Multi-Robot Coordination and Competition Using Mixed Integer and Linear Programs

Curt Alexander Bererton

CMU-RI-TR-04-65

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Robotics.

Thesis Committee

Geoffrey Gordon, Co-Chair, Carnegie Mellon University
Pradeep Khosla, Co-Chair, Carnegie Mellon University
Sebastian Thrun, Stanford University
Hugh Durrant-Whyte, University of Sydney

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

August 30th, 2004

I would like to dedicate this work to my family.

May you remain close even though you are far.

Abstract

Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs) are preferred methods representing complex uncertain dynamic systems and determining an optimal control policy to manipulate the system in the desired manner. Until recently, controlling a system composed of multiple agents using the MDP methodology was impossible due to an exponential increase in the size of the MDP problem representation. In this thesis, a novel method for solving large multi-agent MDP systems is presented which avoids this exponential size increase while still providing optimal policies for a large class of useful problems.

This thesis provides the following main contributions:

A novel description language for multi-agent MDPs: We develop two different modeling techniques for representing multi-agent MDP (MAMDP) coordination problems. The first phrases the problem using a linear program which avoids the exponential state space size of multi-agent MDPs. The second, more expressive representation, expands upon the linear programming representation with the addition of integer constraints. For many problems, these representations exactly represent the original problem with exponentially fewer variables and constraints. This leads to an efficient and optimal solution of the multi-agent MDP.

A novel multi-agent coordination method for multi-agent MDPs: We use the Dantzig-

Wolfe decomposition technique and the branch and bound method to solve the above models efficiently. These solution methods overcome significant drawbacks in related work. We develop a multi-robot towing and foraging model, devise a novel multi-agent path planner and solve coordination problems with quadratic cost on a global resource.

A method to determine the optimal strategies for competing teams: One team of players is allowed to determine the cost function of the MAMDP of the second team. This allows us to solve a zero-sum game played by both teams. One team chooses cost functions and the other coordinates to find the least cost plan given the other team's possible strategies. Each team is coordinated using the newly developed multi-agent coordination method above.

A game of robot paintball: Using the above techniques we solve a game of multi-robot paintball. This game is played with real robots at high speeds.

Acknowledgements

I would like to thank several people who have helped me to achieve my doctoral degree in robotics. First, I would like to thank my parents and family for subtly pushing me to actually get an education and support me when the education went further and further. Without their support and time I would not be where I am today, and for that I am very thankful.

This thesis would never have been without my thesis committee. First and foremost I would like to thank Geoff Gordon for being an amazing advisor. Not only was he supportive from our very first meeting, he has such good ideas that it took me a whole thesis to try out one. Not only that, he is a vast repository of information and skill from which I had to draw many times. Some of the best moments during my degree were thinking of answers that hadn't yet occurred to him, a smarter person I have not yet met. Next my appreciation goes to Pradeep Khosla for supporting my entire academic career. I appreciate the guidance he has given me and the freedom to pursue any subject that I found to be interesting and valuable. Without his support, I doubt that I would have accomplished nearly so much nor been as satisfied with the result. Sebastian Thrun was also a great source of inspiration. No one can come up with clear analogies and explanations of hard theoretical material quite like Sebastian. He also has amazing insight into what motivates people to care about your research. Lastly, I'd like to thank Hugh Durrant-Whyte for his valuable comments and time. Especially in getting up at insane hours to be available for my thesis proposal and

defense.

There are so many students and colleagues I would like to thank that I had best not list them all lest I leave someone out. Without the amazing group of people in the robotics institute and the department of computer science, I doubt that my doctorate would have been anywhere as close to as fun, interesting, challenging and rewarding as it was. Six years is a long time to spend devoting your life to something, and you had better like what you are doing or move on. I'm happy to say that I have no regrets and many fond memories of my doctorate at CMU. I hope that everyone who follows is as fortunate with their experience as I was.

Table of Contents

Chapter 1

Introduction and Motivation

Many real-world problems solved today involve making decisions in uncertain and dynamic environments. Agriculture, manufacturing, policy making, robotics, and many other domains all have this problem in common. Current approaches for decision making in these environments have two main difficulties. The first, is that many approaches do not take into account the *uncertainty* inherent in these problem domains. The second, is that in systems with many cooperating or competing agents, state of the art methods grow to be too cumbersome to be applied or are suboptimal. In this work, we provide a method which builds on the framework of Markov Decision Processes (MDPs) to solve problems involving large and uncertain multi-agent systems in both cooperative and competitive scenarios. Throughout the thesis, we concentrate on the robotic applications of our method.

We begin by motivating the need to deal with dynamic systems that involve uncertainty. We then observe and motivate the need for cooperative and competitive multi-agent system planning. With these two research fields described, we characterize the basic issues faced in both domains. Characterizing the current difficulties in these fields allows us to describe the intuition behind the approach advocated in this thesis.

1.1 Uncertainty in Systems

Most current decision making methods for large dynamic systems assume that one can always accurately determine important information about the system being considered. The "state" of a system is typically a vector of variables which contains all the relevant information required to describe the system under investigation. "Actions" for a given system change the state of the system. Typically, someone planning a control strategy or set of actions for a given system is *uncertain* about the effect of any given *action* on the state of the system. Previous approaches capable of creating plans for uncertain dynamic systems were unable to handle large multi-agent problems. As will be shown below, this work addresses that deficiency by building a much more efficient method on the base of MDPs. To begin, we motivate the need for reasoning about uncertainty in dynamic systems.

For a factory concerned with maximizing output, we might have several possible actions such as turning off a machine, increasing the output of a particular machine or hiring or firing some workers. What will be the overall effect on the factory if these actions are taken? Often we cannot predict the future state of the system given a particular action at the current time. For a robot concerned with moving from location A to location B, it may be that we know the current x and y location of the robot, but we are uncertain that if we try to move straight forward we may end up going slightly to the right and falling off of a cliff. These examples show why we might want to worry about the uncertainty in one's actions.

One can assume that we always exactly know the outcome of a particular action. However, intuition dictates that it makes sense to plan for the case where the result of an action is not known for certain. Parts may not get shipped on time from suppliers even though we ordered a certain number. Machines may break if we increase output. The robot may move several inches too far to the right when we directed it to move straight ahead. In many situations the economic or implications of ignoring the uncertainty inherent in our actions

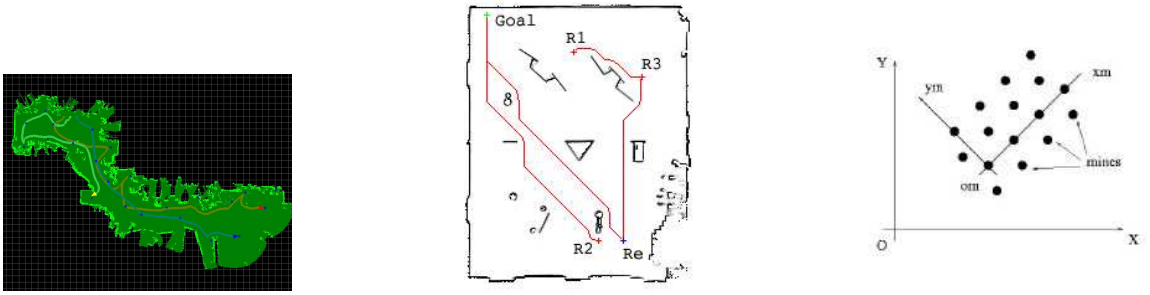


Figure 1.1: Many applications benefit from the use of multiple agents. From left to right: multi-robot exploration (Zlot et al. (2002)), planning for repairable robots (Bererton and Khosla (2002)), Robotic de-mining for a patterned minefield (Zhang et al. (2001))

are dire.

In fact, people intuitively take actions which are robust to uncertainty all the time. Spare parts for machines are kept on hand. Spare stock is kept on hand in case something goes wrong. The robot is made to take a path far away from the stairwell. In general, humans already take this form of uncertainty into account using past experience. Unfortunately, the models and solutions many planners apply to real-world problems do not account for this uncertainty.

The real-world is rife with uncertainty, and people in general make plans "in case something goes wrong". That is, people are taking into account uncertainty in the effect of their actions. One goal of this thesis is to present an approach which can reason about and create effective plans for multi-agent systems involving uncertainty.

1.2 Multi-Agent Systems

Most problems can be solved more efficiently and robustly with multiple agents. Previous work dealing with multi-agent systems can only handle small dynamic systems if they are to be controlled optimally in the face of uncertainty. In order to control a team of agents, it is considered necessary to lose the ability to generate an optimal plan for a large team as

well as losing the capability to generate a plan which could handle uncertainty. The main innovation of this thesis is the further development of an approach whereby a large class of multi-agent systems can be controlled optimally in the face of uncertainty.

Observing many of the tasks an individual agent performs in a given day, we often see two forms of interaction with other agents or other groups of agents: cooperation and competition. In this work, we address both forms of interaction for teams of agents.

We often cooperate with others to perform a number of tasks we could not accomplish ourselves, or would simply take much longer to perform with one person. There are many arguments for the benefits of multi-agent systems. The primary arguments are based on increased output, increased speed and/or increased robustness to mission failure (Dias and Stentz (2001); Gerkey and Mataric (2002)).

These advantages are readily apparent in some of the tasks shown in figure (1.1). Multi-robot exploration increases the speed with which an area can be mapped compared to mapping with a single robot (Zlot et al. (2002)). Allowing the robots to tow or otherwise repair each other increases team robustness (Bererton and Khosla (2002)). In de-mining tasks where some robots may be disabled by an exploding mine one clearly desires more than one robot to complete the task (Dawson-Howe and Williams (1997); Zhang et al. (2001)).

The advantages to multi-robot systems are many. The problem with planning for many agents which are cooperating is that it becomes harder and harder to reason about the combination of every robot's low level action as the number of robots increases. The contribution of this thesis is to develop a method which overcomes this difficulty while still guaranteeing optimality for many problems.

Figure 1.2: Robotic games are becoming a popular sport. Generating successful strategies which are robust to uncertainty is an important area of research. From left to right: multi-robot paintball robots developed by Botrics and a robocup AIBO team

1.2.1 Competitive Games

There are many important tasks where two teams of agents are competing against one another in dynamic and uncertain environments. Sports provide a multitude of such examples and are now becoming a popular testbed for many robotics algorithms. Robot soccer (Kitano (1998)), robot laser tag (Rosencrantz et al. (2002)), and other robot sports all demonstrate examples of these kinds of tasks. Of more practical importance is that of competition in business, and "competition" in the military sense.

Clearly, planning for uncertainty in these environments is crucial since the competing team adds a great deal of uncertainty. Game theory has provided a basic model which has been used very successfully in many applications (Fudenberg and Tirole (1991)). However, game theory models are often very abstract models of the true task and do not model uncertainty. Another contribution of this thesis is to allow the modeling of team competition games within an MDP framework and determine the minimax optimal strategy to play for both teams for a certain class of problems. With this kind of technology, we can begin to model real-world games such as those shown in figure (1.2) much more accurately. Thus, the resulting strategies will become more useful for the game.

1.3 Current Approaches

Given the motivation for the application domains above, we can now look at current methods for dealing with uncertainty and large multi-agent systems. In this section, we will highlight areas where the state of the art cannot adequately handle either uncertainty or large multi-agent systems.

1.3.1 Linear Programming

Of all approaches used for modern day planning, perhaps the most popular is linear programming (Dantzig (1963); Chvatal (1983); Rardin (1998)). Linear programming is extremely powerful and general. However, linear programs are rarely used for multi-robot planning for two reasons. Firstly, it was not clear how to modify standard linear programming models of multi-robot problems to account for uncertainty in state and uncertainty in action. Second, for many problems there are task-specific solutions which are far more efficient than a general linear programming solution. In particular, linear programs are particularly inefficient at solving systems where long sequential plans are required.

There are many problems in which one should account for uncertainty. As a rule of thumb: If a small change to an important system variable due to an action can cause great losses, then it would be wise to model uncertainty for that state variable. An example is robot path planning in a known environment with a cliff. If a small difference in trajectory causes the robot to drive off of a cliff, then you should likely consider uncertainty the motion direction.

There are a large class of problems which require long sequential plans. For problems such as path planning in a grid world, it is possible to formulate a linear program to solve the problem. Unfortunately, such a solution is orders of magnitude slower than A* or other search based methods. This is because the size of a linear program is very large for long sequential plans such as that required for path planning.

While linear programming is slow and inefficient for many problems, new algorithms are often derived by examining the linear programming solutions to a problem. One advantage that certain advanced linear programming methods provide is the solution of large weakly coupled problems using Benders decomposition or its dual Dantzig-Wolfe decomposition (Dantzig (1963)).

As we shall see later in this thesis, we use these decompositions to create a new method for multi-robot planning which maintains the advantages of task-specific solutions and also deals with uncertainty.

1.3.2 Market Based Methods

In many cases, robots in a multi-robot system interact only in limited ways. Robots might seek not to collide with each other (Bennewitz et al. (2001)), coordinate their locations to carry out a joint task (Kitano (1998); Salido et al. (1997)), or consume a joint resource with limited availability (Burgard et al. (2000); Parker (1996); Zlot et al. (2002)). While these problems are not trivially decomposed, they do not necessarily have the worst-case exponential complexity that characterizes the general case.

Handling this sort of limited interaction is exactly the strength of market-based planning algorithms (Zlot et al. (2002); Gerkey and Mataric (2002)): by focusing their attention on a limited set of important resources and ignoring all other interactions, these algorithms reduce the problem of cooperating with other robots to the problem of deciding which resources to produce or consume. Market-based algorithms are particularly attractive for multi-robot planning because many common types of interactions can be phrased as constraints on resources such as space (two robots can't occupy the same location at once) and time (a robot can only work on a limited number of tasks at once).

From the point of view of these market algorithms, the difficult part of the multi-robot planning problem is to compute the probability distribution of the price of each resource at

every time step: the optimal price for a resource at time t depends on how much each robot produces or consumes between now and time t, and what each robot's state is at time t. The resource usage and state depend on the robots' plans between now and time t, which in turn depend on the price. Worse yet, future resource usage depends on random events which can't be predicted exactly.

The typical auction based method uses an auctioneer to determine the price for each resource based on bids from the individual planning robots. This allows for low bandwidth communication between the robots and the auctioneer (they need only send bids). Most importantly, each robot only considers the part of the full problem relevant to its own rewards and capabilities. This allows for a distributed solution which is very efficient. Unfortunately, because each robot is seeking to maximize its own profit while participating in the bidding, a global optimum is no longer guaranteed with standard market based methods.

An important assumption in much recent work in auction based coordination has made is that each agent can determine its bid independent of the actions of the other robots (Gerkey and Mataric (2004)). In the case of multiple robots planning paths in an environment, this is no longer true. If one robot decides to sit in the only doorway between two rooms, then other agents bidding to perform tasks in either room must know this information to bid properly for tasks in either room. There are many tasks in which a robot or agent's valuation of sub-goals cannot be determined independently of other robot's actions. Auction methods which do not account for this are not guaranteed to find the global optimum for the coordination problem. The main method for taking this into account is to hold combinatorial auctions which are typically quite slow.

What we would like is to use the intuition behind market based methods that only a limited set of important resources need be considered jointly by all robots while planning for a large team. The remainder of the planning can be done by each robot individually

preserving the efficiency of a distributed solution.

In addition to maintaining the best features of market based methods, we would like to address its weaknesses. In particular, we want to keep the efficiency of market based methods while still achieving a global optimum.

1.3.3 Markov Decision Processes

A formalism called Markov Decision Processes (MDPs) was developed soon after linear programming and popularized by Bellman (Bellman (1957a,b)). This framework allows for optimal decision making (planning) in the face of action uncertainty. Modern methods for solving Markov Decision Processes have been known to solve problems with millions of states. Two modern texts describing MDPs and their solution methods are: (Sutton and Barto (1998); Puterman (1994)).

Many planning problems in robotics are best phrased as MDPs, defined over world states or—in case of partial observability—belief states (Kaelbling et al. (1998)). However, existing MDP planning techniques generally scale poorly to multi-robot systems because of the "curse of dimensionality": in general, it is exponentially harder to solve an MDP for N agents than it is to solve a single-agent MDP, because the state and action space for N robots can be exponentially larger than for a single-robot system. This enormous complexity has confined MDP planning techniques largely to single-robot systems.

Recent work by Guestrin and Gordon (Guestrin and Gordon (2002)) have recently significantly enhanced the ability to plan for multi-agent MDPs. In this thesis, we build on their work and proceed in a slightly different direction. We arrive at a novel multi-agent coordination method which builds on the idea of a factored representation but tries to circumvent some of the difficulties of prior work.

1.3.4 Game Theory and Team Competition

It is rare to see game theory applied to competitions between teams of agents in dynamic uncertain environments. Though the general theory for solving these kinds of games is available in the form of stochastic games, it is known to be very difficult to solve in the general case (Filar and Vrieze (1992)). Recently, a method was developed whereby game theory has been combined with Markov Decision Processes to solve a class of useful single agent competitions (McMahan et al. (2003); McMahan and Gordon (2003)). The basic idea behind this work is that an adversary can choose the cost function for a player which plans using an MDP.

Using a coordination algorithm, one could see expanding this approach to handle two teams competing in an uncertain dynamic environment. This class of problems can be found in a variety of application from military planning, computer games, robotics games such as robot soccer as well as certain economic scenarios.

A real world example of this class of problems lies in multi-robot path planning while trying to evade a network of surveillance sensors. The "defending" team selects the location of the sensors on a known map, and the "attacking" team of robots must plan the lowest cost path to reach a set of goals while evading the sensors. We have addressed this problem in this thesis in the domain of multi-robot paintball.

In this work we take an approach similar to that proposed by (Lagoudakis and Parr (2002)). Where they build a team competition based on the work of Guestrin, we build upon our multi-agent coordination algorithm which is a different offshoot of Guestrin's work.

1.4 The proposed approach

Reviewing the above subsections we see the following: Linear programs have good methods for solving large problems, but are slow compared to problem specific solutions for long sequential plans and do not handle uncertainty. Most state of the art methods for solving Markov Decision Processes have difficulty with large problems, but can handle uncertainty well. Lastly, market based methods can handle large problems efficiently and are easy to understand and implement, but do not have guaranteed optimal solutions for a large class of useful problems nor do they have convenient mechanisms for handling uncertainty nor global resource constraints based on low level actions.

Intuitively, we would like to combine the best of the above methods into one method while negating the disadvantages of each method. As the reader will see, we have built on previous work to overcome some of the difficulties for planning in multi-agent MDPs.

At its base, the multi-agent coordination method developed in this thesis expands Markov Decision Process solution techniques by using large scale solution methods from linear programming. The result is a method which has many parallels to market based methods in terms of efficiency and understandable operation. For a large class of useful problems this new method is efficient while maintaining optimality and handling long sequential plans and certain kinds of uncertainty.

In brief, we assume that the interaction between agents' actions can be represented linearly. If this is so, we can use a linear program called the "master" program to coordinate a team of agents in such a way that an optimal joint plan for all agents is achieved. The individual agents plan without any knowledge of the other agents in the world. However, the master linear program adjusts the costs of some actions in each agent's plan such that they will coordinate properly to achieve the best reward. Aside from their cost functions being modified, the individual agents need not know the actions of the other agents to achieve an optimal plan. This allows an efficient implementation while still guaranteeing

optimality.

With this new multi-agent coordination in hand, we then combine it with the single agent adversarial method developed in (McMahan et al. (2003)) to achieve a principled planning method for team competitions. This can then be applied to a real-world multi-robot paintball game.

1.5 Thesis Layout

The rest of the thesis is laid out as follows:

Chapter 2 In this chapter we will introduce some background material. We will begin with very quick review of Markov Decision Process notation. This is followed by a brief summary of the linear programming notation used in this thesis. The phrasing of Markov Decision Processes as linear programs are reviewed. Then a review of Dantzig-Wolfe decomposition for linear programs are presented. Lastly, the branch and bound method for mixed integer programs is reviewed. These algorithms form the base for the new methods outlined in this thesis.

Chapter 3 In this chapter, we present our novel multi-agent coordination mechanism. Primarily, our method is based on a combination of Dantzig-Wolfe decomposition and Markov Decision Processes phrased as linear programs. It is recommended to review these subjects in chapter 2 prior to reading this chapter.

Chapter 4 This chapter presents a novel combination of the coordination method outlined in chapter 3 with the single agent adversarial method developed by McMahan Gordon and Blum (McMahan et al. (2003)). This combination allows us to solve a particular game of multi-robot paintball. This game is then played by physical robots demonstrating the applicability of the method to real world tasks.

Chapter 5 In this chapter we address related work mentioned briefly in previous chapters. Specifically, we compare our work to related methods and specify the gap the work is intended to fill.

Chapter 6 Lastly, we summarize the main contributions of this thesis. We also take a brief look at future directions which seem the most promising for the continuation of this work.

Bibliography

Bellman, R. (1957a). *Dynamic Programing*. Princeton University Press.

Bellman, R. (1957b). A markov decision process. *Journal of Mathematical Mechanics*.

Bennewitz, M., Burgard, W., and Thrun, S. (2001). Optimizing schedules for prioritized path planning of multi-robot systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seoul, Korea. ICRA.

Bererton, C. and Khosla, P. (2002). An analysis of towing repair capabilities in a team of robots. In *Robosphere2002*.

Burgard, W., Fox, D., Moors, M., Simmons, R., and Thrun, S. (2000). Collaborative multi-robot exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA. IEEE.

Chvatal, V. (1983). *Linear Programming*. W.H. Freeman and Company.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press.

Dawson-Howe, K. M. and Williams, T. G. (1997). The detection of buried landmines using probing robots.

Dias, M. and Stentz, A. (2001). A market approach to multirobot coordination.

Filar, J. and Vrieze, O. (1992). Weighted reward criteria in competitive markov decision processes. *Methods and Models of Operations Research*, 36:343–358.

Fudenberg, D. and Tirole, J. (1991). *Game Theory*. MIT Press.

Gerkey, B. and Mataric, M. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*.

Gerkey, B. P. and Mataric, M. J. (2002). Sold!: Market methods for multi-robot control.

Guestrin, C. and Gordon, G. (2002). Distributed planning in hierarchical factored MDPs. In Darwiche, A. and Friedman, N., editors, *Uncertainty in Artificial Intelligence (UAI)*, volume 18.

Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134.

Kitano, H., editor (1998). *Proceedings of RoboCup-97: The First Robot World Cup Soccer Games and Conferences*, Berlin. Springer Verlag.

Lagoudakis, M. and Parr, R. (2002). Learning in zero-sum team markov games using factored value functions. In *Advances in Neural Information Processing Systems*, volume 15.

McMahan, H. B. and Gordon, G. (2003). Planning in cost-paired markov decision process games. In *NIPS 2003 Workshop on Planning for the Real-World*.

McMahan, H. B., Gordon, G., and Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. In *Twentieth International Conference on Machine Learning (ICML)*.

Parker, L. E. (1996). On the design of behavior-based multi-robot teams. *Journal of Advanced Robotics*, 10(6).

Puterman, M. L. (1994). *Markov Decision Problems*. Wiley.

Rardin, R. (1998). *Optimization in Operations Research*. Prentice Hall.

Rosencrantz, M., Gordon, G., and Thrun, S. (2002). Locating moving entities in dynamic indoor environments. In *Submitted to: AGENTS ACM 2003*.

Salido, J., Dolan, J., Hampshire, J., and Khosla, P. (1997). A modified reactive control framework for cooperative mobile robots. In *Proceedings of the International Conference on Sensor Fusion and Decentralized Control*, pages 90–100, Pittsburgh, PA. SPIE.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning*. MIT Press.

Zhang, Y., Schervish, M., Acar, E., and Choset, H. (2001). Probabilistic methods for robotic landmine search. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '01)*, pages 1525 – 1532.

Zlot, R., Stentz, A., Dias, M., and Thayer, S. (2002). Multi-robot exploration controlled by a market economy.

Chapter 2

Background Theory

The purpose of this chapter is to review some of the basic mathematical concepts and algorithms required to understand the novel multi-agent coordination mechanism presented in chapter 3. We begin with linear programming and decompositions for large, loosely coupled linear programs. We then proceed on to Markov Decision Processes and how they can be formulated as linear programs. Integer programming capabilities and methods are then reviewed.

It is with these concepts that we will build our multi-agent coordination algorithm. It is not the goal of this chapter to replace a good reference on the above subjects, but merely to review the main ideas and theorems so that the reader may understand the multi-agent coordination method at least at a medium level of understanding. For a deeper understanding of the framework, the author refers the reader to the texts cited in each section below.

2.1 A brief review of some linear programming concepts

This section provides mathematical intuition for the linear programming methods used without requiring a reader unskilled in linear programming having to resort to reading

hundreds of pages of a textbook.

For a deeper understanding of the multi-agent coordination method, those unfamiliar with linear programming should eventually study some linear programming. The author recommends first reading the introduction to linear programming chapters in both (Cormen et al. (2001)) and (Rardin (1998)). Then proceed onto the more complete text by Bertsimas and Tsitsiklis (Bertsimas and Tsitsiklis (1997)). We stay close to the notation used in (Bertsimas and Tsitsiklis (1997)) and have also drawn relevant examples and abbreviated some explanations from this text. For all of the linear programming tools used in this thesis, it is sufficient to refer to (Bertsimas and Tsitsiklis (1997)) for a thorough examination of the material. The author has found this book to be one of the best references for linear programming. The books by (Chvatal (1983)) and (Dantzig (1963)) are also useful references though the presentation is not as clear.

2.1.1 A Brief History

Linear programming was invented by the L.V. Kantorovich in 1939 while trying to optimize a manufacturing schedule. At that time it was relegated to an interesting but not very useful way to optimize economics problems. This was due to the lack of computers which could solve larger linear programs quickly as well as the lack of a principled method to solve linear programs efficiently. Fortunately, George Dantzig developed the first algorithm to solve a general linear program efficiently called the simplex method in 1947. Soon after in 1952, the first linear program was solved on a National Bureau of Standards SEAC computer. Since that time, linear programming has been heavily used as a general optimization tool for a countless number of domains.

2.1.2 Linear programming notation and definitions

Linear "Programming" is in fact a misleading name. By solving a linear program we are finding the values for a set of *decision variables* x_i such that a linear function of those variables is maximized or minimized. The function being minimized or maximized is called the *objective function*. Linear programming also allows the specification of linear *constraints* on the variables over which the objective function is valid. These constraints are linear functions of the decision variables. For example, the following describes a simple linear program:

$$\begin{aligned} \min_{x_1,x_2} \; & 2x_1 + 3x_2 \\ \text{such that:} & \\ x_1 + x_2 &\leq 1 \\ x_1 + x_2 &\geq \frac{1}{2} \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned} \tag{2.1}$$

The objective function of (2.1) is $2x_1 + 3x_2$. We wish to minimize the objective function (as noted by the *min*) while meeting the linear constraints on the variables listed below the objective function.

It is also possible to convert between an inequality constraint and an equality constraint by the addition of a *slack variable*. For example: $x_1 + x_2 \leq 1$ can be changed to $x_1 + x_2 - s_1 = 1$ where we force the slack variable s_1 to be:$s_1 \geq 0$. This changes the program (2.1) to be:

$$\begin{aligned} &\min_{x_1,x_2} 2x_1 + 3x_2 \\ &\text{such that:} \\ x_1 + x_2 - s_1 &= 1 \\ x_1 + x_2 &\geq \frac{1}{2} \\ x_1, x_2, s_1 &\geq 0 \end{aligned} \tag{2.2}$$

The linear program in (2.1) can be changed to a more compact vector and matrix based notation as follows:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{c}'\mathbf{x} \\ A\mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned} \tag{2.3}$$

The following substitutions could be made to represent the linear programming problem in (2.1):

$$A = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -\frac{1}{2} \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

In general, the representation in (2.3) is sufficient to represent *any* linear program. The vector and matrix notation above will be used extensively throughout the thesis. Also, we will sometimes convert matrices to a different form which is shown in (2.4). This form is known as the *standard form* and can represent any linear program described by (2.3).

$$\begin{aligned} \min_{\mathbf{x}} \mathbf{c}'\mathbf{x} \\ A\mathbf{x} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2.4}$$

Any vector $\mathbf{x}$ satisfying the constraints $A\mathbf{x} = \mathbf{b}$ in (2.4) is called a *feasible solution* or *feasible vector*. The set of all feasible solutions is called the *feasible set* or *feasible region*. Any feasible solution $\mathbf{x}^*$ that minimizes the objective function ($\mathbf{c}'\mathbf{x}^* \leq \mathbf{c}'\mathbf{x}$ for all feasible x) is called the *optimal feasible solution* or simply the *optimal solution*. The value of $\mathbf{c}'\mathbf{x}^*$ is the *optimal cost*. For a general $\mathbf{x}$ we may refer to $\mathbf{c}'\mathbf{x}$ as the *value* or *cost* of the linear program.

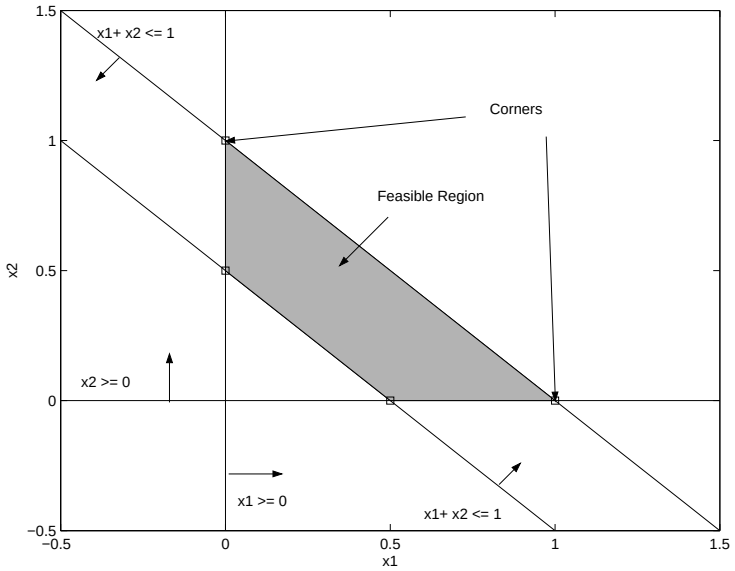


Figure 2.1: The feasible region for the linear program (2.1)

For any linear program, the feasible region of solutions can be represented by a convex polytope. For example, the feasible region of (2.1) is shown in figure (2.1) and is a convex polytope. It is known that there always exists an optimal solution to a linear program on

a *corner* of the feasible region. If the feasible region is empty, then the linear program is said to be *infeasible*.

Any point within the feasible region can be represented as a convex combination of the corners of the feasible region. More formally, let $\mathbf{x^1}, \ldots, \mathbf{x^k}$ be vectors in $\Re^n$ and let $\lambda_1, \ldots, \lambda_k$ be nonnegative scalars such that $\sum_{i=1}^{k} \lambda_i = 1$. If the vectors $\mathbf{x^1}, \ldots, \mathbf{x^k}$ are the corners of a bounded convex polytope, then any point $\mathbf{y}$ within the polytope can be represented as $y = \sum_{i=1}^{k} \lambda_i x_i$ for some set of λ_i. This representation will become important as we examine the decomposition of large linear programs using Dantzig-Wolfe decomposition.

The representation of the feasible region as a convex polytope is also very important for a particular method of solving linear programs. The *revised simplex method* solves a linear program by starting at a corner of the polytope and moves to adjacent corners in such a way that it finds the optimal feasible solution at a given corner. The revised simplex method is a modified version of the simplex method which only needs to remember a subset of the columns of the A matrix when choosing the next corner to move to. This feature of the solution method is important for the Dantzig-Wolfe decomposition described below.

2.1.3 The dual of a linear program

Given any linear program there always exists another linear program which can be used to draw insight and help to solve the first linear program. We will call the original linear program the *primal* linear program and the second linear program the *dual* linear program. Duality theory provides insight into the solution of linear programs and, important for this thesis, enables the formulation of the Dantzig-Wolfe decomposition of large linear programs.

The idea of duality comes from the Lagrange multiplier method used in calculus to minimize a function which has equality constraints. For example to solve the following example from (Bertsimas and Tsitsiklis (1997)):

$$\begin{aligned} \text{minimize} \quad & x^2 + y^2 \\ \text{such that} \quad & x + y = 1 \end{aligned} \tag{2.5}$$

We first introduce a Lagrange multiplier p and form the Lagrangian $L(x, y, p)$. p is the *penalty* or *price* that we pay for violating the constraint.

$$L(x, y, p) = x^2 + y^2 + p(1 - x - y)$$

To solve for the minimum we keep p fixed and minimize the Lagrangian over all x and y by setting $\partial L/\partial x$ and $\partial L/\partial y$ to 0. For the above problem the solution to the unconstrained problem is:

$$x = y = \frac{p}{2}$$

Using the additional constraint $x + y = 1$ from the original program we can solve for p to find that $p = 1$. Therefore, the optimal solution is $x = y = \frac{1}{2}$.

The intuition behind the Lagrange multiplier method is that we do not enforce the hard constraint $x + y = 1$. Instead, we allow that constraint to be violated, but violating the constraint causes us pay the price p times the amount the constraint is violated. When the price p is properly chosen ($p = 1$ in the above example), the solution to the unconstrained minimization is equivalent to that of the constrained minimization. In general, we wish to find the price p for which the the presence or absence of the hard constraints does not affect the optimal solution.

When forming the dual of a linear program, we follow the same procedure. For each constraint in the original (aka primal) linear program we associate a price p with each linear constraint and attempt to determine the set of prices $\mathbf{p}$ for which the absence of presence of the hard constraints no longer matters. Applying this intuition to the standard LP of

(2.4) we obtain:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}) \\ &\mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2.6}$$

Because we have removed the hard constraints, we have essentially increased the feasible region for the problem. Thus, the optimal cost for (2.6) will be lower than the optimal cost for the original standard form LP (2.4). If we let $g(\mathbf{p})$ be the optimal cost for (2.6) then this leads to the following:

$$g(\mathbf{p}) = \min_{\mathbf{x} \geq \mathbf{0}} \mathbf{c}'\mathbf{x} + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}) \leq \mathbf{c}'\mathbf{x}^* + \mathbf{p}'(\mathbf{b} - \mathbf{A}\mathbf{x}^*) = \mathbf{c}'\mathbf{x}^* \tag{2.7}$$

When $\mathbf{c}' - \mathbf{p}'\mathbf{A}$ is less than zero, the minimum in (2.7) is $-\inf$ and so we only concentrate on cases where $\mathbf{p}'\mathbf{A} \leq \mathbf{c}'$. For each value of $\mathbf{p}$ we have a lower bound on the optimal cost $\mathbf{c}'\mathbf{x}^*$. This corresponds to the theorem of *weak duality* which simply states that $\mathbf{p}'\mathbf{b} \leq \mathbf{c}'\mathbf{x}$ for any feasible solutions $\mathbf{p}$ and $\mathbf{x}$. The search for the $\mathbf{p}$ to achieve the tightest bound on the optimal cost leads us to the dual problem:

$$\begin{aligned} &\max_{\mathbf{p}} \mathbf{p}'\mathbf{b} \\ &\mathbf{p}'\mathbf{A} \leq \mathbf{c}' \end{aligned} \tag{2.8}$$

Given the optimal $\mathbf{p}$ from (2.8), finding the $\mathbf{x}$ which minimizes the primal problem is only a matter of unconstrained linear optimization. Also it has been shown that if there exists an optimal solution to the primal problem, there also exists an optimal solution to the dual problem. This fact has been named *strong duality* and has been proven to be true.

The discovery of duality has led to some fast algorithms for solving linear programs. Part

of the method described in chapter 3 relies on a primal-dual interior point algorithm for the solution of a linear program. The advantage of this solution method (in addition to being a fast solution algorithm for linear programs) is that it yields the optimal decision variables $\mathbf{p}$ and $\mathbf{x}$ for both the primal and dual linear programs which is useful for Dantzig-Wolfe decomposition described below.

In summary, the dual formulation of a linear program yields fast solution algorithms for linear programs and enables the Dantzig-Wolfe decomposition method below. The intuition that the prices $\mathbf{p}$ are prices paid to violate a constraint is also useful for understanding the multi-agent coordination algorithm presented in the next chapter.

2.1.4 Linear program solution time

Assume that the A matrix contains m rows and n columns. The revised simplex method requires $O(mn)$ computations per iteration in the worst case and $O(m^2)$ computations per iteration in the best case. While this may seem to be slow, practical linear program solvers can solve problems with tens of thousands of constraints and variables in a few seconds.

The number of iterations required to solve a linear program can be exponential on some problems. However, for most problems encountered in practice, the number of iterations is $O(m)$. Unfortunately, the study of the "average" running time of either simplex method is an open question. This is due mainly to the definition of "average" as it may vary greatly dependent on a given set of tasks.

Recently however, a definition of average was presented as the *smoothed complexity*. The smoothed complexity of a linear program is expected running time of the an algorithm on an arbitrary instance under a random permutation. It has been shown in (Spielman and Teng (2001)) that the revised simplex method has polynomial smoothed complexity.

2.2 Dantzig-Wolfe decomposition

Dantzig-Wolfe decomposition was invented in 1960 by Dantzig and Wolfe and is well described in (Dantzig (1963)) as well as in (Bertsimas and Tsitsiklis (1997)). To understand the Dantzig-Wolfe decomposition, we begin by describing *delayed constraint generation* and its dual cousin *delayed column generation* which is a crucial component of Dantzig-Wolfe decomposition.

2.2.1 Delayed constraint generation

Imagine that we have a problem of the form:

$$\begin{aligned} &\max_{\mathbf{p}} \mathbf{p}'\mathbf{b} \\ &\mathbf{p}'\mathbf{A_i} = \mathbf{c_i} \quad \mathbf{i} = \mathbf{1}, \ldots, \mathbf{n} \end{aligned} \tag{2.9}$$

Unfortunately, the number of constraints n in this problem is very large. A solution to this problem is to solve a *relaxed* problem where only a small subset of the constraints are represented. Thus we solve (2.9) but only represent some subset $I \subset \{1, \ldots, n\}$ of the constraints.

By relaxing the problem in this manner we guarantee that the optimal cost of the relaxed problem is greater than the cost of the original problem. Thus $\mathbf{p}'_{relaxed}\mathbf{b} \geq \mathbf{p}'\mathbf{b}$. Thus, if we solve the relaxed problem and find a $\mathbf{p}_{relaxed}$ that is feasible in the original problem, we are finished. If it is not feasible in the original problem, we add a constraint which was broken in the original problem to the relaxed problem (increasing the size of I). We continue to increase the size of I until we find a $\mathbf{p}_{relaxed}$ which is feasible in the original problem, at which point we terminate. This is guaranteed to lead to an optimal solution to the original problem, though there is no general guarantee on the size I must grow to before the optimal solution is found. This procedure of adding constraints which were broken by the solution

of the previous iteration is called *constraint generation.*

2.2.2 Delayed column generation

$$\begin{aligned} \min_{\mathbf{x}} \mathbf{c}'\mathbf{x} \\ A\mathbf{x} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{2.10}$$

The dual of delayed constraint generation is delayed column generation. We now describe this crucial technique for the decomposition of large linear programs.

Given the standard LP form above (repeated from (2.4)), how does one solve the linear program if the number of columns in the A matrix is so large that it cannot be stored in computer memory? The answer comes from the fact that the revised simplex method mentioned above typically only needs to examine a small subset of the columns in A in order to choose which corner of the feasible region to move to in the next iteration. Determining which corner to move to involves a computation which calculates the *reduced cost* $\bar{c}_i$ of adjacent corners. The reduced cost is the rate at which we can lower the cost of the objective function if we were to move in the direction of corner i.

Once we determine which corner of the feasible region to move to, moving to that corner will correspond to inserting a new column into our matrix A that we previously had not represented and removing another column. Thus, it is possible to solve a linear program where the matrix A has a very large number of columns but only stores a fixed number of columns K at any given time while solving the linear program using the revised simplex method. This process of generating the required columns at the time they are needed is called *delayed column generation.*

A variant is to remember all columns that you have ever seen and keep them in the A matrix which would then grow as the revised simplex algorithm proceeded. Solving the LP in either manner is also guaranteed to converge because we are using a special case of the

revised simplex algorithm which is guaranteed to converge.

2.2.3 Dantzig Wolfe decomposition

Dantzig-Wolfe decompositions is meant to solve problems of the following form:

$$\begin{aligned} \min_{\mathbf{x_1},\mathbf{x_2}} \mathbf{c_1'x_1} + \mathbf{c_2'x_2} \\ \mathbf{D_1x_1} + \mathbf{D_2x_2} = \mathbf{b_0} \\ \mathbf{A_1x_1} = \mathbf{b_1} \\ \mathbf{A_2x_2} = \mathbf{b_2} \\ \mathbf{x_1}, \mathbf{x_2} \geq \mathbf{0} \end{aligned} \tag{2.11}$$

If we were to join all of the constraints in this program into one large constraint matrix it would resemble the structure depicted in figure (2.2). There are two subproblems which are mostly independent of each other depicted by the lower rectangles constraining only the variables of each subproblem. Solving a linear program with these kinds of grouped constraints can be significantly easier than solving a general linear program as we will demonstrate below.

Suppose that $\mathbf{x_1}$ and $\mathbf{x_2}$ have dimensions of n_1 and n_2. Furthermore, $\mathbf{b_0}$, $\mathbf{b_1}$ and $\mathbf{b_2}$ have dimensions of m_0, m_1 and m_2 respectively. $\mathbf{x_1}$ must satisfy the non-negativity constraints and those imposed by $\mathbf{A_1x_1} = \mathbf{b_1}$. Similarly, $\mathbf{x_2}$ must satisfy the non-negativity constraints and those imposed by $\mathbf{A_2x_2} = \mathbf{b_2}$. This problem contains two separate subproblems which are joined by the linear constraints $\mathbf{D_1x_1} + \mathbf{D_2x_2} = \mathbf{b_0}$.

This type of problem arises frequently in the real world. Here the variables $\mathbf{x_1}$ might represent the activity levels of division one's factory machines producing type one widgets produced by some manufacturing firm. The variables $\mathbf{x_2}$ might represent the activity levels of division two's factory machines producing type two widgets. If both use the same raw

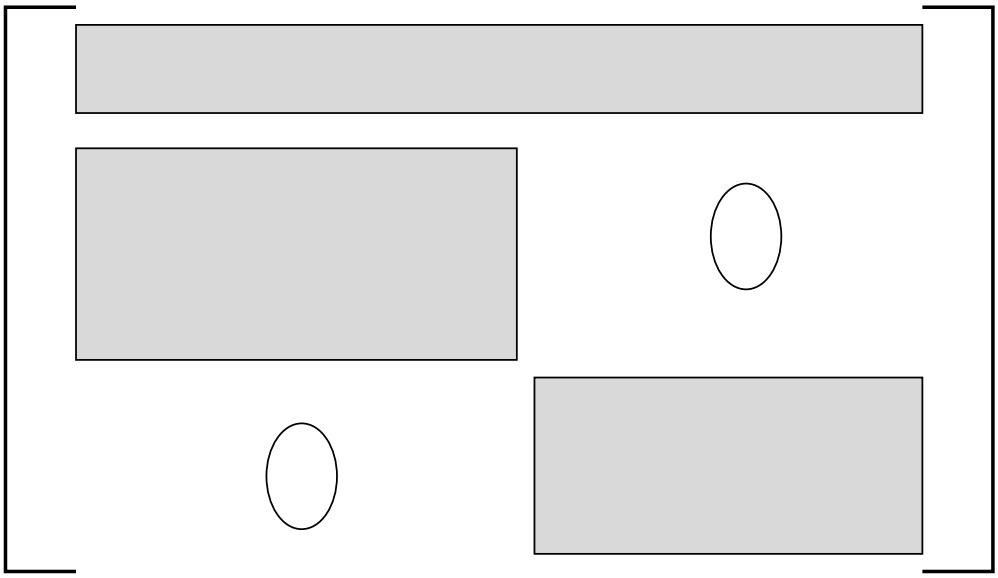


Figure 2.2: A constraint matrix for a loosely coupled linear program. Certain sub-problems have constraints which only affect certain variables belonging to that sub-problem such as the lower rectangles in this figure. The sub-problem variables are only joined by a few interaction constraints depicted as the upper rectangle.

resources then their activity levels will interact but only through variables which use a resource shared by the two divisions.

Imagine further that the two separate divisions in a firm must minimize their own costs according to their own constraints. For division one this might correspond to a linear program of the form:

$$\begin{aligned} \min_{\mathbf{x_1}} \mathbf{c_1'x_1} \\ \mathbf{A_1x_1} = \mathbf{b_1} \\ \mathbf{x_1} \geq \mathbf{0} \end{aligned} \tag{2.12}$$

Division two would have a separate problem which is represented by a similar linear program. They might be linearly coupled by the fact that the amount of resources used should not exceed the resources available to the company. This constraint can be repre-

sented by $\mathbf{D_1x_1 + D_2x_2 = b_0}$. Thus, the divisions interact via shared *resources* or shared *commodities*. Combining this constraint with the individual linear programs from each division (of the form shown in (2.12)) yield the original problem (2.11). The Dantzig-Wolfe decomposition allows us to solve this kind of problem in an efficient way.

We would like to propose a problem equivalent equivalent to (2.11). This problem will have many fewer constraints but many more variables. We begin by writing down the convex polytope defined by the constraints $\mathbf{A_ix_i = b_i}$:

For the subproblems $i = 1, 2$ we define a convex polytope P_i as:

$$P_i = \{\mathbf{x_i} \geq \mathbf{0} \mid \mathbf{A_ix_i = b_i}\}$$

Using this definition we can rewrite the original LP in (2.11) as:

$$\begin{aligned} \min_{\mathbf{x_1},\mathbf{x_2}} \mathbf{c_1'x_1 + c_2'x_2} \\ \mathbf{D_1x_1 + D_2x_2 = b_0} \\ \mathbf{x_1} \in P_1 \\ \mathbf{x_2} \in P_2 \end{aligned} \tag{2.13}$$

Unfortunately, this is not a valid linear program. However, as we have seen above, if the vectors $\mathbf{x^1}, \ldots, \mathbf{x^k}$ are the corners of a bounded convex polytope, then any point $\mathbf{x}$ within the polytope can be represented as $x = \sum_{j=1}^{k} \lambda^j x^j$ for some set of λ^j such that $\sum_j \lambda^j = 1$. Let J_i be the set of all corners of the polytope P_i. For $i = 1, 2$ let $\mathbf{x_i^j}$, $j \in J_i$ be the corners of the bounded convex polytope P_i. Using this representation we can rewrite the above program into the following valid linear program:

$$
\begin{aligned}
\min \sum_{j \in J_1} \lambda_1^j \mathbf{c'}_1 \mathbf{x}_1^j + \sum_{j \in J_2} \lambda_2^j \mathbf{c'}_2 \mathbf{x}_2^j \\
\sum_{j \in J_1} \lambda_1^j \mathbf{D}_1 \mathbf{x}_1^j + \sum_{j \in J_2} \lambda_2^j \mathbf{D}_2 \mathbf{x}_2^j = \mathbf{e} \\
\sum_{j \in J_1} \lambda_1^j = 1 \\
\sum_{j \in J_2} \lambda_2^j = 1 \\
\forall i, j \quad \lambda_i^j \geq 0
\end{aligned}
\tag{2.14}
$$

We call the representation of (2.14) the *master* program. Each "corner" of the feasible region is a valid solution to the linear program. Representing the feasible region of polytopes P_1 and P_2 with the corners of the region usually takes an exponential number of points. Thus, we have transformed the original program (2.11) which had $m_0 + m_1 + m_2$ constraints to a program with only $m_0 + 2$ constraints but possibly exponentially many variables.

Fortunately, we can use the revised simplex method which only requires $m_0 + 2$ columns to solve the LP solve shown in 2.14 and thus only examines $m_0 + 2$ variables of the master program. Let the dual variables of the first m_0 constraints be called the vector $\mathbf{p}$ and the dual variables of the last two constraints be r_1 and r_2 respectively.

We now require a method by which to generate corners of the polytope $\mathbf{x}_1^j$ and $\mathbf{x}_2^j$ for the revised simplex method to examine. This is done by creating one *subproblem* or *slave program* for each polytope P_i of the form:

$$
\begin{aligned}
\min (\mathbf{c}_i - \mathbf{p}\mathbf{D}_\mathbf{i}) \mathbf{x}_i \\
\mathbf{A}_\mathbf{i} \mathbf{x}_\mathbf{i} = \mathbf{b}_\mathbf{i} \\
\mathbf{x}_\mathbf{i} \geq \mathbf{0}
\end{aligned}
\tag{2.15}
$$

This slave program is almost identical to the program shown in (2.12). The only difference is that the objective function of the slave program (2.15) is modified by an extra cost $\mathbf{pD_i}$ it incurs because of its contribution to the constraint $\mathbf{D_1x_1 + D_2x_2 = e}$ in the original program.

Intuitively, the slave program must pay a price determined by $\mathbf{p}$ for any contribution to the constraint in the master program. This price is determined by the dual variable for that constraint. This is consistent with our earlier discussion of dual variables representing a *price* which must be paid to break a constraint. When the Dantzig-Wolfe algorithm iterates it fixes the price $\mathbf{p}$ of the interaction constraints to be constant during the solution of the sub-problems. Fixing the prices is what allows us to separate the master problem into sub-problems.

When each slave program is solved, feasible points (corners) of the polytopes P_1 and P_2 are created which are then added to the master program following the conventions of delayed column generation. While executing the Dantzig-Wolfe algorithm, we will generate several of these corners. We can now state the full Dantzig Wolfe decomposition algorithm:

Dantzig-Wolfe Decomposition

- A typical iteration starts with a total of $m_0 + 2$ corners of P_1 and P_2. Solving the master program yields a set of dual variables $(\mathbf{p}, \mathbf{r_1}, \mathbf{r_2})$.

- Form and solve each slave program which adds an entry to the set of corners $\mathbf{x}_i^1$ and $\mathbf{x}_i^2$. If $\mathbf{x}_i^1 = \mathbf{x}_{i-1}^1$ and $\mathbf{x}_i^2 = \mathbf{x}_{i-1}^2$ then terminate.

- Otherwise, take the corners $\mathbf{x}_i^1$ and $\mathbf{x}_i^2$ from solving the slave programs and add them as variables to the master program. Generate the columns associated with these variables and insert them into the master program constraint matrices.

  Also store the costs c_i^1 and c_i^2 of the slave solutions $\mathbf{x}_i^1$ and $\mathbf{x}_i^2$ neglecting the dual prices $\mathbf{p}$ these become objective function coefficients in the master program.

- Repeat until termination.

In summary, we have transformed the original linear program, into a *master program* which uses several corners of the subproblem feasible regions to find the optimal solution. These corners are generated via solving *slave programs* which are separate subproblem linear programs with modified costs.

We have presented here a version of the Dantzig Wolfe algorithm that requires the subprograms to have feasible regions which are bounded. The algorithm generalizes to unbounded polytopes, but for simplicity of presentation, the details of handling unbounded polytopes are omitted and the reader is referred to (Bertsimas and Tsitsiklis (1997)). The Dantzig-Wolfe algorithm is guaranteed to terminate because it is a special case of the revised simplex method combined with delayed column generation.

While we have developed the algorithm for the case of only two slave programs or subdivisions of a company, it is equally applicable to any number of subdivisions of the problem. In fact, we will use this decomposition to coordinate any number of agents.

Dantzig-Wolfe decomposition solves the original linear program (2.11) efficiently. It is efficient in the same way that linear programming is typically efficient: While it is possible to construct problems where it will take an exponential number of iterations, in general it terminates quickly. This efficiency has been validated experimentally many times by ourselves and others, but the proof of an average case running time remains an open problem. For the problems we have studied, we find that the algorithm iterates approximately 3-10 times before termination. If there are N subproblems requiring $O(t_1)$ time each and the master requires $O(t_2)$ time, then the total running time per iteration is $O(Nt_1 + t_2)$.

2.2.4 An economic interpretation of Dantzig-Wolfe decomposition

The decomposition method can be described in terms of an economic context. Imagine that an organization has two divisions which must meet a common objective of minimizing cost to the company while still producing a certain amount of widgets in each division. The widgets produced in each division are different but use the same raw material which is only available in a limited quantity. This limited quantity can be represented by the constraint $\mathbf{D_1 x_1} + \mathbf{D_2 x_2} = \mathbf{e}$. Each division i must minimize its own cost $\mathbf{c_i' x_i}$ while meeting its own constraints $\mathbf{A_i x_i} = \mathbf{b_i}$ of producing a certain amount of widgets. The Dantzig-Wolfe algorithm is guaranteed to find the optimal solution for this problem such that the total cost to the organization is minimized.

2.3 MDPs, linear programs, and duals

Markov Decision Processes were first introduced in order to solve the "optimal control" problem. In this problem, there is the notion of the state of a system, which describe quantities of interest in the system. There are actions a controller can take which cause the state of the system to transition from one state to another. The result state of a given action may be stochastic, that is taking an action in a given state does not always take you to the same next state. A Markov Decision Process (MDP) is a tuple $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \boldsymbol{\alpha}\}$ defined as follows:

- a set of states $\mathcal{S} \in \{s_1, s_2, \ldots, s_n\}$, $|\mathcal{S}| = N$
- a set of actions $\mathcal{A} \in \{a_1, a_2, \ldots, a_m\}$, $|\mathcal{A}| = M$
- a set of transition probabilities $T(s', a, s) = p(s' \mid s, a)$, where p is a probability distribution

- a set of costs $c : \mathcal{S} \times \mathcal{A} \mapsto \Re$
- a discount factor $\gamma \in [0, 1]$
- a probability distribution over initial states $\boldsymbol{\alpha} : \boldsymbol{\alpha} \in \Re^N, \sum_i \alpha_i = 1$

For any MDP there is a *value function* which indicates how desirable any given state in our system is. The value function may be determined using the *Bellman equation* defined in (2.17):

$$V(s_i) \quad : \quad \mathcal{S} \mapsto \Re \tag{2.16}$$

$$V(s_i) \quad = \quad \min_a \left(c(s_i, a) + \gamma \sum_{j=1}^{N} p(s_j \mid s_i, a) V(s_j) \right) \tag{2.17}$$

Achieving some states is desirable (or less costly), that is we would like to *control* the system to achieve a desired state or set of states. The question to be answered in such a problem is: "What is the best action to take from a given state so that I achieve the lowest possible cost". What we would like is a *policy* to tell us which action to take in any given state. A policy then, is a function which takes a state as an input and tells us what action to take. A *deterministic* policy will only tell us one action we should take with 100% certainty. A *non-deterministic policy* tells us to take each possible action with a certain probability. That is:

- a deterministic policy $\pi(s_i) : \mathcal{S} \mapsto \mathcal{A}$, or $\pi(s_i) = a_i$
- a non-deterministic policy:

$$\forall_{i,j} \quad \pi(s_i) = p(s_i, a_j) \in [0, 1] \tag{2.18}$$

$$\sum_j p(s_i, a_j) = 1 \tag{2.19}$$

The *optimal policy* is the policy which chooses the best possible action in each state. If we know the value function V then the optimal policy can be stated as follows:

$$\pi(s_i) \quad = \quad \arg\min_a \left(c(s_i, a) + \gamma \sum_{j=1}^{N} p(s_j \mid s_i, a) V(s_j) \right) \tag{2.20}$$

One method by which the optimal policy can be computed is to phrase the MDP $\mathcal{M}$ as the Bellman linear program. If we pick a distribution over starting states $\alpha \subset \Re^N$ with $\alpha > 0$ and $\sum_i \alpha_i = 1$ then the Bellman LP was shown in (Puterman (1994)) to be:

$$\begin{aligned} &Maximize\ \alpha \cdot \mathbf{V} \\ \forall_a :\ &\mathbf{V} \leq \mathbf{c_a} + \gamma \mathcal{T}_\mathbf{a} \mathbf{V} \end{aligned} \tag{2.21}$$

where $\mathbf{V}$ is the vector form of the value function whose size is $|\mathcal{S}|$, one value per state in the MDP. $\mathbf{c_a}$ is the vector of immediate costs for taking action a and $\mathcal{T}_\dashv$ is the matrix representation of the transition probabilities. The inequalities between vector quantities above are meant to hold component-wise. Note that the constraints in equation (2.21) come directly from equation (2.17). There is one $\leq$ constraint for each possible action, and this is how the minimum in (2.17) is achieved in the linear program. In solving this linear program (LP) we will determine the value function for the MDP. This can then be used to determine the optimal policy using equation (2.20).

2.3.1 The dual of the Bellman LP

$$\begin{aligned} \min_{\mathbf{f}} \ & \mathbf{c} \cdot \mathbf{f} \\ \mathbf{f} - \gamma T \mathbf{f} &= \alpha \\ \mathbf{f} &\geq \mathbf{0} \end{aligned} \tag{2.22}$$

The dual of the Bellman LP gives us an interesting alternative from which to view the problem of finding an optimal policy directly without finding the value function. The dual of the Bellman LP is shown in (2.22).

Every state s_j has some set of valid actions $a_k \in \mathcal{A}_{s_j}$ which may be executed in the that state. These are called valid *state action pairs*. The vector $\mathbf{f}$ represents the expected number of times we perform a given action from each state. We call each real valued variable f_i in this vector a *state action visitation frequency*. The vector $\mathbf{f}$ contains one real valued state action visitation frequency for every valid state action pair in the MDP. Thus, the size of the vector $|\mathbf{f}|$ is simply the sum over states of the number of actions available in each state: $|\mathbf{f}| = \sum_{\mathbf{s} \in \mathcal{S}} \text{num actions}_{\mathbf{s}}$.

We now rewrite the bellman dual as the following linear program which conforms more closely with the LPs discussed in the linear programming review:

$$\begin{aligned} \min_{\mathbf{f}} \ & \mathbf{c} \cdot \mathbf{f} \\ \text{A } \mathbf{f} &= \mathbf{b} \\ \mathbf{f} &\geq \mathbf{0} \end{aligned} \tag{2.23}$$

Each column of the A matrix represents a unique state action visitation frequency. The objective function has a cost for each state action visitation frequency. These costs are defined by the cost vector $\mathbf{c}$.

Each row of the matrix A represents one state in the MDP and enforces a flow constraint. This constraint ensures that the sum of state action visitation frequencies leading into a state minus the sum of state actions visitation frequencies leading into of a state must be equal the number of times one starts in a given state at the beginning of execution:

$$\forall s_i \quad \sum_k f(s_i, a_k) - \sum_{j,k} f(s_j, a_k) p(s_i \mid s_j, a_k)\gamma = \alpha_i$$

It should be noted that there is a direct mapping from a policy $\pi(s_i)$ to a set of state action visitation frequencies as discussed in ((Guestrin, 2003, Chapter 2)) and (Puterman (1994)):

$$f(s_i, a) = \sum_{t=0}^{\inf} \sum_{s_j} \gamma^t \pi(a \mid s_i) \sum_{a_k} p(s_j^t \mid s_i^{t=0}, a_j)\pi(a_k \mid s_i) \tag{2.24}$$

There is not necessarily a one to one mapping in the other direction. It is possible that there exists a state s_j for which $\sum_k f(s_j, a_k) = 0$. That is the optimal policy does not pass through this state. In this case, when defining the policy $\pi(s_j)$ we can set $\pi(s_j)$ to be any probability distribution over the actions available in that state.

An error bound

It was also shown in (Guestrin, 2003, chapter 6) that given a set of state action visitation frequencies there exists an error bound from the optimal policy. We first determine the *dual violation* $\delta(s_i)$ of a given set state visitation frequencies $f(s_i)$ for

every joint state s_i:

$$\delta(f(s_i)) = \sum_k f(s_i, a_k) - \alpha(s_i) - \gamma \sum_{j,k} f(s_j, a_k) p(s_i \mid s_j, a_k) \tag{2.25}$$

Given these dual violations we can compute the error between a value function V_f induced by a vector of state action visitation frequencies $\mathbf{f}$ and the optimal value function V^* as:

$$\|V^* - V_f\|_{1,\alpha} \leq sum_i \delta(f(s_i)) V_f(s_i) \tag{2.26}$$

where the $\mathcal{L}_1$ norm is defined by $\|V\|_{1,\alpha} = \sum_i \alpha s_i |V(s_i)|$.

This is a powerful tool, as we can now take a given vector of state action visitation frequencies $\mathbf{f}$ and determine an error bound of the value function from the optimal value function. The caveat is that the error bound requires a sum over an exponential number of states.

2.3.2 Converting an MDP to the dual linear program

The above discussion leads to a general procedure for converting a Markov Decision Process to the dual of the Bellman linear program.

Assume that we start with a Markov decision process $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, \boldsymbol{\alpha}\}$. Furthermore, assume that there are n state-action pairs and that the number of valid actions in every state is n_a. Assume an arbitrary ordering on the state/action pairs such that $order(s_i, a_k)$ returns a unique integer. We use the ordering function: $order(s_i, a_k) = i \times n_a + k$. If states and actions are numbered beginning at zero, this function returns a unique positive integer greater than or equal to zero. Given this ordering function we can now convert an MDP to a dual linear program using the

following sequence of steps:

- For every state $s_j \in \mathcal{S}$
  - Create a row in the matrix A which has n columns with every value set to zero
  - For any state s_i with action a_k having non-zero probability of entering state s_j, calculate the column index to be $order(s_i, a_k)$ and set the value of that column to be $-p(s_i \mid s_j, a_k)\gamma$.
  - For every action leaving state s_j calculate the column index to be $order(s_j, a_k)$ and add $+1.0$ the value of that column.
- Set the values of the cost vector $\mathbf{c}$: For every state j and every action k calculate an index $i = order(s_j, a_k)$. Set the cost at c_i to be the cost for executing action k in state j
- Set the vector $\mathbf{b} = \boldsymbol{\alpha}$

Though we have used an ordering function for the special case of having the same number of actions per state, handling a variable number of functions per state simply requires a different ordering function. We will now demonstrate the procedure applied to a simple example.

An example using the dual of the Bellman LP

In general, we will use the dual formulation of the linear program as this is a more efficient representation for our decomposition method. To demonstrate how a Markov Decision Process can be transformed to an equivalent linear program in dual form, we present the example in figure (2.3).

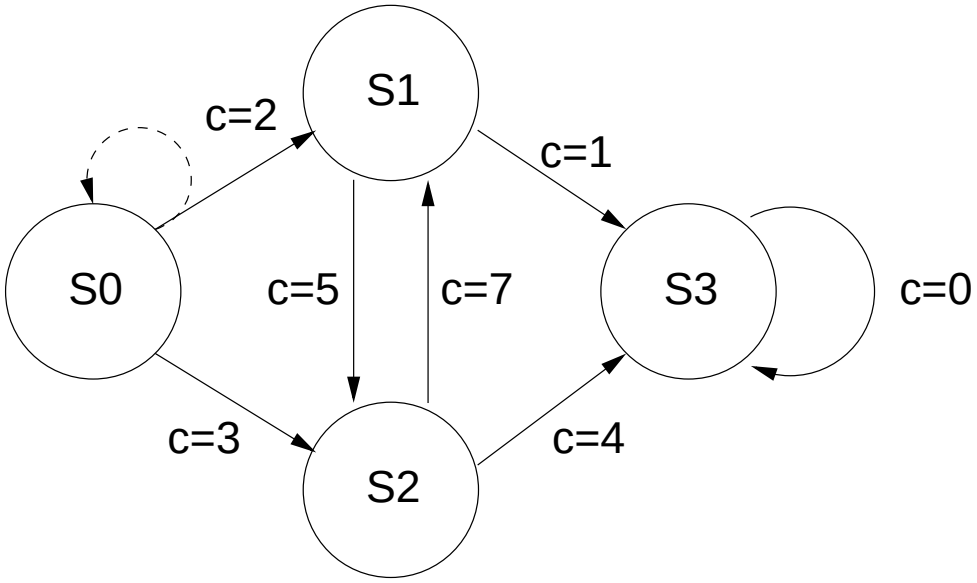


Figure 2.3: A simple MDP, the start state is s_0 and we wish to reach the terminal state s_3 while minimizing cost. The costs are labeled on each each. We can introduce action uncertainty by adding the dashed arc from state s_0.

Figure (2.3) depicts a simple Markov decision process. We begin by neglecting the transition indicated by the dashed line and assuming that the discount factor γ is equal to 1.0. It is unnecessary to represent the terminal state since its value will always be zero. Transitions out of the goal state will always lead back to the goal state and thus there is no need to solve for state action visitation frequencies from any goal state. This leaves us with three states we must represent in the A matrix. Each state has two actions thus the number of state action pairs is six.

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 0 & -1 & 1 & 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 5 \\ 7 \\ 4 \end{bmatrix}$$

We see that transitions leading into a state cause a -1 to be represented in the A matrix. Transitions leading out of the state cause a $+1$ to be present in the A matrix. The decision variables f_i are:

$$\mathbf{f} = \begin{bmatrix} f(s_0, a_0) \\ f(s_0, a_1) \\ f(s_1, a_0) \\ f(s_1, a_1) \\ f(s_2, a_0) \\ f(s_2, a_1) \end{bmatrix}$$

In this case, we arbitrarily decided that the zero'th action in each state is the action which leads towards the top of the page. Representing this as the linear program in (2.23) and using a linear program solver we obtain the optimal solution to be:

$$\mathbf{f} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Though we have shown the conversion for a deterministic MDP, it is possible to represent any MDP. Imagine that the dashed arc is the result of $f(s_0, a_0)$ 50% of the time and the solid arc of $f(s_0, a_0)$ is the resulting state 50% of the time. We can represent this MDP simply by changing the A matrix to be:

$$A = \begin{bmatrix} 0.5 & 1 & 0 & 0 & 0 & 0 \\ -0.5 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 0 & -1 & 1 & 1 \end{bmatrix}$$

2.4 Mixed Integer Programming

We have shown some linear programming techniques and examples above. Sometimes a problem is better modeled with the addition of *integer* variables or constraints involving integer variables. If we take a linear program and force some subset of variables to be integer, then the resulting program is a *mixed integer program*. In general, we can phrase a mixed integer program as the following:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \mathbf{c}'\mathbf{x} + \mathbf{d}'\mathbf{y} \\ A\mathbf{x} + B\mathbf{y} = \mathbf{b} \\ \mathbf{x}, \mathbf{y} \geq \mathbf{0} \\ \mathbf{x} \in \text{integer} \end{aligned} \tag{2.27}$$

Thus, the decision variables $\mathbf{x}$ may only take discrete integer values, while the decision variables $\mathbf{y}$ are continuous real variables. At times, we only allow the integer variables to take on the values zero or one: $x_i \in \{0, 1\}$. This kind of mixed integer program is called a *zero or one integer program* (ZOIP) or a *binary integer program*.

The addition of integer variable allows for a much more expressive modeling framework than linear programming. With integer variables we can model the following details for a task which are not easily modeled (or not possible to model) with a linear program:

Binary Choice Using a binary variable we can select between two alternatives.

Dependent Decisions Sometimes a second decision may only be made if a first decision is chosen to be a particular value. This can be represented using $x \leq y$ where $x = 1$ may only be true if $y = 1$

Disjunctive Constraints Sometimes we would like to satisfy one set of linear constraints *or* a different set of linear constraints. This can be accomplished with:

$$\begin{aligned} A\mathbf{x} &\geq y\mathbf{b} \\ C\mathbf{x} &\geq (1-y)\mathbf{d} \\ y &\in \{0,1\} \end{aligned} \tag{2.28}$$

Setting $y = 0$ we satisfy $C\mathbf{x} \geq \mathbf{d}$. With $y = 1$ we satisfy $A\mathbf{x} \geq \mathbf{b}$. Here we assume that A and b are greater than zero, but this constraint can be relaxed via the addition of a slack variable. Thus we can use (2.28) to satisfy the disjunction of the two sets of constraints.

Relations between variables If we would like at most one variable to be true we can implement $\sum_i x_i \leq 1$ and make $x_i \in \{0,1\}$

Restricting possible values for a variable If we would like a variable x to take on a restricted set of values from the real line ($x \in \{a_1, a_2, \ldots, a_n\}$) we can implement the following constraints:

$$\begin{aligned} x &= \sum_{j=1}^{m} a_j y_j \\ \sum_{j=1}^{m} y_j &= 1 \\ y_j &\in \{0,1\} \end{aligned} \tag{2.29}$$

2.5 The Branch and Bound algorithm

There are several methods for solving mixed integer programs. We will describe only the *branch and bound algorithm* but there are other methods such as Gomory

cuts, simulated annealing and many others which are described in (Bertsimas and Tsitsiklis (1997)). To begin, we describe how one might *relax* a mixed integer program by deleting constraints. In particular, the branch and bound algorithm removes constraints forcing variables to take on integer values. This technique is described below.

2.5.1 Linear Programming Relaxation

If we remove some constraints from a linear program or mixed integer program we obtain a *relaxed* form of that program. If we take the integer program:

$$\begin{aligned} \min_{\mathbf{x_{int}}} \mathbf{c}'\mathbf{x_{int}} \\ A\mathbf{x_{int}} = \mathbf{b} \\ \mathbf{x_{int}} \geq \mathbf{0} \\ \mathbf{x_{int}} \in \text{integer} \end{aligned} \tag{2.30}$$

We obtain a *linear programming relaxation* of (2.30) by removing the integer constraints to obtain:

$$\begin{aligned} \min_{\mathbf{x_r}} \mathbf{c}'\mathbf{x_r} \\ A\mathbf{x_r} = \mathbf{b} \\ \mathbf{x_r} \geq \mathbf{0} \end{aligned} \tag{2.31}$$

Because we are removing constraints we are guaranteed that the optimal cost for (2.31) $\mathbf{c}'\mathbf{x}_\mathbf{r}^*$ will be less than the cost of the integer program $\mathbf{c}'\mathbf{x}_\mathbf{int}^*$. Thus, when

we solve any mixed integer program which has fewer constraints than some original mixed program, we obtain a lower bound on the optimal cost of the original program: $\mathbf{c}'\mathbf{x}^*_{\mathbf{r}} =$ lower bound $\leq \mathbf{c}'\mathbf{x}^*_{\mathbf{int}}$. If we furthermore assume that all entries of c are integer then we know that this lower bound must also be integer.

2.5.2 Branch and Bound

The Branch and Bound algorithm begins by solving the fully relaxed linear program. That is, it solves (2.31). If the solution $\mathbf{x}^*_{\mathbf{r}}$ contains non-integer values, we can then force one or more of them to be integers with a specific value and solve a different relaxation with those variables set to be constants.

We thus *branch* the problem into a set of subproblems where some variables are fixed to be integer constants. Each time we solve a relaxed linear program, we obtain a lower bound on the optimal cost as explained in the previous section. If we solve a program and all variables are integer we obtain an *upper* bound on the value of the mixed integer program.

This is best explained following the explanation of an example from (Trick (1997)). We would like to solve the following integer program:

$$\begin{aligned} \min\ & -8x_1 - 11x_2 - 6x_3 - 4x_4 \\ & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 15 \\ & x_j \in \{0,1\} \quad j = \{1,2,3,4\} \end{aligned} \tag{2.32}$$

If we remove all the integer constraints and solve the linear programming relaxation we obtain $x_1 = 1$, $x_2 = 1$, $x_3 = 0.5$, $x_4 = 0$. The value of the objective function is -22. We now know that no integer solution to (2.32) will have a value less than -22.

There is only one variable which has a fractional (non-integer) value, that is x_3. We now force x_3 to be integer by *branching* on x_3. That is we begin with the top node of a tree being the fully relaxed linear program and create *child nodes* or *subproblems* representing cases where we force $x_3 = 0$ or $x_3 = 1$. In each branch, we solve a relaxed problem, where all variables other than x_3 can take on any real value but x_3 is forced to be 0 or 1. We can see this initial tree shown in figure (2.4). We will use the terms "node" (as in node of the tree), and "subproblems" interchangeably.

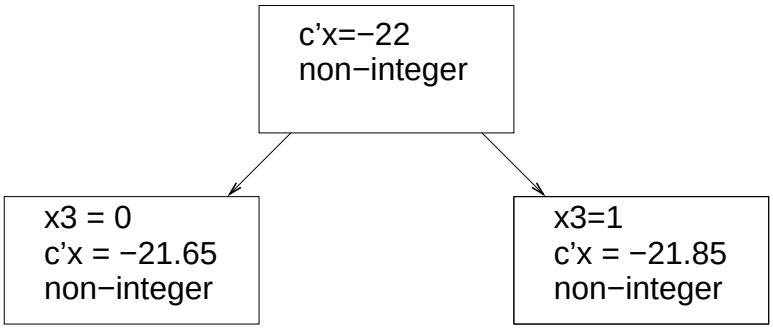


Figure 2.4: A partial tree for the branch and bound algorithm. $c'x$ is the value of the objective function

Solving the branch where $x_3 = 0$ we obtain:

- $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0.667$ with value $c'x = -21.65$

For the branch where $x_3 = 1$ we obtain:

- $x_1 = 1, x_2 = 0.714, x_3 = 1, x_4 = 0$ with value $c'x = -21.85$.

We know that the optimal solution can be no less than -21.85 (since that is the lower of the two branches). In fact, we know that it cannot be any lower than -21 since all the multiplicative constants and decision variables must be integers. Thus the optimal cost of the objective function must also be an integer higher than -21.85.

Unfortunately, neither branch leads to an integer solution, so we must branch again. We must choose whether to further explore the left or right node. We are

choosing an *active subproblem*. An active subproblem is a node of the tree which is feasible and has a "useful" value for the objective function. "Useful" will be defined momentarily.

The choice of an active subproblem can be made with a simple heuristic such as the lower value $c'x$ of the two branches, depth first search, breadth first search, or with domain dependent guidance heuristics.

Choosing the right branch where $x_3 = 1$ we now force a non-integer variable of that branch (remember $x_2 = 0.714$) to be integer by branching again. After solving the subproblems of $x_3 = 1$ node we have:

- $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 1$ with value $c'x = -18$
- $x_1 = 0.6, x_2 = 1, x_3 = 1, x_4 = 0$ with value $c'x = -21.8$

We now have at least one subproblem with an integer solution which is a feasible solution to the original problem. This gives us an upper bound on the value of the original problem. We *know* that we can achieve a minimum value of at least -18 and possibly better. If we have a subproblem with an integer solution we will no longer need to branch on that problem. That problem is said to be *inactive*, since there is no need to search further below that node. The current tree is shown in figure (2.5).

Branching on the non-integer subproblem of the $x_3 = 1, x_2 = 1$ node, we obtain the following subproblem solutions:

- $x_0 = 1, x_2 = 1, x_3 = 1, x_4 = 1$ with value $c'x = -21$
- $x_1 = 1, x_2 = 1, x_3 = 1$ is infeasible. There is no value of x_4 which can meet the linear constraint in (2.32).

Note that given the lower bound produced by the top node of the tree (-21), there is no need to investigate the branch where $x_3 = 0$ because we now have a feasible integer

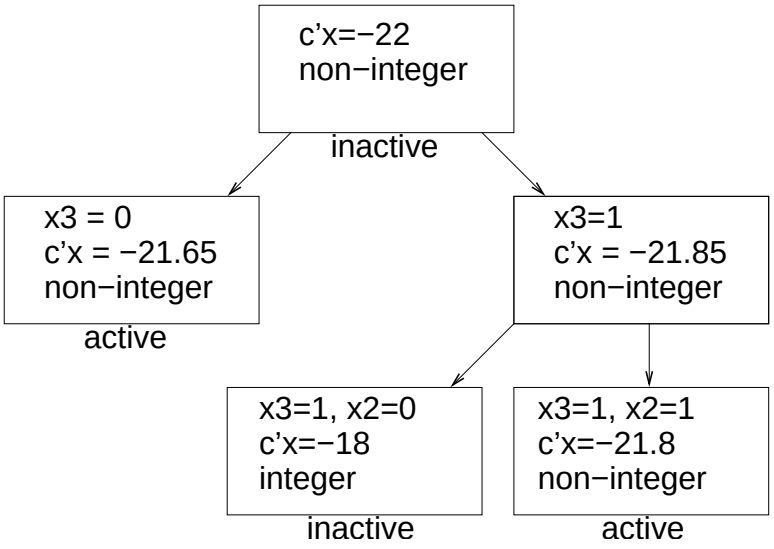


Figure 2.5: A partial tree for the branch and bound algorithm. Nodes labeled as active must still be considered.

solution which achieves this cost. Thus that subproblem (and any child nodes of that subproblem) is said to be *fathomed* meaning it requires no investigation because its best cost is worse than a cost which can already be achieved by some feasible integer solution we have already found. Fathomed subproblems are said to be inactive. Any nodes which are infeasible are also said to be inactive.

We have performed a complete branch and bound algorithm for the problem in (2.32). Figure (2.6) depicts the entire branch and bound tree. Until all nodes are inactive (either fathomed or infeasible), we must continue searching all nodes of the tree. The general algorithm may be stated as follows:

1. Solve the linear relaxation of the currently selected subproblem or node (beginning with the original mixed integer program). If all variables in the solution are integer, then this branch of the tree is terminated with this subproblem and we have an upper bound created by the integer solution. The tree is also terminated by an infeasible subproblem. If the solution is non-integer, then we

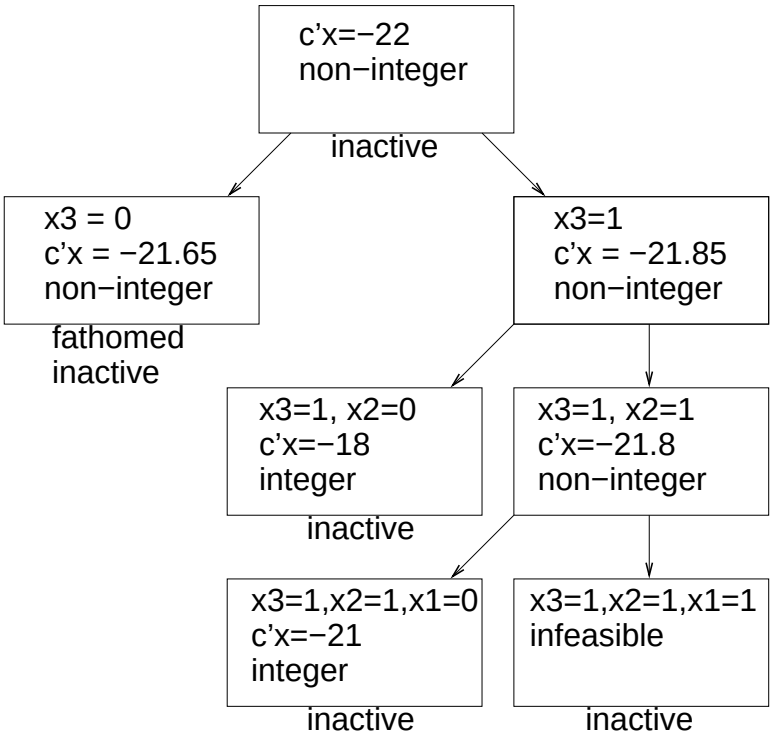


Figure 2.6: The complete branch and bound tree for problem (2.32).

must branch on a non-integer variable.

2. Choose an active subproblem and branch on a non-integer variable. A subproblem or node of the tree is inactive when:

   - The subproblem has already been branched.
   - All variables in the subproblem solution are integer. This leads to an upper bound on the optimal cost which can be used to fathom other subproblems.
   - The subproblem is infeasible.
   - You can fathom this subproblem using an upper bound from another subproblem.

3. Repeat until there are no active subproblems.

The branch and bound algorithm causes us to solve many linear programs. Every time we create a node in the tree, we must solve some relaxed form of the original mixed integer program. Fortunately, it is very rare that we need examine the whole tree. Often there are problem dependent heuristics for guiding the search and we need not resort to full breadth first or depth first search. It is also possible to re-use solutions from the parent node to jump start solutions of the child nodes.

Computational Efficiency of Branch and Bound

Branch and bound is guaranteed to converge to the optimal solution. However, there are no known polynomial time algorithms for solving general mixed integer programs. In fact, because many NP hard problems can be phrased as integer programs, solving a general integer program in polynomial time would mean $P = NP$.

Branch and bound may take exponential time in the worst case, however, it often produces acceptable solutions in a reasonably short period of time. For certain tasks,

good heuristics are available to guide the search, much reducing the computation time and making branch and bound a solid algorithm for practical applications. Many commercial packages use branch and bound as their integer program solver.

2.6 Conclusions

The above discussions are meant to provide sufficient breadth to understand the material in this thesis. We have attempted to gather the main results from linear programming and Markov Decision Process theory in this chapter so that the reader may understand the multi-agent coordination algorithms and multi-agent coordination language presented in the next chapter. The above background material should be sufficient to gain at least an intuition for the algorithm.

The multi-agent coordination algorithms presented in this thesis incorporate most of the methods described in this chapter. More specifically we combine Dantzig-Wolfe decomposition, delayed constraint generation, and at times branch and bound to perform our multi-agent coordination. The resulting algorithms are fast, efficient, have low communication requirements and, for many useful tasks, are guaranteed to converge to the optimal policy.

Bibliography

Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific.

Chvatal, V. (1983). *Linear Programming*. W.H. Freeman and Company.

Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, 2nd edition.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press.

Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Stanford University.

Puterman, M. L. (1994). *Markov Decision Problems*. Wiley.

Rardin, R. (1998). *Optimization in Operations Research*. Prentice Hall.

Spielman, D. and Teng, S. (2001). Smoothed analysis: Why the simplex algorithm usually takes polynomial time. In *Proc. of the 33rd ACM symposium on the theory of computing*.

Trick, M. (1997). A tutorial on integer programming.

Chapter 3

Multi-Agent Coordination Using Dantzig-Wolfe Decomposition and Branch and Bound Search

3.1 Introduction and Motivation

In recent years, the design of cooperative multi-robot systems has become a highly active research area within robotics (Bennewitz et al. (2001); Y.U. et al. (1997); Goldberg and Matarić (2000); Kitano (1998); Roumeliotis and Bekey (2000); Salido et al. (1997)). For many coordination tasks, there does not exist a principled and efficient method for multi-robot coordination. In this chapter, we build on previous work in linear programming and Markov decision processes (MDPs) to create a new method for multi-robot coordination which overcomes some of the difficulties in each of the prior methods while remaining intuitive and easy to use.

Many planning problems in robotics are best phrased as MDPs because of their ability to handle uncertainty in action outcome. These MDPs are often defined over

world states. It is also possible to handle uncertainty in state (also called partial observability) by defining an MDP over belief states (Kaelbling et al. (1998)), but we have not investigated this kind of MDP in this thesis. Unfortunately, any MDP describing a multi-agent problem grows in size exponentially with the number of agents if phrased in a straightforward manner.

Throughout this chapter we will refer to the kinds of problems we are solving as multi-agent coordination problems, multi-agent MDPs, or multi-agent planning problems.

We bring together resource-allocation techniques from the linear programming and MDP literature to solve these problems. In particular, we propose a general technique for decomposing multi-robot MDP problems into "loosely coupled" MDPs which interact only through resource production and consumption constraints. Prices can be attached to any linear function of the state action visitation frequencies of each robot.

This loosely coupled approximation of a general multi-agent MDP is obtained by phrasing the original multi-agent planning problem as a large mixed integer program (MIP) or linear program (LP). In this mixed integer or linear program approximation of the original problem, we only model important inter-robot constraints and resources shared by the robots. These representations avoid the "curse of dimensionality", the exponential growth of multi-agent MDPs. Thus, MIPs and LPs become our modeling language for multi-agent coordination planning problems.

Given a MIP or LP model of a multi-agent problem, we offer a centralized and decentralized solution method for each model. These methods are based on off the shelf MIP and LP solution methods. One of our main contributions is the modeling of multi-agent MDP problems as a MIP or an LP and the discovery that the Dantzig-Wolfe distributed solution method is best suited for solving these kinds of models.

Our approach generalizes a large body of previous literature in multi-robot systems, including prior work by Guestrin and Gordon (Guestrin and Gordon (2002)). Our decentralized algorithm can distribute computation so that each robot reasons only about its own local interactions, and the algorithm always produces the same answer as the centralized planning solution methods.

In summary, we can find the global optimum to an approximate MIP or LP model of the multi-agent coordination problem in a very efficient way. For a large class of planning problems, our MIP or LP model of the problem can also be made to *exactly* represent the original multi-agent coordination problem. When the MIP or LP model is exact, we can guarantee an optimal solution to the original multi-agent planning problem we are modeling.

The rest of the chapter is structured as follows: We first describe straight forward planning for multi-agent MDPs and the curse of dimensionality inherent in that representation. We then describe a novel multi-agent task representation using mixed integer programming along with a centralized solution algorithm which avoids this curse of dimensionality. The simpler linear program model is then described along with its corresponding centralized solution algorithm. We then present the Dantzig-Wolfe decomposition algorithm and how it is applied to both models yielding an efficient solution technique for both the MIP and LP models. Experimental results solving a problem using both models are then presented.

3.2 The standard multi-agent MDP representation

Unfortunately, existing MDP planning techniques generally scale poorly to multi-robot systems because of the curse of dimensionality: in general, it is exponentially harder to solve an MDP for N agents than it is to solve a single-agent MDP because

the state and action space for N robots can be exponentially larger than for a single-robot system. This enormous complexity has confined MDP planning techniques largely to single-robot systems.

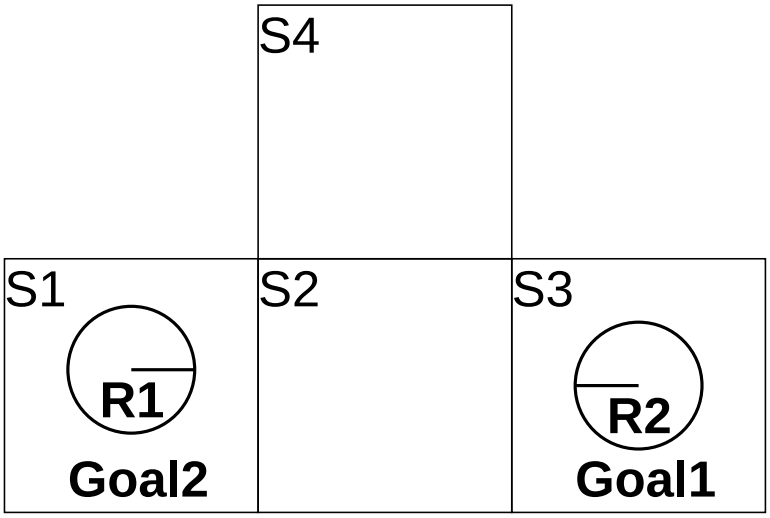


Figure 3.1: A simple two robot path planning problem. Robot R1 must go to the grid cell Goal1 and robot R2 must arrive at the grid cell Goal2. There are four positions: S1,S2,S3, and S4. The two robots must not cross over each other or move into the same grid cell.

To illustrate this complexity, let us describe the state and action space for a two robot path planning problem shown in figure 3.1. There are four grid cells or locations labeled $S1, S2, S3, S4$. There are two robots: $R1$ and $R2$. Robot $R1$ must plan a path to $Goal1$ and robot $R2$ must plan a path to $Goal2$. The robots cannot cross over each other, nor may they be in the same cell at one time. The robots have actions: move north, move east, move south, move west, and wait. Actions are deterministic: when you attempt to perform an action you have a 100% probability of succeeding. With a cost of 1 per action, the objective is for the robots to reach their goals while minimizing cost and not crashing into each other or the walls.

If Robot $R1$ totally ignored robot $R2$, we could model the problem for $R1$ using an MDP with 4 states: the robot being in each possible position. Similarly the robot

would have 5 actions (north, east, south, west, wait) in each state. If robot $R2$ ignored $R1$, then $R2$ would have a separate MDP with 4 states and 5 actions. Thus, totally ignoring other robots in the world, we have two separate MDPs with small state and action spaces. Unfortunately, they cannot ignore each other since we could not then guarantee a collision free path.

A simple way to solve this problem is to model the state of both agents in one much larger MDP. The location of *both* the robots at any time is the state of this new larger problem. In this MDP, we keep track of the simultaneous location of both robots called the *joint location* or *joint state*. We also keep track of which actions both robots are executing simultaneously at a given time (called the *joint action*).

For each of the 4 possible states (locations) of the first robot we must keep track of the position of the second robot which might be one of four states. This leads to a *joint state space* of $4 \times 4 = 16$ states. Similarly, keeping track of the simultaneous actions of both agents the *joint action space* would be $5 \times 5 = 25$ joint actions.

By controlling the joint actions (eg. disallowing some joint actions) for each joint state we can ensure collision free paths. This allows us to minimize the cost for both robots while avoiding collisions. For example, when the robots are in joint state ($R1$ position = S1, $R2$ position = S3), we can disallow the joint action ($R1$ action = east, $R2$ action= west) which would lead to a collision.

In general, by reasoning simultaneously about several robots the *size of the joint state space* grows exponentially in the number of robots. Similarly, the number of the joint actions or *size of the joint action space* also grows exponentially in the number of agents. Since MDP solution methods are polynomial time in the number of states and actions, these methods quickly become too slow and cumbersome for even small multi-agent problems. MDP methods can handle millions of states with tens of actions on todays computers. This could handle a total of two robots performing coordinated

path planning on a grid 100 by 100 in size. This problem has a joint state space size of 10^8, which can barely be solved with today's computers and standard MDP algorithms. This is insufficient for most any useful multi-robot planning. To address this problem of the exponential state space, we will show how these problems are actually only loosely coupled, and how they may be represented by a mixed integer program.

3.3 Loosely Coupled MDPs and the mixed integer programming representation

3.3.1 Intuition and relation to market based approaches

In many cases, robots in a multi-robot system interact only in limited ways. Robots might seek not to collide with each other in specific locations (Bennewitz et al. (2001)), coordinate their locations to carry out a joint task (Kitano (1998); Salido et al. (1997)), or consume a joint resource with limited availability (Burgard et al. (2000); Parker (1996); Zlot et al. (2002)). While these problems are not trivially decomposed, they do not necessarily have the worst-case exponential complexity that characterizes the general case we have described in section 3.2.

Handling this sort of limited interaction is exactly the strength of market-based planning algorithms (Zlot et al. (2002); Gerkey and Mataric (2002)): by focusing their attention on a limited set of important resources and ignoring all other interactions, these algorithms reduce the problem of cooperating with other robots to the problem of deciding which resources to produce or consume. Market-based algorithms are particularly attractive for multi-robot planning because many common types of interactions can be phrased as constraints on resources such as space (two robots

can't occupy the same location at once) and time (a robot can only work on a limited number of tasks at once).

We have described keeping track of the positions and actions of both robots at every time step to ensure they do not run into each other in a path planning task. Another possibility for planning in this domain which avoids the exponential growth in the joint state space is to let the robots plan paths individually, but penalize plans that actually conflict in specific cells at specific times. After changing the penalty *only in specific cells and times where they conflict* we ask the robots to re-plan. Thus, there would be a shared resource per cell (the right to be in that grid cell), but this number of resources would not grow as we added more robots.

In our path planning example, we might decide that a *resource* is the ownership of a particular grid cell. One way we might get the robots to coordinate is to have them "pay" to own or *consume* a particular grid cell so that they might plan a path which includes that grid cell. For example, $R1$ might buy grid cells $S1, S2, S3$ in the example of figure (3.1). All robot $R1$ would need to know is that it owned those cells, and could ignore robot $R2$ when doing its path planning. We could also introduce the notion of time so that a robot could own a particular cell at a particular time step. To solve the problem shown in figure (3.1) we would need a planner to *allocate the resources* of grid cells at particular time steps. An optimal allocation would lead to an optimal joint path plan for the robots. In this case it would correspond to one robot waiting in the grid cell $S4$ as the other robot passed by.

Current market algorithms make the strong assumption that robots can calculate the *utility* of each sub-goal or resource required to complete the overall task *independently of other robots' actions*. In the path planning problem this might mean that each agent would be able to determine how much each grid cell is worth to them independently of all other robots' desired grid cells. The example of figure (3.1) shows

one such example where the robots must both own grid cells at various times and share them with the other robot. From the point of view of these auction/market algorithms, this assumption leads to two difficulties for many multi-robot planning problems.

The first is to compute the probability distribution of the price of each resource at every time step: the optimal price for a resource at time t depends on how much *each* robot produces or consumes between now and time t, and what each robot's state is at time t. The resource usage and state depend on the robots' plans between now and time t, which in turn depends on the price. In our path planning example if one robot buys/consumes a grid cell which is important to another robot's path plan, then the second robot's utility function must somehow model the actions of the first robot.

Worse yet, future resource usage depends on random events which can't be predicted exactly. Assuming independent utility calculations for resources means that auction algorithms cannot model certain forms of future interaction between agents very well. Creating utility functions to model future interactions is very difficult and is generally avoided in the market literature.

In the example of figure 3.1, let us examine the case where robot $R1$ might buy positions $S1, S2, S3$ and keep them for all time. Unfortunately, $R2$ would never be able to reach its goal as it would never be able to use those grid cells. Thus, we may need to reason about owning a particular resource at a given time step which is equivalent to having one resource per grid cell per time step. It may be difficult to resolve conflicts if two robots wish to own the same resource at the same time. Resolving this conflict may lead to other conflicts. How does one resolve these conflicts while still obtaining an optimal joint plan for both robots such that the total cost to the team is minimized? This example shows some of the difficulties inherent in

reasoning about future resource usage.

The second problem comes about when sub-goals truly are dependent upon each other which violates a main assumption of many auction and market algorithms (Gerkey and Mataric (2004)).

Ideally, we require some representation which is not exponential in size as the number of agents increases, and yet can represent future resource usage and easily handle dependent sub-goals. We hope to convince the reader that the mixed integer programming model of multi-agent coordination problems meets these objectives.

3.3.2 The mixed integer programming representation

Our models are designed for multi-robot problems that can be decomposed into separate single-robot MDPs which interact through the production or consumption of fictitious resources. These resources may be physical goods such as fuel; or they may be logical resources such as the right to pass over a bridge at a particular time, the right to explore an area of the environment, or the right to collect reward for achieving a particular subgoal. Time may be part of the individual robot states, in which case a resource could be the right to consume a unit of fuel at a particular time (a futures contract). In the example in figure (3.1), the resources were the right to use a particular grid cell at a particular time.

More formally, we will phrase each individual robot's MDP as the dual linear program described in chapter 2. Each robot (indexed with i) has a vector of state-action visitation frequencies $\mathbf{f}_i$ which must satisfy its own local dynamics $A_i\mathbf{f}_i = \mathbf{b}_i$. These dynamics ignore any other agents in the environment. Each robot will also have a cost $\mathbf{c}_i$ for each state-action visitation frequency $\mathbf{f}_i$. This cost ignores any interaction with other robots. Summarizing the above, if each robot ignores other robots then they solve the mixed integer program (3.1).

This mixed integer program typically will represent a single agent Markov decision process, and is thus capable of handling uncertainty in action as well as uncertainty in state. Sometimes, we may wish to force some of the state action visitation frequencies to take on integer values. For example, to represent a deterministic policy in a domain where time is included in the state we require that every state action visitation frequency f_{ij} be either 0 or 1. We represent these integer constraints more generally by stating that for some j it may be that $f_{ij} \in integer$.

$$\begin{aligned} \min_{\mathbf{f}_i} \mathbf{f}_i \mathbf{c}_\mathbf{i} \\ A_i \mathbf{f}_\mathbf{i} = \mathbf{b}_\mathbf{i} \\ \mathbf{f}_\mathbf{i} \geq \mathbf{0} \\ \text{For some i,j:} \quad f_{ij} \in integer \end{aligned} \tag{3.1}$$

In a multi-agent MDP where agents must coordinate, the robots are not independent and the model in the mixed integer program (3.1) is insufficient. The robots must interact through the consumption and production of resources. We define the production or consumption of resources by robot i by a matrix D_i. Element (j, k) of D_i is the amount of resource j which is produced or consumed by robot i in state-action pair f_{ik}. (So, $D_i \mathbf{f}_i$ is the vector of expected resource usages for robot i. The sign is arbitrary, so we will assume positive numbers correspond to consumption.)

Given a description of how much of resource j each possible state action pair f_{ik} of every robot generates or consumes, we can now write constraints on how these resources may be generated or consumed. Specifically, we will write *linear constraints* on resource production and consumption.

Using linear constraints we can write constraints of the form: $f_1(s_0, a_1) + f_2(s_3, a_4) \leq$

1. Here, we have indexed the state action visitation frequency pair, f_{ik} with the corresponding state and action $f_i(s_j, a_k)$, where i represent the robot identifier, j represents an integer state index, and k represents an integer index for the actions available to that robot in that state. Thus, the number of times robot one executes action a_1 in state s_0 plus the number of times robot two executes action a_4 in state s_3 must be less than or equal to one. In a similar manner we can write any linear constraint between state action visitation frequencies of any robot. We will call these kinds of constraints resource constraints. We can summarize all of these resource constraints with one matrix equation shown in (3.2). We will also call these resource constraints the inter-robot constraints as they directly constrain the actions of the robots with respect to one another.

$$\sum_i D_i \mathbf{f_i} = \mathbf{e} \tag{3.2}$$

The representation of (3.2) uses matrix multiplication to represent *all* of the linear inter-robot constraints in one matrix equation.

A mixed integer program to represent loosely coupled multi-agent MDP coordination problems

If we combine the inter-robot constraints of (3.2) with the individual robot dynamics and individual robot objectives of (3.1), we have the following full mixed integer program representation of the multi-agent MDP:

$$
\begin{aligned}
& \min_{\boldsymbol{f}_i} \sum_i \boldsymbol{f}_i \mathbf{c_i} \\
\forall i: \quad & A_i \mathbf{f_i} = \mathbf{b_i} \\
(*) \qquad & \sum_i D_i \mathbf{f_i} = \mathbf{e} \\
\forall i: \quad & \mathbf{f_i} \geq \mathbf{0} \\
\text{For some i,j:} \quad & f_{ij} \in integer
\end{aligned}
\tag{3.3}
$$

This mixed integer program is exactly the representation that we use to represent any multi-agent coordination problem. To obtain this mixed integer program we examine the original multi-agent problem. At first we model the state spaces of the individual robots ($A_i\mathbf{f_i} = \mathbf{b_i}$). We then note constraints between the robots as linear functions of the individual robot state action visitation frequencies. These inter-robot constraints are modeled by the constraints labeled (*) in the MIP (3.3). Lastly, we must decide which variables must be forced to be integer values using integer constraints. We then have a loosely coupled multi-agent MDP represented as the mixed integer program in (3.3). We use the term loosely coupled because the coordination linear program (3.3) has a number of constraints linear in the number of robots.

If we use the MIP model shown in (3.3), then in order to model any multi-agent coordination problem it suffices to define the following variables: c_i, A_i, $\mathbf{b_i}$, D_i, $\mathbf{e}$ and any integer constraints on the f_{ij}. To illustrate how one goes about defining these variables we present the following example.

3.3.3 A full example of a multi-robot path planner

We use the example of the multi-robot path planner throughout this chapter to explain some of the aspects of modeling loosely coupled MDPs. We will now give the full formulation of a simple multi-robot path planner for the very small domain shown in figure (3.1) and repeated here in figure (3.2).

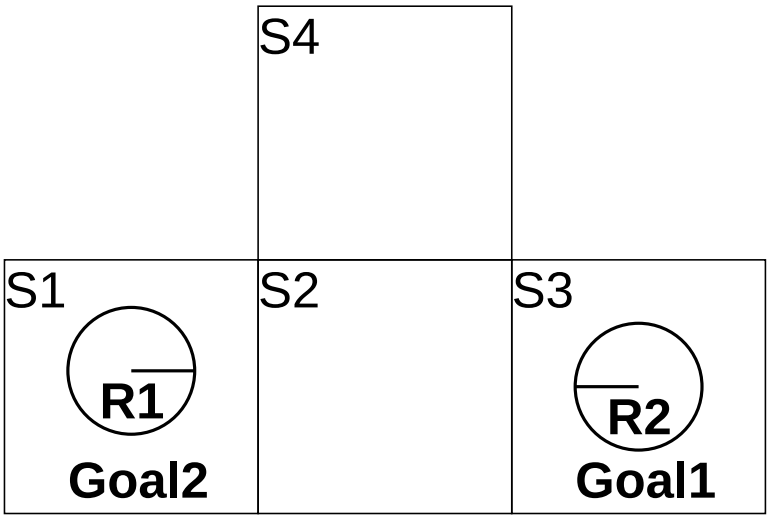


Figure 3.2: A small path planning problem.

We will define the state and action spaces for each robot individually and then determine the resources shared by the robots and how they constrain the state action visitation frequencies. To begin with, we will not include time in the state but describe this addition at the end of the example.

The state space for each robot is simply the positions: $s_i \in \{S1, S2, S3, S4\}$. The action space is the four cardinal directions, north, east, south, west, and do nothing (wait in a given state). We order the states starting in $S1$ and order the actions starting with north and going clockwise. If we reason about every action in every state, we have 20 state action visitation frequencies per robot. We abbreviate the direction of the action with the first letter of the direction (so "north" becomes 'n'

etc.) and the "do nothing" action with 'd'. Ignoring time in the state for now, we have the following full state action visitation frequency vector $\mathbf{f}_i$ for each robot $i \in \{1, 2\}$:

$$\begin{aligned}\mathbf{f}_i' = [&f_i(S1, n), f_i(S1, e), f_i(S1, s), f_i(S1, w), f_i(S1, d),\\ &f_i(S2, n), f_i(S2, e), f_i(S2, s), f_i(S2, w), f_i(S2, d),\\ &f_i(S3, n), f_i(S3, e), f_i(S3, s), f_i(S3, w), f_i(S3, d),\\ &f_i(S4, n), f_i(S4, e), f_i(S4, s), f_i(S4, w), f_i(S4, d)]\end{aligned}$$

The individual robot dynamics (A_i and $\mathbf{b_i}$) simply consist of moving the robot to the correct new position for a given motion direction and current position. Attempting to move off the boundary leaves the robot in the same position. There is one flow constraint per state leading to four flow constraints (one for each position). Each flow constraint requires one row in the A_i matrix, an action leading out of a state is represented with a $+1$ and an action entering a state results in a -1. If an action both exits and enters the same state, then it is represented with a 0 in the matrix. Each column in the A_i matrix represents a state action visitation frequency. Specifically, we could label the columns $f_i(S1, n), f_i(S1, e), \ldots, f_i(S4, d)$ corresponding to the order in the $\mathbf{f}_i$ vector. This leads to an A_i matrix of the following for each robot:

$$A_i = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Using the knowledge that the start state for robot 1 is $S1$ and the start state for $S2$ is $S3$ we have the $\mathbf{b_i}$ vectors:

$$\mathbf{b}'_1 = [1, 0, 0, 0]$$

$$\mathbf{b}'_2 = [0, 0, 0, 1]$$

Valid motions have a cost of one. We allow the robots to attempt to move outside the boundary of the world but penalize them with a cost of 100. In general, solution time is faster if less constraints are used and changes are made to the cost function, and that is the practice we have followed here. If the robot is waiting in or moving to the goal state, the cost is zero. This leads to the following cost vectors $\mathbf{c_i}$ for each robot i, where each value indexed by k is the cost for a particular state action visitation frequency $\mathbf{f}_i$ for that robot.

$$\mathbf{c}'_1 = [100, 1, 100, 100, 1, 0, 1, 100, 1, 1$$
$$100, 100, 100, 1, 0, 100, 100, 1, 100, 1]$$

$$\mathbf{c}'_2 = [100, 1, 100, 100, 0, 1, 1, 100, 0, 1$$
$$100, 100, 100, 1, 1, 100, 100, 1, 100, 1]$$

Now we require the inter-robot or resource constraints (D_i and $\mathbf{e}$). We need to ensure that the two robots are never in the same cell, and that they do not attempt to pass through each other on the boundaries between cells. Thus the resource constraint states that only one robot has the right the use the resource (a grid cell) at a time). To stop the robots from ending up in the same cell we ensure that the sum of any actions leading into a given cell is less than or equal to one. For example for position $S1$:

$$f_1(S1, n) + f_1(S1, s) + f_1(S1, w) + f_1(S1, d) + f_1(S2, w) +$$
$$f_2(S1, n) + f_2(S1, s) + f_2(S1, w) + f_2(S1, d) + f_2(S2, w) \leq 1$$

This is repeated for each position leading to four constraints. To stop the robots from crossing over each other we require two constraints for each boundary leading to six cross-over constraints. For the boundary between $S1$ and $S2$ we have:

$$f_1(S1, e) + f_2(S2, w) \leq 1$$
$$f_1(S2, w) + f_2(S1, e) \leq 1$$

Combining all of the above inter robot constraints we obtain the D_i matrices and

the **e** vector.

We now would like to represent a deterministic policy for each robot. That is, each robot must choose any action with 100% probability. This can be enforced by making every state action visitation frequency be either 0 or 1 which are the integer constraints on the f_{ik} variables. The reason we would like a deterministic policy will be made clear in section 3.4.3.

Unfortunately, for this problem, it is necessary to include time in the state since the robots must be able to share the grid cell $S2$ at different time steps (one robot cannot own that resource for all time). This means that both state action visitation frequencies and constraints must be replicated for every time step. Instead of having a state action variable for position $S1$ and action n: $f(S1, n)$, we now have a state action variable for each time step $t_1, t_2, \ldots, t_k$. The state action variable $f(S1, n)$ becomes: $f(S1, t_1, n)$, $f(S1, t_2, n)$,..., $f(S1, t_k, n)$. Similarly we must replicate all state action variables, and rewrite the constraints to include time. $f_1(S1, e) + f_2(S2, w) \leq 1$ would become: $\forall i \quad f_1(S1, t_i, e) + f_2(S2, t_i, w) \leq 1$.

If we have T time steps, then we have $20 \times T$ state action visitation frequencies per robot. Similarly we have $(4 + 6 + 4) \times T$ constraints. If we use $T = 6$ then we arrive at a linear program with 240 state action visitation frequencies and 84 constraints.

Now that we have a fully described integer program model of this multi-agent coordination problem, we require one or more methods to solve the mixed integer program.

3.3.4 Solving the multi-agent mixed integer program: The centralized algorithm

There are several solution methods for mixed integer programs of the form shown in (3.3). Choosing any of these methods leads us to a centralized algorithm for multi-

Centralized algorithm for solving multi-agent planning problems

- Determine the resources which must be shared (produced or consumed) by the robots.
- Approximate the true resource constraints using linear constraints of the form in (3.2).
- Write the individual agent MDPs as dual linear programs such as (3.1).
- Combine the resource constraints and single agent problems into the larger integer program shown in (3.3).
- Solve this integer program with any off-the-shelf integer program solver.

Figure 3.3: The centralized algorithm for solving loosely coupled MDPs as a mixed integer program

agent coordination. The core of the algorithm is phrasing the problem as the mixed integer program of (3.3) and then solving it with any mixed integer program solver. This centralized approach leads to our first multi-agent coordination method shown in figure 3.3:

Solving the integer program (3.3) as part of the algorithm in figure 3.3 yields a set of state-action visitation frequencies ($\mathbf{f}_i$) for each robot i. Typically, we use the branch and bound method to solve the MIP for reasons which will be made clear when we describe how Dantzig-Wolfe decomposition is used to speed the solution of the mixed integer program.

Using these frequencies, we can obtain the value function for each robot. This can be used to obtain a joint policy for the robots in the original multi-agent coordination problem.

If our model exactly describes the original multi-agent planning problem, these frequencies induce an optimal value function over the joint state of the multi-agent MDP. Solving the loosely coupled MDP represented by the integer program (3.3) thus tells us the value of being in any given joint state. If the linear program is an exact

representation of the multi-agent MDP, then we are guaranteed to have the optimal value function as the dual violation defined in chapter 2 will be zero for every state.

If our model does not exactly describe the original problem, then the quality of the value function obtained will depend on how closely we approximated the original problem. We discuss this further in section 3.4.4.

In summary, this method solves an approximation of a multi-agent coordination problem. We approximate the original problem by writing a model of the problem defined by the MIP (3.3). The solution of the MIP yields a joint policy for each of the robots. The quality of the joint policy depends on how accurately our linear constraints on resources (the constraints labeled $(*)$ in (3.3)) model the inter-robot constraints in the original multi-agent MDP we are attempting to solve. Understanding the quality of our loosely coupled MDP model requires examining the capabilities of modeling a general multi-agent MDP with a mixed integer program. We examine the modeling capabilities by examining the kinds of problems that cannot be well modeled as well as the kinds of problems that may be solve.

3.3.5 The capabilities of the multi-agent MIP model

Let us examine the implications of phrasing a multi-agent MDP as the mixed integer program in (3.3) and what kinds of multi-agent MDPs we may accurately model with a MIP. Our model has two main assumptions which affect the quality of our approximation to the original problem: First, we only allow a linear sum of the agent costs to be represented in the objective function ($\sum_i \boldsymbol{f}_i \mathbf{c_i}$). Secondly, we only allow linear constraints on the state action visitation frequencies between robots. There are both advantages and limitations of this approach if we wish to represent a general multi-agent MDP.

These two assumptions along with the selection of integer programming as our

modeling method lead to some important limitations and advantages of this approach. We describe them in detail below.

Limitations of the mixed integer program model

Computation Time: First and foremost, the computation time of using integer programs can be expensive. There are no known polynomial time algorithms for general integer programs. This said, extremely large (millions of constraints and variables) integer programs are solved regularly by the operations research community, and thus integer programming should not be abandoned outright. A simpler modeling language (based on an LP) and a distributed solution algorithm for both the MIP and the LP models described in this chapter can mitigate these computation costs.

Representing non-linear interactions: It is possible that the robots in the original problem interact in non-linear ways. If this is the case, then these non-linear interactions would need to be approximated with linear constraints in the MIP in (3.3).

For some non-linear functions, this can be done efficiently. Reasonable approximations to many quadratic functions and in general many convex functions can be made with relatively few linear constraints. Other non-linear functions may be very difficult to model with linear constraints.

An example of non-linear interactions between robots might be the XOR or the multiplication of two variables. The XOR of two variables and the conjunction of two variables require a multiplication which is non-linear but may also be represented with linear constraints. However, it may require many linear constraints for a reasonable approximation. This will cause the size of the MIP to

grow and we may not be able to solve it in a reasonable amount of time.

Non-linear reward functions: Quadratic reward functions may be solved using quadratic programming which is a polynomial time method. Other non-linear reward functions would need to be approximated.

Tightly coupled coordination: In our path planning example, we need only represent a number of interaction constraints which is linear in the size of the grid world. If the robots truly need to interact in every joint state and every joint action, then the number of linear constraints required to represent these interactions would be exponential. This tight coordination would force us to use the standard multi-agent MDP representation as discussed in section 3.2. The exponential growth in joint state and action space could not be avoided by our MIP model.

Stochastic policies: If there is an optimal joint policy which uses deterministic policies for all agents, then the representation of (3.3) is sufficient. Unfortunately, some problems have only stochastic policies which are optimal, and coordination becomes more difficult. This is described in more detail in section 3.4.3.

In summary, the main disadvantage of this approach is that integer programming can be quite slow. The main advantage is that integer programming is a very flexible modeling language and we can exactly model a very large class of multi-agent tasks and guarantee an optimal solution to the original multi-agent coordination problem. We address the issue of speeding up the solution of integer programs in section 3.4 with a simpler modeling language and in section 3.6 with a distributed solution algorithm to greatly increase the speed of the solution. Given that we will address these main disadvantages, let us now look at what we can already easily model.

Strengths of the mixed integer program model

Reduced problem size: The main strength of the MIP model is that we have reduced the size of the problem from being exponential in the number of agents to being linear in the number of agents. This is significant, as it now allows us to solve problems with many agents that could not have been solved with previous MDP solution techniques.

Accurate modeling of linear interactions: If the agents truly interact in linear ways, then we have an accurate representation of the problem which is exponentially smaller than representing the true joint problem. An exact representation means that we can solve the MIP model of the problem to obtain an *optimal* joint plan for all robots.

An efficient solution: Because we have exponentially reduced the size of the original problem, we can now solve the multi-agent coordination problem efficiently using the algorithms described in this chapter.

Modeling global resource constraints: Using linear constraints, we can also create global resource constraints. We can set limits on global resource production or consumptions which cannot be exceeded by the robot team as a whole. This is useful for representing limited fuel for a team of path planning robots, or limited cash reserves for a business with multiple agents. Many multi-agent problems involving coordination require global resource constraints which are not handled by other multi-agent planning methods.

Representational capabilities of integer programs: Though we have not depicted many of these capabilities in the program presented in (3.3), there are several important capabilities that are available to us when modeling a prob-

lem with integer programming. We can represent binary choices, dependent decisions, disjunctive constraints, relations between variables, and restricting a variable to a set of constants. These capabilities are detailed in chapter 2.

Representing deterministic policies: The representation of deterministic policies requires the use of integer constraints. We can represent both stochastic and deterministic policies with the MIP shown in (3.3).

Specifically, we could include time in the state and force every state action visitation frequency f_{ij} to be zero or one using the program in (3.3), then we achieve deterministic policies which avoids issues with coordinating stochastic policies described in section 3.4.3.

In general, the closer our mixed integer program model represents the original problem, the more useful the policies generated by our method will be. Using this model we can represent most useful multi-robot coordination tasks exactly. Our mixed integer program provides a very flexible modeling language, and thus can model most multi-agent coordination problems closely, if not exactly.

Unfortunately, the cost we exchange for our flexible mixed integer program model is the length of time it takes to find a solution. Addressing this lack of speed leads us to our next multi-agent planning model.

3.4 Modeling a multi-agent MDP with a linear program

To see how we might make a model of a multi-agent problem which is faster to solve, we look now at the branch and bound algorithm. Observing the way that branch and bound goes about solving a MIP, we see that it forms relaxations of the MIP by removing all integer constraints and solving a linear program where certain variables

are forced to be constants. Linear programs (unlike MIPs) can be solved in polynomial time and so this sub-step of branch and bound is fast.

Observing this functionality of branch and bound leads one to ask: "What happens if we *only* solve the first node of the branch and bound tree?". Solving this first node corresponds to solving a single linear program in which there are no integer constraints and no variables are forced to be constants. If we only solve this one node and use the state action visitation frequencies that result, will we have a useful joint policy for the robots?

The short answer is yes. Intuitively, this relaxation is still a very close approximation of the original MIP. We still model the inter-robot constraints and the individual robot MDPs, but we do not allow integer constraints. The longer answer is that certain problems may arise.

To examine these problems let us examine this new model more formally. Solving the first node of the branch and bound tree for the mixed integer program model of (3.3) leads to the simplified model for multi-agent MDPs shown in (3.4). The only difference from (3.3) is that we do not allow integer constraints.

$$\begin{aligned} & \min_{\mathbf{f}_i} \sum_i \mathbf{f}_i \mathbf{c_i} \\ \forall i: \quad & A_i \mathbf{f_i} = \mathbf{b_i} \\ (*) \qquad & \sum_i D_i \mathbf{f_i} = \mathbf{e} \\ \forall i: \quad & \mathbf{f_i} \geq \mathbf{0} \end{aligned} \tag{3.4}$$

There are some clear advantages and disadvantages to this new model. We now describe some of the high-level limitations and advantages of this model which are

different from those listed for the MIP.

3.4.1 Limitations of the multi-agent LP in (3.3):

Integer constraints: As discussed in chapter 2, some kinds of relationships are best modeled as integer constraints. Given that this representation only contains real-valued variables and constants, integer constraints may only be approximated with a combination linear constraints.

Meeting resource constraints in expectation: By representing the state action frequencies as continuous real variables, we have made an implicit assumption. We are assuming that it is acceptable to meet the linear constraints on state-action visitation frequencies *in expectation over the entire execution time* of the MDP. This limitation has some important implications and is explained further in section 3.4.3.

Stochastic policies: It is possible that the linear program may generate valid stochastic policies which meet all coordination constraints in LP (3.4), but that the generated policies do not behave as expected. This is described in more detail in section 3.4.3.

3.4.2 Advantages of using the multi-agent LP:

The main advantage to using the LP model over our MIP model is that we can solve any linear program in polynomial time. Fortunately, we also know from chapter 2 that any MDP can be phrased as a linear program and thus a linear program still allows us to model a general multi-agent MDP which a powerful representation.

If the linear program (3.4) accurately represents our original multi-agent coordination problem, then this is a good representation which can be solved with linear

programming techniques.

We encountered some subtle problems while coordinating multi-agent tasks using the LP model. We now describe the two most prominent problems in detail.

3.4.3 An example of problems with meeting constraints in expectation

We now solve the simple problem of figure (3.2) using our LP model. We discover that the linear program comes up with an interesting but flawed solution. We include time in the state, and so by using an LP we are meeting inter-robot constraints *in expectation* on every time step. In particular, we are saying that "in expectation" no two robots will be in the same cell on the same time step. Using our example we can show how this can lead to invalid joint policies for the robot team.

We use the notation $f_i([position, time], action)$ as the state action visitation frequencies when time is included in the state.

Solving the LP yielded the following policy for robot $R1$:

$$f_1([S1,1],e) = 0.5, f_1([S1,1],d) = 0.5$$
$$f_1([S1,2],e) = 0.5, f_1([S2,2],e) = 0.5$$
$$f_1([S3,3],d) = 0.5, f_1([S2,3],e) = 0.5$$
$$f_1([S3,4],d) = 1.0$$
$$f_1([S3,5],d) = 1.0$$
$$f_1([S3,6],d) = 1.0$$

Similarly for robot $R2$ it generated:

$$f_2([S3,1],w) = 0.5, f_2([S3,1],d) = 0.5$$
$$f_2([S3,2],w) = 0.5, f_2([S2,2],w) = 0.5$$
$$f_2([S1,3],d) = 0.5, f_2([S2,3],w) = 0.5$$
$$f_2([S1,4],d) = 1.0$$
$$f_2([S1,5],d) = 1.0$$
$$f_2([S1,6],d) = 1.0$$

All state action visitation frequencies not listed above were zero. Having $f_1([S1,1],e) = 0.5, f_1([S1,1],d) = 0.5$, is equivalent to saying that robot 1 will roll a die before beginning to move. Fifty percent of the time, it will wait in the grid cell $S1$ and the other fifty percent of the time, it will move east.

We see above that no robot ever moves north or south, which would be necessary if they were to wait in the alcove. The robots did not generate the required plan of having one of the robots move into grid cell $S4$ (the alcove) to wait while the second robot passed by. In fact they move straight over each other as they head toward their goals. The robots each generated a stochastic policy which allowed them to meet the constraints. The stochastic policy is valid for each robot, but the joint policy for the robots would be invalid.

We note that these policies meet the MDP individual constraints $A_i\mathbf{f}_i = \mathbf{b}_i$ described in section 3.3.3. They are valid stochastic policies for the individual robots. Furthermore, these policies also meet the inter-robot constraints that we have set for the robots. For example:

$$f_1([S1,1],e) + f_2([S3,1],w) \leq 1.0$$

The linear program solver has not erred, in fact it has done exactly what we requested by our use of the linear constraints. If we examine the cost of the above policy, we see that the total cost is 5.0. The cost of the policy we desired (waiting in the alcove as the other robot passes by) is 7.0. Our LP has thus found a lower cost policy that met the constraints we requested. To do so, it met our inter-robot constraints *in expectation*. That is it used the capability of each robot to choose to move with fifty percent probability.

If we look at how this affects the policies above, we see that this allows the robots to *fuzz out* while passing each other. Half of robot 1 passes straight across, and the other half of robot 1 waits one time step and then goes straight across. If robot 2 has a similar strategy, then the robots can fuzz out to meet the constraint that only one robot (where each half of each robot adds to one) is in a given cell at a given time step.

A way to fix this problem is to request a deterministic policy, thus disallowing the robots from choosing to take an action with fifty percent probability. This can only be done by using integer constraints and the MIP model. With time included in the state, this corresponds to having each $f_i(s, a)$ be zero or one for every state action visitation frequency. Having a deterministic policy for each robot ensures that constraints will be met exactly on each time step. In the path planning example, we could then guarantee collision free paths for a deterministic domain.

The lesson is that we need to be careful when phrasing our coordination constraints. We must be aware of the fact that a stochastic policy might meet the linear constraints we set in expectation while not achieving the intended coordination.

In practice, we have found that this problem of generating stochastic policies to sidestep linear constraints met in expectation is relatively rare. We have seen this problem arise in our multi-robot path planner when robots must wait extended

periods of time for other robots to pass.

This is one possible problem with the LP model. We now examine a problem which applies to both the MIP model as well as the LP model.

The implications of modeling time in the state

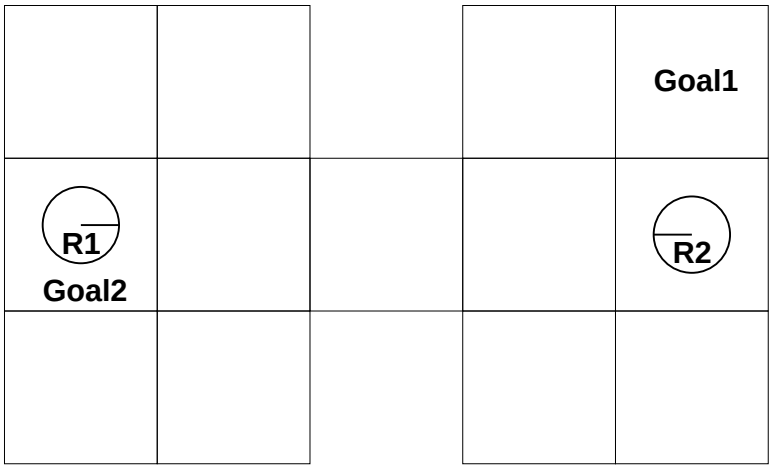


Figure 3.4: A multi-robot path planning problem with a bridge. If one robot owns the right to use the bridge for all time, then the second robot cannot achieve its goal

There are many problems where the approximation of using one resource for all time steps is insufficient. Multi-robot path planning is one such example. In the bridge example of figure 3.4, we cannot generate an optimal plan unless the robots can each own the bridge resource at a different point in time. Thus, we must add time to the state space. This does not affect the form of the LP (3.4), but increases the state space size and the number of resource constraints.

This has considerable implications. If we have a state space size of $|\mathcal{S}|$ before adding time then we have a state space size of $|\mathcal{S}| \times T_{max}$ after adding time to the state. T_{max} represents the maximum amount of time required to real a goal state or, more generally, to represent a full policy for the problem. This may make the number

of constraints in the full multi agent LP (3.4) become quite cumbersome and slow the solution of the linear program.

The solution to this problem is to resort to *constraint generation* as we have discussed in chapter 2. We solve the linear program without the constraints, and after examining the solution, we add in the constraints which were broken and re-solve. This is iterated until we have arrived at the optimal solution to the linear program with only the necessary constraints ever being represented.

The main advantage of including time in the state is that constraints will now be met in expectation on every time step as opposed to over all time. This will be a more accurate model of the original problem than meeting constraints in expectation over all time. This implies a more accurate model of the original problem and thus a more accurate joint value function for the team.

We have described the addition of time to the state to generate a more accurate model of the original problem. One question is, "What can one do with a joint policy generated from a fairly accurate (but not exact) model?" The joint policy may not be executable in the real world such as the example from section 3.4.3. The answer is that we can add an extra phase of processing to policies generated from an approximate model which guarantees that the output policies for each robot will be executable in the real world. We call this extra phase, action selection.

3.4.4 Action selection for overcoming modeling inaccuracies

Though a given model may not exactly represent the original multi-agent MDP, all is not lost. If our model is "close" to representing the original problem, then the solution to (3.3) or (3.4) will yield a value function which can likely be used to generate a very reasonable policy for the robot team (although there would be no optimality guarantee).

Because the value functions incorporate information about future actions and random events, the robots only need to look ahead a short time to choose good actions. The robots can run a simple auction to determine their best joint action at each step in the execution of the multi-agent MDP: each individual robot estimates its future cost for each action by a single-step backup from its value function. The difference between these future costs then tells the robot how much it is willing to bid for the right to execute each action. The optimal joint action is then the feasible action with the highest sum of bids.

Let us assume that we model some multi-agent coordination MDP using the mixed integer program of (3.3). If the linear loosely coupled representation is a close approximation to the original problem, then the value function obtained by solving the MIP (3.3) will be close to the true value function. Action selection methods which use the approximate value function will likely lead to good policies for the original problem.

We now present an alternative explanation of our MIP/LP models which gives some intuition as to how the MIP/LP modeling method works.

3.4.5 An alternate intuition for the linear programming representation

The linear programming representation of a loosely coupled MDP of (3.4) is in many ways related to an undirected dynamic Bayes network: each node of the network corresponds to the state and action of a single MDP, and a resource constraint involving a subset of the MDPs plays the role of a clique potential on the corresponding nodes. In this way, it is similar to the representation of (Guestrin and Gordon (2002)); but, we do not assume any particular form for the D_i matrices, while (Guestrin and Gordon (2002)) effectively assumes that they are indicator functions of particular state or action variables.

In the same (trivial) sense as Bayes nets, our representation is completely general: by collapsing all robots into a single giant agent we can represent an arbitrary MDP. More importantly, in the more-typical case that some pieces of our model can be written as resource constraints, we can achieve an exponential savings in representation size compared to the monolithic planning problem.

3.5 Examples of modeling loosely coupled MDPs

While alternative explanations are useful for providing intuition to the MIP/LP modeling of multi-agent problems, we feel that examples are typically the best way to become familiar with a new method. To this end, we now provide two more examples of modeling multi-agent coordination problems. We describe models for a multi-robot repair problem and a multi-robot patrol problem.

3.5.1 Towing repairable robots

Figure 3.5: A simple example (left panel): the objective is to have all robots (R1,R2,R3) reach the goal (G) where they receive a reward. Any action may result in a robot becoming disabled, in which case it must be towed to the repair area (Re) to continue with the task. The grid shown here is significantly smaller than of the real world maps which can be solved using Dantzig-Wolfe coordination described in section 3.6 (right panel).

Figure (3.5) shows a simulator which contains 3 robots. Each robot receives a large reward upon reaching the goal but incurs a small cost for each step it takes. Robots can break whenever they take a step, but a functioning robot may tow a

failed robot to the repair area and the repaired robot may then proceed to the goal. Each robot has the action set $\mathcal{A} = \{$ 8-connected move, pickup for towing, request tow$\}$. The state of each robot is its x position, its y position and its status {towing, going to goal, being towed, doing nothing}. If the grid is 300 by 300, then the state space size is $|S| = 300 \times 300 \times 4 = 360000$. The action space size is $|A| = 10$. The joint state space of all three robots is $|S_{joint}| = |S|^3$ and the joint action space is $|A| = 10^3$. Clearly, this problem size is such that ordinary MDP solution methods will be insufficient to determine the optimal value function.

However, this problem lends itself to resource-based decomposition because the robots only interact through towing. Specifically, we design our D_i matrices to represent the constraint that the expected number of times a robot executes a pickup action $f(s, a_{pickup})$ at a position should be equal to the expected number of times some other robot executes a request-tow action $f(s, a_{requesttow})$. For each grid cell or position p_k and indexing robots with i we have:

$$\forall k \quad \sum_i \mathbf{f}_i(p_k, a_{pickup}) - \mathbf{f}_i(p_k, a_{requesttow}) = 0$$

Using constraints of this form for each cell, we have decoupled the problem resulting in a weakly coupled MDP with robot interactions that can be modeled by linear constraints and solved as the linear program in (3.4).

3.5.2 A surveillance problem

We can also phrase the surveillance problem depicted in figure (3.6) as a loosely coupled MDP modeled as a linear program. The state of each robot is its position which could be one of seven patrol point locations as well as the starting points for each robot. For this example the size of the state space for each robot would be

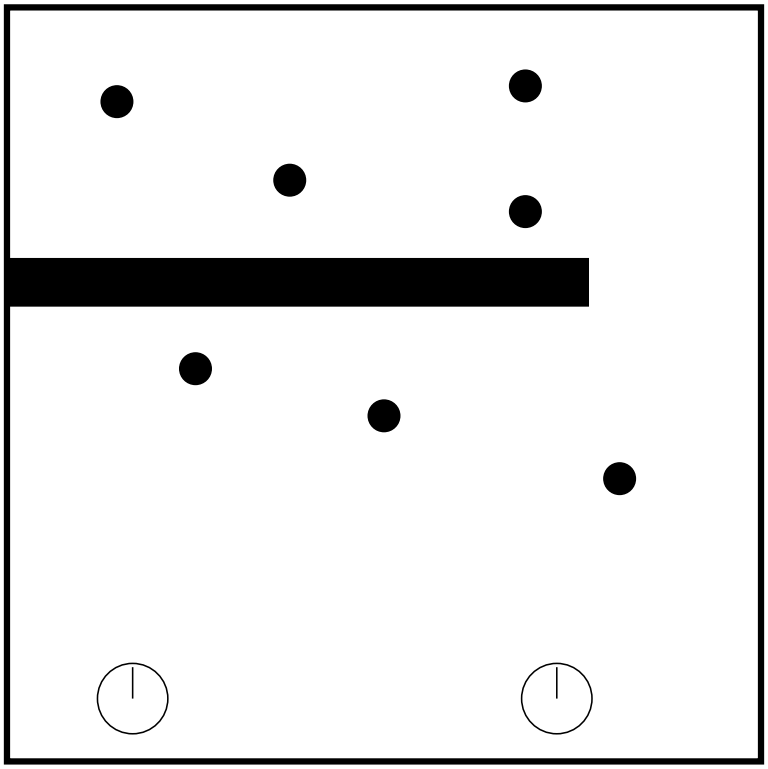


Figure 3.6: A surveillance problem in which the two robots must investigate each of the observation points labeled with black circles to complete the task.

$|\mathcal{S}| = 9$. Actions would consist of going to a given point by traveling to that point. The action space size for each robot would then be $|\mathcal{A}| = 9$. Thus, the state action visitation frequencies for each robot i would be $\mathbf{f}_i(s, a) : s \in \mathcal{S}, a \in \mathcal{A}$. This leads to each vector $\mathbf{f_i}$ being of size $9 \times 9 = 81$. The costs for each action are simply the travel distances of moving to each point. All observation points must be observed by one of the robots at least once to complete the task.

The robots only interact through the observation points. This implies the creation of a resource for each observation point and ensuring that at least one robot visits each point. For each point k we create the constraint:

$$\sum_i f_i(s_j, a_k) \geq 1$$

If each robot minimizes their travel cost while achieving the constraint above, we will have a model for this problem which can be solved using the linear program model (3.4). One could also model different amount of importance for each observation point by giving a different reward for observing each patrol point.

3.6 Dantzig-Wolfe decomposition

We have now examined two modeling languages for multi-agent coordination problems. The first model uses a mixed integer program, and the second uses a linear program. The advantages and limitation of each model were presented along with examples to better understand the models. We will now describe how the MIP and LP programs can be solved very efficiently using Dantzig-Wolfe decomposition. We will begin by describing how we can solve the LP in (3.4) efficiently, and then describe the extension to solving the MIP in (3.3).

One possible planning algorithm for solving the LP in (3.4) is to pass the LP to an off-the-shelf linear-program solver. This is the same approach we used for our mixed integer program model as shown by the algorithm in figure (3.3). This planning algorithm can be fairly efficient, but it is completely centralized: each agent must communicate its entire dynamics to a central location and wait to receive its value function in return.

Instead of using this centralized algorithm, we want to produce the same outcome using a decentralized planner which distributes as much computation as possible to the individual agents. To do so, we will apply Dantzig-Wolfe decomposition described in (Dantzig, 1963, chapter 24), (Bertsimas and Tsitsiklis (1997)) and in chapter 2. This decomposition splits our original LP (3.3) into a master LP shown in (3.5) and one slave LP shown in (3.6) for each robot i. It then solves each slave program repeatedly, generating a new value for each vector $\mathbf{f}_i$ each time.

These $\mathbf{f}_i$ vectors generated by the slave LP are the corners of the polytope P_i described in chapter 2. The master program finds a convex combination of these corners by inserting them into the master LP and solving it. Solving the master LP modifies prices on shared resources at which point the master program asks for a new state action visitation frequency vector $\mathbf{f}_i$ from each slave. It repeats this process until it has met all the inter-robot constraints while achieving minimal cost for the team as a whole.

More formally, the master and slave LPs are shown in (3.5) and (3.6).

$$
\begin{aligned}
&\min_{\boldsymbol{\lambda}_i} \sum_i \mathbf{c}_i^{\mathrm{T}} F_i \boldsymbol{\lambda}_i \\
(*) \quad &\sum_i D_i \left(F_i \boldsymbol{\lambda}_i\right) = \mathbf{e} \\
&\forall i: \ \boldsymbol{\lambda}_i \geq 0 \\
&\forall i: \ \sum_j \lambda_{ij} = 1
\end{aligned}
\tag{3.5}
$$

$$
\begin{aligned}
&\min_{\mathbf{f}_i} \left(\mathbf{c}_i^{\mathrm{T}} - \mathbf{p}^{\mathrm{T}} D_i\right) \mathbf{f}_i \\
&A_i \mathbf{f}_i = \mathbf{b}_i \\
&\mathbf{f}_i \geq \mathbf{0}
\end{aligned}
\tag{3.6}
$$

The master LP is the same as the original LP shown in (3.4) except that $\mathbf{f}_i$ has been replaced by $F_i \boldsymbol{\lambda}_i$ and individual robot dynamics and cost functions have been moved to the slave LP. Each column of F_i is one of the solutions (corners of the polytope) $\mathbf{f}_i$ which we have computed for the ith slave LP. (For efficiency, instead of storing F_i we keep $G_i = D_i F_i$ and $\Phi_i = \mathbf{c}_i^{\mathrm{T}} F_i$. This speeds up communication.) Solving the master LP means finding a convex combination $\boldsymbol{\lambda}_i$ of the known solutions for each slave LP.

The slave LP of (3.6) is the same as a single-robot planning problem (3.1) except that its costs have been altered by subtracting $\mathbf{p}^{\mathrm{T}} D_i$. The vector $\mathbf{p}$ are the dual variables for the constraints $(*)$ from the last time we solved the master LP. The entire Dantzig-Wolfe algorithm is shown in figure 3.7.

The Dantzig-Wolfe solution will always be identical to the solution obtained by solving the LP in (3.4) using a centralized method. The Dantzig-Wolfe decomposition algorithm is guaranteed to terminate in a finite number of steps with the correct solution to our original LP and therefore with the correct local value functions for each slave/robot. Though there is no proof on the number of iterations required for the Dantzig-Wolfe decomposition to converge, our experiments show that the number of iterations is typically quite small. We thus compare it to the simplex

$\mathbf{p} \leftarrow 0 \quad G_i = [\,] \quad \Phi_i = [\,]$
repeat
  done $\leftarrow$ true
  for $i \leftarrow 1 \ldots n$
    send prices $\mathbf{p}$ to robot i
    $\mathbf{f} \leftarrow$ frequencies from planning for robot i with costs $\mathbf{c}_i - D_i^{\mathrm{T}}\mathbf{p}$
    send expected usage $\mathbf{g} = D_i\mathbf{f}$ and cost $\phi = \mathbf{c}_i \cdot \mathbf{f}$ to master
    if $\mathbf{g}$ is not already in G_i
      $G_i \leftarrow [G_i, \mathbf{g}] \quad \Phi_i \leftarrow [\Phi_i, \phi] \quad$ done $\leftarrow$ false
    end if
  end for
  $\mathbf{p} \leftarrow$ new dual variables from solving (3.5) with current G_i and Φ_i
until done

Figure 3.7: The Dantzig-Wolfe decomposition applied to the full linear program of (3.3).

algorithm: though it is possible to generate an example for which the algorithm may take an exponential number of iterations to converge, in practical problems it converges quickly. The computational efficiency of Dantzig-Wolfe decomposition was discussed in chapter 2.

Note that each slave LP is the same as the corresponding individual robot's MDP except that it has different state-action costs (modified by the price vector $\mathbf{p}$). This is a very important feature of the algorithm. If each robot sub-problem is only modified to have different costs then each robot can run a standard MDP planner (which is often much faster than a general LP solver) to produce their plans.

Another significant advantage of this decomposition is that robots communication requirements to the master program are minimal. Instead of sending whole MDPs and value functions back and forth, the Dantzig-Wolfe decomposition only needs to send resource prices $\mathbf{p}$ from the master to the slave and expected usages of resources (values from the vector $G_i = D_i\mathbf{f}_i$ which appear in the coordination constraint) from the slave programs back to the master. The total amount of communication is minimal.

The master program can be located on a separate agent, or on an arbitrary robot. If we desire the procedure to be more robust, we can have two master programs each receive the same information and ensure they arrive at the same answer.

We have shown how the Dantzig-Wolfe algorithm can solve a linear program model of a multi-agent coordination problem in a distributed manner. We will now show how the Dantzig-Wolfe algorithm can greatly speed up the solution of a mixed integer program model.

Solving a mixed integer program using Dantzig-Wolfe decomposition

To solve the mixed integer program of (3.3) we will combine the Dantzig-Wolfe decomposition with the branch and bound solution method for mixed integer programs.

At every node of the branch and bound tree, we will solve a linear program. Some of the state-action visitation frequency variables may be forced to be constants, but in essence, every node of the branch and bound search tree requires the solution of a linear program which looks almost identical to (3.4) with the only difference being that some variables are forced to be constants with extra linear constraints.

What this means is that we can use Dantzig-Wolfe decomposition for every node of the branch and bound tree. To further save computation, the sub-plans (sets of state action visitation frequencies $\mathbf{f_i}$ generated by the slaves) can be re-used in most every node solved during branch and bound. This leads to our decentralized algorithm for solving a multi-agent MDP shown in figure 3.8.

Though we have named this a decentralized algorithm, it is important to note that while all of the computation is decentralized, the master program does constitute a centralized component of the algorithm. It must remember all of the sub-plans passed to it by the individual agents. It is by keeping this small centralized portion that it can guarantee an optimal solution to the LP in (3.4).

Decentralized algorithm for solving a multi-agent MDP

- Determine the resources which must be shared (produced or consumed) by the robots.
- Approximate the true resource constraints using linear constraints of the form in (3.2).
- Write the individual agent MDPs as dual linear programs such as (3.1).
- Combine the resource constraints and single agent problems into the larger integer program shown in (3.3).
- Solve this integer program with branch and bound. Every node of the branch and bound tree is solved with Dantzig-Wolfe decomposition.

Figure 3.8: The decentralized algorithm for solving loosely coupled MDPs as a mixed integer program

3.6.1 An economic interpretation of Dantzig-Wolfe decomposition

We have described how to use the Dantzig-Wolfe decomposition to derive an efficient distributed planning algorithm for loosely-coupled MDPs represented as the LP in (3.4) and MIP in (3.3). In addition to being efficient and distributed, our algorithm has an intuitive economic interpretation which leads to interesting links with existing work on market architectures.

It is well known that the dual variables of a linear program have economic significance (Rardin (1998); Chvatal (1983)). As we have discussed in chapter 2, associated with each row of the constraint matrices D_i in the master program (3.5) is a dual variable; that is, there is one dual variable p_j for each resource j. We can interpret this dual variable as a price for resource j. To see why, notice that the slave program charges robot i a cost of $p_j[D_i]_{j,k}$ each time it visits state-action pair k, and that visiting state-action pair k consumes an amount $[D_i]_{j,k}$ of resource j.

The Dantzig-Wolfe algorithm can be interpreted as a search for optimal resource prices. The master agent repeatedly asks the robots what they would do if the prices

were $\mathbf{p}$, then tries to combine their answers to produce a good plan for all the robots together. As it combines the single-robot plans, it notices whether it could achieve a higher reward by increasing or decreasing the supply of each resource; if there is an under-supply of a resource the master agent assigns it a high price, and if there is an oversupply the master agent assigns it a low price.

We now describe the algorithm in terms of how it is applied specifically to the multi-robot path planning example.

3.6.2 The decentralized algorithm applied to multi-robot path planning

Revisiting the multi-robot path planner in section 3.3.3, we now describe how the decentralized method in figure (3.8) is applied to a general multi-robot path planning problem. For our multi-robot path planner, we chose to only model the coordination problem with our linear program model of (3.4).

We have already defined the matrices and vectors $A_i, \mathbf{b}_i, D_i, e, \mathbf{c}_i$ for a small example in section 3.3.3. In general, we will use the same kinds of constraints for our general multi-robot path planner. Specifically, we say that no more than one robot may be in a 3x3 tile at any given time step. This allows the robots to have a width corresponding to 30cm, and still not run into each other.

We need not use a linear program to solve the individual robot path planning sub-problems described by the slave LP (3.6). In fact we can simply use an A* path planner for each robot thus solving the robot sub-problems very efficiently. We can use A* in this case because the subproblems for each robot are deterministic.

At a high-level, the master program will determine a set of prices on each grid cell in the world for each time step. These prices are set when the master linear program (3.5) is solved. The prices are the dual variables of the coordination constraints. The master program observes the robots path plans sent to it in response to a particular

set of prices. The master program also notes where robots are running into each other. It modifies the prices until robots no longer run into each other but still have low cost paths.

We also use constraint generation: at first we do not explicitly represent any coordination constraints in the master program. Only when the slave programs generate path plans with possible conflicts do we add the constraints to the master program.

In more detail the Dantzig-Wolfe decomposition solves the multi-robot path planning problem as follows:

- Initialize the prices on the grid cells at every time step to be 0. $\mathbf{p} \leftarrow \mathbf{0}$
- Initialize the set of coordination constraints (the rows of D_i and e) to be empty.
- Repeat
  - Send the vector of prices to the robots.
  - Plan a path for each robot in the world from its start state to its goal state with a cost vector modified by the prices $\mathbf{p}$ using A*. This creates a path plan for each robot which is represented as a vector of state action visitation frequencies $\mathbf{f}_i$ for each robot. If the path plan for robot i is new (not seen previously), then this plan vector is added to the matrix F_i as another column. Expected resource usage g_i and total cost ϕ_i are calculated for this plan. The expected resource usage for path planning is just noting which cells are occupied on which time steps.
  - Examine the paths generated for this vector of prices $\mathbf{p}$. In any grid cell where paths of different robots cross, create the coordination constraint to prevent collisions from occurring. Creating these constraints entails adding a row to each D_i and adding a value to the vector $\mathbf{e}$.

  - If we have seen all path plans previously, then terminate. Otherwise add their expected resource usage and plan costs to the matrices G_i and Φ_i.
  - Solve the master program to determine a new set of prices $\mathbf{p}$

- When no new paths are generated, we examine the values of the λ_{ij} to determine the joint policy for all robots.

Because the prices $\mathbf{p}$ induce a price on each grid cell at each time step, we can simply allow each robot to plan a path in time/position space using A*. There is one price per state action visitation frequency. The master program need only send prices p_i which are non zero. Because non-zero prices occur only where there is a possible conflict, communication is only required for parts of the joint state space where robot paths may conflict. That is, this method only needs to communicate information to resolve conflicts.

When each robot plans a path, it is planning in a state space which includes time. Though this could mean that the state space is huge, there is an efficient technique for planning in such state spaces. We first generate a heuristic for A* over the entire state space which does not include time. That is, we know exactly the most efficient path to get from anywhere on the grid to the goal. When the robot path plans it uses this as a heuristic when planning in time/state space. This method allows us to have prices on any grid cell on any time step, but still have efficient path planners.

At termination we desire that for each robot i only one of the λ_{ij} will be one and the rest will be zero. If that is not the case, then the Dantzig-Wolfe decomposition has created a stochastic policy for the robots which might result in a collision if executed. If this occurs, we must resort to mixed integer programming to enforce deterministic policies.

The above is an efficient multi-robot path planner shown to be effective by our experimental results. These results are shown below.

3.7 Experimental results

3.7.1 Multi-Robot path planner

We have implemented the multi-robot path planner using our LP model described in section 3.6.2. We found it to be quite efficient for a number of problems and works well up to ten robots in a 235 x 280 grid generated from a real environment 23.5 x 28.0 meters in size. These results are shown in figure 3.9.

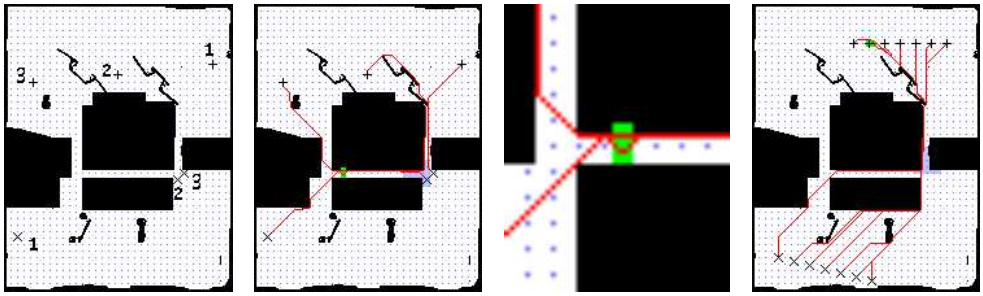
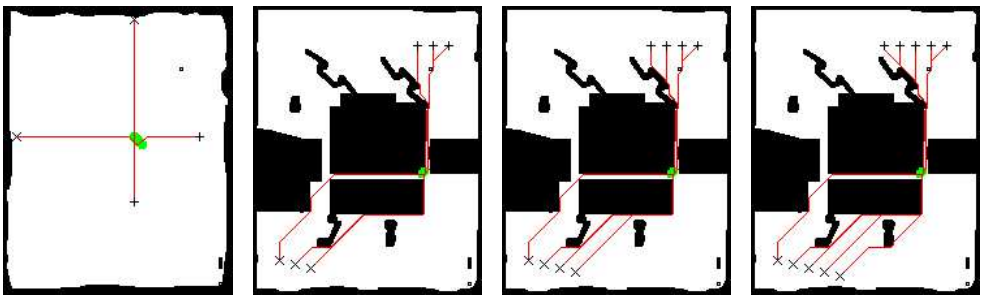


Figure 3.9: Some sample path plans performed using our algorithm. Figure (a) displays the initial map created from a 30m by 20m real environment modified by the addition of rectangles where an 'x' indicates a start for a robot, and a '+' indicates a robot goal and the number indicates which robot the start or goal belongs to. Figure (b) shows a joint path. Figure (c) shows an expanded portion of figure (b) where robots successfully navigated around each other. The last figure shows the largest problem which was solved.

The multi-robot path planning algorithm was also used to coordinate real robots in a game of multi-robot paint ball described in chapter 4.



a b c d

Figure 3.10: Some sample paths generated from our planner for which we calculated the wall clock time of the algorithm.

3.7.2 Timing Results

To give the reader some idea of our multi-robot path planner run times with certain problems, we provide the examples shown in figure 3.10. The run time results are shown in table 3.1. We first define the meaning of the various headers in the table and then highlight some features of the algorithm as they relate to these problems.

Timing results table description

Column and row headers: The column headers correspond to the path planning problems shown in figure 3.10. The row headers indicate the size of the LP model which was solved using Dantzig-Wolfe as well as how long it took each component of the algorithm to solve the problem. All numbers in the table are averaged over 30 runs for each problem. Mainly, the wall clock times varied between runs, which is why the times are averaged over 30 runs.

num lp rows, num lp cols: The size of the master LP or MIP solved at every iteration will typically grow as more sub-plans are returned by the robots. For this reason, table 3.1 shows the average number of rows as well as the average number of columns (`num lp rows` and `num lp cols` respectively) in the constraint

matrix solved over the entire planning procedure.

For example, on the first iteration it might be a 6x6 matrix and then a 40x100 matrix the second iteration and 50x150 matrix the third iteration. The number reported for `num lp rows` would be (6+40+50)/3, and for `num lp cols` would be (6+ 100 + 150)/3, the num of lp solves and iters would be 3. `num lp solves` multiplied by the number of rows or columns is an integer though this may not seem to be the case as we have rounded some of the numbers in the table.

num lp solves/ iters: This is the number of times the Dantzig-Wolfe decomposition iterated before arriving at the global solution for this problem.

lp time: The total wall clock time spent solving the master program LPs during the Dantzig-Wolfe solution.

astar time: The total wall clock time spent solving the individual sub-problems using the A* path planner.

constraint gen time: The total time spent reasoning about which constraints should be added to the master LP as well as inserting them into the master LP.

total time: The total time spent planning for this particular problem. This may not be the sum of previous times as we included in this row time spent saving figures out to files.

Discussion of timing results

We now note several interesting features from table 3.1. Firstly, the execution time is dominated by the linear program solve time even though the linear programs being solved are fairly small. This is due primarily to the fact that we are using a homegrown

infeasible start point interior point method. Large speedups could likely be achieved if we were to use a commercial linear program solver package such as CPLEX.

Note that the seemingly simpler problem from figure 3.10(a) actually takes longer to execute than the more complicated problems (b)-(d) involving more robots. The reason for this is that when two robots try to cross paths directly, a larger number of inter-robot constraints are added thus slowing the solution time of the linear program. It is difficult to predict for a given problem how many constraints may be added. A useful intuition however, is that if the robot's paths cross more often, then it will take longer to solve the multi-robot path planning problem as many more constraints will be added.

Typically, one would assume that the planner will take longer when there are more robots. The time difference between figure (b) and (c)-(d) shows that this may not be the case. While the A* path planning time is approximately linear in the number of robots, the inter-robot constraints add variance to how long it may take to solve the master LP. The master LP solution times for problem (b) were much higher than those of problem (c) and (d). This occurred because, in this particular case, having more robots caused the prices on more cells to be higher on the first iteration, causing the robots to generate non-intersecting paths faster on the second iteration.

In summary, a faster LP solver would greatly decrease our execution time. Also, we have found that it is very difficult to predict just how large the size of the master LP will grow. Depending on the problem domain, it may be necessary to also solve the master LP itself using Dantzig-Wolfe decomposition. Essentially, this would be equivalent to having a hierarchy of robot teams coordinating. For large robot teams, this would likely be the best course of action.

	Fig. a	Fig. b	Fig. c	Fig. d
num lp rows	86.1	47.0	48.0	49.0
num lp cols	176.4	92.0	93.0	94.6
num lp solves / iters	7.0	2.0	2.0	3.0
lp time	22.83s	1.67s	0.215s	0.622s
astar time	0.22s	0.16s	0.22s	0.30s
constraint gen time	0.012s	0.002s	0.002s	0.003s
total time	23.1s	1.84s	0.44s	0.93s

Table 3.1: The timing results for the problems shown in figure 3.10.

3.7.3 Modeling multi-robot path planning with quadratic fuel cost

We also used our decentralized multi-robot planning method on a separate problem generated synthetically by randomly placing high cost circular regions inside a bounded arena to create a maze. We place 15 robots in random starting locations and ask them to plan paths to 10 random goals of different value. Each robot can choose whichever goal it wants, but must pay a random goal-specific price. The robots are coupled through a constraint on fuel usage: there is a quadratic penalty on total fuel usage which is a linear function of total path length. The robots in this problem must coordinate to minimize the fuel cost while achieving the most reward by arriving at the lowest cost goals.

We represent the quadratic penalty using an objective function which approximates the quadratic function by using multiple linear constraints. We do this using a first order Taylor expansion of the quadratic objective function about the total path length tpl_i returned at any given iteration i of the Dantzig-Wolfe decomposition. In general we add a constraint of the form:

$$\forall i \quad obj \geq (tpl_i)^2 + 2(tpl - tpl_i)$$

This corresponds to a first order Taylor series expansion $f(x) = f(a) + f'(a)(x-a)$,

where $x = tpl$ is the total path length returned on any iteration and $a = tpl_i$ is the path length returned on a specific iteration i. This demonstrates the representation of a quadratic function using only linear constraints. An interesting feature of this approach is that, at convergence, we can guarantee that the quadratic penalty is met. This is because we always add a constraint at that final iteration centered about the final solution total path length. If this constraint is met, then we know that the quadratic cost function is satisfied and we can terminate.

In this problem, our algorithm starts from an arbitrary initial guess at the value of a unit of fuel (which causes the individual robots to make poor policy decisions) and rapidly improves the estimated value by examining the individual robot plans. We averaged the performance of our algorithm on 20 random instances; the results are shown in Figure (3.11).

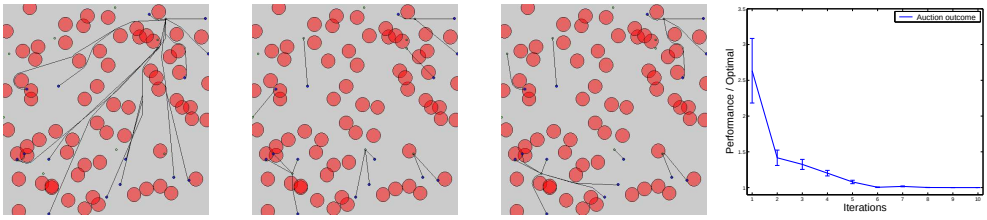


Figure 3.11: Multi-robot path planning with limited fuel usage. Left to right: in an iteration based on the assumption of cheap fuel, all robots go to the globally most tempting goal. If we assume very expensive fuel, each robot crashes through obstacles and goes to its closest goal. With the optimal fuel price, the Dantzig-Wolfe decomposition trades goal quality against distance to achieve the best possible total cost. As our algorithm learns better prices, the Dantzig-Wolfe method iterates toward the optimal policy.

The last figure shows a typical convergence graph for our algorithm. It rapidly converges towards the optimal policy requiring up to 15 iterations of the master program. Error bars are determined over multiple runs of the algorithm.

3.8 Results using mixed integer program modeling

Although we have presented first our MIP model and then our LP model, the ideas were developed in the reverse order of that presented in this chapter. As such, we have more results using the LP model than those using the MIP model. Specifically, while using the LP model for multi-robot path planning we discovered the problems inherent in the LP model as we have described in section 3.4.3.

We then developed the MIP model to address the difficulties observed in the path planning domain. At first, we were very hesitant to include integer constraints in our multi-agent coordination models. Integer programming is significantly more difficult and time consuming to perform than linear programming. As mentioned in chapter 2 there is no known polynomial time algorithm for general mixed integer programming. However, mixed integer programming is used every day in many applications and is included in popular linear and integer programming applications such as Matlab and Cplex.

In particular, we found this representation to be very appealing when we discovered that the branch and bound algorithm *can be combined* with the Dantzig-Wolfe decomposition algorithm. In fact, it is relatively common to do so in the operations research literature.

The first MIP model we have solved with this method is shown in (3.7) (copied from equation (3.3)).

$$
\begin{aligned}
& \min_{\mathbf{f}_i} \sum_i \mathbf{f}_i \mathbf{c}_\mathbf{i} \\
\forall i: \quad & A_i \mathbf{f}_\mathbf{i} = \mathbf{b}_\mathbf{i} \\
(*) \qquad & \sum_i D_i \mathbf{f}_\mathbf{i} = \mathbf{e} \\
\forall i: \quad & \mathbf{f}_\mathbf{i} \geq \mathbf{0} \\
\forall i,j: \quad & f_{ij} \in \{0,1\}
\end{aligned}
\tag{3.7}
$$

This model allowed us to model any deterministic policy for the path planning domain and led us to the more general mixed integer program model for multi-agent coordination. Unfortunately, we only have preliminary results using this model.

3.8.1 Preliminary mixed integer modeling results

We used the simple MIP model shown in (3.7) to solve the simple crossover example shown in figure 3.2. The branch and bound method was able to correctly solve this problem. A partial branch and bound tree is shown in figure 3.12. In this figure we have shown the successful branches in which we force robot 1 to first go right ($f_1(S1, t_1, e) = 1$) and then up ($f_1(S2, t_2, n) = 1$). Note that we need not enforce all integer constraints to arrive at the optimal policy, only the above two integer constraints are sufficient for the branch and bound method to find an optimal solution where robot 1 waits in the alcove as robot 2 passes underneath.

For multi-robot path planning there are also good heuristics available to bias the branch and bound search. Specifically, we can assign each of the planning robots a priority, and make all robots defer to higher priority robots when performing their path plans. This approach was advocated in (Bennewitz et al. (2001)) for a multi-

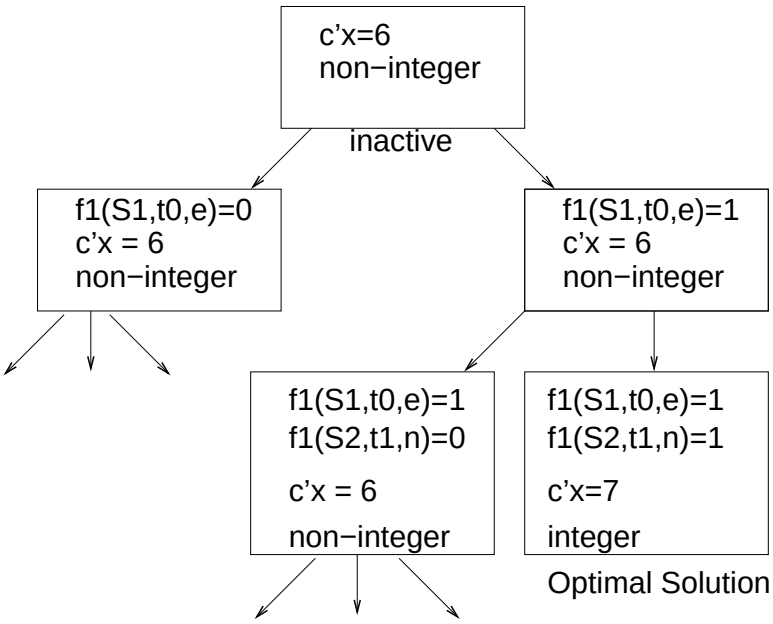


Figure 3.12: A partial tree for the branch and bound algorithm applied to the small path planning problem from figure 3.2. $c'x$ is the value of the objective function .

robot path planner. The main difference being, they would *only* use the heuristic, there was no other coordination algorithm on top of it.

Though the prioritized path planning method of (Bennewitz et al. (2001)) has been shown to work well in most cases, there are many coordination examples which it cannot solve. The simple example in section 3.3.3 is such an example. At first robot $R1$ must have priority over the grid cell $P2$ in order to move into the alcove. Then robot $R1$ must defer to robot $R2$ as $R2$ passes by to the goal. Any problem where priorities must vary with time cannot be solved by the fixed priority scheme suggested in (Bennewitz et al. (2001)). This demonstrates the need for a more flexible multi-robot coordination technique such as the one developed in this thesis.

3.9 The continuum between the MIP model and LP model

Though we have presented the MIP model and LP model as entirely separate methods by which to model a multi-agent coordination problem, LP models are in fact a subset of the MIP model and they lie on a continuum as shown in figure 3.13.

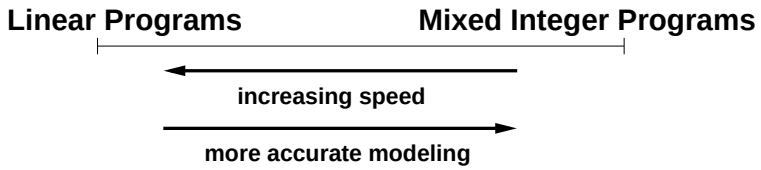


Figure 3.13: As one adds more integer constraints solution time becomes slower but model accuracy increases.

The continuum is between the slow but accurate modeling technique of the MIP model and the fast but less accurate LP model. It may be that we only model a few integer constraints of the MIP model requiring only a few nodes of branch and bound. In this manner, we could implement something such as disjunction over a few variables, even though we might not be able to enforce deterministic policies (an integer constraint on every variable) for each robot.

In general, we suggest modeling as many integer constraints as the available computation time allows. Obviously, we begin by modeling the most important integer constraints required by a given multi-agent coordination problem. One need not model all constraints, and we can still use the technique of constraint generation to generate only those constraints needed for a particular problem.

3.10 Conclusions

We have developed new, efficient techniques for solving large loosely-coupled multi-robot planning problems. In particular we have described the MIP and LP models

for multi-agent coordination as well as examples of each. For each model, there exists off the shelf centralized solution methods (branch and bound and revised simplex respectively). Lastly, we have shown how each model can be solved in a distributed and efficient manner using Dantzig-Wolfe decomposition.

Though the centralized algorithm for each model is the easiest to implement, it will be much slower than using Dantzig-Wolfe decomposition. Dantzig-Wolfe is guaranteed to arrive at the same answer as the first, centralized, algorithm for each model but does so in a much more efficient and distributed manner. As such, we highly recommend that the reader use the decomposition if there are more than a few agents in the coordination problem.

Another important factor is that the Dantzig-Wolfe decomposition can be combined with previous MDP decomposition methods and task specific solution methods (such as A*), allowing the user to mix and match which methods are best suited to their problem. The Dantzig-Wolfe decomposition also has an intuitive economic interpretation which facilitates its application to new problems. We have applied our algorithm to optimal use of fuel in a multi-robot path planning problem, and path planning for multi-robot paintball. We have also described models for other multi-agent coordination problems such as multi-robot repair and the multi-robot patrol problem. Combining branch and bound search with Dantzig-Wolfe decomposition for a MIP model gives us an efficient algorithm which can tackle a much broader array of coordination problems than the previous state of the art.

In summary, we have shown a flexible modeling language for multi-robot coordination problems as well as efficient techniques for solving these models.

Bibliography

Bennewitz, M., Burgard, W., and Thrun, S. (2001). Optimizing schedules for prioritized path planning of multi-robot systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, Seoul, Korea. ICRA.

Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific.

Burgard, W., Fox, D., Moors, M., Simmons, R., and Thrun, S. (2000). Collaborative multi-robot exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA. IEEE.

Chvatal, V. (1983). *Linear Programming*. W.H. Freeman and Company.

Dantzig, G. B. (1963). *Linear Programming and Extensions*. Princeton University Press.

Gerkey, B. and Mataric, M. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*.

Gerkey, B. P. and Mataric, M. J. (2002). Sold!: Market methods for multi-robot control.

Goldberg, D. and Matarić, M. (2000). Robust behavior-based control for distributed multi-robot collection tasks. Technical Report IRIS-00-387, USC Institute for Robotics and Intelligent Systems.

Guestrin, C. and Gordon, G. (2002). Distributed planning in hierarchical factored MDPs. In Darwiche, A. and Friedman, N., editors, *Uncertainty in Artificial Intelligence (UAI)*, volume 18.

Kaelbling, L., Littman, M., and Cassandra, A. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134.

Kitano, H., editor (1998). *Proceedings of RoboCup-97: The First Robot World Cup Soccer Games and Conferences*, Berlin. Springer Verlag.

Parker, L. E. (1996). On the design of behavior-based multi-robot teams. *Journal of Advanced Robotics*, 10(6).

Rardin, R. (1998). *Optimization in Operations Research*. Prentice Hall.

Roumeliotis, S. and Bekey, G. (2000). Distributed multi-robot localization. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS 2000)*, pages 179–188, Knoxville, Tenneessee.

Salido, J., Dolan, J., Hampshire, J., and Khosla, P. (1997). A modified reactive control framework for cooperative mobile robots. In *Proceedings of the International Conference on Sensor Fusion and Decentralized Control*, pages 90–100, Pittsburgh, PA. SPIE.

Y.U., C., A.S., F., and A.B., K. (1997). Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4:1–23.

Zlot, R., Stentz, A., Dias, M., and Thayer, S. (2002). Multi-robot exploration controlled by a market economy.

Chapter 4

Multi-Robot Competition

4.1 Introduction and Motivation

Recently there has been much activity in the field of adversarial multi-robot systems: (Kitano (1998); Bowling and Veloso (2003); Buro (2003)). Many planning problems are best phrased as MDPs define over either world states or belief states, but it was not until recently that a method was developed for adversarial MDP settings. In our previous work described in chapter 3, we present a novel method for coordinating agents in large multi-agent MDP problems which overcomes the curse of dimensionality for a large class of problems.

Recently, there has also been work on adversarial cost function selection for MDPs and cost paired MDPs: (McMahan et al. (2003); McMahan and Gordon (2003)), the larger subset of which is stochastic games (Filar and Vrieze (1992)). To our knowledge, team competition (two or more teams competing against one another) has not been studied using Markov Decision Processes as a framework.

Both the coordination method developed in the last chapter and the single-agent adversarial method developed by (McMahan et al. (2003)) allow for solving new types

of problems of practical use in robotics applications. However, what we propose here is to combine these two bodies of work in a principled manner so that the user can solve an entirely new class of problems where cooperation in the face of adversity is required.

Specifically, we would like to solve the class of problems where there are multiple cooperating agents playing a game against an opposing team which also uses cooperation among its agents. We call this type of problem *team competition*. We model this problem by allowing one team to choose the cost function of the MDP for the second team. A real world example of this (shown in (McMahan et al. (2003))) lies in multi-robot path planning while trying to evade a network of surveillance sensors. The adversarial team selects the locations of the sensors on a known map, and the path planning team must plan the lowest cost path to reach a set of goals while evading the sensor network as best it can.

Adversarial cost function selection can be applied to a variety of application from military planning, computer games, robotics games such as robot soccer to domains such as economics. In this chapter, we develop an efficient algorithm which solves problems of the form described above. Furthermore, the algorithm is easy to implement; requiring only a linear program solver and the knowledge of how to solve matrix games using linear programs.

We begin by describing adversarial cost function selection in the single agent case. We then describe how we can treat the teams as two large opposing single agents and use the methods described in chapter 3 to coordinate the individual agents in each team. Lastly, we apply our new team competition algorithm to a particular variant of physical robots playing multi-robot paintball and show some results from this domain.

4.2 Background

4.2.1 Planning for an MDP with an adversary choosing the cost function

Assume that we are given a single agent MDP with known dynamics. We create an adversarial game where player one will attempt to solve this MDP in order to reduce its cost. Assume also that there exists an adversarial agent (player two) who chooses a cost vector $\mathbf{c}_i$ from the set $K = \{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$ in order to maximize the cost to player one's MDP. The objective of player one is to minimize its plan cost, and player two must maximize the plan cost.

It was shown in (McMahan et al. (2003)) that this problem can be phrased as a zero sum matrix game as follows: Let player one have a set of pure strategies π_i which correspond to stationary deterministic policies Π_D in the MDP. Choosing a given policy π_i determines a set of state action visitation frequencies $f(\pi_i)$ which result from the dual formulation of the MDP as mentioned in chapter 2. We then let player two choose the pure strategy c_j from the finite set K. The value of the game for player one playing π_i and player two player playing c_j is $f(\pi_i) \cdot c_j$. Thus, the value of the game is the same for both players leading to a zero-sum game.

If player one chooses a mixed strategy over stationary deterministic policies $u : \Pi_D \rightarrow [0, 1]$, this is equivalent to choosing a stochastic policy with state action visitation frequencies $f = \sum_{\pi \in \Pi_D} f(\pi)u(\pi)$. Similarly player two can choose a mixed strategy $v : K \rightarrow [0, 1]$. Player two's strategies are in the convex set V whose corners are the elements of K. We will also define a matrix S whose columns are the cost vectors $c_1, c_2, \ldots, c_k$.

$$\min_{\pi \in \Pi} \max_{c \in V} f(\pi) \cdot c = \max_{c \in V} \min_{\pi \in \Pi} f(\pi) \cdot c = f(\pi^*) \cdot c^* \quad (4.1)$$

$$\begin{aligned} &\min_{\mathbf{f}, z} \; z \\ &A\mathbf{f} = \mathbf{b} \\ &\mathbf{1} \cdot z \geq S^T \mathbf{f} \\ &\mathbf{f} \geq 0 \end{aligned} \quad (4.2)$$

To find the minimax equilibrium of this two player zero sum game, we will need to solve the problem shown in (4.1). We would like to find the minimax optimal solution represented by π^* and c^*. $f(\pi^*)$ must meet the MDP constraints $A\mathbf{f} = \mathbf{b}$.

We can achieve the maximum in (4.1) by adding a set of constraints $\mathbf{1} \cdot z \geq S^T \mathbf{f}$ where z is the value of the game $\mathbf{f} \cdot \mathbf{c_i}$.

We can find the solution to (4.1) by extending the single agent dual linear program with another variable z and constraints on that variable. z is the value of the game and it is forced to be greater than the value of the game for every possible cost function $\mathbf{c_i}$. The solution of the LP in (4.2) yields the optimal π^* and c^* which is the minimax solution to the game.

As shown in (McMahan et al. (2003)), solving the new LP shown in (4.2) is equivalent to performing the double oracle algorithm shown in figure (4.1).

To describe the double oracle algorithm, we first introduce some notation. Let S be a matrix whose columns are $c_1, \ldots, c_k$. Furthermore, let us define a matrix M to be a payoff matrix defining our two player zero sum game. An entry $M(r, c)$ is the cost to the row player (player one) when row r (a set of state action visitation frequencies $\mathbf{f}$) is played while player two (the column player) plays a strategy $\mathbf{c}$ (a particular cost function). Let the value of the game be V_G and let $V_M(u, v)$ be the value of the game under (possibly mixed) strategies u and v. Assume furthermore that we have a set

of best response oracles $\mathcal{R}$ and $\mathcal{C}$ for player one and player two respectively. Given a mixture v on the columns of M, $\mathcal{R}(v) = \mathbf{f}_r$ computes a single pure row strategy which consists of a vector of state action visitation frequencies $\mathbf{f_r}$ that is the best response to v. Similarly, given a mixture u on the rows of M, $\mathcal{C}(u) = \mathbf{c}_r$ computes the single pure column strategy $\mathbf{c}_r$ (a cost function) that is the best response to u. We will use the term row strategy interchangeably with a set of state action visitation frequencies $\mathbf{f}$. We will also you the term column strategy interchangeably with the choice of a cost function $\mathbf{c} \in K$

The double oracle algorithm works by maintaining subsets of the pure row and column strategies $\bar{R}$ and $\bar{C} \in K$. It starts with a single random row and a single column strategy, solves a reduced matrix game $\bar{M}$ which only considers the row and column strategies in $\bar{R}$ and $\bar{C}$. The solution of the reduced game yields a mixed strategy u for the row player and a mixed strategy v for the column player. The best row response and column response to each is computed and added to the sets $\bar{R}$ and $\bar{C}$. This repeats until no new strategies are generated. This leads to the *double oracle algorithm* shown in figure 4.1.

In figure (4.1), we begin with any valid strategy for player one ($\mathbf{f_0}$) and any valid strategy for player two ($\mathbf{c_0}$). The starting game matrix M is only a one by one matrix which is the value of the game when $\mathbf{f}_0$ and $\mathbf{c_0}$ are played.

Given a current game matrix of any size, the *solvegame*() function solves the game resulting in two probability distributions over policies. u is a distribution over player one policies $\mathbf{f}_i$ such that $u : \mathbf{f} \rightarrow [0 \dots 1], \quad \sum_i u_i = 1.$ and v is a distribution over player two policies (cost functions) $v : \mathbf{c}_i \rightarrow [0 \dots 1], \quad \sum_i v_i = 1.$ The *solvegame*() function can be implemented with a linear program as shown in (Arsham (2003))). The average cost function $\bar{\mathbf{c}}$ is computed to be $\bar{\mathbf{c}} \leftarrow \sum_{i=1}^{|\bar{C}|} v_i \mathbf{c}_i$, and the average state action visitations frequencies $\bar{\mathbf{f}}$ of player two are computer to be $\bar{\mathbf{f}} \leftarrow \sum_{i=1}^{|\bar{R}|} u_i \mathbf{f}_i$.

$\bar{R}^0 \leftarrow \mathbf{f}_0$ \\ $\mathbf{f}_0$ is any deterministic policy for player one
$\bar{C}^0 \leftarrow \mathbf{c}_0$ \\ $\mathbf{c}_0$ is any cost function $\in K$
$\bar{M}_{0,0} \leftarrow V(\mathbf{f}_0, \mathbf{c}_0)$ \\ $\bar{M}$ is a 1x1 matrix
repeat
  $t \leftarrow t + 1$
  $(u, v) \leftarrow$ solvegame($\bar{M}$) \\ u, v are minimax strategies for $\bar{M}$
  $\bar{\mathbf{f}} \leftarrow \sum_{i=1}^{|\bar{R}|} u_i \mathbf{f}_i$ $\bar{\mathbf{c}} \leftarrow \sum_{i=1}^{|\bar{C}|} v_i \mathbf{c}_i$
  $\mathbf{f}_r \leftarrow \mathcal{R}(\bar{\mathbf{c}})$ $\mathbf{c}_r \leftarrow \mathcal{C}(\bar{\mathbf{f}})$
  $\bar{R}^{t+1} \leftarrow \bar{R}^t \cup \{\mathbf{f}_r\}$ $\bar{C}^{t+1} \leftarrow \bar{C}^t \cup \{\mathbf{c}_r\}$
until ($\bar{R}^t == \bar{R}^{t-1}$ AND $\bar{C}^t == \bar{C}^{t-1}$)
return ($\bar{\mathbf{f}}$,$\bar{\mathbf{c}}$)

Figure 4.1: Double-Oracle Algorithm

Player one then plans a best response $\mathbf{f}_r$ to the average cost function $\bar{\mathbf{c}}$. This is done using the row oracle $\mathcal{R}$. Similarly, player two plans a best response to average state action frequencies $\bar{\mathbf{f}}$. These plans are stored in $\bar{R}^t$ and $\bar{C}^t$. Then a larger game matrix is built by calculating the value of the game for every combination of row and column strategies that were stored. This is repeated until no new strategies $\bar{\mathbf{f}}$ and $\bar{\mathbf{c}}$ are generated.

The final results of the double oracle algorithm are the following: The strategies stored in $\bar{R}^t$ and $\bar{C}^t$ as well as the distributions u and v over the stored strategies. To play optimally, players one and two each generate a strategy according to u and v respectively.

The double oracle algorithm is proven to converge and is also proven to correctly solve the game shown in (4.2). The main caveat is that if all possible state action visitation frequencies $\mathbf{f}$ and all possible cost functions $\mathbf{c}$ of the game matrix M are computed, the double oracle algorithm will be much slower than solving the entire game shown in (4.2) directly. Fortunately, for many problems, many pure strategies

will never enter the set of row strategies $\bar{R}$ nor the set $\bar{C}$. This is what makes the double oracle algorithm efficient in practice.

The two key pieces required to use the double oracle algorithm are the row oracle $\mathcal{R}$ and column oracle $\mathcal{C}$. The row oracle $\mathcal{R}$ determines a best response set of state action visitation frequencies $\mathbf{f}_r$ to a fixed cost function $\bar{\mathbf{c}}$. Similarly, the column oracle $\mathcal{C}$ determines a best response cost function $\mathbf{c}_r$ to a fixed vector of state action visitation frequencies $\bar{\mathbf{f}}$

In summary, to specify a complete zero sum game which can be played optimally by the double oracle algorithm, a user need only specify optimal best response oracles $\mathcal{R}$ and $\mathcal{C}$.

4.2.2 A possible application

To see how the double oracle algorithm might be useful, we present a small example of a single robot planning a path to a goal. There is an opponent who places a single sensor with a square sensor area one grid cell in area who wishes to "see" the path planning robot as it moves towards the goal. It can only place the sensor in one of a few locations $s_1, \ldots, s_7$. This example is shown in figure (4.2). The path planning robot $R1$ has a cost of one for every step it takes and a cost of zero in the goal state when it arrives. If $R1$ steps on the sensor placed by player two it incurs an additional cost of 100. The sensor player wishes to maximize the cost to $R1$ and the robot $R1$ wishes to minimize the cost. Using the double oracle algorithm, we can solve this as a zero sum game.

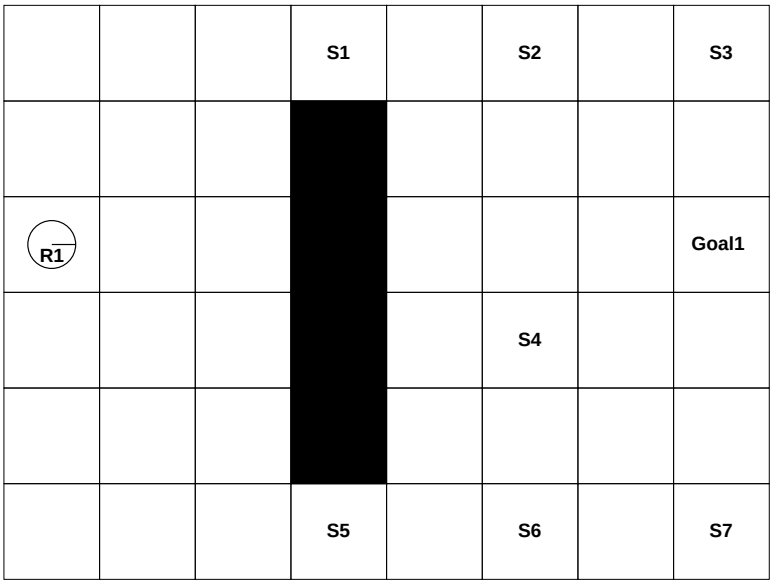


Figure 4.2: A simple game in which one player (the robot $R1$) must plan a path to the goal $Goal1$ while avoiding a sensor placed by the second player in one of seven possible locations (S1-S7)

4.3 Combining multi-agent coordination with the Double Oracle Algorithm

Imagine we wish to solve a game in which two teams compete such as the example shown in figure (4.3) which is an extension of the problem shown in figure (4.2). In this game, there are multiple agents performing coordinated path planning while the opposing team places multiple sensors. The path planning team must coordinate by not colliding with each other and minimizing total path cost. In a similar manner, the sensor placement team must coordinate to best catch the path planning team as they try to move to their goals.

We will combine our multi-agent coordination algorithm with the double oracle algorithm proposed by McMahan to solve problems of this type. It turns out that this can be done in a fairly simple manner while maintaining the convergence guarantees

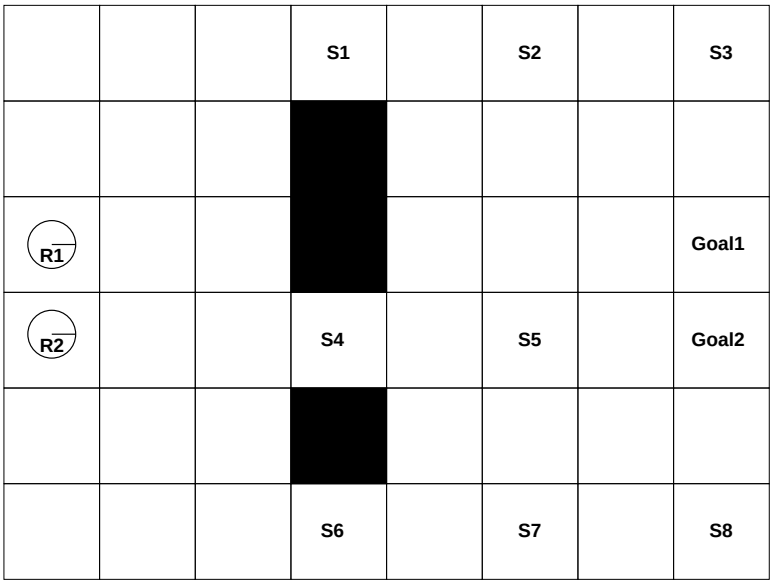


Figure 4.3: A multi-player game in which team one robots $R1$ and $R2$ must reach their goals $Goal1$ and $Goal2$. Team two may place two robot guards $R3$ and $R4$ in any of the locations (S1-S7) to attempt to catch $R1$ and $R2$ as they proceed towards the goal.

of both algorithms.

We have already discussed in section 3.2 of chapter 3 the standard method for representing a team of agents using one very large MDP. Essentially, we propose having "player one" in the double oracle algorithm be played by "team one" which is represented as a large multi-agent MDP, and "player two" in the double oracle algorithm be played by the second team as a joint agent.

Thus, as far as the double oracle algorithm is concerned, team one is planning in the joint MDP space, and team two selects a cost function for that joint MDP. However, a detail hidden from the double oracle algorithm is that each team will actually be coordinated by the coordination method presented in chapter 3.

The planner for team one is the row oracle $\mathcal{R}$ and a planner for team two is the column oracle $\mathcal{C}$ required by the double oracle algorithm. As mentioned previously, the definition of these two oracles allows the solution of the adversarial game using

the double oracle algorithm.

In more detail, we allow team two to choose the cost function $\mathbf{c}$ for team one. It does so by choosing a cost function from a finite set of cost functions $K = \{c_1, \ldots, c_k\}$ for the joint team. Team two may mix between strategies in the set K by using a strategy $\mathbf{v}$ which is a probability distribution over the set K. Let S be a matrix where the columns of the matrix are the cost vectors $c_1, c_2, \ldots, c_k$. A cost vector $\mathbf{c}$ for the cooperating team is generated as $\mathbf{c} = S\mathbf{v}$. As far as the multi-agent planner for team one is concerned, all it need know is that there is a fixed cost function.

If we factor team one using our linear programming representation developed in chapter 3 we arrive at the maximin game shown in (4.3).

$$\max_{v} \min_{\mathbf{f}_i} \sum_i (S \cdot \mathbf{v}) \cdot \mathbf{f}_i \tag{4.3}$$
$$\forall i: \quad A_i \mathbf{f}_i = \mathbf{b}_i$$
$$\sum_i D_i \mathbf{f}_i = \mathbf{e}$$
$$\forall i: \quad \mathbf{f}_i \geq \mathbf{0}$$
$$\mathbf{v} \geq 0$$
$$\mathbf{1} \cdot \mathbf{v} = 1$$

$$\min_{\mathbf{f}_i, z} z \tag{4.4}$$
$$\forall j: \quad z \geq \sum_i (S \cdot \mathbf{v}_j) \cdot \mathbf{f}_i$$
$$\forall i: \quad A_i \mathbf{f}_i = \mathbf{b}_i$$
$$\sum_i D_i \mathbf{f}_i = \mathbf{e}$$
$$\forall i: \quad \mathbf{f}_i \geq \mathbf{0}$$
$$\mathbf{v} \geq 0$$
$$\mathbf{1} \cdot \mathbf{v} = 1$$

In order to guarantee an optimal solution to the game, the double oracle algorithm requires optimal row and column oracles. Thus the factored representation must exactly represent the original multi-agent MDP. We have discussed the limitations of the linear program representation in chapter 3 as well as the more expressive integer programming representation which could be used as well if required.

As in the single agent case, the program shown in (4.3) is not a linear program,

but can be converted to a valid linear program through the use of the additional variable z which is the value of the game. This form is identical to the form shown in (4.2), where we can obtain the form of (4.2) by collapsing the multi-agent formulation into one single agent and arrive at exactly the linear program in (4.2). Thus, we can use the results from (McMahan et al. (2003)) to verify that we may use the double oracle algorithm of figure (4.1) to find the optimal solution to the game. Once again this assumes that the linear or integer program we are using represents exactly the multi-agent MDP.

In order to use the double oracle algorithm in figure (4.1), we require two things, a row oracle which generates a multi-agent plan for a fixed cost function and a column oracle which chooses a cost function given a set of fixed state action visitation frequencies. We now describe the formulation of these two oracles.

4.3.1 The multi-agent MDP planner

From the point of view of the row oracle $\mathcal{R}$ (the multi-agent MDP planner), it is solving a multi-agent MDP problem shown in (4.4) (or a mixed integer program representation) with a fixed cost function. The cost function is simply $\mathbf{c} = S\mathbf{v}$.

We have already discussed how to solve these kind of linear programs using the Dantzig-Wolfe algorithm (and solving mixed integer programs with branch and bound) in chapter 3. We will not repeat the discussion here, but refer the reader to chapter 3 for details.

Solving the multi-agent coordination problem shown in (4.4) yields a set of joint state-action visitation frequencies $\mathbf{f}$. This is the only function required of the row oracle $\mathcal{R}$ in the double oracle algorithm for team one.

In summary, from the point of view of the double oracle algorithm we have one large team agent which plans a best response to a fixed cost function. When actually

generating the joint policy for the robots however, we use our multi-agent coordination algorithms from chapter 3.

4.3.2 Choosing a cost function using a coordinated team of adversaries

We now require a method by which team two may generate a cost function $\mathbf{c}$ for team one given a set of joint state visitation frequencies $\mathbf{f}$. That is, we need to know how to make the column oracle $\mathcal{C}$ required by the double oracle algorithm.

More formally, we would like to choose some particular joint cost function $\mathbf{c}_j$ from the set of possible cost functions K. By using our multi-agent coordination algorithms from chapter 3. To maintain optimality guarantees, the cost functions $\mathbf{c}_j$ must be linear in the joint state action visitation frequencies $\mathbf{f}$ of team one.

We will phrase our team two problem (the column oracle) as a multi-agent MDP with only one state but many joint actions. Team two must perform the maximization in (4.5).

$$\begin{aligned} &\max_{\boldsymbol{c}} \boldsymbol{f}\mathbf{c} \\ &\mathbf{c} \in \mathbf{K} \end{aligned} \tag{4.5}$$

We will now allow the joint cost function $\mathbf{c}_j$ be chosen by multiple coordinating agents who are on team two. A joint cost function will be selected by agents on team two by linearly combining sub-cost functions $\mathbf{sc}$ from a set of sub-cost functions $\mathbf{sc} \in \mathbf{SK} = \{\mathbf{sc_1}, \mathbf{sc_2}, \ldots, \mathbf{sc_k}\}$. If each agent i can select a single cost function from SK, then the number of joint cost functions in K will be exponential in the number of agents i.

The agents on team two will each have a vector of binary selector variables $\mathbf{h_i}$

which allows them to choose one or more of the sub-cost vectors sc_j. Each agent i will have constraints on how the selection vector $\mathbf{h_i}$ may be chosen represented by the constraint represented by $A_i\mathbf{h}_i = b_i$. Each value of the selection vector $\mathbf{h}_i$ may be one or zero leading to a set of integer constraints.

In addition, we will allow linear interaction constraints $\sum_i D_i\mathbf{h}_i = \mathbf{e}$ between the agent selector variables $\mathbf{h}_i$ which constrain how the agents on team two may choose the sub cost functions. We can now phrase the factored multi-agent version of (4.5) as the mixed integer program of (4.6).

$$\begin{aligned} \max \boldsymbol{h_{ij}} \boldsymbol{f} & \sum_{ij} \mathbf{h}_{ij}\mathbf{sc}_j \\ & \sum_i D_i\mathbf{h}_i = \mathbf{e} \\ \forall i, \quad & A_i\mathbf{h}_i = b_i \\ & \sum_j h_{ij} = 1.0 \\ & h_{ij} \in \{0, 1\} \end{aligned} \tag{4.6}$$

Applying the magic of branch and bound combined with Dantzig-Wolfe decomposition we can solve this in exactly the same manner that we solved the multi-agent coordination problem (3.7) in chapter 3. The reader is referred to chapter 2 for a discussion of these methods.

We now have a method for the column oracle $\mathcal{C}$ of the double oracle algorithm. Using the mixed integer program of (4.6) we have a planner for team two which will choose a best response cost function $\mathbf{c}_j$ for a given set of state-action visitation frequencies $\mathbf{f}$.

4.3.3 Summary of the team competition method

We have left the double oracle algorithm essentially unchanged aside from having much larger joint team agents play against each other. These team agents are coordinated using the methods described in chapter 3. Because of this, the convergence proofs given in (McMahan et al. (2003)) still hold and the algorithm is guaranteed to converge to the optimal minimax equilibrium of the game.

The main caveat is that the convergence proofs require optimal row and column oracles. This will be true if the assumptions made by our multi-agent coordination algorithms hold as discussed in chapter 3.

The combination of the the coordination methods from the previous chapter and the adversarial method developed by McMahan, Gordon, and Blum provides the user with a powerful method for performing distributed multi-agent planning in the presence of an adversarial team which cooperates to choose the worst possible cost function for the first team.

We will now show how this algorithm may be used to represent a particular game of multi-robot paintball.

4.4 Applying the team competition double oracle algorithm to multi-robot paintball in a physical environment

We now apply the double oracle algorithm for team competition to the problem of multi-robot paintball. We begin by describing the environment, robots, and rules used to play the game. We then describe the multi-robot path planner for the first team and the sensor/defense location selector for the second team.

4.4.1 Multi-Robot paintball environment and rules

The robots

We used four robots produced by Botrics: (Atwood et al. (2004)) in our game of paintball. These differential drive robots have a top speed of two meters per second, turn in place, and have high turning speeds. Real paintball guns on tilt mounts were added to the robots. The ammunition for the robots is small nerf balls made to paintball specifications and can be found at: (Lazerball (2004)). To detect hits from the nerf balls, each robot has 8 hit detection panels around the robot. The robots are shown in figure (4.4).

The robots used Sick laser range finders as sensors and are controlled with the CARMEN robot software: (Roy et al. (2004)).

Figure 4.4: Four Obot robots developed by Botrics modified for use in paintball. The robots fire nerf balls out of paintball guns powered by compressed CO_2. The black panels detect hits from other robots.

The Environment

The robots play the game in a room called Rangos hall at Carnegie Mellon University. The room is approximately 23 meters by 28 meters in size. A robot-generated map of this room with additional obstacles is shown in figure (4.5). The map is represented as an occupation grid where one grid cell represents a 10x10cm rectangle.

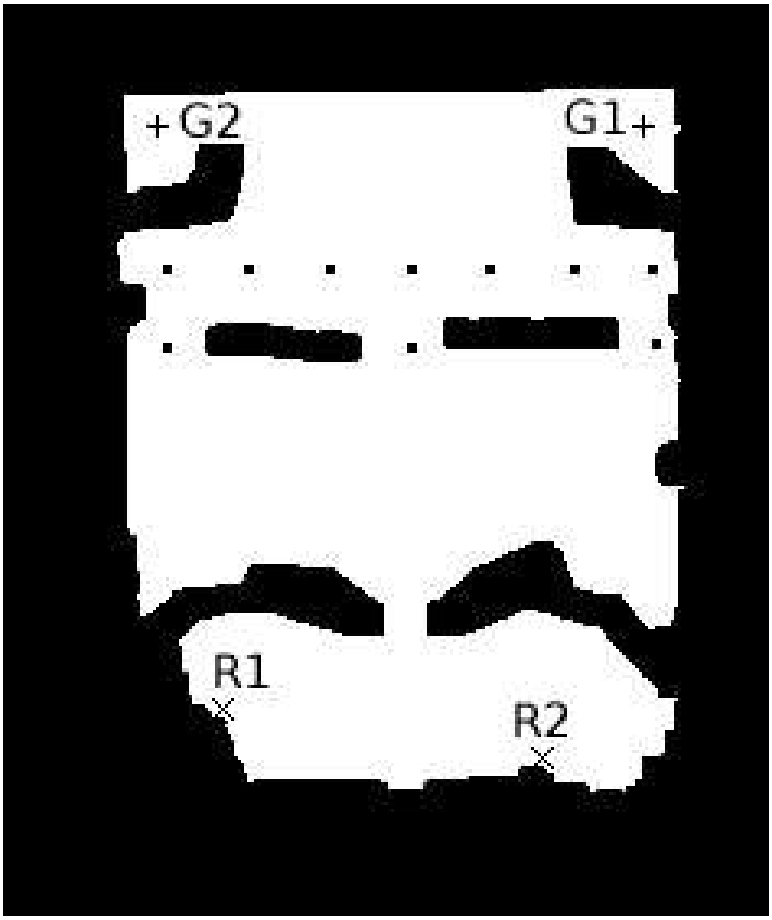


Figure 4.5: A map made of rangos hall at Carnegie Mellon University. The 'x' indicate the start positions of team one robots $R1$ and $R2$. The '+' signs indicate the goals for team one, $G1$ is the goal for $R1$ and $G2$ is the goal for $R2$. Team two may place robots $R3$ and $R4$ which acts as defense turrets. $R3$ and $R4$ may be place on any of the locations marked with a square.

The Rules

This restricted game of paintball works as follows. There are four robots, two belong to team one and two robots belong to team two. Team one consists of the red robots $R1$ and $R2$ shown in figure (4.4). Robot $R1$ must arrive at its goal $G1$ in the upper right corner and robot $R2$ must arrive at its goal $G2$ in the upper left corner. Team one and team two robots can only "see" each other if they are within two meters of each other.

The team two robots $R3$ and $R4$ must choose positions to defend in the map before the game starts. Robots $R3$ and $R4$ may only choose to defend positions marked by the squares in figure (4.5). When positioned, they act as turrets which may not move, but will shoot at the team one robots if they come within two meters.

This is a full knowledge game. That is $R3$ and $R4$ know the start and goal location of $R1$ and $R2$. $R1$ and $R2$ know the possible locations of $R3$ and $R4$. Both teams know the map.

The objective of team one is to reach their goals without being hit. The objective of the team two robots is to position themselves so that they are more likely to hit the team one robots as they try to reach their goals. Team one robots, do not shoot back, nor do they "die", they simply incur a penalty if hit.

4.4.2 Designing the planners for multi-robot paintball

To play the game described above, we require a planner for the each of the teams corresponding to the row and column players required by the double oracle algorithm. We must specify which team plans using the MDP and which team chooses the cost function for that MDP. In this case, $R1$ and $R2$ will be using a multi-robot path planner. Robots $R3$ and $R4$ will be affecting the joint cost function of $R1$ and $R2$.

Specifically, when a robot $R3$ or $R4$ is present at a particular location it increases the cost of $R1$ or $R2$ planning a path nearby. This is equivalent to the opponents placing "sensors" in the example of figure (4.3). In this case, planning through an area where $R1$ or $R2$ can be hit by a member of team two is modeled simply by a square region centered on the squares in figure (4.5). The size of the region is 20 grid cells by 20 grid cells thus covering a 2 meter by 2 meter region. Every grid cell in the square area increases the cost to robot $R1$ and $R2$ by 100.

Thus, team one consisting of $R1$ and $R2$ will be playing the row oracle $\mathcal{R}$ which is a multi-robot path planner whose objective is to minimize the total cost. Team two consisting of robots $R3$ and $R4$ is attempting to maximize the cost to team one by guarding areas in the most likely paths of $R1$ and $R2$. Choosing the areas to guard will induce a cost function $\mathbf{c}$ on robots $R1$ and $R2$.

Multi-robot path planning using Dantzig Wolfe decomposition

We have already discussed a complete formulation for a multi-robot path planner in chapter 2. There are a few small modifications required to use the planner on real robots.

Instead of constraining robots so that they do not enter a single grid cell, the constraints between the state action visitation frequencies are constrained so that no two robots may enter any given 5 by 5 tile at the same time. The size of the tile is chosen because the robots are 30cm in diameter and a 5 by 5 tile corresponding to a region 50cm by 50 cm in size. If we center a tile on each grid cell, then this generates the same number of constraints as there are cells in the grid.

The sum of the state action visitation frequencies entering any given tile must be less than or equal to one. To further reduce the number of constraints, we use the technique of constraint generation to further reduce the number of constraints.

Specifically we only model a tile constraint when robots plan paths that conflict.

The path planning team (team one) must be able to plan a path under any probability distribution v over the joint guard locations of $R3$ and $R4$. For every joint guard location i there is a probability v_i that team two will play this strategy and guard a given pair of locations. To account for any probability distribution v of joint guard locations, robots $R1$ and $R2$ can modify their path planning cost grid for by adding a cost of $v_i \times 100$ to every grid cell within two meters of the sensor location indicated by v_i. This modified cost grid is created prior to beginning the multi-robot path planning on each iteration of the double oracle algorithm.

When physically following a joint path, the robots $R1$ and $R2$ are each given a sub-goal point on the joint path. When both robots have reached their goal points (to within a certain radius) they are then passed their next subgoals from the joint path. At times, due to network lag, one robot may not receive its next goal point in a timely manner. However, this did not greatly affect the performance of the system.

In summary, given a distribution v over possible joint guard locations of $R3$ and $R4$, the path planning team $R1$ and $R2$ creates a modified cost grid and creates a multi-robot path plan using this new cost grid. For more details on the implementation of the path planner, the reader is referred to the discussion in chapter 3.

Adversarial sensor placement and the double oracle algorithm

The objective of team two is to choose where to place the guard robots $R3$ and $R4$ in response to a distribution u over joint paths. We see in figure (4.5) the possible guard locations (indicated by squares) shown by the grid of dots on the map.

Intuitively, this will mean placing a guard robot $R3$ or $R4$ in such a way as to maintain visibility over a region that the robots $R1$ and $R2$ pass through, or by placing the guard so that $R1$ and $R2$ must take a much longer path around the guard

location.

Deciding where to place both guards is equivalent to the step $y \leftarrow \mathcal{C}(\bar{x})$ in the double oracle algorithm in figure (4.1). This is a fairly simple task for a single agent. Given a mixture of joint paths $\mathbf{f}_i$ weighted by a probability u_i, the guard placement player simply sums up the state action visitation frequencies in each rectangular guard area available to it and chooses the area with the highest value.

For multiple guard robots we cannot allow two guards choosing the same location as they cannot both be physically present in the same space. To solve this problem we write a much simpler version of the general problem shown in (4.6).

$$\begin{aligned} \max \boldsymbol{h_i j f} \sum_{ij} \mathbf{h}_i j \mathbf{sc}_j \\ \sum_i D_i \mathbf{h}_i = \mathbf{e} \\ \sum_j h_{ij} = 1.0 \\ h_{ij} \in \{0, 1\} \end{aligned} \tag{4.7}$$

Guard robots $R3$ and $R4$ can be placed on *any* of the square locations marked in figure (4.5) locations so the individual constraints ($A_i \mathbf{h}_i = \mathbf{b}_i$) for each guard robot may be deleted. The constraints $D_i \mathbf{h}_i = \mathbf{e}$ simply state that guards cannot choose the same location. The state-action visitation frequency $\mathbf{f}$ in the mixed integer program (4.7) is obtained by multiplying each of the deterministic plans $\mathbf{f}_i$ generated by the player one team by its probability of playing that plan u_i. Thus $\mathbf{f} = \sum_i \mathbf{f}_i u_i$. Solving the problem in (4.7) then yields a set of selection variables $\mathbf{h}_i$ for robots $R3$ and $R4$. These induce a particular cost function c for the path planning robots $R1$ and $R2$.

This constitutes our column oracle $\mathcal{C}$ required by the double oracle algorithm.

4.4.3 The system operation

We have described the planners for the row and column players of the double oracle algorithm. Running the double oracle algorithm for our two robot paintball game leads to the following sequence of events for our game of paintball:

- Team one plans a joint path
- Team two choose the best place to put two guard locations in response to that path
- repeat
  - Calculate the value of the game for every combination of stored joint paths and guard locations, store this in a game matrix
  - Solve the game to get a probability distribution over paths and guard locations
  - Find the best response joint path
  - Find the best response joint guard locations
  - Store the new paths and guard locations if they are new
- until no new paths or guard locations are generated

Upon termination, we solve the game matrix one more time to determine the mixed strategies u and v played by each team. The teams then generate a plan according to u and v accordingly and execute those plans.

In the next section we present the results of our two versus two paintball game.

4.4.4 Experimental results on two versus two paintball

For the problem shown in figure (4.5), running our double oracle algorithm leads to the series of joint path plans generated on each iteration shown in figure (4.6).

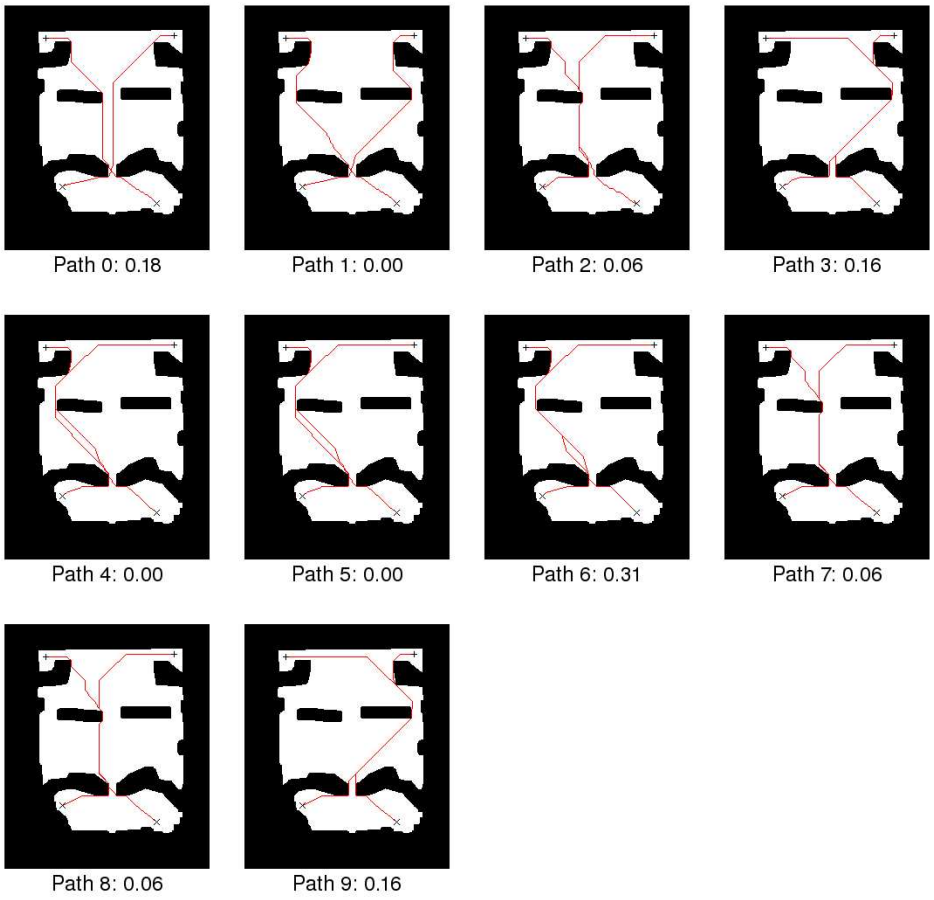


Figure 4.6: The paths generated by team one as the double oracle algorithm iterates. The first path is in the top left and the order proceeds left to right top to bottom. Each path is the best response to the joint guard locations shown in figure (4.7). The numbers from 0 to 1 below each figure indicate the probability of that strategy being played in the final game.

Similarly, the guard locations generated by the double oracle algorithm are shown in figure (4.7).

We implemented the above algorithm for several simulation problems as well as our

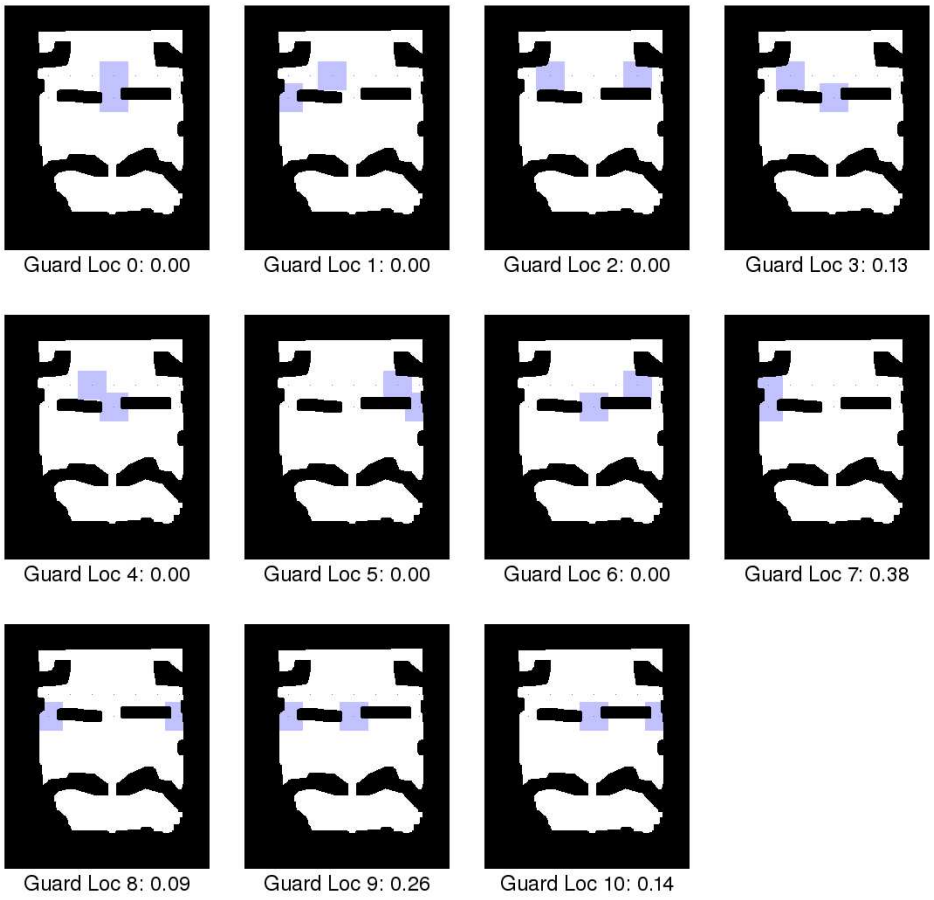


Figure 4.7: The guard locations which are the best responses to the paths in figure (4.6). The squares are the areas which are covered by the guard robots on team two. The probability of playing each strategy is labeled below each image.

two robot paintball game and found it to be robust in a wide variety of problems. It was implemented on a Pentium IV 1.2GHz machine with 1 Gigabyte of main memory. For the problem shown in figure (4.5) the wall clock time for execution was about ten seconds on average for all path plans and sensor locations to be generated.

The execution time is largely dependent on the amount of interaction between the path planning robots. If robots must coordinate every action, then the size of the master linear program will be such that it takes a long time to solve.

There is also a limited form of re-planning which can be done with this method. When team one gains some information about the strategy that team two has chosen, it can re-solve the matrix game with the knowledge that team two has chosen a particular strategy, and find the best response to that strategy and switch to that plan.

The robots physically played the strategies recommended by the double oracle algorithm. The results can be seen in a movie of the two versus two robot paintball available at: http://www-2.cs.cmu.edu/∼curt/research.html.

The movie demonstrates what happens when the robots $R1$ and $R2$ do not coordinate while path planning (they collide). It also demonstrates some of the re-planning capabilities; When team one gets shot at, it switches to a different plan to avoid the guard robots.

4.5 Discussion

The double oracle algorithm is specialized to the case where robots from team one are not able to make observations of the strategy chosen by team two. Allowing team one to make observations and change their actions accordingly would result in a full POMDP which is known to be difficult to solve. We implemented a simple form of

observation in which we simply re-plan when we have observed that a guard is in a given location. While not ideal, such a solution would work well in many practical cases.

There are two main contributions of this work. The first is the combination of two prior approaches into a novel framework for multi-robot or multi-agent coordination in the presence of an opposing team. The second is the application of this method to a real game of multi-robot paintball.

Bibliography

Arsham, H. (2003). Introduction to game theory: Winning business in a competitive environment.

Atwood, C., Kirby, B., and Lauwers, T. (2004).

Bowling, M. and Veloso, M. (2003). Simultaneous adversarial multi-robot learning. In *International Joint Conference on Artificial Intelligence*.

Buro, M. (2003). Real-time strategy games: A new ai research challenge. In *International Joint Conference on Artificial Intelligence*.

Filar, J. and Vrieze, O. (1992). Weighted reward criteria in competitive markov decision processes. *Methods and Models of Operations Research*, 36:343–358.

Kitano, H., editor (1998). *Proceedings of RoboCup-97: The First Robot World Cup Soccer Games and Conferences*, Berlin. Springer Verlag.

Lazerball (2004). http://www.lazerball.com.

McMahan, H. B. and Gordon, G. (2003). Planning in cost-paired markov decision process games. In *NIPS 2003 Workshop on Planning for the Real-World*.

McMahan, H. B., Gordon, G., and Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. In *Twentieth International Conference on Machine Learning (ICML)*.

Roy, N., Montemerlo, M., and Thrun, S. (2004). http://www-2.cs.cmu.edu/ carmen/.

Chapter 5

Related Work

In this chapter, we discuss some related work and the hole that our new method is attempting to address. We first discuss related work in the field of general Markov Decision Process multi-agent planning. In particular, there is one piece of work named factored MDP coordination (on which our work is partially based) which deserves specific discussion. Market based methods and how they are related to this work are then discussed. Lastly, we discuss how our work might be combined with other methods and where one might use our work when designing a multi-agent system from the ground up.

5.1 Factored MDP Coordination methods

The closest related work to that presented in this thesis is multi-agent coordination using the factored value representation developed by Carlos Guestrin and described best in his thesis: ((Guestrin, 2003, Chapter 3)). These factored MDPs are used in a multi-agent coordination mechanism developed by Guestrin and Gordon (Guestrin and Gordon (2002)).

5.1.1 The Factored MDP representation

We first describe the factored MDPs which are at the base of the multi-agent coordination algorithm. Factored MDPs use the primal linear program representation of an MDP. There is a value function $V(x)$ which has one real value per state x. The value function $V(x)$ can be represented as a vector $\mathbf{V}$ with one value per state. Solving an MDP then consists of solving the following LP:

$$\begin{aligned} &\max_{\mathbf{V}} \boldsymbol{\alpha}'\mathbf{V} \\ \forall x, a \quad &V(x) \leq c(x,a) + \gamma \sum_{x'} P(x'|x,a)V(x') \end{aligned} \tag{5.1}$$

$V(x)$ is the value of state x, $c(x,a)$ is the cost function and $P(x'|x,a)$ is the probability distribution over next states. The factored MDP method then approximates this linear program by representing every state value using a linear combination of basis functions $h_i(x)$ replacing $V(x)$ with $V(x) = \sum_x w_i h_i(x)$:

$$\begin{aligned} &\max_{w_i} \alpha(x) \sum_i w_i h_i(x) \\ \forall x, a \quad &\sum_i w_i h_i(x) \leq c(x,a) + \gamma \sum_{x'} P(x'|x,a) \sum_i w_i h_i(x') \end{aligned} \tag{5.2}$$

This new linear program only has as many variables w_i as there are basis functions h_i which is generally much smaller than the number of variables in the original LP. The basis functions are selected to accurately represent the value function. Unfortunately, the selection of good basis functions is currently a difficult problem. There is no known algorithm for selecting basis function for a general problem such that there

is a guaranteed error bound between the approximate value function and the true value function. de Farias and Roy (2001) have made good progress in this direction. Currently, they assume the existence of a probability distribution over constraints in the original exponentially sized linear program representation of the multi-agent MDP. If a user selects a probability distribution which is close to the optimal, then they can give and error bound, but the selection of such a probability distribution is currently an unsolved problem.

In a multi-agent problem, the number of joint states and joint actions is exponential in the number of robots. The main problem with the linear program in 5.2, is that it contains an exponential number of constraints as there is one constraint per setting of each of the joint state variables.

The solution proposed by factored MDPs, was to replace the exponential number of constraints with an equivalent *max* function:

$$\phi \geq \max_{x} \sum_{i} w_i c_i(x) - b(x) \tag{5.3}$$

Here, the constraints from 5.2 have been divided into coefficients $c_i(x)$ of the weights w_i and constants with respect to the weights $b(x)$. The main innovation was that the maximum in 5.3 can be achieved much more efficiently by realizing that the coefficients and constants are functions of limited scope over the state variables x.

We can re-write this maximum defined over all joint states as a series of linear constraints. For example, consider finding the maximum the following function over 4 binary variables x_1, x_2, x_3, x_4 taken from Guestrin (2003):

$$F(x_1, x_2, x_3, x_4) = f_1(x_1, x_2) + f_2(x_1, x_3) + f_3(x_2, x_4) + f_4(x_3, x_4)$$

To find the maximum of this function we could write all 64 possible joint values of $x_1 \ldots x_4$ or we could perform the following steps:

$$\max_{x_1,x_2,x_3,x_4} f_1(x_1,x_2) + f_2(x_1,x_3) + f_3(x_2,x_4) + f_4(x_3,x_4)$$
$$= \max_{x_1,x_2,x_3} f_1(x_1,x_2) + f_2(x_1,x_3) + \max_{x_4}[f_3(x_2,x_4) + f_4(x_3,x_4)]$$

Let e_1 (elimination function 1) be the maximum over x_4 of $[f_3(x_2,x_4)+f_4(x_3,x_4)]$. We then have:

$$\max_{x_1,x_2,x_3} f_1(x_1,x_2) + f_2(x_1,x_3) + e_1 x_2, x_3$$
$$= \max_{x_1,x_2} f_1(x_1,x_2) + \max_{x_3}[f_2(x_1,x_3) + e_1 x_2, x_3]$$
$$= \max_{x_1,x_2} f_1(x_1,x_2) + e_2(x_1,x_2)$$

By defining the elimination functions e_i which are required as we eliminate variables, we arrive at a representation which is not exponential in size.

Unfortunately, the factored MDP representation must represent each maximum over e_i using one constraint for every possible discrete value of the joint state variables in the scope of e_i. Let there be a state variable x_i such that $e_i = f(x_i)$. If the domain of x_i is (for example) a finely discretized continuous variable, then we require a constraint for each possible value of x_i.

It is also possible that the order in which we eliminate variables still leads to a large number of constraints. This would occur if there is a large number of constraints which depends on all of the state variables (such as a global resource constraint). The selection of an elimination order to minimize the number of added constraints is itself an NP-hard problem. This fact limits the usefulness of the approach.

5.1.2 The limitations of the Guestrin factored MDP representation

Choosing an elimination order : Choosing an elimination order of variables is a difficult task. If we choose a poor elimination order, it is possible that the factored MDP method will have a number of constraints close to that of the naive method of representing all constraints. The selection of an elimination order is an NP hard problem as described by Guestrin in his thesis ((Guestrin, 2003, Chapter 4)).

Complex state or action variables : Guestrin assumes each state or action variable is assumed to have a small number of discrete values for this method to be tractable. If not, a very large number of constraints must be added to ensure proper coordination. Specifically, one must add a number of constraints which grows as the cross product of the size of state or action space for every variable in a coordination constraint. By allowing any linear constraint on state action visitation frequencies, our algorithm is more robust to large numbers of discrete states and actions.

Large "induced width" constraints : The Guestrin method is limited in that it currently handles only a small number of discrete interacting actions and states well. We call a constraint which depends on many variables a constraint which has large "induced width". The Guestrin method of (Guestrin and Gordon (2002)) does not handle such constraints efficiently as each discrete joint value of variables causes an additional constraint to be added to the LP. This is related to the prior limitation of handling only state and action variables with a small number of discrete values.

Examples of constraints with large induced width are global resource constraints.

In a multi-robot path planning problem with limited fuel, the factored MDP representation would require an exponential number of constraints as the number of agents increases. This limitation applies to any problem with a global resource constraints.

There are several other methods that attempt to handle exponentially large constraint sets. The book on linear optimization by Bertsimas and Tsitsiklis Bertsimas and Tsitsiklis (1997) has the most common approaches (including the ones we have used in this thesis). Patrascu et al. (2002) builds on the representation developed by Guestrin where a variable elimination cost network is used to find only the necessary violated constraints in the exponential sized problem. These constraints are then added into the LP. The closest previous work to Guestrin's is that of Yannanakis (1991), in which they show that some exponentially sized LPs can be modeled using a polynomial number of constraints using appropriate variable substitution.

5.2 Other MDP based planning approaches

Several methods such as have been proposed to solve weakly coupled MDPs. In general there are two possible ways to decompose an MDP, serial decomposition and parallel decomposition. For multi-agent planning problems, we are particularly concerned with the parallel decomposition methods of which we discuss some below. The basic idea behind parallel decomposition methods is that they take a large problem and break it up into smaller interacting components as we have proposed here.

Singh and D.Cohn (1998) constrain feasible joint actions in a multi-agent MDP by eliminating certain joint actions from the full joint MDP. While they can guarantee optimality, their method requires explicitly enumerating the entire joint state and action space for a multi-agent MDP which limits the method to very small problems.

Meuleau et al. (1998) performs a parallel decomposition of the MDP where the sub-problems are joined only via global resource constraints such as fuel. More complex interactions (such as linear constraints between two specific robots), could not be planned with this method.

Kushner and Chen (1974) was the first to apply Dantzig-Wolfe decomposition to Markov chains, though their method could not be applied to multi-agent coordination problems such as those discussed in this thesis.

There are a wide variety of hierarchical MDP based planners David and S. (2001); Hauskrecht et al. (1998); Ryan and Reid (2000); Parr and Russell (1997) but none of these explicitly examine how the method would apply to multi-agent coordination. Many of these methods however could be combined with our method to solve one of the single agent planning problems. We surmise that this would be a good way to solve very large problems where even an individual agent's planning problem is too large for conventional MDP planning methods.

In fact, this is a significant advantage of our method. Once we have used Dantzig-Wolfe decomposition to break a problem into a master linear program and several slave MDPs, we can then use any other MDP solution methods proposed in the literature to solve these slave MDPs.

5.3 Market based methods

As we have described in chapter 3, the Dantzig-Wolfe decomposition has several similarities to auction based methods. The dual variables of the resource constraints correspond to prices each robot must pay (or receive) when they produce or consume a given resource. Whereas we have a master linear program coordinating multiple agents, an auction based method would have an auctioneer determine the optimal

resource allocation by observing the bids sent to it by the robots and clearing the auction. One main difference is that our master program needs to remember all previous vectors of state action visitation frequencies and thus constitutes a centralized component. Though most of the computation is performed by the individual agents, it is possible (as we have shown in chapter 3) that the master LP would grow to be very large. The solution mentioned in the previous chapter is to apply the Dantzig-Wolfe decomposition hierarchically. The main advantage of auction methods is that they do not necessitate a centralized component. Their greatest fault is the lack of strong optimality guarantees.

We will first describe the basic theory behind auction based methods and then discuss a problem domain in which both methods may be used and how they each would work on the same problem.

5.3.1 A brief review of auction methods

The basic idea behind auction methods for multi-robot coordination is that a task to be performed by a team is somehow divided up into a series of subtasks that must be executed by the team. The robots then bid on the subtasks until all of the subtasks are owned by a given robot. Robots then perform the subtasks that they own until the task is completed.

Much work has been done using auction methods for multi-robot coordination. Some of the earliest work was done by Dias and Stentz (Dias and Stentz (2001); Zlot et al. (2002); Dias (2004)) as well as Gerkey and Mataric (Gerkey and Mataric (2002, 2004)) and Golfarelli et al. (1997); Botelho and Alami (1999). Much of this work is based on the contract net protocol developed by Smith (1980).

We present one formal description of auction methods following the presentation of Gerkey and Mataric (2004) We require an *allocation* of tasks among robots. If

there are a set of N tasks $T =< t_1, t_2, \ldots, t_N >$ to be performed, we can define an allocation of those tasks to a set of K robots by defining a subset of those tasks that each robot will perform which we term T_i where $T_i \subset T$ and no two robots may own the same task ($T_i \cap T_j = \emptyset$).

It is assumed that each robot i can calculate for each task j the quality q_{ij}. The quality with which a robot can complete a task typically implies how well that task is done for that domain. It might be how efficiently a robot can use resources, the accuracy of a map made by a robot, how many blocks can be moved from A to B in some period of time etc. The concept of quality is very similar to the concept of rewards in Markov decision processes and in fact is sometimes called the revenue of a robot. Thus, in an exploration task, you might get a reward per meter squared of area explored, or some reward for every point reached in a traveling salesman problem.

With every task there is also a cost c_{ij} of performing each task j by robot i. This might be the distance traveled by a robot, how much fuel is used etc. We now define the utility u_ij of a task T_j for robot i as $u_{ij} = q_{ij} - c_{ij}$. The objective then is to maximize the utility of the entire team by finding a *task allocation* such that the team utility is maximized.

If we assume that each robot can execute at most one task at a given time then this problem can be approximated by phrasing it as the optimal assignment problem (OAP). If we have m robots and n tasks, assume that we have a prioritized list of tasks to be accomplished. If this is the case, then we can assign at most one robot to each task using the following integer program.

$$\max_{\alpha_{ij}} \sum_{i=1}^{m} \sum_{j=1}^{n} (\alpha_{ij})(u_{ij})$$

such that

$$\sum_{i=1}^{m} \alpha_{ij} = 1, \quad 1 \leq j \leq n$$

$$\sum_{i=1}^{n} \alpha_{ij} = 1, \quad 1 \leq i \leq m$$

$$\alpha_{ij} \in 0, 1$$

The α_{ij} are indicator variables which decide which robot is assigned to which task. The first constraint ensures that no more than one robot is assigned to a given task, and the second constraint ensures that no more than one task is assigned to a given robot. This can be solved in $O(mn^2)$ time using the Hungarian method developed in Kuhn (1955). This requires a centralized method however since all of the utilities from each robot for every task must be sent to a single agent.

Another alternative to this problem is to once again gain insight from the dual program. This leads to market based methods for coordination. Essentially, the assignment problem is approximated using a *task market*. The tasks are each sold by an auctioneer which could be located on any given robot. One can auction off each task in order of preference to the robots. For each task T_j, each robot i submits a bid for that task based on their utility u_{ij} if they do not already have a task. In this manner, a fully distributed system can solve the assignment problem. Often tasks are auctioned in a greedy manner, that is the highest utility tasks are offered for auction first to the highest bidder. This approach is detailed in Gerkey and Mataric (2004).

Although most auction based coordination mechanisms auction off tasks rather

than resources, the concept is the same for the kind of resource allocation we have described in this thesis. If we have shared resources and are seeking an optimal resource allocation, then the same method can be used to find an optimal assignment of resources.

The problem with allocating a single task or resource at a time as shown above, is that certain task or resource utilities may be dependent. If you are buying a hotel room in Paris, it will be worth less to you if you do not also have the airplane ticket to get you to Paris. Thus, it is likely that auctioning off single tasks or resources are unlikely to achieve an efficient solution to the problem.

Recent work in auction methods addresses this issue by performing a combinatorial auction. In this kind of auction, multiple tasks or resources may be auctioned off at the same time. Work by Hunsberger and Grosz (2000); Nair et al. (2002); Zlot and Stentz (2005) uses combinatorial auctions to perform multi-agent planning. The problem with combinatorial auctions is that they are very difficult to clear optimally. It is an NP hard problem, although much work has been done towards developing polynomial time approximations. As such, there are typically only weak guarantees of optimality when using this method. This said, much success has been achieved coordinating agents using auction algorithms.

Although a general combinatoric option might be slow, recent work by Zlot and Stentz (Zlot and Stentz (2003, 2005)), addresses this by using an AND/OR tree description of the task where higher levels of the tree are abstract tasks. The leaf nodes of the tree corresponds to tasks that may actually be executed by the robots. Using this tree, they can avoid the exponential complexity of having to try to bid on all possible combinations of tasks.

To give the reader an intuitive feel for how a particular auction might work as well as show how our method might solve the same problem, we present the following

example. After showing how each method might solve the problem separately, we show how the two methods could be combined.

5.3.2 An illustrative traffic flow example

Imagine the following example of traffic crossing bridges going from source locations or entry points to a particular goal location. Robots randomly (according to some probability distribution) appear at the entry points with a given probability on every time step. The objective is to minimize the average number of steps for all robots to reach the goal, equivalently we want to maximize throughput of robots appearing at the start and arriving at the goal. Intuitively, this means that some robots should go left over the larger but more distant bridge and some robots will go over the smaller but closer bridge on the right.

Modeling the traffic problem as a MIP

We model this problem using our multi-robot path planning model from chapter 3. We first describe the planning phase which will attempt to calculate the cost of going over either bridge as robots head toward the goal. Ideally, we would want a smaller number of robots mostly from the entry point on the right to take the small bridge, and the rest of the robots to take the larger bridge because there would be too much congestion on the smaller bridge. We will use our linear program model of the problem due to its higher efficiency, though it will only yield an approximate joint value function as we will only be matching constraints in expectation. Because the planning phase can only produce an approximate joint value function, we introduce an execution phase which is run at execution time to ensure that no collisions between robots occur.

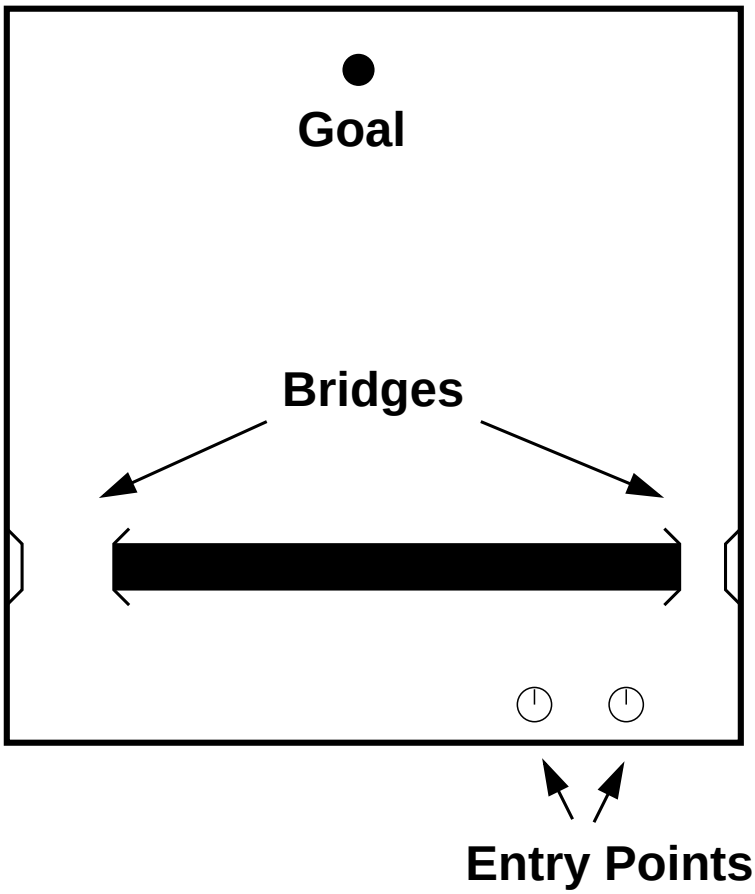


Figure 5.1: A simple traffic flow problem in a finely discretized grid world. Robots appear in the bottom right at either of the two entry points according to some probability distribution. We want to maximize traffic flow so that the average time for robots to reach the goal is minimized.

There are some important differences from our previous multi-robot path planner. Firstly, we cannot enforce deterministic policies because we will be dealing with some "average" number of robots entering the map at each of the entry points and so we use our simpler LP model without any integer constraints. We will treat all robots entering each point as an "average agent", thus we will be planning as if there are only two agents. One agent entering from the left entry point and one from the right.

Instead of a single robot starting in a particular position, we set the starting number of robots at each entry point position to be the mean number of robots (obtained from the probability distribution for that entry point) entering that state on a given time step. This can be done by modifying the $\mathbf{b}$ vector for each entry point to be the mean of the entry distribution for that point.

The inter-robot constraints state once again that no two robots can be in the same grid cell at a given time step except for the entry points. Thus, multiple robots may enter the world at the entry point but their first action must be to move off of each other. Time is not included in the state and so the constraints are met over the entire plan in expectation.

Other than these exceptions, the model is the same as that used in chapter 3. We can now solve the LP model with the Dantzig-Wolfe decomposition. To speed things up, we pre-compute the cost to go function from the goal position. That is we know exactly how much it costs to reach the goal from every position in the world. This can be used as a very accurate heuristic for the A* path planner of the individual robots.

The Dantzig Wolfe algorithm would iterate by selecting multiple paths for each of the "average" number of robots entering the world at the entry point and combines the paths using a probability distribution over each agents' path such that the inter-robot constraints are met. This will cause the joint value function to be more costly

to cross the narrow bridge on the right and less costly to cross the larger bridge on the left. This is the planning phase for the traffic control problem.

Now the execution phase. If we were to try to execute the above generated plan with stochastic policies directly, then the robots may try to run into each other because the constraints are only met in expectation.

We solve this problem *during execution*. Whenever there are two robots that want to use the same grid cell, we hold a mini auction where robots bid on the right to use a grid cell based on the distribution over value functions they were told to use by the master program to determine the cost of owning or not owning that cell at that particular time. This auction would resolve robots running into each other while still keeping them close to their average optimal paths.

An auction method for the traffic problem

While our method attempts to control robots at very fine level of control, an auction designer for the same problem would not do so. This is because a very large combinatorial auction would be required to auction off each of the fine level grid cells. In this method, we only require an execution phase auction method.

A plausible auction method would be as follows. "Tickets" to use the small bridge on the right are issued by an auctioneer (which might be a robot) which has tickets for the bridge which are valid for particular time-slice (time steps 5-10, 10-15).The time slice length for a ticket is tuned by the auction designer. With a ticket, you have the right to cross the small bridge during that time slice. Without one, you cannot cross the small bridge on the right. The area around the bridge that belongs to the ticket holder is also tuned by the auction designer.

For each entry point, we generate an individual value function ignoring the robots coming from the other entry point. They generate one value function for when they

do not have a ticket and one value function for when they do have a ticket for each time slice. These value functions will assume that a robot coming from a particular entry point cannot sell its ticket later nor buy another – that is, the value for a ticket for 5-10 minutes from now assumes that the robot can only cross the bridge 5-10 minutes from now rather than buy a different ticket. With these value functions, the robots can bid on any ticket, but will only want to bid on tickets that make it cheaper for them to take the bridge on the right than go around.

For example, let us say that you are a robot which has just entered the world at the left entry point. Examining your value function you know that it would cost a path length of 50 to go left over the larger bridge and 30 over the smaller bridge to the right. Examining your plan, you seek to buy the ticket over the smaller bridge for the time slice when you reach it. You would be willing to pay up to 20 points for the ticket at the time slice required to cross the bridge immediately when the robot arrived at the bridge. If the robot cannot get a ticket at the correct time slice, it could can similarly examine the next available time slice (meaning that you might have to wait to cross the bridge). If you successfully bid and acquire a useful ticket you proceed towards the right bridge, if not you proceed towards the left bridge.

At the start, tickets will be sold to the first bidder at any price. Every time step during execution every robot will offer the ticket that it owns for auction. Any other robot can bid on it. Bids on a ticket are made comparing the cost to the goal with that ticket versus going around the left bridge or buying another ticket for a later time slice. Only one ticket is auctioned at a time and the tickets are auctioned in sequence. Thus, robots may acquire a ticket to cross the bridge, but then another robot appears who is willing to pay more for it and the first robot would then sell his ticket to the second for a profit.

During execution, the robots may also attempt to run into each other. When a

conflict arises for a particular grid cell, the robots bid for the right to use a grid cell based on the cost to wait or take another grid cell. Ties are broken randomly.

This auction method would likely generate a good traffic flow control method for this problem.

Combining the two methods

Though we have begun with a MIP model with more pre-planning effort and a lightweight execution phase, we can also modify the execution phase of the LP plan to be almost identical to that of the auction method.

Essentially, we put the exact same ticket auction system into the execution phase of the LP method. However, when the robots bid on the tickets, they will bid using a different value function (determined by the master program) which would more accurately represent the congestion around the bridge and thus should do better than the plain auction mechanism which doesn't take account the congestion around the bridge into their value functions when considering how much to pay for the tickets.

Note that one could consider this combined method as using a better execution phase method for the MIP model or as using the MIP to calculator more accurate utility functions for the auction based methods. However one views it, there are significant possibilities for combining the two approaches.

Adding more bridges and expanding the problem

Clearly, we can expand this problem to be many bridges and many tickets. An interesting experiment would be to expand this problem to be traffic planning for a city. This could be done by adding many entry points and goals, multiple intersections instead of bridges.

5.3.3 The main differences between MDP based coordination and Market based

In this section we try to highlight the main differences between planning with auction methods and solving our MIP or LP models of multi-agent MDPs. We use the example above to highlight some of these differences.

5.3.4 Main Advantages of MDP based coordination

A more natural way to specify the task One only need specify low level rewards and low level action costs for particular actions in particular states. Resources can be made for many low level items such as grid cells used in path planning, even if they are never used.

Typically, auction methods will try to coordinate the robots at a much higher level. Having so many resources would make the clearing of a combinatorial auction very slow. Thus, auction/market methods typically coordinate robots through the use of higher level resources such as task allocation. Other market based methods have robots bid on predefined roles. In the multi-robot soccer domain Vail and Veloso (2003) had robots bid for the roles of attacker, defender and goalie.

In the example above, the auction designer needs to come up with the idea of having tickets for the bridge which robots will then bid upon. Using the Dantzig-Wolfe decomposition method, we need only specify the low level interaction rules, and our method will do most of the work.

Very recently, there has been some work Kalra and Stentz (2003) in auction mechanisms to coordinate robots at a much finer scale. This method looks to add the capability to reason about fine grained interactions between robots using

auction methods. It does so in a manner very similar to the one proposed in this thesis, though it does not offer any optimality guarantees.

Optimality guarantees The guarantees for out multi-agent coordination method based on our LP and our MIP models tend to be stronger than those given for auction methods. Given an exact model, we can guarantee an optimal solution to the multi-agent problem. There has been some recent work on optimality guarantees for single task single robot auctions as well Gerkey and Mataric (2004).

Efficient search Though we may define many thousands of constraints in our MIP model, only the necessary constraints will ever be generated according to when the robots determine there are conflicts. This can be done by using constraint generation techniques. In our traffic flow example, constraints would be generated only at bridges over which robots wished to cross at the same time as other robots. While our method discovers this automatically, an auction method would have to use a similar method to discover which resources are contested.

We believe that similar constraint generation methods could be applied to market/auction methods. Essentially, resources would only be auctioned when desired by two or more robots.

A well defined mathematical framework for MDPs: Because we have based our method on Markov Decision Processes phrased as linear programs, we gain many of the advantages inherent in using MDPs. Firstly, we need only define the dynamics of the world and the reward structure. Once this is done, the method will generate value functions and or policies.

Auction methods on the other hand must carefully design the utility functions

for the robots. For many tasks, the quality of the coordination depends strongly on how these utility functions are designed. Currently, such functions are hand designed for each task and optimality guarantees can only be given for certain utility function designs.

There is certainly ambiguity that arises when designing the rewards for an MDP, but for most any tasks the reward structure is very easy to define, and this has been born out in practice. For multi-robot coordination using auction methods however, more of the method must be defined by a highly skilled auction designer.

In the above example, we simply define the individual robot dynamics of movement, the fact that each action costs one point, and the inter-robot constraints that no two robots may be in the same place at the same time. The algorithm does the rest. The auction method however needed to identify the bridge as an important resource and specifically design an auction in which tickets are sold to that bridge in a specific way. If the designer neglects to identify an important resource, then solutions will be much significantly worse than optimal.

Global resource constraints: Our multi-agent coordination algorithm has a much more natural way of handling global resource constraints. Imagine that we have a limited amount of fuel. We can add a single constraint to the master LP to accommodate this requirement.

5.3.5 Main Advantages of Market based coordination

Speed of planning: One of the clear advantages of auction based methods is that they will likely be much faster than our LP and MIP model based multi-robot coordination. Auctions can be designed so that the auction clearing is very fast.

Thus many more auctions can be performed and in general the process can be performed in real time.

The difficulty is that more thought must go into the utility function design so that the auctions can be cleared quickly. A combinatorial auction with hundreds or thousands of commodities cannot, in general, be cleared quickly.

Robustness to death of robots and change in environments: Largely due to the speed with which the auctions can be cleared, teams of robots coordinated with this method can react quickly to changes in the environments. When a robot fails, for example, one can simply auction off the tasks that robot was to perform. Other changes in the environment which affect only certain tasks can be handled by simply holding a new auction for those tasks.

No centralized component is required: One of the biggest advantages is that no centralized component is required for typical auction based multi-robot coordination. This means that there is no single point to which all information must flow, and thus become a single failure point. Though we could replicate the master LP/MIP on several different agents to avoid this single failure point, each agent must still store all previous plans. In our experience, the amount of data transferred and the size of the master LP/MIP has been very reasonable.

Auction methods on the other hand need not worry about sending all information to a single agent. This ensures that there is no single point of failure. It also means, that it is less likely that any given auction becomes extremely difficult to solve.

We have described in our small example that it is possible to combine the two methods. There has also been work by several people which attempts to bring auction

method coordination closer to Dantzig-Wolfe coordination done by Kutanoglu and Wu (1999); Jose et al. (1997) as well as many others.

5.4 Engineering a complete system with MIP method

As we have mentioned many times throughout this thesis, our multi-agent coordination method combines well with several other algorithms. One could use our algorithm in a number of ways for a situation involving multi-robot coordination.

The master LP/MIP coordination could be used as a higher level component in a system where the solution of the individual slave robot problems is provided by a custom algorithm. The individual slave robot problems could be solved as MDPs, but the master LP/MIP replaced with a custom algorithm. Teams of robots coordinated using a different coordination method (such as auction methods) could be internally coordinated with our method. We now discuss some of these possibilities in more detail.

5.4.1 Replacing the slave MDPs

At a higher level, one could use the master LP/MIP to coordinate the actions of multiple slave robots. In general, we recommend solving the slave programs using an MDP specific solver, but it is also possible to instead use a custom algorithm to generate state action visitation frequencies and costs. This is what we have done with our multi-robot path planner, where instead of solving an MDP we use an A* path planner.

This idea could be extended in a number of ways. We can replace the lower level slave programs with a variety of methods: Factored MDPs (Guestrin (2003)), macro actions (Hauskrecht et al. (1998)), state abstraction (David and S. (2001)), temporal

abstraction (Sutton et al. (1999)), Hierarchies of abstract machines (Makar et al. (2001)) or any number of proposed methods for solving large weakly coupled MDPs.

Alternatively, if the slave robots are performing a task for which there exists a custom algorithm (such as A* in our path planning example), that algorithm could be used instead of a generic MDP solver. One could also use hand-coded behaviors here. Often a behavior is the fastest way to get some lower level functionality of a robot to work.

5.4.2 Replacing the master MIP

At a high level, the master MIP performs a fairly generic task. Given a list of all previously generated plans by the slaves, choose a set of prices on resources such that the slave planners will generate plans which meet the coordination constraints. For a given task, one could replace the master program with a faster task-specific algorithm which would perform the same function.

5.4.3 Hierarchies of coordination

We have already mentioned that our method can coordinate a hierarchy of robot teams. Each layer of the hierarchy could be coordinated using MIP/LP models and Dantzig-Wolfe coordination. It would also be feasible to replace some layers with other coordination methods. For example, if the master LP at the highest level was becoming simply too large, a system designer might replace the master LP with a non-optimal but fast auction algorithm. In general the coordination algorithm at any point in a coordination hierarchy can be replaced with a task specific coordination algorithm.

To see how one might replace the master program with an equivalent task specific

coordination mechanism we see that there are several things which the master program does. First, it must remember each of the plans generated by the slaves. Each sub-plan tells the master program that a particular robot can generate or consume some set of resources with a specific cost. It must then match supply and demand for resources by combining sub-plans in such a way as to obtain minimal cost.

Bibliography

Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific.

Botelho, S. and Alami, R. (1999). M+: A scheme for multi-robot cooperation through negotiated task allocation and achievement. In *International Conference on Robotics and Automation (ICRA)*.

David, A. and S., R. (2001). State abstraction for programmable reinforcement learning agents. In *Neural Information Processing Systems*.

de Farias, D. and Roy, B. V. (2001). The linear programming approach to approximate dynamic programming.

Dias, B. (2004). *TraderBots: A New Paradigm for Robust and Efficient Multirobot Coordination in Dynamic Environments*. PhD thesis, Carnegie Mellon University.

Dias, M. and Stentz, A. (2001). A market approach to multirobot coordination.

Gerkey, B. and Mataric, M. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*.

Gerkey, B. P. and Mataric, M. J. (2002). Sold!: Market methods for multi-robot control.

Golfarelli, M., Maio, D., and Rizzi, S. (1997). A task-swap negotiation protocol based on the contract net paradigm. Technical report, CSITE (Research center for informatics and telecommunication systems), university of bologna.

Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Stanford University.

Guestrin, C. and Gordon, G. (2002). Distributed planning in hierarchical factored MDPs. In Darwiche, A. and Friedman, N., editors, *Uncertainty in Artificial Intelligence (UAI)*, volume 18.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., and Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, pages 220–229.

Hunsberger, L. and Grosz, B. (2000). A combinatorial auction for collaborative planning. In *Proceedings of the fourth international conference on multi-agent systems (ICMAS)*.

Jose, R., Harker, P., and Ungar, L. (1997). Coordinating locally constrained agents using augmented pricing. working paper.

Kalra, N. and Stentz, A. T. (2003). A market approach to tightly-coupled multi-robot coordination: First results. In *Proceedings of the ARL Collaborative Technologies Alliance Symposium*.

Kuhn, H. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*.

Kushner, H. and Chen, C. (1974). Decomposition of systems governed by markov chains. *IEEE Transactions on Automatic Control*.

Kutanoglu, E. and Wu, S. (1999). On combinatorial auctions and lagrangean relaxation for distributed resource scheduling. *IIE transactions*.

Makar, R., Mahadevan, S., and Ghavamzadeh, M. (2001). Hierarchical multiagent reinforcement learning.

Meuleau, N., Hauskrecht, M., Kim, K., Peshkin, L., Kaelbling, L. P., Dean, T., and Boutilier, C. (1998). Solving very large weakly-coupled Markov decision processes. In *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 165–172, Madison, WI.

Nair, R., Ito, T., Tambe, M., and Marsella, S. (2002). Task allocation in the rescue simulation domain: A short note. In *Robocup 2001: the fifth robot world cup games and conferences*.

Parr, R. and Russell, S. (1997). Reinforcement learning with hierarchies of machines. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

Patrascu, R., Poupart, P., Schuurmans, D., Boutilier, C., and Guestrin, C. (2002). Greedy linear value-approximation for factored markov decision processes. In *AAAI*.

Ryan, M. and Reid, M. (2000). Learning to fly: An application of hierarchical reinforcement learning. In *Proc. 17th International Conf. on Machine Learning*, pages 807–814. Morgan Kaufmann, San Francisco, CA.

Singh, S. and D.Cohn (1998). How to dynamically merge markov decision processes. In *NIPS*, volume 10.

Smith, R. (1980). The contract net protocol: high level communication and control in a distributed problem solver. *IEEE transactions on computers*.

Sutton, R. S., Precup, D., and Singh, S. P. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.

Vail, D. and Veloso, M. (2003). Multi-robot dynamic role assignment and coordination through shared potential fields.

Yannanakis, M. (1991). Expressing combinatorial optimization problems by linear programming. *Journal of Computer and System Sciences*, 43:441–466.

Zlot, R. and Stentz, A. (2003). Multirobot control using task abstraction in a market framework. In *International conference on field and service robotics*.

Zlot, R. and Stentz, A. (2005). Complex task allocation for multiple robots. In *IEEE Proceedings of the International Conference on Robotics and Automation (ICRA)*. submitted.

Zlot, R., Stentz, A., Dias, M., and Thayer, S. (2002). Multi-robot exploration controlled by a market economy.

Chapter 6

Conclusions

6.1 Contributions

There are several main contributions of this thesis:

Two novel modeling methods for multi-agent MDPs: Naive multi-agent MDPs create a state and action space size which is exponential in the number of robots. We have presented two modeling languages for multi-agent MDPs which avoid this exponential increase in size.

The first modeling method uses a mixed integer program to represent a multi-agent coordination task. Each robot or agent plans using separate MDPs with modified cost functions. The agents are connected via linear constraints on their state action visitation frequencies and the possible addition of integer constraints on the variables. This mixed integer program represents the original multi-agent planning problem with a total number of variables and constraints which is exponentially smaller than the naive multi-agent MDP model. Because of the flexibility of the MIP model, we can guarantee optimality for a large class of problems. The main drawback of this method is that mixed integer programs

can be slow to solve. To address this concern we introduced the Dantzig-Wolfe decomposition algorithm. A further simplification of the mixed integer model leads to our second modeling method for multi-agent coordination tasks.

The second modeling method removes some of the modeling flexibility of the MIP model by removing the integer constraints on action visitation frequencies yielding a linear program model for multi-agent coordination. While not as flexible as the MIP model, we can still meet inter-robot constraints in expectation. For several problems, this generates a very good joint plan in a very reasonable amount of time.

If a multi-agent task cannot be specified exactly using our models, then it is still possible to approximate the original problem closely by grouping highly-interacting agents into joint agents.

It is also possible to approximate non-linear interaction constraints with linear or integer constraints. The quality of the resulting policy when using such approximations will vary depending on the given task. For several of the problems we have investigated, we were able to exactly model the task, thus guaranteeing an optimal joint policy for the agents in the team.

Efficient solution methods for our multi-agent models: Both the mixed integer and linear program models can be solved efficiently using Dantzig-Wolfe decomposition. In the case of the mixed integer program we combine Dantzig-Wolfe decomposition with the branch and bound method. Using these techniques we can solve both of our models in an efficient distributed manner.

The Dantzig-Wolfe decomposition also allows for an intuitive economic model which makes it easier for an end user to model a multi-agent task. Specifically, the Dantzig-Wolfe decomposition has many parallels to market based coordina-

tion mechanisms. As such, a user who has previously used these methods would have less difficulty using our method should they wish stronger optimality guarantees.

Though we have no guarantees on the number of iterations required by the Dantzig-Wolfe algorithm, we have found that the number of iterations required is quite small (less than ten) and the computation required per iteration is exponentially less than that for the full problem.

In summary, we have found that these methods can solve real-world multi-agent coordination problems efficiently as well as guaranteeing an optimal solution to the model if the model exactly depicts the original multi-agent problem.

A method to determine the optimal strategies for competing teams: (McMahan et al. (2003)) solves a particular form of single agent game where an opponent chooses the cost function for a player planning using an MDP. By using our multi-agent coordination method to represent opposing teams of agents, we have developed an efficient method for solving a certain class of team competition games.

A game of robot paintball: In particular, we have used our algorithms to solve a two versus two game of multi-robot paintball and shown that our algorithms can be efficiently implemented and deployed on real robots.

Our method is that that it avoids several of the major hurdles encountered previous work. In particular, we can easily represent global resource constraints, avoid problems with "high induced width", and can handle larger state and action spaces.

6.2 Future Work and Open Problems

6.2.1 Partial Observability

We have not previously discussed how to handle partial observability using our multi-agent coordination mechanism. The first method that could be tried is representing an MDP over belief states as opposed to regular state space as proposed by (Roy and Thrun (1999)) or planning over the compressed belief space as proposed by (Roy and Gordon (2002)).

6.2.2 Combination with other methods

As we described in chapter 5, once we have used Dantzig-Wolfe decomposition to break the multi-agent problem into a separate set of slave problems, we can then apply any method to solve the sub-MDPs. We could solve the sub-MDPs with Factored MDPs (Guestrin (2003)), macro actions (Hauskrecht et al. (1998)), state abstraction (David and S. (2001)), temporal abstraction (Sutton et al. (1999)), Hierarchies of abstract machines (Makar et al. (2001)) or any number of proposed methods for solving large weakly coupled MDPs.

In his thesis, Guestrin also discusses the dual linear program representation and defines an approximated factored dual linear program (Guestrin (2003)). It might be possible to approximate some interactions using factored MDPs while leaving other interactions exactly specified.

6.2.3 Representing non-linear functions

The main weakness of our approach is that it relies on linear constraints of real or integer variables. Thus, non-linear functions of variables cannot be directly represented with this method.

One possibility is simply to allow non-linear constraints and reward functions, but then we are forced into the hard problem of non-linear optimization for which there are no general algorithms which guarantee optimality. In (Gordon (1999)), several possible methods for representing these types of nonlinearities are discussed which are well suited to linear program models.

A possible promising direction is to approximate non-linear constraints using linear constraints. In particular, we should be able to represent many convex functions efficiently using linear constraints. We have shown that this is feasible using the quadratic cost on fuel for multi-robot path planning.

6.2.4 Drawing on mixed integer programming tools

The main tools used in this thesis are drawn from linear and mixed integer programming solution techniques. Our contribution is determining how best to apply these techniques to multi-agent coordination as well as determining in what ways they limit the kinds of problems which can be solved.

There are several current linear and mixed integer solution techniques which are promising directions for future work. The first is the determination of the first feasible corner for the Dantzig-Wolfe decomposition described in (Bertsimas and Tsitsiklis (1997)). This would speed the solution of the Dantzig-Wolfe decomposition.

The method of Gomory cuts for mixed integer programming is also a promising avenue. Instead of using branch and bound, this method attempts to guarantee an integer solution to a mixed integer program by adding constraints which eliminate non-integer solutions. We chose branch and bound because it can be easily combined with Dantzig-Wolfe decomposition, but Gomory cuts can also be combined with branch and bound (Balas et al. (1996)). This combination might lead to a more effective solution method for solving mixed integer multi-agent coordination models.

6.2.5 Error bounds

Our coordination method yields optimal policies for a large class of problems which can be represented exactly using our linear and mixed integer program models. Unfortunately, if the original problem is not represented exactly, then we have no guarantee on the difference between the induced value function and that of the optimal value function. Determining such an error bound is an important next step.

6.3 Summary

In this thesis, we have developed novel multi-agent coordination models and solution methods. We can compute the policies for all agents in a system distributing as much of the computation as possible while still maintaining the guarantees of centralized methods. It overcomes several limitations of other state of the art methods. For many problems, we can guarantee an optimal solution.

We have applied this method to multi-agent competitive games developing a novel team competition algorithm. This algorithm was shown to compute reasonable strategies for a physically embodied robots in a game of two vs. two paintball.

Lastly, there has been interest by other researchers in applying our method to new problems. Multi-agent pursuit evasion problems as well as different solution methods for our integer models using rounding are currently being considered. This is a promising start for the representations and methods developed here.

Bibliography

Balas, E., Ceria, S., Cornuéjols, G., and Natraj, N. (1996). Gomory cuts revisited. *Operations Research Letters*, 19:1–9.

Bertsimas, D. and Tsitsiklis, J. (1997). *Introduction to Linear Optimization*. Athena Scientific.

David, A. and S., R. (2001). State abstraction for programmable reinforcement learning agents. In *Neural Information Processing Systems*.

Gordon, G. (1999). *Approximate Solutions to Markov Decision Processes*. PhD thesis, Carnegie Mellon University.

Guestrin, C. (2003). *Planning Under Uncertainty in Complex Structured Environments*. PhD thesis, Stanford University.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., and Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, pages 220–229.

Makar, R., Mahadevan, S., and Ghavamzadeh, M. (2001). Hierarchical multiagent reinforcement learning.

McMahan, H. B., Gordon, G., and Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. In *Twentieth International Conference on Machine Learning (ICML)*.

Roy, N. and Gordon, G. (2002). Exponential family PCA for belief compression in POMDPs. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing 15 (NIPS)*, pages 1043–1049, Vancouver, Canada.

Roy, N. and Thrun, S. (1999). Coastal navigation with mobile robots. In *Advances in Neural Processing Systems 12*, volume 12, pages 1043–1049.

Sutton, R. S., Precup, D., and Singh, S. P. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.