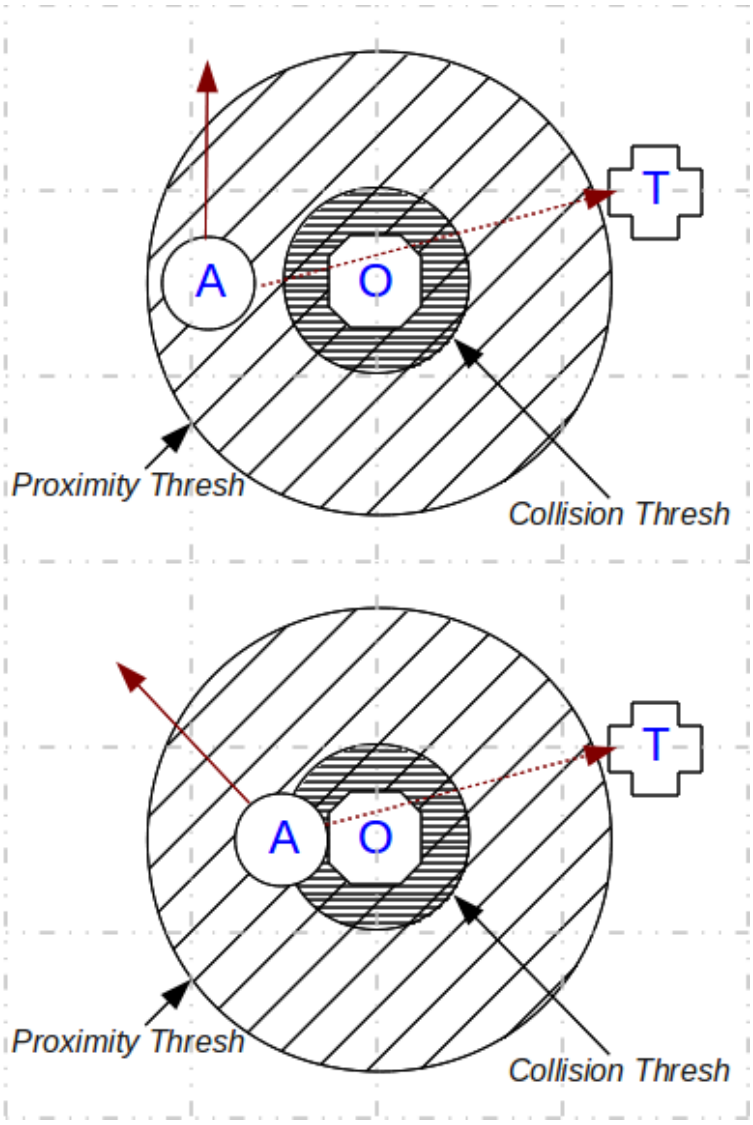


Figure D.1: Collision avoidance examples where agent A is traveling to target T but wants to avoid colliding with obstacle O. The top diagram shows how the agent's path is adjusted if it enters the *proximity* threshold of the obstacle while the bottom diagram depicts the agent's movement when entering the *collision* threshold. The dotted arrow is the agent's desired path while the solid arrow is the agent's corrected path to avoid a collision.

ball, do not result in sudden quick turns in place that might destabilize the agent as it adjusts its position to the revised target. Facing the ball when stopped also allows for agents to quickly move to the ball, without needing to turn, should they become the closest agent to the ball as determined in Appendix D.1.1.

D.1.4 Ball Approach

When the agent approaches the ball to dribble or kick the ball it moves toward a target position a little behind the ball that is in line with the direction the agent wants to move the ball. Should the agent be in front of the ball, meaning that if it were to walk straight to its desired target behind the ball it would end up walking through the ball, the agent instead picks a target to move to that is .5 meters either left or right from the ball along a line perpendicular to the direction from the agent to the ball. This target provides a waypoint for the agent to move through, along an efficient path around the ball, as opposed to directly walking up to the ball and then having to walk all the way around it.

If an opponent agent is within a meter of the ball, and the ball is between the opponent agent and our goal, it is likely that the opposing agent is going to move the ball toward our goal. If our agent were to walk straight toward the ball, and the opponent agent does start dribbling, chances are that the opponent agent will move the ball past our agent and need to be chased after. Our agent recognizes this situation when going to the ball and adjusts its target position to be further behind the ball, and along the anticipated path that the opponent agent is projected to dribble the ball, so as to be position to intercept the ball should the opponent agent move it. This target adjustment can be thought of as approaching the ball with a good angle for pursuit.

D.1.5 Reflex-based Strategy for Navigation with the Ball

By default our agents attempt to drive (move) the ball toward the opponent's goal with the target destination being the center of the opponent's goal. However, in many cases, the fastest way to drive a ball to this target point is different than just dribbling/kicking it directly to the target. For instance, when the agent is approximately aligned behind the ball facing the target direction, frequently it is faster to start dribbling the ball and slightly adjust the path of the ball while dribbling, then trying to align exactly in the target direction before starting to dribble. Other times when it is better to start dribbling the ball immediately, instead of waiting to align in the target direction, include when an opponent is close by, and there is no time to turn to face the exact target direction, and when an opponent is blocking the path to the target. This section describes a simple, reflex-based navigation strategy used when driving the ball.

The general idea behind this strategy is that given a desired target direction for the ball to move in, and given the agent's current direction facing the ball, a decision is made as to whether the agent should just move the ball in the direction of its current heading based on the current state of the game. This strategy is encapsulated in a function named *shouldMoveBallInCurrentDirection()* which returns `true` when the agent should move the ball forward along its current direction relative to the ball. This function is roughly implemented as follows:

```
function shouldMoveBallInCurrentDirection(agentDirection,
                                          desiredDirection, ...)
  if ballWouldGoInsideGoal
    return true;
  else if ballGoingOutsideFieldBounds
```

```
  return false;
else if agentDirection is too backwards
  return false;           // allow to dribble only mildly backwards
else if opponentsAreFar
  return false;                     // we have time to better align
else if opponentGetsClose
        and opponentDoesNotBlockAgentPath
        and goingForwardGetsBallCloserToOpponentGoal
  return true;
else if opponentIsNearBall
  return true;                // do not let opponent reach the ball
else
  return false;          // on all other cases, try to align better
```

Using this method, the agent is able to quickly navigate between obstacles, without the need for a complex path planning algorithm. Note that at each moment, the agent ignores all but the closest obstacle, making this a reflex-based strategy rather than a planning with lookahead strategy.

D.1.6 Dribbling

Dribbling the ball amounts to walking through the center of the ball in the desired direction that the agent wants to move the ball. When the agent is close to the ball, and is attempting to position itself behind the ball, it always faces the ball, as mentioned in Appendix A.1.1, so that it can quickly walk forward and move the ball should an opposing agent approach. When circling the ball to dribble the agent

uses collision avoidance (Appendix D.1.2) with a *proximity* threshold of .5 meters and a *collision* threshold of .35 meters to avoid running into the ball.

D.2 Kicking Strategy

The following subsections describe the UT Austin Villa team's strategy for kicking the ball using the team's walk engine discussed in Section 8.2.7. These strategy considerations include when to kick the ball in Appendix D.2.1, where to kick the ball and passing in Appendix D.2.2, and set plays in Appendix D.2.3.

D.2.1 When to Kick

Before deciding where to kick the ball, first a decision must be made as to whether to kick or dribble the ball. The 2014 UT Austin Villa team chose to always dribble if an opponent is within two meters of the ball—it was assumed that an agent might not have enough time to complete a kick if an opponent is less than two meters from the ball.

Rather than using a hand-picked value to determine if there is enough time to kick the ball, the 2015 UT Austin Villa trained a logistic regression classifier to predict the probability of a kick being successful given the current state of the world.[56] To do so, the team played many games against a common opponent in which agents were instructed to always try and kick the ball. During the course of kick attempts the following state features were recorded and then labeled as positive or negative kick examples based on whether kick attempts were successful.

1. Difference between angle of ball and the orientation of agent
2. Difference between angle of kick target and orientation of agent

[56] Thanks to Jason Liang for training the logistic regression classifier.

3. Angle difference between closest opponent to ball (OPP*) and ball from agent's point of view

4. Difference between angle of ball (from OPP*'s point of view) and the orientation of OPP*

5. Is OPP* fallen or not

6. Magnitude of OPP* velocity

7. Angle between OPP* velocity and ball velocity

8. Distance from agent to ball / OPP* distance to ball

9. Distance from agent to ball / OPP* distance to agent

10. OPP* distance to ball / OPP* distance to agent

11. Distance from agent to ball - OPP* distance to ball

12. Distance from agent to ball - OPP* distance to agent

13. OPP* distance to ball - OPP* distance to agent

14-24. Same features as 3-13 except OPP* is the second closest opponent to ball

25-35. Same features as 3-13 except OPP* is the third closest opponent to ball

The output from a trained classifier is a probability of a kick attempt being successful. A threshold value for this probability, for which kicks are attempted when the probability of a successful kick exceeds this value, was chosen after experimenting with different threshold values while playing 100s of games against multiple opponents. Metrics monitored during these games were average goal differential, number of kicks performed, goals against, and the probability of a tie or loss.

Figure D.2 shows example data for these metrics when playing against a common opponent using different classifier thresholds for deciding when to kick.

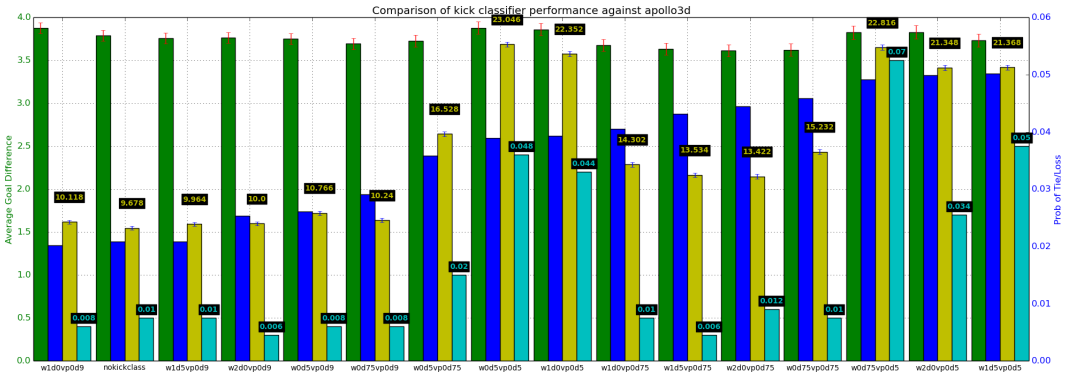


Figure D.2: Data for average goal differential (green), number of kicks performed (yellow), goals against (light blue), and the probability of a tie or loss (dark blue) when playing against the apollo3d team using different classifier thresholds for deciding when to kick.

D.2.2 Where to Kick the Ball and Passing

For the 2015 competition the UT Austin Villa team added a set of 13 new kicks to its agents with each of the kicks optimized to travel a fixed distance of 3 to 15 meters in 1 meter increments. This series of new variable distance kicks allow a robot to kick the ball within half a meter of any target 2.5 to 15.5 meters away. The kicks, represented as a series of parameterized joint angle poses as discussed in Section 8.2.6, were optimized using the CMA-ES algorithm [57] and the team's optimization framework incorporating overlapping layered learning presented in Chapters 3 and 4. During learning of a d meter kick the robot attempts to kick the ball to a target position d meters directly in front of the robot, and a kick attempt is awarded a negative fitness value equal to the euclidean distance of the ball relative to the target position. Each kick was optimized for 400 generations of CMA-ES with a population size of 150.

After optimization of each kick the top 300 highest fitness kick parameter sets were evaluated again over 300 kick attempts each to check for consistency. Finally, a parameter set with both high accuracy and low variance for the target distance was identified from collected data and chosen as the kick to use. This learning process was performed for each kick distance, and run across all five heterogeneous agent types, resulting in a total of 13 X 5 = 65 kicks learned.

Variable distance kicks allow for a richer set of passing options as robots can select from many potential targets to kick the ball to as shown in Figure D.3. When deciding where to kick the ball, the UT Austin Villa agent first checks to see if it can kick the ball and score from the ball's current location. If the agent thinks it can score then it tries to do so. If not, the agent then samples kicking the ball at targets in 10 degree direction increments and, for all viable kicking direction targets (those which don't kick the ball out of bounds or too far backwards), the agent assigns each a score based on Equation D.1. The location with the highest score is chosen as the location to kick the ball to. Equation D.1 rewards kicks for moving the ball toward the opponent's goal, penalizes kicks that have the ball end up near opponents, and also rewards kicks for landing near a teammate. All distances in Equation D.1 are measured in meters.

$$\begin{aligned} \texttt{score}(target) = \; & -\|opponentGoal - target\| \\ & \forall opp \in Opponents, -\max(25 - \|opp - target\|^2, 0) \\ & +\max(10 - \|closestTeammateToTarget - target\|, 0) \end{aligned} \tag{D.1}$$

Once an agent has decided on a target to kick the ball at it then broadcasts this target to its teammates. A couple agents then use "kick anticipation" where they run toward locations on the field that are good for receiving the ball based on

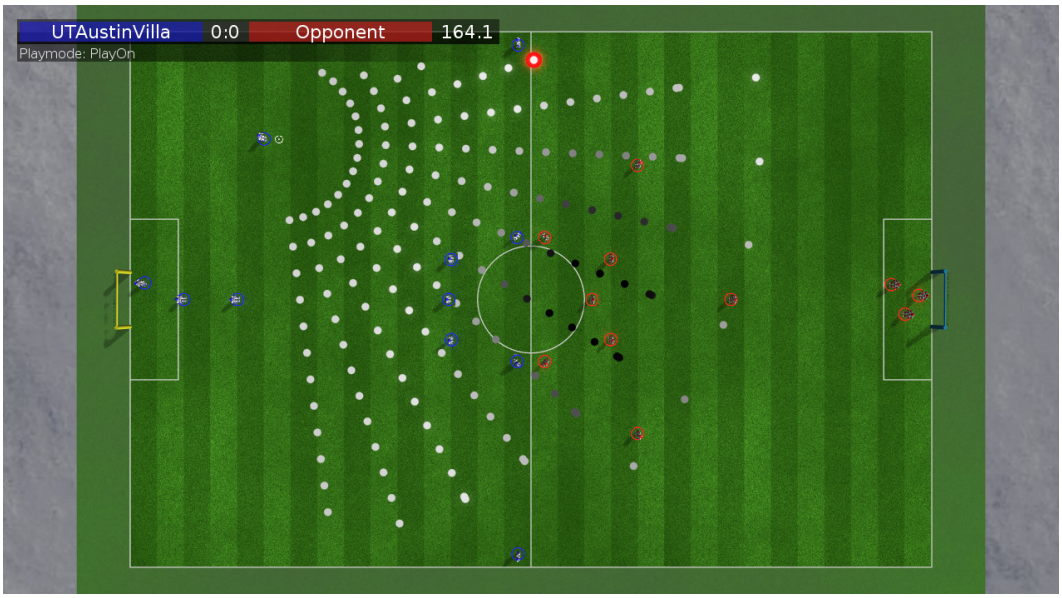


Figure D.3: Potential kick target locations with lighter circles having a higher score. The highest score location is highlighted in red.

the ball's anticipated location after it is kicked.[57] The agents assigned to run to these anticipated positions are chosen by a dynamic role assignment system described in Chapter 7. Also, agents avoid getting in the way of the projected trajectory of the ball before it is kicked to prevent them from accidentally blocking the kick.

D.2.3 Set Plays

During the 2014 RoboCup competition the UT Austin Villa team used a multi-robot behavior to score goals immediately off an indirect kickoff as discussed in Section 4.1.3. This behavior consisted of having one robot lightly touch the ball before a second robot kicked the ball into the opponent's goal. As rules were changed

[57] Videos of kick anticipation being used for passing can be found at
http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2014/html/kickanticipation.html

for the 2015 competition, and now a teammate is required to touch the ball outside of the center circle before a goal can be scored, this kickoff tactic is no longer allowed. Instead the team created legal set plays for kickoffs to try and quickly score.

The first kickoff set play, shown in Figure D.4, has the player taking the kickoff kick the ball slightly forward and to the left or right side of the field to a waiting teammate ready to run forward and take a shot. The player taking the kickoff chooses which side target to kick the ball to based on which target is furthest from any opponent. If there are opponents near both side targets then the player taking the kickoff instead chooses the kickoff set play shown in Figure D.5. In this set play the ball is first kicked backwards and to the side to a waiting teammate. The player who receives this backwards pass then kicks the ball forward and across to the other side of the field where a teammate is waiting for a pass. It is expected that the player who receives the second pass will be in a good position to take a shot on goal as opponent agents will have been drawn to the other side of the field after the initial backwards pass off the kickoff.

In addition to kickoff set plays, the UT Austin Villa team also created set plays for offensive corner kicks. These set plays, shown in Figure D.6, consist of having three teammates move to positions on the midline at the center and both sides of the field. The player taking the corner kick chooses to kick the ball to whichever of these three players is most open. If none of these players are open then the player taking the corner kick just chooses the default option of kicking the ball to a position in front of the goal where several teammates are waiting.

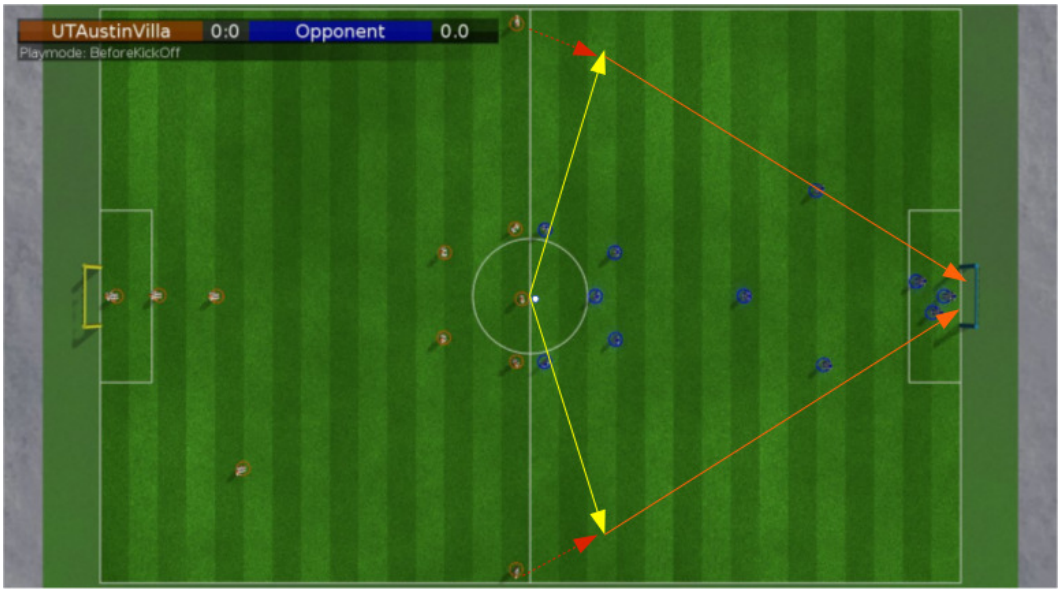


Figure D.4: Kickoff set play to the sides. Yellow lines represent passes and orange lines represent shots. Dashed red lines represent agent movement.

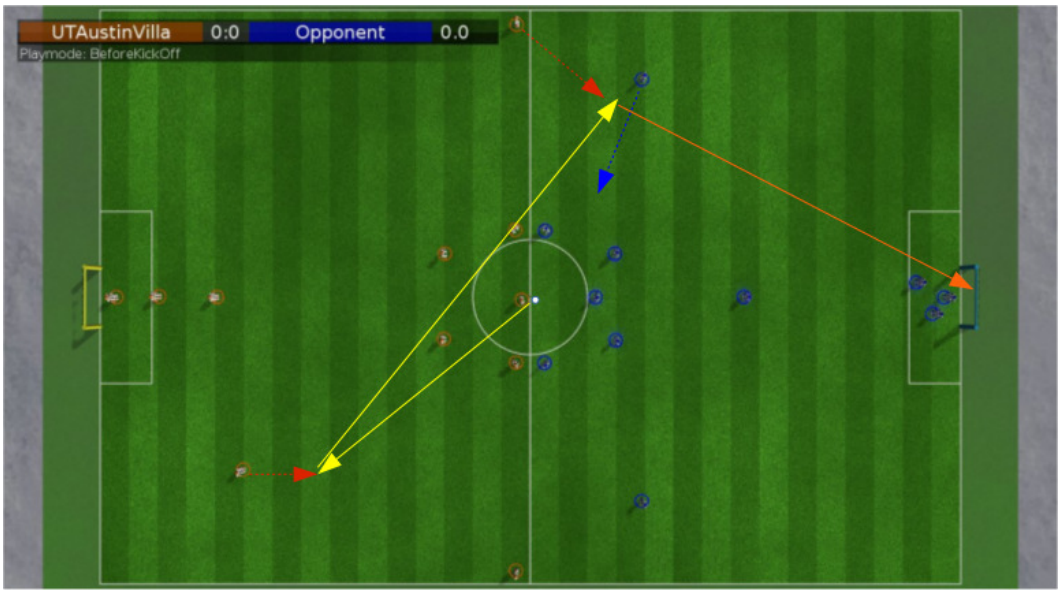


Figure D.5: Kickoff set play for passing backwards. Yellow lines represent passes and orange lines represent shots. Dashed lines represent agent movement (red for teammates and blue for opponents).

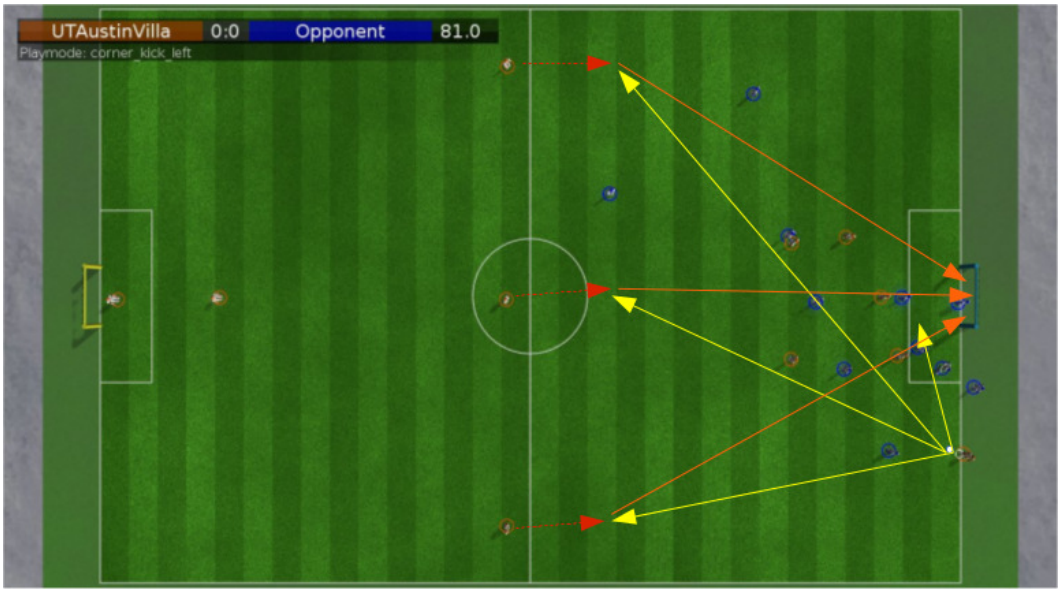


Figure D.6: Corner kick set plays. Yellow lines represent passes and orange lines represent shots. Dashed red lines represent teammate movement. In the example shown the ball would be passed to the teammate waiting for the ball near the bottom of the image as that teammate is most open.

All set plays require passing the ball to specific locations on the field though the use of learned variable distance kicks discussed in Appendix D.2.2. Approaching and kicking the ball must be quick as a team has only 15 seconds to kick the ball once a set play starts.

D.3 Goalie

The goalie is the last line of defense and is the only agent allowed to purposely dive to try and stop a ball when the opposing team shoots on goal. The following sections describe the behavior of the goalie.

D.3.1 Positioning

The goalie agent is designed to stay on a line .5 meters above its own goal line and always position itself between the ball and the goal so as to minimize the maximum angle between either goal post, ball, and the goalie. As the goalie moves it is instructed to always face the ball so that it can both keep track of the current position of the ball and also be in position to dive left or right at angles perpendicular to the direction of the ball for maximum angular coverage. Should the ball enter the goal box, and the goalie is determined to be the closest agent to the ball, the goalie will assume the *onBall* role (discussed in Section 7.1) and go to the ball. Otherwise the goalie is instructed to always stay within its goal box and position itself to best be ready to block shots.

D.3.2 Kalman Filter

The goalie needs to quickly and accurately respond to balls traveling in toward the goal. Because accuracy is paramount in the estimation of the ball's position, and the goalie needs a way of smoothing out noise present in observations of the ball's location, the goalie uses a Kalman filter to track the ball's position and velocity.

D.3.3 Dives

The goalie is equipped with a special set of diving skills in order to effectively use its body to stop a ball that is headed toward the goal. Since the goalie tracks the ball velocity with a Kalman filter (Appendix D.3.2), these dives are invoked only when the goalie evaluates that the ball is indeed headed with a certain threshold velocity toward the goal; otherwise, the goalie merely intercepts the ball by running toward it.

The key desiderata of a dive are that the goalie lower its body to the ground as quickly as possible, and that the angular range the goalie is able to "cut off" with its dive be as large as possible. We achieve these two objectives by designing three separate types of dives for the goalie; screenshots of these dives are depicted in Figure D.7. Figure D.7(a) shows a "central split", which results in the goalie reaching the ground with its legs split. This dive is programmed as a sequence of keyframes mainly manipulating joints in the robot's legs. Since the keyframes have left-right symmetry, the robot remains more-or-less centered at its original position when its legs split and touch the ground. Figure D.7(b) shows a slight variation, a "side-wards split" that is essentially the same as a central split, but by introducing slight asymmetry in the keyframes of the skill, results in a net displacement of the robot either to the left or the right. Both the central and the side-ward splits typically take less than 1.5 seconds to complete. The third diving skill, shown in Figure D.7(c), is a more human-like lateral lunge, which accomplishes significantly larger lateral coverage than the side-wards dive.

(a) Central split (b) Side-wards split (c) Lateral lunge

Figure D.7: Screenshots of the goalie diving.

The strategy controlling the goalie's dives is dependent on the Kalman filter's prediction of the ball's trajectory. Depending on the line predicted to be taken by the

ball, as well as the ball's speed, a manually designed set of rules determines whether a dive is to be undertaken, and if so, which of the five available dives (central split, left-wards split, right-wards split, left lunge, right lunge) is to be deployed.

Appendix E

RoboCup Competition Results

This appendix provides RoboCup competition results of the UT Austin Villa RoboCup 3D simulation team from 2011–2017. During those seven years the team won the competition six times while finishing second in 2013. Across those competitions the team accrued a record of 124 wins, 3 losses, and 6 ties, and scored a total of 640 goals while only conceding 7. In addition to the main RoboCup competition, the UT Austin Villa RoboCup 3D simulation team has also participated in and won the IranOpen RoboCup competition every year from 2012–2017. During those IranOpen competitions the team accrued an undefeated record of 70 wins and 3 ties, and scored a total of 348 goals without conceding any.

The relatively few number of games played at competitions, coupled with the complex and stochastic environment of the RoboCup 3D simulator, make it difficult to determine one team being better than another team by a statistically significant margin. At the end of every competition however, all teams are required to release their binaries used during the competition.[58] Using these released binaries, we are

[58] Released binaries from competitions are available at
https://chaosscripting.net/files/competitions/RoboCup/WorldCup/

able to play many games after competitions have ended to further analyze the results of competitions. In addition to the following sections providing competition results and analysis, Appendix E.8 lists members of the UT Austin Villa RoboCup 3D simulation team who have been instrumental in the team's success.

E.1 2011 RoboCup Competition

The main changes for the 2011 RoboCup competition format from the previous year's competition were to add players increasing the size of teams from 6 to 9, and to increase the size of the field from 18 m X 12 m to 21 m X 14 m in length and width.

UT Austin Villa won all 24 of its games during the RoboCup 2011 3D simulation competition, scoring 136 goals and conceding none [19].[59] In order to validate the results of the competition, in Table E.1 we show the performance of our team when playing 100 games against each of the other 21 teams' released binaries from the competition. UT Austin Villa won by at least an average goal difference of 1.45 against every team. Furthermore, of these 2100 games played to generate the data for Table E.1, our agent won all but 21 of them which ended in ties (no losses). The few ties were all against three of the better teams: apollo3d, boldhearts, and robocanes. We can therefore conclude that UT Austin Villa was the rightful champion of the competition.

UT Austin Villa's performance at the 2011 RoboCup competition was a massive improvement in the team's performance at the 2010 competition in which the team finished with a record of 4 wins, 6 losses, and 1 tie while scoring 11 goals and conceding 17. The primary reason for the team's improvement was the development

[59] Results of every game played by UT Austin Villa at the 2011 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup11

Table E.1: UT Austin Villa's 2011 released binary's performance when playing 100 games against the released binaries of all other teams at RoboCup 2011. This data includes place (the rank—or range of a rank if multiple teams were eliminated from the competition at the same time—a team achieved at the competition) and average goal difference (values in parentheses are the standard error).

Opponent	Place	Avg. Goal Diff.
apollo3	3	1.45 (0.11)
boldhearts	5-8	2.00 (0.11)
robocanes	5-8	2.40 (0.10)
cit3d	2	3.33 (0.12)
fcportugal3d	5-8	3.75 (0.11)
magmaoffenburg	9-12	4.77 (0.12)
oxblue	9-12	4.83 (0.10)
kylinsky	4	5.52 (0.14)
dreamwing3d	9-12	6.22 (0.13)
seuredsun	5-8	6.79 (0.13)
karachikoalas	13-18	6.79 (0.09)
beestanbul	9-12	7.12 (0.11)
nexus3d	13-18	7.35 (0.13)
hfutengine3d	13-18	7.37 (0.13)
futk3d	13-18	7.90 (0.10)
naoteamhumboldt	13-18	8.13 (0.12)
nomofc	19-22	10.14 (0.09)
kaveh/rail	13-18	10.25 (0.10)
bahia3d	19-22	11.01 (0.11)
l3msim	19-22	11.16 (0.11)
farzanegan	19-22	11.23 (0.12)

and optimization of an omnidirectional walk engine [103] (discussed in Section 8.2.5 and Appendix A.1). The 2011 team was able to beat a team using the 2010 team's fixed frame walk [172] by and average goal difference of 6.32 goals across 100 games with a standard error of 0.13.

E.2 2012 RoboCup Competition

The 2012 competition saw the addition of two more players to each team from 2011 allowing for full 11 vs 11 games, and also an increase in the size of the field from 21 m X 14 m to 30 m X 20 m in length and width.

In winning the 2012 RoboCup competition UT Austin Villa finished with a record of 12 wins, 2 losses, and 3 ties [104].[60] During the competition the team scored 39 goals and only conceded 4. This performance was not nearly as dominant of a performance as was seen in the 2011 competition when the team won all 24 games it played while scoring 136 goals and conceding none. Several reasons can be attributed to this dip in performance. There was a general decrease in goals scored during the tournament due to the larger field and increase in the number of agents on a team. Additionally early in the tournament there were network problems causing instability that resulted in many teams' agents losing their balance and having trouble walking. This instability was very noticeable during the first round when UT Austin Villa suffered both of its losses. UT Austin Villa eventually beat both the teams it lost to (magmaOffenburg and RoboCanes) during the semifinals and finals rounds. A large amount of credit must also be given to the other teams in the tournament as they exhibited a substantial improvement in overall play from the previous year.

[60] Results of every game played by UT Austin Villa at the 2012 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup12

As seen in Table E.2, the 2012 UT Austin Villa team was only able to beat the released binary of the 2012 2nd place team (RoboCanes) by an average of 0.88 goals and tied them 32 times across 100 games. Although the data in Table E.2 shows that UT Austin Villa winning the 2012 RoboCup competition was statistically significant, and that the team didn't lose any games or concede any goals when playing 100 games against the other top teams' released binaries, there was a decent chance of the tournament being decided by penalty kicks due to UT Austin Villa tying the 2nd place team almost 1/3 of the time. It is thus not surprising that the championship game wasn't decided until the second half of extra time (which UT Austin Villa won 2-0).

Table E.2: UT Austin Villa's 2012 released binary's performance when playing 100 games against the released binaries of the 2nd, 3rd, and 4th places teams at RoboCup 2012. This data includes place (the rank a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
RoboCanes	2	0.88 (0.08)	68-0-32	88/0
Bold Hearts	3	1.64 (0.09)	89-0-11	164/0
magmaOffenburg	4	1.87 (0.10)	94-0-6	187/0

The main changes to the 2012 UT Austin Villa team from the previous year were to improve the teams kickoff, formations, and getup behaviors [104]. The 2012 team, when computing the combined average goal difference after playing 100 games against each of the other three teams in the semifinals listed in Table E.2, had an average goal difference 0.54 goals greater than that of a version of the 2012 team without these improvements.

E.3 2013 RoboCup Competition

At the 2013 RoboCup competition two new robot body types were introduced: one with longer legs and arms and another with quicker moving feet. Teams were given the option of using up to three robots each of the new robot body types, however there was no requirement to do so.

UT Austin Villa took second place at the RoboCup 2013 competition finishing with a record of 19 wins, 1 loss, and 1 tie.[61] The team scored a total of 67 goals without conceding any until the final match which it narrowly lost 0-1. In order to validate the results of the competition, in Table E.3 we show the results of the UT Austin Villa team playing at least 100 games against each of the other participating teams' released binaries. UT Austin Villa had a positive average goal difference against all of the other teams, and only had a $\approx 5\%$ chance of losing to the the first and third place teams.

The main source of improvement from the previous year's team was the optimization of skills for the new robot body models. The 2013 team, when computing the combined average goal difference after playing 1000 games against the first and third place teams' release binaries listed in Table E.3, had an average goal difference 0.297 goals greater than that of the 2012 team when playing against the same binaries.

E.4 2014 RoboCup Competition

The 2014 RoboCup competition brought about the introduction of two additional robot body types: one with even longer arms and legs and another with a toe on

[61] Results of every game played by UT Austin Villa at the 2013 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup13

Table E.3: UT Austin Villa's 2013 released binary's performance when playing at least 100 games against the released binaries of all other teams at RoboCup 2013. This data includes place (the rank—or range of a rank if multiple teams were eliminated from the competition at the same time—a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
FCPortugal	3	0.465 (0.023)	459-52-489	633/168
Apollo3D	1	0.698 (0.027)	568-50-382	858/160
SEUJolly	4	1.133 (0.027)	772-13-215	1185/52
magmaOffenburg	5-8	1.447 (0.026)	887-0-113	1457/10
Bold Hearts	5-8	1.607 (0.029)	908-0-92	1607/0
RoboCanes	5-8	1.828 (0.031)	974-0-26	1830/2
Karachi Koalas	5-8	2.507 (0.031)	994-0-6	2509/2
ITAndroids	9-12	4.200 (0.080)	100-0-0	420/0
HfutEngine3D	9-12	4.530 (0.086)	100-0-0	453/0
Photon	9-12	4.590 (0.081)	100-0-0	459/0
ODENS	13-16	4.820 (0.092)	100-0-0	482/0
FUT-K	13-16	5.440 (0.084)	100-0-0	544/0
Paydar3D	9-12	5.990 (0.099)	100-0-0	599/0
L3MSIM	13-16	6.050 (0.098)	100-0-0	605/0
Mithras3D	13-16	8.330 (0.098)	100-0-0	833/0
Bahia3D	17	9.800 (0.110)	100-0-0	980/0

each foot. Unlike in the previous year's competition, teams were required to use different robot body types as described in Section 2.2.

In winning the 2014 RoboCup competition UT Austin Villa finished with an undefeated record of 13 wins and 2 ties [105].[62] During the competition the team scored 52 goals without conceding any. In order to validate the results of the competition, in Table E.4 we show results of UT Austin Villa playing 1000 games against each of the other 11 teams' released binaries from the competition.

UT Austin Villa finished with at least an average goal difference greater than

[62]Results of every game played by UT Austin Villa at the 2014 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup14

Table E.4: UT Austin Villa's 2014 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2014. This data includes place (the rank—or range of a rank if multiple teams were eliminated from the competition at the same time—a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, goals for/against, and the percentage of own kickoffs which the team scored from.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)	KO Score %
BahiaRT	5-8	2.075 (0.030)	990-0-10	2092/17	96.2
FCPortugal	4	2.642 (0.034)	986-0-14	2748/106	83.4
magmaOffenburg	3	2.855 (0.035)	990-0-10	2864/9	88.3
RoboCanes	2	3.081 (0.046)	974-0-26	3155/74	69.4
FUT-K	5-8	3.236 (0.039)	998-0-2	3240/4	96.3
SEU_Jolly	5-8	4.031 (0.062)	995-0-5	4034/3	87.6
KarachiKoalas	9-12	5.681 (0.046)	1000-0-0	5682/1	87.5
ODENS	9-12	7.933 (0.041)	1000-0-0	7933/0	92.1
HfutEngine	5-8	8.510 (0.050)	1000-0-0	8510/0	94.7
Mithras3D	9-12	8.897 (0.041)	1000-0-0	8897/0	90.4
L3M-SIM	9-12	9.304 (0.043)	1000-0-0	9304/0	93.7

two goals against every opponent. Additionally UT Austin Villa did not lose a single game out of the 11,000 that were played in Table E.4. These game results show that UT Austin Villa winning the 2014 competition was far from a chance occurrence. UT Austin Villa also won the league technical challenge—introduced for the first time—consisting of three separate challenges: running robot challenge (first place), drop-in player challenge (first place), and free challenge (second place) [105].

A key to the team winning the competition was the use of overlapping layered learning as described in Chapter 4, and in particular new longer kicks with kick anticipation for passing (described in Appendix D.2.2), the ability to score on kickoffs (discussed in Section 4.1.3), and use of the robot body model with toes increased the team's performance [105]. The 2014 UT Austin Villa team improved substantially from the previous year as it was able to beat the team's 2013 second place binary by an average of 1.525 goals (with a standard error of 0.034) over 1000 games, and

also beat the 2013 first place team (Apollo3D) by an average of 2.726 goals (with a standard error of 0.041) across 1000 games.

E.5 2015 RoboCup Competition

The primary change for the 2015 RoboCup competition was to require that an opponent first touches the ball, or a teammate touches the ball outside the center circle, before a goal can be scored on a kickoff. This rule was added to prevent the ability to use multiagent behaviors to score directly off kickoffs as described in Section 4.1.3.

In winning the 2015 RoboCup competition UT Austin Villa finished with a perfect record of 19 wins and no losses.[63] During the competition the team scored 87 goals while only conceding 1. In order to validate the results of the competition, in Table E.5 we show results of UT Austin Villa playing 1000 games against each of the other 11 teams' released binaries from the competition.

UT Austin Villa finished with at least an average goal difference greater than two goals against every opponent. Additionally UT Austin Villa only lost 7 games out of the 11,000 that were played in Table E.5 with a win percentage greater than 92% against all teams. These results show that UT Austin Villa winning the 2015 competition was far from a chance occurrence. UT Austin Villa also won the league technical challenge consisting of three separate challenges (the team took first place in each of them): free challenge, kick accuracy challenge, and drop-in player challenge [108].

A large factor in UT Austin Villa's success in 2015 was improvements in kicking and the coordination of set plays described in Appendix D.2.3 [108]. The

[63] Results of every game played by UT Austin Villa at the 2015 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup15

Table E.5: UT Austin Villa's 2015 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2015. This data includes place (the rank a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
FUT-K	2	2.082 (0.036)	927-2-71	2178/96
FCPortugal	3	2.399 (0.040)	945-4-51	2624/225
BahiaRT	4	2.496 (0.044)	944-1-55	2501/5
Apollo3D	5	3.803 (0.046)	995-0-5	3805/2
magmaOffenburg	6	4.167 (0.051)	999-0-1	4171/4
RoboCanes	7	4.187 (0.049)	998-0-2	4235/48
Nexus3D	8	5.571 (0.044)	1000-0-0	5573/2
CIT3D	9	6.321 (0.050)	1000-0-0	6321/0
ITAndroids	11	10.125 (0.041)	1000-0-0	10125/0
Miracle3D	12	10.521 (0.056)	1000-0-0	10521/0
HfutEngine3D	10	11.897 (0.068)	1000-0-0	11897/0

2015 UT Austin Villa team improved dramatically from 2014 as it was able to beat a version of the team's 2014 champion binary that does not attempt the now illegal behavior of scoring on a kickoff by an average of 1.838 goals (with a standard error of 0.047) across 1000 games.

E.6 2016 RoboCup Competition

The main changes for the 2016 RoboCup competition were to penalize robots for charging into each other through the use of an automated referee foul model, and also to make free kicks indirect—a goal can not be scored until a second player touches the ball after it is kicked. These changes promoted better play by discouraging players from running into each other and encouraging more passing respectively.

In winning the 2016 RoboCup competition UT Austin Villa finished with a

perfect record of 14 wins and no losses.[64] During the competition the team scored 88 goals while only conceding 1. In order to validate the results of the competition, in Table E.6 we show results of UT Austin Villa playing 1000 games against each of the other eight teams' released binaries from the competition.

Table E.6: UT Austin Villa's 2016 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2016. This data includes place (the rank a team achieved at the 2016 competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
FUT-K	2	1.809 (0.036)	888-3-109	1872/63
FCPortugal	3	2.431 (0.040)	954-1-45	2452/21
BahiaRT	4	3.123 (0.040)	985-0-15	3123/0
magmaOffenburg	5	3.921 (0.049)	996-0-4	3926/5
KgpKubs	8	7.728 (0.046)	1000-0-0	7729/1
ITAndroids	6	9.022 (0.053)	1000-0-0	9024/2
HfutEngine3D	9	10.192 (0.056)	1000-0-0	10192/0
Miracle3D	7	11.126 (0.059)	1000-0-0	11126/0

UT Austin Villa finished with at least an average goal difference greater than 1.8 goals against every opponent. Additionally UT Austin Villa only lost 4 games out of the 8000 that were played in Table E.6 with a win percentage greater than 88% against all teams. These results show that UT Austin Villa winning the 2016 competition was far from a chance occurrence. UT Austin Villa also won the league technical challenge consisting of three separate challenges (the team took first place in each of them): free challenge, keepaway challenge, and Gazebo running robot challenge [114].

A critical component in UT Austin Villa's success in 2016 was the incorporation of a marking system using prioritized role assignment [113] detailed in

[64] Results of every game played by UT Austin Villa at the 2016 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup16

Section 7.3. The 2016 UT Austin Villa team also improved dramatically from 2015 as it was able to beat the team's 2015 champion binary by an average of 0.561 goals (with a standard error of 0.029) across 1000 games.

E.7 2017 RoboCup Competition

Outside of a few bug fixes to the server, the removal of crowding rules (previously too many players crowded around the ball caused a player to be penalized and beamed to the sideline), and a couple small changes to the charging foul model, the 2017 RoboCup competition environment was unchanged from the previous year.

In winning the 2017 RoboCup competition UT Austin Villa finished with a perfect record of 23 wins and no losses.[65] During the competition the team scored 171 goals while conceding none. UT Austin Villa also won the league technical challenge consisting of three separate challenges (the team took first place in each of them): free challenge, Gazebo running robot challenge, and passing and scoring challenge. As of the time of writing this dissertation the binaries of the other teams participating in the 2017 competition have yet to be released, and thus we are not yet able to compare the performance of our team when playing thousands of games against the other teams' released binaries.

An important factor in UT Austin Villa's success in 2017 was the introduction of a fast walking kick that does not require the robot to assume a standing position before kicking, takes less than 0.25 seconds to execute, and can kick the ball over 18 meters. The 2017 UT Austin Villa team improved dramatically from 2016 as it was able to beat the team's 2016 champion binary by an average of 1.339 goals (with a standard error of 0.039) across 1000 games.

[65] Results of every game played by UT Austin Villa at the 2017 RoboCup competition available at http://www.cs.utexas.edu/~AustinVilla/?p=competitions/RoboCup17

E.8 UT Austin Villa RoboCup 3D Simulation Team Members

The following is a list of former and past members of the UT Austin Villa RoboCup 3D simulation team—their combined contributions have been instrumental to the success of the team:

- Patrick MacAlpine - Graduate Student (2010-present)
- Peter Stone - Professor (2007-present)
- Min Bi - Undergraduate Student (2016)
- Mahmut Tarik Ozkaya - Undergraduate Student (2016)
- Jordan Torres - Undergraduate Student (2016)
- Matt Union - Undergraduate Student (2016)
- Xinyi Wang - Undergraduate Student (2016)
- Josiah Hanna - Graduate Student (2015)
- Jason Liang - Graduate Student (2014-2015)
- Samuel Barrett - Graduate Student (2011-2014)
- Mike Depinet - Undergraduate Student (2013-2014)
- Andrew Sharp - Undergraduate Student (2013)
- Nick Collins - Undergraduate Student (2011-2012)
- Adrian Lopez-Mobilia - Undergraduate Student (2011-2012)

- Michael Quinlan - Research Scientist (2011)
- Shivaram Kalyanakrishnan - Graduate Student (2007-2011)
- Daniel Urieli - Graduate Student (2010-2011)
- Frank Barrera - Undergraduate Student (2011)
- Art Richards - Undergraduate Student (2011)
- Nicu Stiurca - Undergraduate Student (2011)
- Victor Vu - Undergraduate Student (2011)
- Yinon Bentor - Graduate Student (2009-2010)

Appendix F

Acronyms

Acronym	Definition
CILB	Combining Independently Learned Behaviors (Section 3.3)
CLL	Concurrent Layered Learning (Section 3.3)
CMA-ES	Covariance Matrix Adaptation Evolution Strategy (Section 2.1.2)
MMDR	Minimum Maximal Distance Recursive (Section 5.3.1)
MMD+MSD^2	Minimum Maximal Distance + Minimum Sum Distance^2 (Section 5.3.2)
MSD	Minimize Sum of Distances (Section 5.4)
MSD^2	Minimize Sum of Distances^2 (Section 5.4)
PCLL	Partial Concurrent Layered Learning (Section 3.3)
PLLR	Previous Learned Layer Refinement (Section 3.3)
RL	Reinforcement Learning (Section 2.1)
SCRAM	Scalable Collision-avoiding Role Assignment with Minimal-makespan (Chapter 5)
SLL	Sequential Layered Learning (Section 3.3)

Appendix G

Online Materials

UT Austin Villa RoboCup 3D simulation team homepage:

http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/

Information and videos on overlapping layered learning:

http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/overlappingLayeredLearning.html

Information and videos of SCRAM role assignment in action, as well as C++ implementations of the role assignment algorithms:

http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/scram.html

Information and videos of SCRAM prioritized role assignment and the marking system in action:

http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/marking.html

UT Austin Villa RoboCup 3D simulation base code release:

https://github.com/LARG/utaustinvilla3d

Bibliography

[1] More robots coming to u.s. factories. http://www.usatoday.com/story/money/2015/02/09/bcg-report-on-factory-robots/23143259/. Accessed: 2015-04-12. 1, 140

[2] A. Abdolmaleki, N. Lau, L. P. Reis, J. Peters, and G. Neumann. Contextual policy search for generalizing a parameterized biped walking controller. In *Autonomous Robot Systems and Competitions (ICARSC), 2015 IEEE International Conference on*, pages 17–22. IEEE, 2015. 133, 148

[3] A. Abdolmaleki, N. Lau, L. P. Reis, J. Peters, and G. Neumann. Contextual policy search for linear and nonlinear generalization of a humanoid walking controller. *Journal of Intelligent and Robotic Systems*, 83(3-4):393–408, 2016. 133

[4] A. Abdolmaleki, B. Price, N. Lau, L. P. Reis, and G. Neumann. Deriving and improving cma-es with information geometric trust regions. 2017. 132

[5] A. Abdolmaleki, D. Simoes, N. Lau, L. P. Reis, and G. Neumann. Learning a humanoid kick with controlled distance. In S. Behnke, D. D. Lee, S. Sariel, and R. Sheh, editors, *RoboCup 2016: Robot Soccer World Cup XX*, LNAI. Springer, 2016. 133, 148

[6] S. Abeyruwan, A. Seekircher, and U. Visser. Dynamic role assignment using general value functions. In *IEEE Humanoid Robots, HRS workshop*, 2012. 139

[7] A. K. Abu-Affash, P. Carmi, M. J. Katz, and Y. Trabelsi. Bottleneck non-crossing matching in the plane. In *Algorithms–ESA 2012*. Springer, 2012. 65

[8] K. Adolph, W. Cole, M. Komati, J. Garciaguirre, D. Badaly, J. Lingeman, G. Chan, and R. Sotsky. How do you learn to walk? thousands of steps and dozens of falls per day. *Psychological Science*. 146

[9] S. Akella. Assignment algorithms for variable robot formations. In P. Abbeel, K. Bekris, K. Goldberg, and L. Miller, editors, *12th International Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, Dec. 2016. 138

[10] Y. Akimoto, Y. Nagata, I. Ono, and S. Kobayashi. Theoretical foundation for cma-es from information geometry perspective. *Algorithmica*, pages 1–19, 2012. 14, 131

[11] H. Akiyama, K. Dorer, and N. Lau. On the progress of soccer simulation leagues. In *Robot Soccer World Cup*, pages 599–610. Springer, 2014. 16

[12] H. Akiyama, D. Katagami, and K. Nitta. Team formation construction using a gui tool in the robocup soccer simulation. In *SCIS & ISIS SCIS & ISIS 2006*, pages 80–85. Japan Society for Fuzzy Theory and Intelligent Informatics, 2006. 154

[13] H. Akiyama and T. Nakashima. Helios base: An open source package for the robocup soccer 2d simulation. In S. Behnke, M. Veloso, A. Visser, and R. Xiong, editors, *RoboCup 2013: Robot World Cup XVII*, pages 528–535. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 93, 124

[14] H. Akiyama and I. Noda. Multi-agent positioning mechanism in the dynamic

environment. In *RoboCup 2007: Robot Soccer World Cup XI*, pages 377–384. Springer, 2008. 79, 93, 154

[15] J. Alonso-Mora, A. Breitenmoser, M. Rufli, R. Siegwart, and P. Beardsley. Image and animation display with multiple mobile robots. *The International Journal of Robotics Research*, 31(6):753–773, 2012. 52, 137, 159

[16] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37(4):237–240, 1991. 63

[17] E. Angel. *Interactive Computer Graphics*. Pearson Education, Inc., 5th edition, 2009. 121

[18] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469 – 483, 2009. 3, 133

[19] A. Bai, X. Chen, P. MacAlpine, D. Urieli, S. Barrett, and P. Stone. Wright Eagle and UT Austin Villa: RoboCup 2011 simulation league champions. In T. Roefer, N. M. Mayer, J. Savage, and U. Saranli, editors, *RoboCup-2011: Robot Soccer World Cup XV*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2012. 219

[20] S. Behnke, M. Schreiber, J. Stückler, R. Renner, and H. Strasdat. See, walk, and kick: Humanoid robots start to play soccer. In *Proc. of the Sixth IEEE-RAS Int. Conf. on Humanoid Robots (Humanoids 2006)*, pages 497–503. IEEE, 2006. 15

[21] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013. 158

[22] Y. Bengio, I. J. Goodfellow, and A. Courville. Deep learning. *Nature*, 521:436–444, 2015. 132, 154

[23] D. P. Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of operations research*, 14(1):105–123, 1988. 152

[24] J. Boedecker and M. Asada. Simspark–concepts and application in the robocup 3d soccer simulation league. *Autonomous Robots*, pages 174–181, 2008. 16

[25] M. Broucke. Disjoint path algorithms for planar reconfiguration of identical vehicles. In *American Control Conference*, 2003. 53, 65

[26] M. Broucke. Reconfiguration of identical vehicles in 3d. In *Decision and Control. IEEE Conf. on*, 2003. 53

[27] R. E. Burkard and F. Rendl. Lexicographic bottleneck problems. *Operations Research Letters*, 10(5), 1991. 59

[28] J. G. Carlsson and B. Armbruster. A bottleneck matching problem with edge-crossing constraints, 2010. 65

[29] L. Chaimowicz, M. F. Campos, and V. Kumar. Dynamic role assignment for cooperative robots. In *Robotics and Automation. Proceedings. ICRA'02. IEEE International Conference on*, 2002. 51, 137

[30] W. Chen and T. Chen. Multi-robot dynamic role assignment based on path cost. In *2011 Chinese Control and Decision Conference (CCDC)*, 2011. 82, 139

[31] A. Cherubini, F. Giannone, and L. Iocchi. Layered learning for a soccer legged robot helped with a 3d simulator. In *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *Lecture Notes in Computer Science*, pages 385–392. Springer Berlin Heidelberg, 2008. 134

[32] S. Chopra and M. Egerstedt. Multi-robot routing under connectivity constraints. *IFAC Proceedings Volumes*, 45(26):67–72, 2012. 137

[33] S. Chopra and M. Egerstedt. Heterogeneous multi-robot routing. In *American Control Conference (ACC), 2014*, pages 5390–5395. IEEE, 2014. 137

[34] S. Chopra and M. Egerstedt. Spatio-temporal multi-robot routing. *Automatica*, 60:173–181, 2015. 137

[35] F. D. Croce, V. T. Paschos, and A. Tsoukias. An improved general procedure for lexicographic bottleneck problems. *Operations research letters*, 24(4):187–194, 1999. 59

[36] M. Cutler, T. J. Walsh, and J. P. How. Reinforcement learning with multi-fidelity simulators. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3888–3895. IEEE, 2014. 2, 133

[37] B. C. da Silva, G. Baldassarre, G. Konidaris, and A. Barto. Learning parameterized motor skills on a humanoid robot. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 5239–5244. IEEE, 2014. 3, 133, 147, 148

[38] B. C. da Silva, G. Konidaris, and A. G. Barto. Learning parameterized skills. 2012. 148

[39] M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011. 131, 133

[40] M. P. Deisenroth, G. Neumann, J. Peters, et al. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142, 2013. 132

[41] M. Depinet, P. MacAlpine, and P. Stone. Keyframe sampling, optimization, and behavior integration: Towards long-distance kicking in the robocup 3d simulation league. In H. L. Akin, R. A. C. Bianchi, S. Ramamoorthy, and K. Sugiura, editors, *RoboCup-2014: Robot Soccer World Cup XVIII*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2015. 43, 124, 125, 165, 166, 167, 169

[42] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July 2008. 121

[43] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13:227–303, 2000. 135

[44] X. Dong, B. Yu, Z. Shi, and Y. Zhong. Time-varying formation control for unmanned aerial vehicles: Theories and applications. *IEEE Transactions on Control Systems Technology*, 23(1):340–348, 2015. 159

[45] Y. Elmaliach, N. Agmon, and G. A. Kaminka. Multi-robot area patrol under

frequency constraints. *Annals of Mathematics and Artificial Intelligence*, 57(3-4):293–320, 2009. 159

[46] A. Farchy, S. Barrett, P. MacAlpine, and P. Stone. Humanoid robots learning to walk faster: From the real world to simulation and back. In *Proc. of 12th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2013. 2, 133, 156

[47] D. Ford and D. R. Fulkerson. *Flows in networks*. Princeton university press, 2010. 61

[48] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, pages 343–349, 1999. 102

[49] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. 152

[50] A. Goldberg and R. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 7–18. ACM, 1987. 138

[51] C. Graf, A. Härtl, T. Röfer, and T. Laue. A robust closed-loop gait for the standard platform league humanoid. In *Proc. of the 4th Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS Int. Conf. on Humanoid Robots*, pages 30 – 37, 2009. 109, 110, 114

[52] S. M. Gustafson and W. H. Hsu. *Layered learning in genetic programming for a cooperative robot soccer problem*. Springer, 2001. 134, 159

[53] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003. 148

[54] J. Hanna and P. Stone. Grounded action transformation for robot learning in simulation. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, February 2017. 2, 133, 157

[55] J. P. Hanna, M. Albert, D. Chen, and P. Stone. Minimum cost matching for autonomous carsharing. In *Proceedings of the 9th IFAC Symposium on Intelligent Autonomous Vehicles (IAV 2016)*, June 2016. 160

[56] N. Hansen. *The CMA Evolution Strategy: A Tutorial*, January 2009. http://www.lri.fr/~hansen/cmatutorial.pdf. 14

[57] N. Hansen, S. Müller, and P. Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, 11(1):1–18, 2003. 13, 36, 119, 130, 163, 180, 209

[58] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014. 158

[59] T. Henn, J. Henrio, and T. Nakashima. Optimizing players formations for corner-kick situations in robocup soccer 2d simulation. *Artificial Life and Robotics*, pages 1–5, 2017. 155

[60] T. Hester and P. Stone. Negative information and line observations for monte carlo localization. In *IEEE Int. Conf. on Robotics and Automation*, May 2008. 103

[61] T. Hester and P. Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *Machine learning*, 90(3):385–429, 2013. 133

[62] N. T. Hien, N. X. Hoai, and B. McKay. A study on genetic programming with layered learning and incremental sampling. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 1179–1185. IEEE, 2011. 159

[63] M. H. Hindsley. The marching band. *Music Supervisors' Journal*, 17(2):15–17, 1930. 159

[64] P. F. Hokayem, M. W. Spong, and D. D. Siljak. Cooperative avoidance control for multiagent systems. *Urbana*, 51:61801, 2007. 51, 137

[65] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. 61

[66] D. Jackson and A. Gibbons. Layered learning in boolean gp problems. In *Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 148–159. Springer Berlin Heidelberg, 2007. 134

[67] S. Jaishy, Y. Fukushige, N. Ito, K. Iwata, and Y. Kawabe. An evaluation of bar: Breakdown agent replacement algorithm for scram. In *Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD), 2016 4th Intl Conf on*, pages 319–324. IEEE, 2016. 139

[68] S. Jaishy, Y. Fukushige, N. Ito, K. Iwata, and Y. Kawabe. Assessment of bar: Breakdown agent replacement algorithm for scram. *International Journal of Software Innovation (IJSI)*, 5(3):1–17, 2017. 139

[69] S. Jaishy, N. Ito, and Y. Kawabe. Bar: Breakdown agent replacement algorithm for scram. In *Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence (ACIT-CSI), 2015 3rd International Conference on*, pages 393–398. IEEE, 2015. 139

[70] M. Ji, S.-i. Azuma, and M. B. Egerstedt. Role-assignment in multi-agent coordination. *International Journal of Assistive Robotics and Mechatronics*, (1):32–40, 2006. 51, 137

[71] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011. 145

[72] L. Johnson and D. Ballard. Classifying movements using efficient kinematic codes. *Annual Meeting of the Cognitive Science Society*, 2014. 132

[73] L. Johnson and D. H. Ballard. Efficient codes for inverse dynamics during walking. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014. 132

[74] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016. 158

[75] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987. 152

[76] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 135

[77] S. Kalyanakrishnan and P. Stone. Learning complementary multiagent behaviors: A case study. In *RoboCup 2009: Robot Soccer World Cup XIII*, 2010. 15

[78] I. Karpov, J. Sheblak, and R. Miikkulainen. Opennero: A game platform for ai research and education. In *AIIDE*, 2008. 158

[79] I. V. Karpov, V. K. Valsalam, and R. Miikkulainen. Assisting machine learning through shaping, advice and examples. In *IJCAI Workshop on Agents Learning Interactively from Human Teachers (ALIHT)*, July 2011. 158

[80] H. Kimura, Y. Fukuoka, and K. Konaga. Adaptive dynamic walking of a quadruped robot using a neural system model. *Advanced Robotics*, 15(8):859–878, 2001. 132

[81] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In *Systems, Man, and Cybernetics. IEEE Int. Conf. on*, 1999. 50

[82] J. Kober and J. Peters. Reinforcement learning in robotics: A survey. In M. Wiering and M. van Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 579–610. Springer Berlin Heidelberg, 2012. 132

[83] J. Kober and J. R. Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009. 131

[84] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004. (IROS*

2004). Proceedings. 2004 IEEE/RSJ International Conference on, volume 3, pages 2149–2154 vol.3, Sept 2004. 128, 149

[85] S. Koos, J.-B. Mouret, and S. Doncieux. Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 119–126. ACM, 2010. 2, 133

[86] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. 57, 84

[87] A. Kupcsik, M. P. Deisenroth, J. Peters, A. P. Loh, P. Vadakkepat, and G. Neumann. Model-based contextual policy search for data-efficient generalization of robot skills. *Artificial Intelligence*, 2014. 133, 147, 148

[88] N. Lau, L. Lopes, G. Corrente, and N. Filipe. Multi-robot team coordination through roles, positionings and coordinated procedures. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)*, oct. 2009. 139

[89] T. Laue, K. Spiess, and T. Röfer. Simrobot-a general physical robot simulator and its application in robocup. In *RoboCup*, pages 173–183. Springer, 2005. 150

[90] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001. 51, 137

[91] J. Lehman, K. O. Stanley, and R. Miikkulainen. Effective diversity maintenance in deceptive domains. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2013*, 2013. 133

[92] D. L. Leottau, J. R. del Solar, P. MacAlpine, and P. Stone. A study of layered learning strategies applied to individual behaviors in robot soccer. In L. Almeida, J. Ji, G. Steinbauer, and S. Luke, editors, *RoboCup-2015: Robot Soccer World Cup XIX*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2016. 134

[93] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015. 132

[94] S. Levine and V. Koltun. Guided policy search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1–9, 2013. 132

[95] R. Lioutikov, G. Neumann, G. Maeda, and J. Peters. Probabilistic segmentation applied to an assembly task. In *Proceedings of the International Conference on Humanoid Robots (HUMANOIDS)*, 2015. 149

[96] L. Liu and D. A. Shell. Optimal market-based multi-robot task allocation via strategic pricing. In *Robotics: Science and Systems*, 2013. 152

[97] I. Loshchilov, M. Schoenauer, and M. Sebag. Comparison-based optimizers need comparison-based surrogates. In *Parallel Problem Solving from Nature, PPSN XI*, pages 364–373. Springer, 2010. 146

[98] I. Loshchilov, M. Schoenauer, and M. Sebag. Self-adaptive surrogate-assisted covariance matrix adaptation evolution strategy. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 321–328. ACM, 2012. 147

[99] I. Loshchilov, M. Schoenauer, and M. Sebag. Intensive surrogate model exploitation in self-adaptive surrogate-assisted cma-es (saacm-es). In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 439–446. ACM, 2013. 147

[100] H. Ma and S. Koenig. Optimal target assignment and path finding for teams of agents. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 1144–1152. International Foundation for Autonomous Agents and Multiagent Systems, 2016. 138

[101] Z. Ma and S. Akella. Coordination of droplets on light-actuated digital microfluidic systems. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2510–2516. IEEE, 2012. 159

[102] P. MacAlpine, F. Barrera, and P. Stone. Positioning to win: A dynamic role assignment and formation positioning system. In X. Chen, P. Stone, L. E. Sucar, and T. V. der Zant, editors, *RoboCup-2012: Robot Soccer World Cup XVI*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2013. 79, 82, 124, 195

[103] P. MacAlpine, S. Barrett, D. Urieli, V. Vu, and P. Stone. Design and optimization of an omnidirectional humanoid walk: A winning approach at the RoboCup 2011 3D simulation competition. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*, July 2012. 35, 38, 96, 124, 125, 146, 168, 180, 185, 221

[104] P. MacAlpine, N. Collins, A. Lopez-Mobilia, and P. Stone. UT Austin Villa: RoboCup 2012 3D simulation league champion. In X. Chen, P. Stone, L. E. Sucar, and T. V. der Zant, editors, *RoboCup-2012: Robot Soccer World Cup*

XVI, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2013. 38, 78, 81, 96, 97, 164, 165, 221, 222

[105] P. MacAlpine, M. Depinet, J. Liang, and P. Stone. UT Austin Villa: RoboCup 2014 3D simulation league competition and technical challenge champions. In H. L. Akin, R. A. C. Bianchi, S. Ramamoorthy, and K. Sugiura, editors, *RoboCup-2014: Robot Soccer World Cup XVIII*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2015. 48, 86, 96, 97, 125, 175, 176, 224, 225

[106] P. MacAlpine, M. Depinet, and P. Stone. UT Austin Villa 2014: RoboCup 3D simulation league champion via overlapping layered learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI)*, volume 4, pages 2842–48, January 2015. 23, 35, 81, 124

[107] P. MacAlpine, K. Genter, S. Barrett, and P. Stone. The RoboCup 2013 drop-in player challenges: Experiments in ad hoc teamwork. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, September 2014. 94

[108] P. MacAlpine, J. Hanna, J. Liang, and P. Stone. UT Austin Villa: RoboCup 2015 3D simulation league competition and technical challenges champions. In L. Almeida, J. Ji, G. Steinbauer, and S. Luke, editors, *RoboCup-2015: Robot Soccer World Cup XIX*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2016. 45, 81, 88, 89, 97, 226

[109] P. MacAlpine, E. Liebman, and P. Stone. Simultaneous learning and reshaping of an approximated optimization task. In *AAMAS Adaptive Learning Agents (ALA) Workshop*, May 2013. 133, 145

[110] P. MacAlpine, E. Liebman, and P. Stone. Adaptation of surrogate tasks for bipedal walk optimization. In *GECCO Surrogate-Assisted Evolutionary Optimisation (SAEOpt) Workshop*, July 2016. 146

[111] P. MacAlpine, E. Price, and P. Stone. SCRAM: Scalable collision-avoiding role assignment with minimal-makespan for formational positioning. In *Proc. of the Twenty-Ninth AAAI Conf. on Artificial Intelligence (AAAI)*, January 2015. 44, 50, 77, 124, 186

[112] P. MacAlpine and P. Stone. Using dynamic rewards to learn a fully holonomic bipedal walk. In *AAMAS Adaptive Learning Agents (ALA) Workshop*, June 2012. 133

[113] P. MacAlpine and P. Stone. Prioritized role assignment for marking. In S. Behnke, D. D. Lee, S. Sariel, and R. Sheh, editors, *RoboCup 2016: Robot Soccer World Cup XX*, Lecture Notes in Artificial Intelligence. Springer, Berlin, 2016. 70, 74, 77, 82, 228

[114] P. MacAlpine and P. Stone. UT Austin Villa: RoboCup 2016 3D simulation league competition and technical challenges champions. In S. Behnke, D. D. Lee, S. Sariel, and R. Sheh, editors, *RoboCup 2016: Robot Soccer World Cup XX*, Lecture Notes in Artificial Intelligence. Springer, 2016. 45, 81, 97, 228

[115] P. MacAlpine and P. Stone. UT Austin Villa robocup 3D simulation base code release. In S. Behnke, D. D. Lee, S. Sariel, and R. Sheh, editors, *RoboCup 2016: Robot Soccer World Cup XX*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2016. 49, 96

[116] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, and P. Stone. UT Austin Villa 2011 3D Simulation

Team report. Technical Report AI11-10, The Univ. of Texas at Austin, Dept. of Computer Science, AI Laboratory, December 2011. 96, 124

[117] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, and P. Stone. UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition. In *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, June 2012. 47, 81, 97, 124

[118] E. A. Macdonald. *Multi-robot assignment and formation control*. PhD thesis, Georgia Institute of Technology, 2011. 138

[119] D. J. Mankowitz, T. A. Mann, and S. Mannor. Adaptive skills adaptive partitions (asap). In *Advances in Neural Information Processing Systems*, pages 1588–1596, 2016. 149

[120] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher-student curriculum learning. *arXiv preprint arXiv:1707.00183*, 2017. 158

[121] D. Mellinger, A. Kushleyev, and V. Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. In *Robotics and Automation (ICRA)*, 2012. 51, 137

[122] M. Mesbahi and M. Egerstedt. *Graph theoretic methods in multiagent networks*. Princeton University Press, 2010. 137

[123] N. Michael, M. M. Zavlanos, V. Kumar, and G. J. Pappas. Distributed multi-robot task assignment and formation control. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 128–133. IEEE, 2008. 51, 137

[124] O. Michel. Cyberbotics ltd. webots: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004. 150

[125] G. A. Mills-Tettey, A. Stentz, and M. B. Dias. The dynamic hungarian algorithm for the assignment problem with changing costs. Technical Report CMU-RI-TR-07-27, Carnegie Mellon University, 2007. 152

[126] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 158

[127] S. Mondesire and R. Wiegand. Evolving a non-playable character team with layered learning. In *Computational Intelligence in Multicriteria Decision-Making (MDCM), 2011 IEEE Symposium on*, pages 52–59, April 2011. 134

[128] S. C. Mondesire. *COMPLEMENTARY LAYERED LEARNING*. PhD thesis, Univ. of Central Florida Orlando, Florida, 2014. 134

[129] C. Nam and D. Shell. When to do your own thing: Analysis of cost uncertainties in multi-robot task allocation at run-time. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, 2015. 152

[130] S. Narvekar, J. Sinapov, M. Leonetti, and P. Stone. Source task creation for curriculum learning. In *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016)*, May 2016. 147, 149

[131] S. Narvekar, J. Sinapov, and P. Stone. Autonomous task sequencing for customized curriculum design in reinforcement learning. In *Proceedings of the*

26th International Joint Conference on Artificial Intelligence (IJCAI), August 2017. 147, 149

[132] S. H. Nguyen, J. Bazan, A. Skowron, and H. S. Nguyen. Layered learning for concept synthesis. In *Transactions on Rough Sets I*, pages 187–208. Springer, 2004. 134

[133] T. H. Nguyen and X. H. Nguyen. Learning in stages: a layered learning approach for genetic programming. In *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on*, pages 1–4. IEEE, 2012. 159

[134] S. Niekum, S. Osentoski, G. Konidaris, and A. G. Barto. Learning and generalization of complex tasks from unstructured demonstrations. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5239–5246, 2012. 3, 133, 147, 149

[135] Y. Ollivier, L. Arnold, A. Auger, and N. Hansen. Information-geometric optimization algorithms: A unifying picture via invariance principles. *Journal of Machine Learning Research*, 18(18):1–65, 2017. 14, 131

[136] M. Omidvar and X. Li. A comparative study of cma-es on large scale global optimisation. *AI 2010: Advances in Artificial Intelligence*, pages 303–312, 2011. 154

[137] J. B. Orlin and R. K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Mathematical programming*, 54(1-3):41–56, 1992. 152

[138] D. W. Pentico. Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, 176(2):774 – 793, 2007. 4, 52, 55, 59

[139] J. Peters, J. Kober, K. Mülling, D. Nguyen-Tuong, and O. Kroemer. Robot skill learning. In *20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 1–6, 2012. 3, 130

[140] J. Peters, K. Mülling, and Y. Altun. Relative entropy policy search. In *AAAI*, 2010. 131

[141] J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008. 131

[142] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt. The robotarium: A remotely accessible swarm robotics research testbed. *arXiv preprint arXiv:1609.04730*, 2016. 160

[143] L. P. Reis, N. Lau, and E. C. Oliveira. Situation based strategic positioning for coordinating a team of homogeneous agents. In *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, pages 175–197. Springer, 2000. 139

[144] A. Richards and J. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *American Control Conference.*, 2002. 51, 137

[145] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009. 15

[146] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick,

K. Kavukcuoglu, R. Pascanu, and R. Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. 136

[147] T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017. 132

[148] S. Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive Motion of Animals and Machines*, pages 261–280. Springer, 2006. 132

[149] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015. 132, 154

[150] J. C. Setubal et al. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report IC-96-09, University of Campinas, 1996. 63

[151] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *AAAI*, 2012. 51, 137, 138

[152] S. P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4):323–339, 1992. 24

[153] P. Sokkalingam and Y. P. Aneja. Lexicographic bottleneck combinatorial problems. *Operations Research Letters*, 23(1):27–33, 1998. 59

[154] J. Stoecker and U. Visser. Roboviz: Programmable visualization for simulated soccer. In T. Rfer, N. Mayer, J. Savage, and U. Saranl, editors, *RoboCup*

2011: Robot Soccer World Cup XV, volume 7416 of *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin Heidelberg, 2012. 125

[155] P. Stone. *Layered learning in multiagent systems: A winning approach to robotic soccer*. MIT Press, 2000. 4, 7, 23, 24, 25, 27, 134, 142

[156] P. Stone, G. A. Kaminka, S. Kraus, and J. S. Rosenschein. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *AAAI '10*, July 2010. 94

[157] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005. 128

[158] P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, June 1999. 80, 105, 139

[159] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998. 11

[160] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999. 135

[161] M. Svetlik, M. Leonetti, J. Sinapov, R. Shah, N. Walker, and P. Stone. Automatic curriculum graph generation for reinforcement learning agents. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, February 2017. 147, 149

[162] G. Synnaeve, N. Nardelli, A. Auvolat, S. Chintala, T. Lacroix, Z. Lin, F. Ri-

choux, and N. Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016. 158

[163] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009. 136

[164] C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, pages 1553–1561, 2017. 158

[165] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. 36

[166] E. Theodorou, J. Buchli, and S. Schaal. A generalized path integral control approach to reinforcement learning. *The Journal of Machine Learning Research*, 11:3137–3181, 2010. 131

[167] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012. 150

[168] P. Toth and D. Vigo. *The vehicle routing problem*. Siam, 2001. 137, 153

[169] M. Turpin, N. Michael, and V. Kumar. Capt: Concurrent assignment and planning of trajectories for multiple robots. *The International Journal of Robotics Research*, 33(1):98–112, 2014. 137, 151

[170] M. Turpin, N. Michael, and V. Kumar. An approximation algorithm for time

optimal multi-robot routing. In *Algorithmic Foundations of Robotics XI*, pages 627–640. Springer, 2015. 153

[171] M. Turpin, K. Mohta, N. Michael, and V. Kumar. Goal assignment and trajectory planning for large teams of interchangeable robots. *Autonomous Robots*, 37(4):401–415, 2014. 137

[172] D. Urieli, P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, and P. Stone. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, volume 2, pages 769–776. IFAAMAS, May 2011. 36, 109, 132, 221

[173] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011. 51, 137

[174] S. Whitehead, J. Karlsson, and J. Tenenberg. Learning multiple goal behavior via task decomposition and dynamic policy merging. In *Robot learning*, pages 45–78. Springer, 1993. 24

[175] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving soccer keepaway players through task decomposition. *Machine Learning*, 59(1-2):5–30, 2005. 24

[176] S. Whiteson and P. Stone. Concurrent layered learning. In *Second Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 193–200, New York, NY, July 2003. ACM Press. 24, 27, 135

[177] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 131

[178] P. R. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008. 50, 160

[179] L. Xiong, C. Wei, G. Jing, Z. Zhenkun, and H. Zekai. A new passing strategy based on q-learning algorithm in robocup. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 1, pages 524–527. IEEE, 2008. 156

[180] Y. Xu and H. Vatankhah. Simspark: An open source robot simulator developed by the robocup community. In S. Behnke, M. Veloso, A. Visser, and R. Xiong, editors, *RoboCup 2013: Robot World Cup XVII*, pages 632–639. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 16