Optimization Techniques for Task Allocation and Scheduling in Distributed Multi-Agent Operations

by

Mark F. Tompkins

Submitted to the
Department of Electrical Engineering and Computer Science

In partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science

at the
Massachusetts Institute of Technology

June 2003
© Mark Freeman Tompkins, MMIII. All Rights Reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author_____
Department of Electrical Engineering and Computer Science
May 24, 2003

Certified by_____
John J. Turkovich
Member of the Technical Staff, Charles Stark Draper Laboratory
Technical Supervisor

Certified by_____
Munther Dahleh
Professor of Electrical Engineering and Computer Science, MIT
Thesis Advisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Studies

-This Page Intentionally Left Blank-

Optimization Techniques for Task Allocation and Scheduling in Distributed Multi-Agent Operations

By
Mark F. Tompkins

Submitted to the Department of Electrical Engineering and Computer Science on
May 21, 2003, in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science

Abstract

This thesis examines scenarios where multiple autonomous agents collaborate in order to accomplish a global objective. In the environment that we consider, there is a network of agents, each of which offers different sets of capabilities or services that can be used to perform various tasks. In this environment, an agent formulates a problem, divides it into a precedence-constrained set of sub-problems, and determines the optimal allocation of these sub-problems/tasks to other agents so that they are completed in the shortest amount of time. The resulting schedule is constrained by the execution delay of each service, job release times and precedence relations, as well as communication delays between agents. A Mixed Integer-Linear Programming (MILP) approach is presented in the context of a multi-agent problem-solving framework that enables optimal makespans to be computed for complex classifications of scheduling problems that take many different parameters into account. While the algorithm presented takes exponential time to solve and thus is only feasible to use for limited numbers of agents and jobs, it provides a flexible alternative to existing heuristics that model only limited sets of parameters, or settle for approximations of the optimal solution. Complexity results of the algorithm are studied for various scenarios and inputs, as well as recursive applications of the algorithm for hierarchical decompositions of large problems, and optimization of multiple objective functions using Multiple Objective Linear Programming (MOLP) techniques.

Technical Supervisor: John J. Turkovich
Title: Technical Staff, Charles Stark Draper Laboratory

Thesis Supervisor: Munther Dahleh
Title: Professor of Electrical Engineering and Computer Science, MIT

-This Page Intentionally Left Blank-

Acknowledgments

This thesis was prepared at The Charles Stark Draper Laboratory, Inc., under internal Research and Development.

Publication of this report does not constitute approval by the Draper Laboratory or any sponsor of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I wish to acknowledge Mr. John J. Turkovich for giving me the opportunity to study at Draper, for introducing me to autonomous multi-agent systems, and for providing guidance and detailed feedback during my work on this project.

I also wish to acknowledge Prof. Munther A. Dahleh for providing his expertise on the ideas that motivated this project, and for patiently reading and reviewing my thesis.

I would also like to acknowledge Dash Optimization, Inc. for graciously providing me the use of their optimization software, without which much of the work in this thesis would not have been possible.

I would like to thank my friends Raj, John, and Greg for keeping me motivated with their constant encouragement and positive reinforcement.

Finally, I would like to thank my family for all of their love and support, and Mary for proofreading my thesis and for always being there to help me keep my chin up.

Permission is hereby granted by the Author to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

_____

Mark F. Tompkins
May 21, 2003

-This Page Intentionally Left Blank-

Table of Contents

-This Page Intentionally Left Blank-

Chapter 1

Introduction

Serious research on task allocation and scheduling problems began in the 1960's, and has become a popular research topic in the past few decades as multiprocessor systems have emerged and demanded efficient parallel programming algorithms. Many studies have been done on the complexities, classifications, and techniques required for solving these problems. Due to the inherent complexity of scheduling problems, however, most researchers have limited their studies to very simple classes of problems and approximation heuristics for solving them. While these types of problems and their solution methods are useful for establishing precise mathematical worst-case bounds and theorems, many of them lack the complexity necessary for modeling any sort of real-world scheduling problem that might occur in manufacturing, business, or military scenarios.

For an example, consider a scenario where a military commander needs to acquire data about potential targets before forming a battle plan. This data could include enemy locations, weather conditions, logistics, and terrain conditions. Suppose that there are various human and robotic agents stationed throughout the area of operations. Each

agent has the ability to conduct information processing tasks such as collect different types of data, aggregate data, analyze patterns in data, and send their results to other agents. The commander wants to assign these information-processing tasks to the agents so that the results are returned to him in the shortest amount of time. The optimal schedule produced is subject to the processing power/speed of each agent, the available communication bandwidth between the agents, and the order in which the data must be collected. This scenario can be modeled in a manner similar to a multiprocessor scheduling problem, where each agent is modeled as a computing entity that has specific execution and communication delay times for each job it is assigned.

Many existing scheduling techniques cannot accurately model multi-agent scenarios such as this one because the problem includes many more parameters than the techniques' limited problem models take into account. Due to the $\mathcal{NP}$ -hardness of all but the simplest classifications of scheduling problems [17], most researchers have chosen to develop scheduling algorithms that sacrifice optimality for a heuristic solution that runs in polynomial time. However, even these heuristic algorithms usually make some assumptions and simplifications for the problem to make it computationally feasible. This thesis focuses on a more comprehensive definition of the multi-agent scheduling problem in the context of a general problem-solving framework, and presents a technique for computing these schedules that runs in exponential time, but produces optimal solutions.

1.1 Problem Overview

This thesis examines problems where there is an environment in which one or more *agents* are given an *objective* to complete in the shortest amount of time possible. This is called minimizing the *makespan* of a multi-agent schedule. The objective can be divided into sub-tasks, or *jobs*, which can be assigned to other agents in the environment. Each agent offers different *services* that can be used to complete these jobs. The following terminology is used to describe this particular class of problems:

- **Agent, $a \in \{A\}$:** Any physical or functional entity that is capable of acting in an environment, and communicating with other agents in that environment. An agent can provide a set of *services* to other agents, and can use these services to help satisfy an *objective*. An agent can be a person with an assigned role (such as a soldier), a piece of software, an autonomous robotic vehicle, or an entity that represents a collection of other agents.

- **Objective:** A final goal state to be reached. This thesis looks at situations where an objective can be achieved by completing a finite set of *jobs.*

- **Job, $j \in \{J\}$:** An individual task that must be completed to help satisfy an objective. Jobs may be related to one another through a *job precedence graph*, and can sometimes be sub-divided into smaller sets of jobs.

- **Job Precedence Graph:** A directed, acyclic graph (DAG) where each node corresponds to a job, and each arrow indicates that one job must be completed before the next job begins. A job precedence graph may correspond to a data-flow graph, where data from the result of one job is needed as input to another job before it begins.

- **Service:** A functionality provided by an agent, which may be the ability to complete a particular job or set of jobs.

- **Job Start Time, S_j:** The time at which job j begins execution

- **Execution Delay, $D^{j,a}$:** The time it takes for agent a to execute service that completes job j. Execution delay depends on both the amount of data that needs to be processed in order to complete a job, as well as the processing power of the agent performing the service.

- **Communication Delay, $C^{j,a,b}$:** The time that it takes to communicate the results of job *j* from agent *a* to agent *b*. Communication delay depends on the amount of data being transmitted and the available bandwidth between pairs of agents.

- **Release Time, R^j:** The time that job *j* can begin once its predecessors have completed. If job *j*'s predecessors are completed before the release time, that job's execution still may not begin until the release time has passed.

- **Makespan, C_{max}:** The total time needed to complete a set of jobs that has been scheduled on one or more agents. Given a set of jobs {J} and agents {A}, the makespan can be defined as $C_{\max} = \max_{j \in J, a \in A} (S_j + D^{j,a})$. In the problems this thesis explores, finding an optimal schedule means assigning jobs to agents such that the schedule's makespan is minimized.

1.2 Approach

The goal of this thesis is to develop a methodology for solving complex scheduling problems that take many parameters into account. A new approach is presented for solving the classic multiprocessor scheduling problem using Mixed Integer-Linear Programming (MILP) techniques. These techniques can be used to compute optimal solutions to the most complex classes of scheduling problems by taking into account parameters such as communication delay, job release times, and multiple job precedence graphs. Most existing algorithms and heuristics only take a few of these parameters into consideration, and often place limits on the complexity or structure of the problem such as setting all execution times the same or requiring that the job precedence graph be in the form of a rooted tree. These heuristics also sacrifice completely optimal solutions in exchange for approximations in order to achieve a faster running time.

Chapter 2 of this thesis begins by reviewing relevant background information, including complexity theory, scheduling notation, problem classification, existing algorithms and heuristics, and Mathematical Programming theory. Chapter 3 presents the general

framework that is used in this thesis to model problem-solving in a multi-agent system, and uses theoretical examples to illustrate the formulation, decomposition, and allocation of tasks to multiple agents. Chapter 4 introduces the MILP technique that is used by this thesis to solve various multi-agent scheduling problems, and demonstrates how this algorithm can be used to minimize the makespan of these schedules as well as optimizing multiple additional objective functions. Chapter 5 examines the scheduling algorithm's run-time performance based on various inputs, and illustrates an example where schedule optimization is used to choose an optimal agent organization. Chapter 6 discusses conclusions drawn from this project and possible future research in this domain.

-This Page Intentionally Left Blank-

Chapter 2

Background

To understand scheduling problems and techniques used to solve them, it is essential to understand the theoretical background that these problems are based on. This chapter provides an overview of numerical complexity theory, scheduling theory and classification, as well as Mathematical Programming theory, all of which are used as a research basis for this thesis.

2.1 Complexity Theory

Computational complexity theory deals with the resources required during computation to solve a given problem. In terms of scheduling problems, we are concerned with the time required to solve the problems to optimality as they get larger. To examine this, we must look at the *running time* of the algorithms developed to solve these problems. The running time describes the number of operations that must be performed as a function of the number of input variables to the algorithm. For example, given n points in a plane, the number of operations required to find the two closest points in the plane is $n*(n-1)/2$,

which corresponds to a running time of $O(n^2)$, since only the most significant term in the polynomial needs to be considered.

In general, we want to determine whether an algorithm can be run in polynomial time or exponential time. A polynomial-time algorithm is one where the running time is proportional to $O(n^k)$ for some constant k, while an exponential-time algorithm has running time proportional to $O(k^n)$. As the number of inputs to a problem increases, polynomial-time algorithms tend to run much faster than exponential-time algorithms because they have a slower growth rate. This is not always the case, as a polynomial-time algorithm that runs in $10e^{100*}n$ steps takes much longer to solve than an exponential-time algorithm that runs in $10e^{-100}*2^n$ steps even for relatively large inputs. In general, however, this distinction holds true.

Different classifications of problem complexity were developed by Cook [5] and Karp [12] in the 1970's to help determine which types of problems were likely to have fast solution techniques. They define a decision problem as belonging to the class $\mathcal{P}$ (Polynomial) if the algorithm to solve it can be run in polynomial time on a deterministic machine. A decision problem is considered $\mathcal{NP}$ (Non-deterministic Polynomial) if its solution can be found in polynomial time on a non-deterministic machine, or equivalently, if a positive or "yes"-solution to the problem can be checked in polynomial time for correctness. It follows that $\mathcal{P} \in \mathcal{NP}$, since the solution to any problem in $\mathcal{P}$ can be checked for correctness by simply running the polynomial time algorithm that generated the solution. Whether or not $\mathcal{P} = \mathcal{NP}$, however, is still an open question that has not yet been proven or disproven [11].

The hardest class of problems to solve in $\mathcal{NP}$ are known as $\mathcal{NP}$-complete problems. Cook [5] defines a problem as $\mathcal{NP}$-complete if it is in $\mathcal{NP}$, and every other problem in $\mathcal{NP}$ is reducible to it in polynomial time. $\mathcal{NP}$-complete problems are considered to be harder than other problems in $\mathcal{NP}$ because, presently, all known $\mathcal{NP}$-complete problems have exponential running times with respect to the problem size, and are therefore likely not to be in $\mathcal{P}$.

The classes $\mathcal{P}$, $\mathcal{NP}$, and $\mathcal{NP}$-complete deal with *decision problems*, or problems where the answer is always "yes" or "no". At first, it may seem that the scheduling problems presented in this thesis would not fall into the category of decision problems because they are combinatorial; the answer to a makespan minimization problem is a time value along with a mapping of jobs to agents. However, it has been shown that almost any type of optimization problem that can produce a Turing Machine output, including combinatorial scheduling problems, can be converted into a decision problem [8]. For example, consider a general problem that outputs n bits of information given an input (such as a scheduling problem where each bit corresponds to whether a job is assigned to a particular agent). This combinatorial problem can be converted into a set of decision problems where each decision problem corresponds to a single bit position in the answer, indicating whether that bit is 1 or 0. Therefore, solving a decision version of a problem cannot be more than n times harder than solving the combinatorial version of the problem.

An $\mathcal{NP}$-hard problem is a combinatorial problem whose decision version is $\mathcal{NP}$-complete. It has been proven that most classes of scheduling problems, even greatly simplified ones, belong to the $\mathcal{NP}$-hard class [12,17]. Because there are currently no polynomial-time algorithms known to solve most of these scheduling problems to optimality, many researchers have instead focused on developing heuristics or approximation techniques that lead to sub-optimal answers but run in polynomial time. However, most of these heuristics only look at simpler versions and special cases of the multi-agent scheduling problem, and don't include a number of various parameters needed to accurately model many classes of real-world problems. Some examples of these heuristics and their limitations are given in Section 2.3.

The algorithm presented in this thesis uses Mixed Integer-Linear Programming (MILP) techniques to solve the most complicated classifications of the multiprocessor scheduling problem. Using MILP techniques has several advantages. First of all, MILP produces exact optimal solutions instead of approximate ones. Second, being a general-purpose

optimization method, software tools such as Dash Optimization's XpressMP™ are available to efficiently solve an MILP problem once it has been formulated (see Appendix B). The disadvantage of using MILP techniques is that they are $\mathcal{NP}$–hard, and therefore may be infeasible to use for solving larger scheduling problems. Chapter 5 explores the limits of the problem input size that the MILP scheduling algorithm can solve in a reasonable amount of time.

2.2 Multiprocessor Scheduling Theory

Much of the work done for this thesis is based on multiprocessor scheduling theory, which is concerned with optimally allocating sets of agents (or machines) to complete a set of tasks over time. Many variations of scheduling problems exist, as do techniques for solving them. Some of these techniques may consider cases where agents are a scarce resource, while others assume that there are an infinite number of available agents. Other techniques may assume that execution time for each job on each agent is constant [7], or that communication time between agents is zero [4]. Many of these algorithms are heuristics tailored to specific instances of scheduling problems, but it has been shown that no single one of these approximation techniques works best for every problem instance [16]. While many of these techniques refer to the computing entities that the jobs are assigned to as "machines" or "processors", this thesis refers to them as "agents", in order to reflect the fact that scheduling problems are applicable to a wide variety of fields, and that the "agents" may represent people or organizations as well as machines.

2.2.1 Classification of Scheduling Problems

The most general case of multiprocessor scheduling problems studied in this thesis involves a set {J} of n jobs $j_1 \ldots j_n$, and a set {A} of m agents $a_1 \ldots a_m$, each of which can perform a subset of the jobs. $E(j_i,a_i)$ represents the execution time for a job j_i on agent a_i, and $C(j_i,a_i,a_j)$ represents the communication delay for sending the results of job j_i from agent a_i to agent a_j.

Lawler, Lenstra, and Kan [15] developed a classification technique for modeling scheduling problems using notation of the form **α | β | γ**, three variables that specify different aspects inherent in scheduling problems.

α represents the agent environment, which can consist of a single agent or a set of multiple agents acting in parallel. $\alpha \in \{P,Q,R\}$. When $\alpha = \{P,Q,R\}m$, this means that the problem instance is dealing with exactly *m* agents.

> $\alpha =$ P: *Identical parallel agents*. Each agent runs at the same speed, therefore job processing time is independent of whichever agent is chosen. $E(j_i,a_i) = f(j_i)$.
>
> $\alpha =$ Q: *Uniform parallel agents*. Each agent may run at a distinct speed, so the job processing time is uniformly related to the speed of each agent. $E(j_i,a_i) = f(a_i)$.
>
> $\alpha =$ R: *Unrelated parallel agents*. A job's processing time is an arbitrary function of the job chosen and the agent chosen to run it. $E(j_i,a_i) = f(j_i,a_i)$.

β represents the job characteristics, such as precedence constraints and execution costs and consists of the subset $\{\beta_1, \beta_2, \beta_3\}$. $\beta_1 \in \{ø,PREC,TREE\}$, $\beta_2 \in \{ø,E(j)=1\}$,and $\beta_3 \in \{ø,r_j\}$.

> $\beta_1 = ø$: Jobs are not precedence constrained, and may be executed in any order.
>
> $\beta_1 =$ TREE: Precedence constraints for jobs have a rooted tree structure.
>
> $\beta_1 =$ PREC: Precedence constraints may exist among jobs, and the job precedence tree may form any type of directed, acyclic graph (DAG).
>
> $\beta_2 = E(j)=1$: All jobs have execution cost of 1 time unit.
>
> $\beta_2 = ø$: Jobs may have distinct, arbitrary execution costs

$\beta_3 = r_j$: Jobs may have set release times before which they may not be processed, regardless of whether their predecessors have been completed or not.

$\beta_3 = \emptyset$: Jobs have no release times and may be scheduled as soon as their predecessors are completed.

γ represents optimality criteria, or the type of objective function to be minimized or maximized for the particular scheduling problem. $\gamma \in \{C_{max}, \Sigma C_i\}$.

$\gamma = C_{max}$: The objective is to minimize the maximum cost (usually completion time or makespan) of the schedule.

$\gamma = \Sigma C_i$: The objective is to minimize the sum of a set of parameters across all jobs, i.e. the sum of the weighted completion times for each job.

McDowell [17] in his thesis suggests an additional notation that represents communication delay between agents, $\delta \in \{k,P,J,JP\}$:

$\delta = k$: Communication cost is a constant (or 0 if there is no communication cost) between all pairs of agents for each job. $C(j_i,p_i,p_j) = k$.

$\delta = P$: Communication cost is a function only of the pair of agents chosen, but independent of the job (the amount of data transmitted is constant for all jobs, but other factors, such as the available bandwidth between agents may change). $C(j_i,p_i,p_j) = f(p_i,p_j)$.

$\delta = J$: Communication cost is a function only of the job, and is independent of the agents between which the results of the job are transmitted. $C(j_i,p_i,p_j) = f(j_i)$.

δ = JP: Communication cost is dependent not only on the agents communicating the results of the job, but also on the job itself. $C(j_i,p_i,p_j) = f(j_i,p_i,p_j)$.

The class of scheduling problems addressed in this thesis is defined as:

R|PREC,r_j|C_{max}|JP: This problem is to schedule a set of generally precedence-related jobs on an arbitrary number of agents, with the objective being to minimize the schedule's makespan. Each job has a distinct number of processing units and each agent has a set amount of processing power, which corresponds to job execution time being a function of both the job chosen and the agent chosen to execute it. There may also be distinct communication delay between each pair of agents, and that delay may also be a function of the job whose results are being communicated between the agents. Jobs may also have distinct release times.

This is the most complex class of scheduling problems defined by Lawler, Lenstra, Kan, and McDowell [2,6], and it is also the most difficult to solve. This type of problem can easily be proven to be $\mathcal{NP}$–hard, based on the fact that most simpler classifications of scheduling problems such as P| |C_{max}|0 have been proven to be $\mathcal{NP}$–hard also [15]. While the MILP scheduling algorithm presented in this thesis runs in exponential time, it is not only capable of solving the R|PREC,r_j|C_{max}|JP problem, but it can also solve every other class of scheduling problems defined above.

2.3 Other Solution Techniques

Algorithms to solve scheduling problems fall into two different categories: *static* and *dynamic*. Static, or off-line approaches assume that all information about the scheduling problem is known *a priori*, and doesn't change as the schedule is being computed or carried out. Using this approach, the complete solution to the schedule is computed before the schedule actually begins. Static scheduling is most useful for cases when the algorithm computing the schedule is computationally intensive, or if the schedule is going to be run multiple times with the same agents and jobs. Dynamic, or on-line

scheduling, on the other hand, is more flexible because its computations take place as the current schedule is executing, and it can usually take into account unexpected changes in the agent organization or job hierarchy. Because the objective of this thesis is to provide exact optimal solutions to scheduling problems, the algorithm presented is static, and assumes that all information is known *a priori* or can be computed beforehand.

Both static and dynamic scheduling problems have been proven to be $\mathcal{NP}$ –hard and are therefore computationally intensive for larger problems. As a result, most scheduling techniques rely on faster but less accurate heuristics and approximations. Below is a sample and analysis of some of these existing solution techniques.

2.3.1 List Scheduling

List scheduling is a stochastic approximation technique that was first examined in detail in 1966 by Ron Graham [9], who looked at the following classification of scheduling problems: $P|PREC|C_{max}|0$. This formulation assumes that each agent is identical, and therefore execution delay is a function of the job only. Also, there is assumed to be no communication delay between agents.

In list scheduling, each job is first assigned a *priority level*, and is placed into a list that is ordered by this level. Also, a *global time* is defined that starts at 0. The algorithm then runs according to the following steps:

1) The job tree is analyzed and any jobs whose predecessors have been completed or which have no predecessors are designated as *executable*.
2) As soon as an agent becomes *available* (when it has completed all of its previous jobs and is idle), the highest executable job in the priority list is assigned to it.
3) That job is deleted from the priority list.
4) Steps 1-3 are repeated until there are either no more executable jobs or no more available agents. The *global time* is then incremented until either a job or an agent becomes available.

Graham proved that this heuristic produces a schedule whose makespan is no more than 2 times the length of the makespan of the optimal schedule [9]. It should be noted, however, that list scheduling does produce optimal solutions for some very simple classes of problems [17]. Some examples of these simple classes of problems include:

- $P|E(j)=1|C_{max}|0$. When using an arbitrary number of parallel identical agents with no communication delay or job precedence constraints and an execution delay fixed at 1 unit for all jobs, simply scheduling each task on an agent as soon as that agent is free automatically minimizes the makespan.

- $P|TREE,E(j)=1|C_{max}|0$. This is the same formulation as the previous problem, except that jobs are now precedence-constrained in a rooted tree structure. List scheduling can be used again, this time giving preference to the jobs furthest from the root of the tree to produce a schedule with an optimal makespan.

Many variations of the List scheduling algorithm exist, and they differ mainly in the way in which they assign priority levels to jobs. In general, the strategy is to find the most critical jobs and place them higher on the priority list. For example, the *Modified Critical Path* algorithm [22] assigns priorities to jobs according to the how far they are from the root of the tree, therefore giving preference to jobs that are likely to be on the *critical path*. The critical path is defined as the longest single chain of jobs in the precedence tree, and its length is a lower bound on the optimal schedule that can be produced.

One of the problems with regular list-based scheduling algorithms is that they don't usually take inter-agent communication delay into account, as pointed out by Kohler [13] and Sih [21]. List scheduling is a greedy algorithm that attempts to assign all executable tasks to processors before advancing the global time to the next step. With large communication delay between agents, however, this can often result in sub-optimal makespans. For example, in a situation where there is large communication delay and short execution time, a better makespan may be achieved by scheduling all tasks on one agent rather than trying to distribute the computation among other agents. McDowell

also proved that adding even constant communication delay to the simplest types of scheduling problems ($P|E(j)=1|C_{max}|K$) makes them NP –hard [17].

2.3.2 Dynamic Programming

Dynamic programming is a technique that models a problem as a set of sequential decisions and a set of *states* that are reached as a result of these decisions. The objective is to define an *optimal policy*, based on Bellman's Principle of Optimality [2]:

> *An optimal policy has the property that, whatever the initial state and the initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

In other words, an optimal policy is a set of rules that are guaranteed to produce an optimal solution regardless of the starting state.

Engels, Feldman, Karger, and Ruhl [7] developed an optimal scheduling algorithm using dynamic programming for $Pm|TREE,E(j)=1|C_{max}|JP$ problems which is polynomial with respect to the number of jobs, under the constraint that the communication delay is bounded by a constant *D*. Using their algorithm each state is defined by the set {A,P} where A is the set of executable jobs, and P is a "window" showing the last *D* steps of the schedule. At each state, jobs are considered to be "active", "waiting", "scheduled", or "set aside", and the algorithm essentially finds the shortest path between the states. There are $O(n^{3mD+2m-1})$ states, which means that the algorithm is polynomial with respect to the number of jobs *n*, but it grows exponentially with *D* and the number of processors *m*.

A variation of the List scheduling algorithm, called *Dynamic Critical Path* scheduling [14], computes a new critical path at every step of the list-scheduling algorithm and rearranges the priority levels of jobs. It then selects a suitable agent for each executable job based on the predicted start time of the "critical child" of that job.

Dynamic programming is a useful technique for conceptualizing scheduling problems and setting them up for computation, but there is no single method used to solve all instances of them. Dynamic programming may be used to develop polynomial-time approximation algorithms or optimal solutions for very simple scheduling classifications, but to solve schedules to optimality it still runs in exponential time which is no better than MILP techniques.

2.4 Mathematical Programming

The algorithm presented in this thesis uses Mathematical Programming (MP) techniques to solve a variety of classifications of scheduling problems. Mathematical Programming is a very useful tool for solving complex problems such as these that can be modeled as an objective function with a set of mathematical constraints. A wide variety of research disciplines currently use MP techniques to aid in complicated decision-making, from management science to engineering to the military. Because Mathematical Programming is concerned with finding optimal ways of using limited resources to achieve an objective, it is often simply referred to as *optimization*.

In Mathematical Programming, the objective function to be optimized is of the form

$$\text{MIN (or MAX):} \qquad f(X_1, X_{2, \ldots,} X_n)$$

where $\{X_1, X_2, \ldots, X_n\}$ is the set of *decision variables* that characterizes the problem. These variables can represent anything from the number of certain types of products that need to be produced by a factory to starting times for various jobs in a schedule. The objective is usually to minimize or maximize a linear function of these variables, subject to mathematical constraints of the form:

$$f_1(X_1, X_{2, \dots, } X_n) \leq b_1$$
$$\dots.$$
$$f_k(X_1, X_{2, \dots, } X_n) \geq b_k$$
$$\dots.$$
$$f_m(X_1, X_{2, \dots, } X_n) = b_m$$

$\{b_1, b_2, \dots, b_m\}$ is a set of constants and $\{f_1, f_2, \dots, f_m\}$ is a set of functions with parameters $\{X_1, X_{2, \dots, } X_n\}$ that represent either an equality or inequality constraint. These functions may be linear or non-linear, and these two types of problems have different optimization techniques for solving them. This thesis focuses on linear programming techniques only, but an overview of non-linear programming can be found in [3].

2.4.1 Linear Programming

Linear Programming (LP) problems are MP formulations where the objective and each constraint are a linear function of $\{X_1, X_{2, \dots, } X_n\}$. Therefore, an LP formulation would look like this:

MIN (or MAX):	$c_1X_1 + c_2X_2 + \dots + c_nX_n$
Subject to:	$a_{11}X_1 + a_{12}X_2 + \dots + a_{1n}X_n \leq b_1$
	$\dots.$
	$a_{k1}X_1 + a_{k2}X_2 + \dots + a_{kn}X_n \geq b_k$
	$\dots.$
	$a_{m1}X_1 + a_{m2}X_2 + \dots + a_{mn}X_n = b_m$

2.4.1.1 Example LP Problem

Suppose a company makes two products, P1 and P2, and has two factories that produce these products. Factory 1 produces 4.5 kg of P1 per day, and 4 kg of P2 per day. Factory 2 produces 7 kg of P1 per day but only 3 kg of P2 per day. Product P1 can be sold for \$2/kg, and product P2 can be sold for \$3/kg. Assuming that both factories operate at a

constant rate, this means that operating at full capacity the total *processing time* necessary for Factory 1 to produce X units of P1 and Y units of P2 would be $\frac{2}{9}X + \frac{1}{4}Y$. Likewise, the total time needed to produce X units of P1 and Y units of P2 for Factory 2 would be $\frac{1}{7}X + \frac{1}{3}Y$. Because it is not possible for the factories to produce negative quantities of each product, we specify that $X \geq 0$ and $Y \geq 0$. Since we want to show that the daily processing requirements do not exceed production capacity at each factory, the problem can be modeled as:

Minimize:	$-2X - 3Y$
Subject to:	$\frac{2}{9}X + \frac{1}{4}Y \leq 1$
	$\frac{1}{7}X + \frac{1}{3}Y \leq 1$
	$X, Y \geq 0$

The linear constraints and optimal objective function value are modeled in Figure 2-1, below:

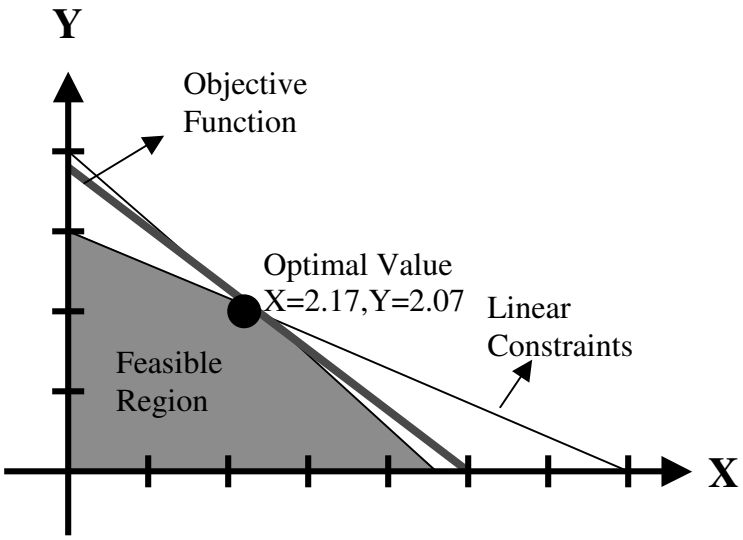

Figure 2-1: A Linear Programming problem and solution

In this figure, the shaded region represents the *feasible region*, or the set of points which, when chosen, satisfy all of the constraints specified in the problem. As seen in the graph, the feasible region is a polygon whose edges correspond to the linear constraints. The objective function line is moved outward (increasing its X and Y intercept values) until it is tangent to the feasible region at one of its *corner points*. This point represents the optimal value for X and Y, the point at which the value of the objective function is maximized.

Certain LP problem formulations may have multiple or infinitely many optimal solutions, such as when the objective function is parallel to one of the linear constraints. An LP problem may also have an unbounded optimal solution, or it may be considered *infeasible* when the constraints do not allow for any solution that satisfies all of them.

2.4.1.2 Simplex Method

The Simplex Method is a widely used computational tool for solving LP problems. It is a fairly simple algorithm, and it has been proven to find optimal solutions (if there are any) to all instances of LP problems. In addition, as a byproduct it provides useful data, such as how an optimal solution varies with respect to the problem data and reports when a solution is unbounded or infeasible. The Simplex Method works because a single optimal solution to any LP problem always has to occur on a corner point of the feasible region [3]. Therefore, only these points need to be considered in the search for an optimal solution.

The Simplex Method works by first converting all constraints that are inequalities into equalities. It does this by defining an extra positive-valued *slack variable* for each of these constraints, subtracting it from each $\geq$ constraint and adding it to each $\leq$ constraint. Using the previous example, the resulting LP formulation with slack variables S_1 and S_2 would look like this:

Minimize: $-2X - 3Y$

Subject to: $\frac{2}{9}X + \frac{1}{4}Y + S_1 = 1$

$\frac{1}{7}X + \frac{1}{3}Y + S_2 = 1$

$X, Y, S_1, S_2 \geq 0$

The constraints are now a series of 2 equations with 4 variables. The Simplex Method then proceeds to set any 2 of these variables to 0 (or their lower bound), and solves for the other 2 variables. This solution is guaranteed to be on one of the corner points of the feasible region, and is called a *basic feasible solution*.

Rather than enumerating all basic feasible solutions and choosing the best one, the Simplex Method begins instead at a random corner point, and moves to an adjacent corner point. It does this only if the solution there is better than the previous one until it can find no better solution. By using this "greedy" approach, it is possible to find the optimal solution using fewer iterations than would otherwise be needed.

2.4.2 Mixed Integer-Linear Programming

Many MP problems exist where it is necessary to restrict the decision variables to integer or binary values. Examples include cases where the decision variable represents a non-fractional entity such as people or bicycles, or where a decision variable is needed to model a logical statement (such as whether or not to assign task A to agent B). These problems are called Mixed Integer-Linear Programming (MILP) problems, and are often much harder to solve than LP problems. This is because instead of having feasible solution points at the easily computed corners of the feasible region, they are instead usually internal and more difficult to locate. For example, constraining X and Y from the previous LP formulation to have integer values, the feasible solution points are shown in Figure 2-2:

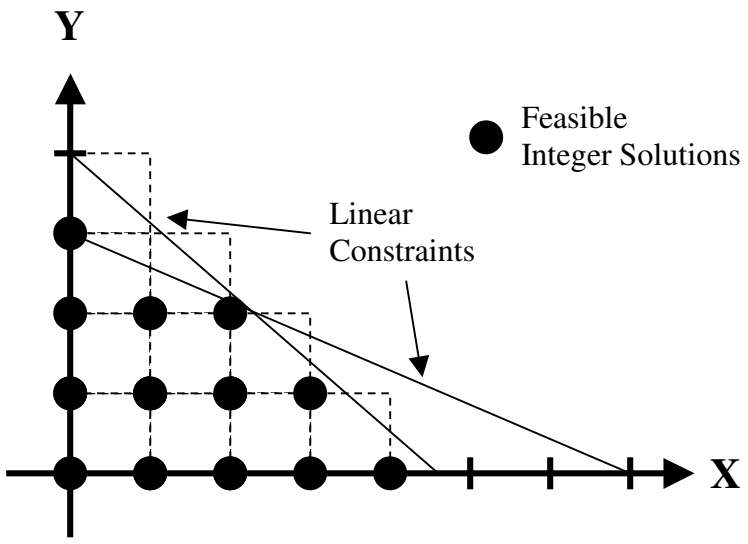

Figure 2-2: A Mixed Integer-Linear Programming problem showing all feasible integer solutions

The first step in solving a MILP problem such as this is to solve the *linear relaxation* of that problem. This simply means removing the constraints that any decision variables have integer values and solving the resulting LP problem using an algorithm such as the Simplex Method. The result is one of the following outcomes:

a) The LP problem is infeasible, so the MILP problem is also infeasible.
b) The LP is unbounded and is probably not a well-posed problem.
c) The LP has a feasible solution and all integrality constraints are satisfied, so the MILP has also been solved.
d) The LP has a feasible solution, but not all of the decision variables have integer values.

No further work is necessary for outcomes a), b), and c), but more work is required to find the optimal solution for case d). In most cases the Branch and Bound technique is used in this case to find the integer solution, as explained in the next section.

2.4.2.1 Branch and Bound

The idea behind Branch and Bound is to divide the feasible region into sub-problems according to integer values and to use LP techniques recursively to tighten upper and

lower bounds around the optimal solution until they meet or until an LP relaxation results in an integer solution. After the initial LP relaxation of the problem has been solved, the resulting decision variable values are analyzed. A non-integer-valued variable is chosen, and then two sub-problem "branches" are created which are the same as the previous LP problem, except that the variable chosen is constrained to be greater than or equal to its upper integer bound (rounding the decimal solution up) in one case, and less than or equal to its lower integer bound (rounding the decimal solution down) in the other case. Each of these sub-problems is recursively solved and "branched" again using LP techniques and choosing a different variable, until either:

a) The LP problem of the branch is infeasible,
b) The solution to the LP problem is integer-valued, or
c) The solution to the LP problem is worse than a previously computed solution.

In a maximization problem, the optimal objective function value of the LP relaxation is always an upper bound on the optimal integer solution. Likewise, any integer feasible point found is always a lower bound on the optimal integer solution [3]. These values are used to update the upper and lower bounds at every step, narrowing down the possible solutions until the optimal integer solution is found.

Branch and Bound works better for MILP problems than completely enumerating every possible integer solution, but its running time may still grow exponentially with respect to the problem size in some cases. The total running time of Branch and Bound depends mostly on how well the LP relaxation approximates the *convex hull* of the feasible integer set. The convex hull of a feasible set of points S of a MILP is defined as the smallest polyhedron that contains S. Mathematically, the convex hull for S is defined as:

$$\mathrm{C} = \left\{ \sum_{i=1}^{K} \lambda_i x^i \;\middle|\; x^i \in S, \lambda_i \geq 0, \sum_i \lambda_i = 1 \right\} \tag{2-1}$$

The closer the LP relaxation is to this bound, the less time it takes for Branch and Bound to find the optimal integer solution.

-This Page Intentionally Left Blank-

Chapter 3

Multi-Agent Problem Solving

In order to provide a context for the multi-agent scheduling algorithm presented in this thesis, this chapter presents a general problem-solving framework that explores job decomposition, optimal task allocation, and solution formulation. In addition, examples from various problem domains are used to illustrate each of these techniques.

3.1 Problem Solving Framework

The process of multi-agent planning and problem solving can be modeled by a 4-phase closed-loop command and control architecture [10], shown in Figure 3-1(a). This architecture contains the following sub-functions: Plan, Execute, Monitor, and Diagnose. The arrows represent the flow of data between each of these functions, which forms a continuous feedback mechanism.

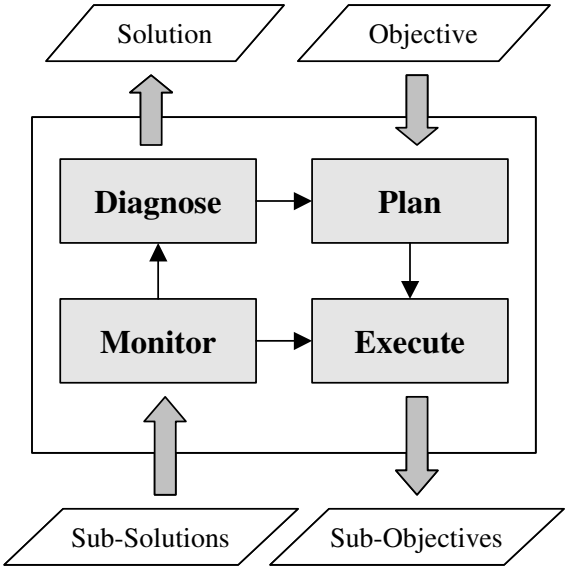

Figure 3-1(a): Problem solving framework represented by a closed-loop command and control architecture.

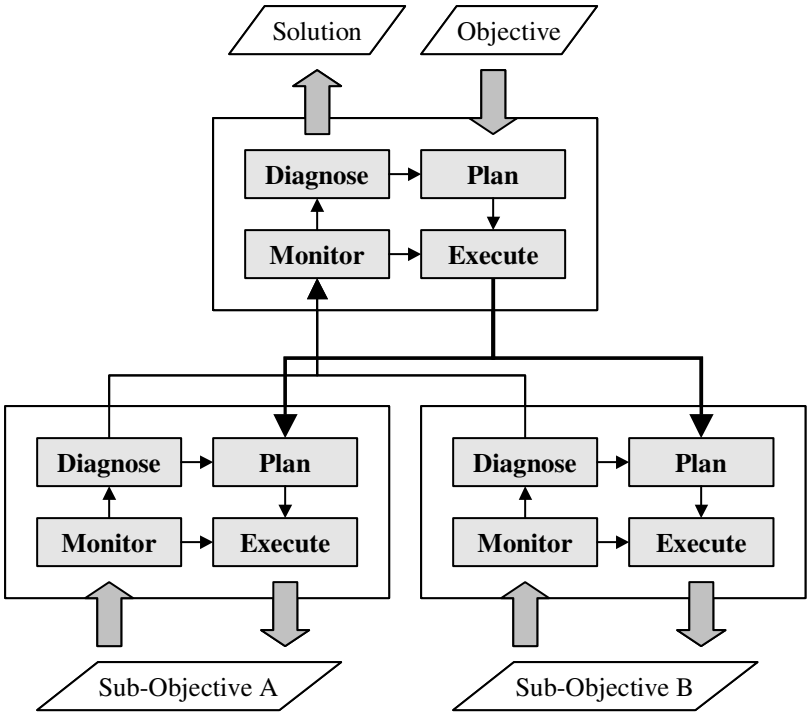

Figure 3-1(b): Hierarchical application of the problem-solving framework in (a).

Figure 3-1(b) shows a hierarchical application of this closed-loop architecture, where the top level can be seen as a controller that decomposes a problem into two sub-problems, and sends these sub-problems to be planned, executed, monitored, and diagnosed in a similar fashion. This hierarchy can be extended to multiple levels depending on how a problem is decomposed.

Each of the functions in Figure 3-1(a) can be further decomposed to address the more specific problem of multi-agent planning and scheduling, as shown in Figure 3-2. In this figure, the specific tasks of job decomposition, job/agent mapping, optimal schedule computation, task allocation, and solution formulation are represented within this higher-level architecture.

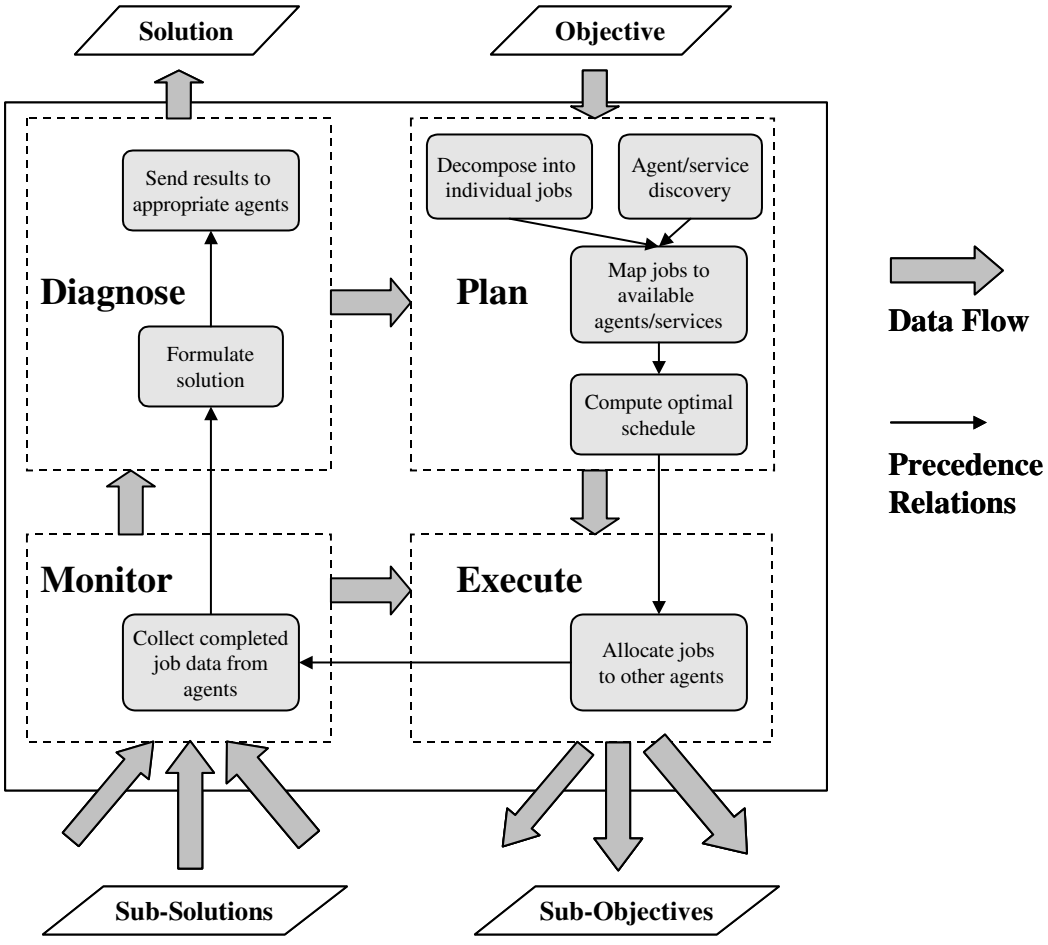

Figure 3-2: Multi-agent planning and scheduling process in the context of the closed-loop command and control architecture.

As shown in Figure 3-2, these steps consist of:

- Formulating or receiving from another source a global objective to be achieved.
- Structuring this objective in a form where it can be easily decomposed into a partially-ordered set of sub-problems or jobs.
- Surveying the environment for available agents and services that may be used to complete these jobs.
- Mapping jobs to available services or sets of services that are capable of completing them.
- Determining the allocation of jobs to agents, such that the resulting schedule is optimized according to user-defined parameters.
- Formulating the solution based on results from all participating agents, and forwarding this solution to the appropriate agents.

It is important to note that the small arrows between the specific tasks in Figure 3-2 represent *precedence relations*, while the large arrows between the high-level functions represent continuous *data flow* and feedback between these phases. It should be noted, however, that the feedback provided by the data flow could represent precedence relations at a higher level itself in the context of this architecture, as one phase may not be allowed to begin its next cycle until it has received data from another phase. In this case, to represent this high-level data flow in the form of precedence constraints it is necessary to view the feedback graph in Figure 3-1(a) as a continuous cycle that runs in multiple iterations. Therefore, in order to show precedence relations between the 4 main functions each iteration must be shown separately, with the feedback arrows pointing to the next *iteration* of each function instead of back to the original version. Figure 3-3 shows an example of these precedence relations between a 2-level deep hierarchy of command and control nodes; it should be noted that the four functions in each node are shown in a different orientation to provide a clearer picture of the precedence constraints between each iteration. These high-level precedence relations in the command and control architecture are explored further in a scheduling application example in Chapter 5.

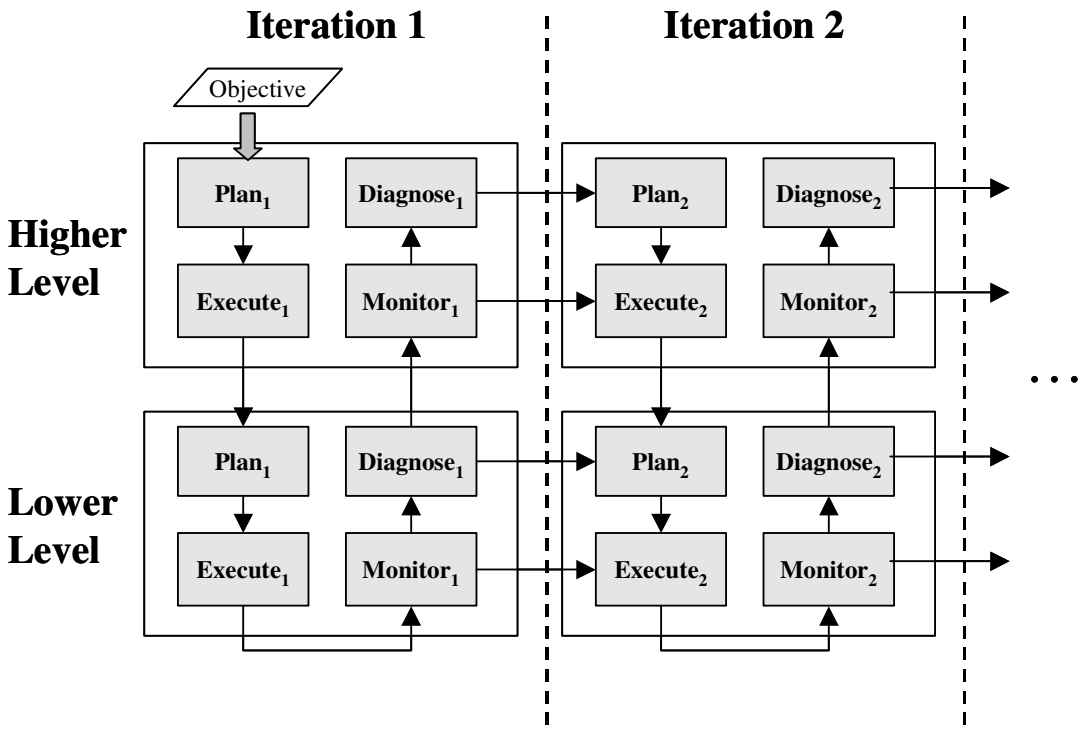

Figure 3-3: Precedence relations between each main function in the closed-loop hierarchy, shown in multiple iterations to represent feedback between functions at each stage.

The following sections describe the planning process from Figure 3-2 in detail, including finding the allocation of jobs to agents such that the resulting schedule is optimized, which is the main focus of this thesis.

3.2 Problem Formulation and Decomposition

In the multi-agent framework that we are considering, there is a set of heterogeneous agents {A} in a network, each with a different set of available abilities and services. A single agent is given an objective to complete, possibly from another agent, and it wishes to take advantage of the resources provided by these other agents in the network to complete the objective more efficiently. The planning agent's first step is to decompose its objective into a set of jobs {J} that can be allocated to other agents in the network and completed in parallel.

However, there are often many possible problem formulations for a given objective and choosing the best way to decompose the objective may depend on the structure of the agent organization and the number of different service types provided by these agents. In the scenario that this thesis addresses, the planning agent's main goal is to choose the job precedence graph that produces the schedule with the shortest possible makespan.

3.2.1 Functional Decomposition

Large-scale problems from many different domains can be decomposed by defining them using generalized functions of the form {F(x), G(y), H(z) …}. A computer program or systems engineering problem may be described this way where each function corresponds to an atomic operation or set of operations. These operations may be ordered relative to each other or may iterate through other operations recursively. As an example, consider the following control system for an unmanned autonomous vehicle (UAV) shown in Figure 3-4.

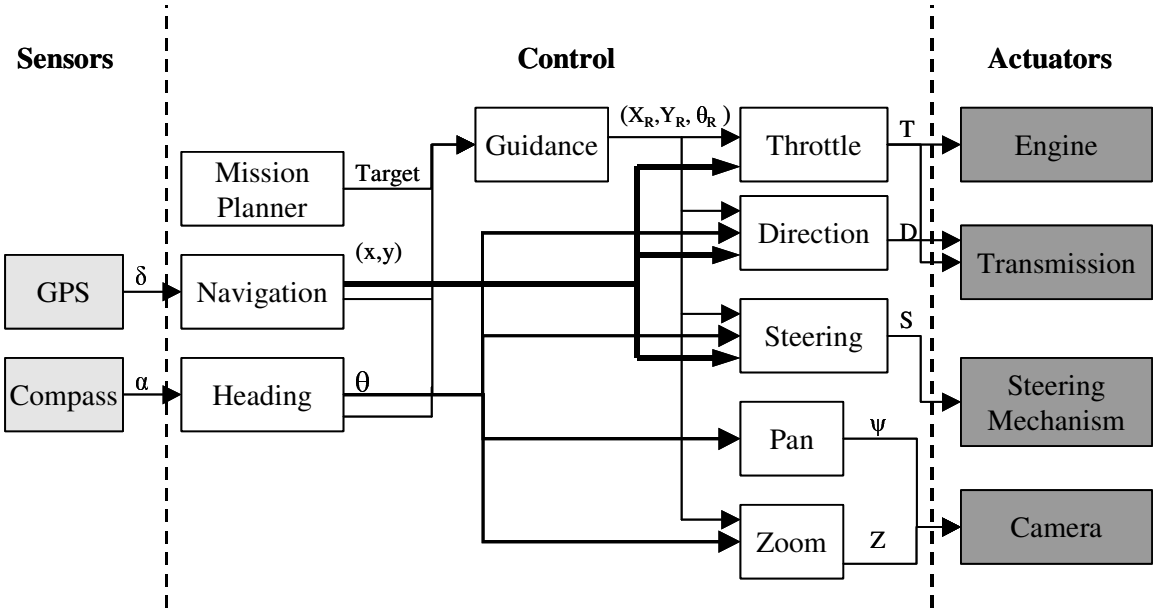

Figure 3-4: Internal control and navigation system of an autonomous vehicle.

Each node in the Control section of Figure 3-4 represents an internal procedure with a specific input and output, and the arrows between these procedures represent data dependencies. These processes can be seen as abstract functions, with the data being passed between them as the outputs and inputs to each function. Suppose the first letter of each process is used to name the function that it performs, i.e. Guidance output $(X_R Y_R, \theta_R) = G(target,x,y,\theta)$. In this case multiple data dependencies between processes can be represented using nested functions, i.e. the Camera Zoom setting would be defined as $Z = Z(H(\alpha),G(target,N(\delta),H(\alpha)))$.

These functions can be used to form different precedence-constrained chains of jobs that are used to achieve the same objective. For example, if the objective given to the UAV is to observe a certain point (x_1,y_1) on a map with its camera, it may choose to formulate and decompose a plan to complete this objective in a variety of ways. One possibility would be to change the camera's pan and zoom settings to point at the new target (x_1,y_1) without moving the vehicle itself, in which case the functions used to complete the objective could be modeled as

Camera: $\{Z(H(\alpha), G(target,N(\delta),H(\alpha))),P(H(\alpha))\}$.

An alternative approach to this may be to move the vehicle in a straight line to a new position such that the camera is pointed at the target without changing the actual camera settings. In this case the functional decomposition would instead be:

Transmission: $\{T(G(target,N(\delta),H(\alpha)),N(\delta)),\ D(G(target,N(\delta),H(\alpha)),N(\delta),H(\alpha))\}$

Engine: $\{T(G(target,N(\delta),H(\alpha)),N(\delta))\}$

These two functional decompositions and others may be used to achieve the same objective, and choosing which one to use may depend the efficiency and reliability of each method, as well as a number of external environmental factors such as terrain conditions and visibility.

3.2.2 Arithmetic Example

Arithmetic problems are another domain where multiple functional decompositions may be used to achieve a single objective, or in this case a solution to a compound problem. The operations add (x+y), subtract (x-y), and multiply (x*y) can be modeled by the

functions F(x,y), G(x,y), and H(x,y) respectively. Each of these functions obeys different mathematical laws, which allow the same arithmetic problem to be decomposed multiple ways. For example, the Commutative Law applies to multiplication and addition, and states that F(x,y) = F(y,x). Similarly, the Associative Law states that F(F(x,y),z) = F(x,F(y,z)). Finally, the Distributive Law states that H(x,F(y,z)) = F(H(x,y),H(x,z)).

Suppose that in a particular environment there are multiple agents that can provide three possible services: add, subtract, or multiply. After one agent formulates and conceptualizes an arithmetic problem, it may then decompose the problem and send the sub-problems to other agents to solve in parallel. Determining which agents to send the problem to may depend on a number of factors such as available bandwidth, physical proximity, and processing speed of each agent. Figure 3-5 shows this agent network.

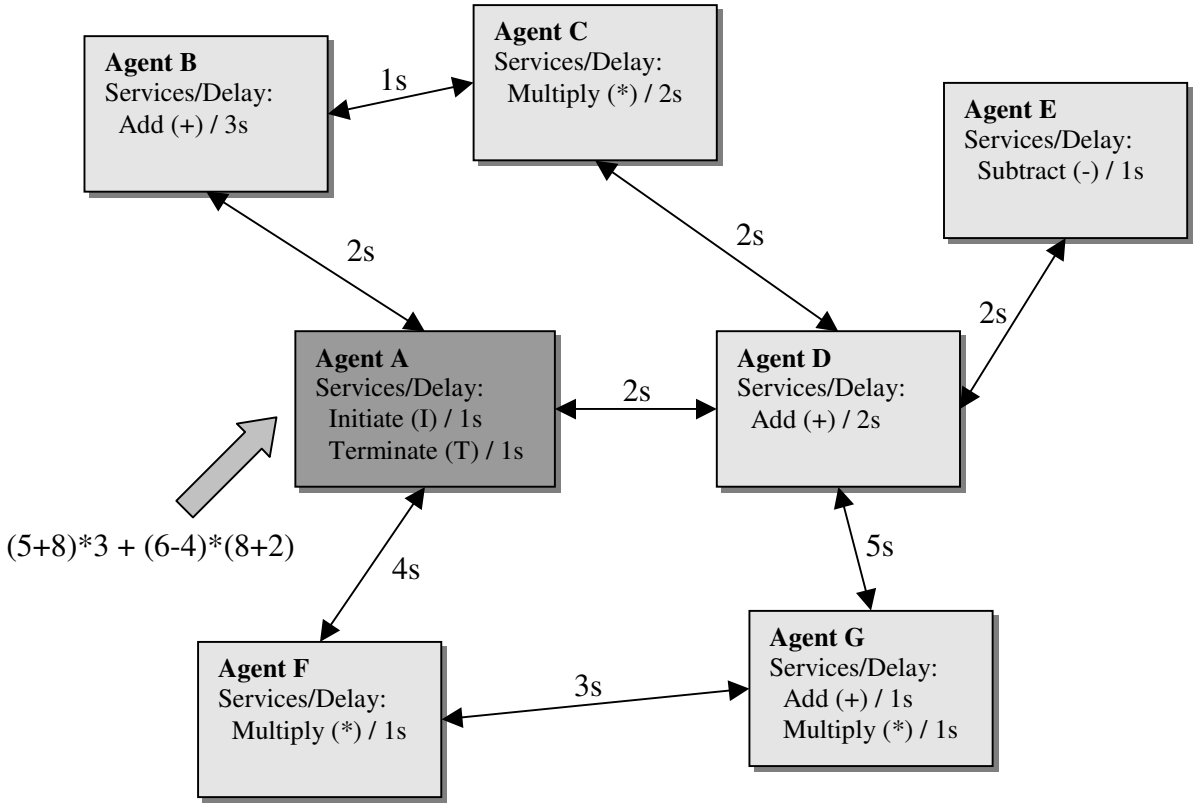

Figure 3-5: An agent network, indicating execution and communication delays and available services.

In Figure 3-5, suppose that Agent A is given the responsibility to compute the solution to the problem (5+8)*3 + (6-4)*(8+2). Agent A does not have the ability to add, subtract, or

multiply, so it determines that it must seek out the services of other agents to solve the problem. The agents are connected to one another according to the communication network shown in the figure, and each other agent provides one or more of these services. The weights on each edge in the graph represent the aggregate time needed to send data over that link. This model assumes that the results of each job contain the same amount of data, so communication delay is a function of available bandwidth between agents only, and is independent of the job data being transmitted. The services offered by each agent are shown, as well as the execution delay required to perform each service. Agent A's services, *Initiate* and *Terminate*, reflect the fact that data must be sent from Agent A to all other agents before the rest of the problem can begin, and that the final solution to the problem must be sent back to Agent A to be processed.

Using the functional notation defined above, Agent A's problem can be written as F(H(F(5,8),3),H(G(6,4),F(8,2))). Using the Distributive Law, however, the same problem can be decomposed as G(F(H(5,3),H(8,3),H(6,8),H(6,2)),F(H(4,8),H(4,2))), or (5*3 + 8*3+6*8 + 6*2) – (4*8 + 4*2). The job precedence graphs for these two equivalent formulations are shown below in Figure 3-6.

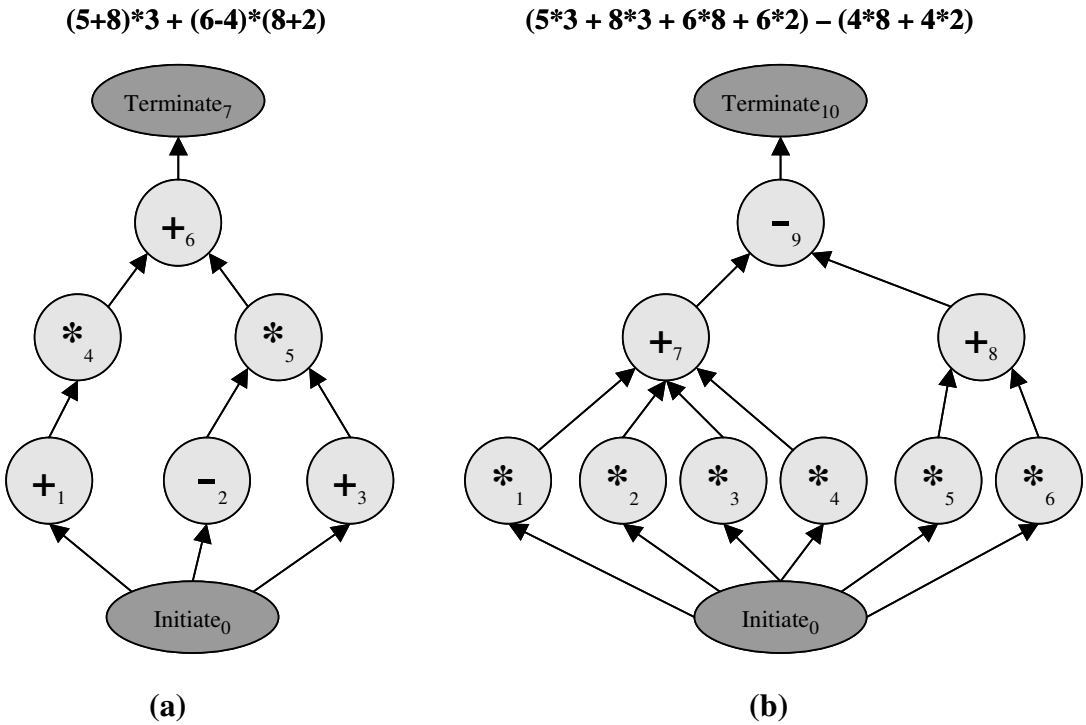

Figure 3-6: Two possible formulations and job precedence graphs for the same arithmetic problem.

In Figure 3-6, each node represents an individual job that must be completed and the arrows represent precedence constraints between the jobs. For example, in the Figure 3-6(a), job ($+_1$) must be completed and the data sent to the agent assigned to process job ($*_4$) before that job can begin execution. Each job is identified by the service needed to complete it (add, subtract, or multiply), and a unique identification number. Once a job precedence graph has been constructed, agents in the network must be identified that have the ability to complete each of these jobs using their provided services, and communication delays must be computed between each pair of agents (see Appendix C). In some cases the services used to complete a job may differ from agent to agent, and in other cases a combination of services may be required for more complicated jobs.

One strategy for determining the best problem decomposition is to look at each different formulation, calculate the optimal agent allocation for it, and choose the decomposition that produces the shortest schedule. However, since there are often many different possible job precedence graphs for any single objective and calculating schedules for each of them is an $\mathcal{NP}$-hard problem as shown in Chapter 2, it may not be feasible to calculate each one individually. In this case a heuristic technique may be used based on the properties of the agent/job ratio and the structure of the agent network. Some of these possible techniques include:

- Choosing the job precedence graph with the smallest number of nodes (minimizing the total number of operations to be performed). This would be the best choice if there were a small agent/job ratio or large communication delays, and the number of available agents and services was a limiting factor in the problem. It follows that in an environment with limited computational resources, the best problem formulation is usually the one with the fewest computations.
- Choosing the job precedence graph with the shortest depth (minimizing the critical path). This would be the best choice if there was a large agent/job ratio and communication delays between agents were small. With computational resources no longer being a limiting factor, the optimal problem formulation would be the one with the most jobs performed in parallel on different agents.

- Choosing the job precedence graph whose jobs most closely correspond to the number of corresponding service types offered by different agents in the network, or to the agent structure itself.

While none of these strategies is guaranteed to produce the optimal problem formulation, they are all likely to rule out many of the obviously sub-optimal job precedence graphs. It should be noted that there are other approaches to this problem where instead of formulating the problem to best fit the agent organization, the organization is modeled around the problem structure. As Rifkin [20] points out,

"…there is an inextricable inter-dependence between structure of the work and structure of the team. The structure of the work is optimal relative to the structure of the team, and the structure of the team is optimal relative to the particular structure of the work."

Because of this interdependency between the agent organization and the problem structure, there is currently no single best algorithm for determining either; heuristics are used instead to calculate one based upon the other. Future work on this topic may focus on developing optimization techniques for finding the best job decomposition for an agent network, but that is beyond the scope of this thesis.

3.3 Schedule Optimization and Task Allocation

After choosing an acceptable job decomposition and gathering all relevant information about the agent network, a problem-solving agent must determine which other agents to allocate tasks to. When computing an optimal schedule, an agent may be trying to achieve one or more of the following objectives:

- Minimize the makespan of the schedule
- Minimize the weighed completion time of each job
- Minimize the total number of agents used to solve the problem
- Minimize the amount of communication bandwidth used

- If there is a value associated with assigning a task to a particular agent, maximize the aggregate value
- If there is a cost (other than time) associated with assigning a task to a particular agent, minimize the aggregate cost

The first of these objectives, minimizing the schedule's makespan, is the subject of most existing research on multiprocessor and job-shop scheduling problems. The makespan is also the primary focus of the algorithm presented in this thesis, but Multiple Objective Linear Programming (MOLP) approaches are presented in Chapter 4 that can be used to optimize the scheduling problem with respect to other objectives as well.

Continuing the arithmetic example from the previous section, assume that Agent A chooses the smaller job precedence tree from Figure 3-6(a): (5+8)*3 + (6-4)*(8+2). Agent A then wants to compute the schedule with the optimal makespan based on the agent organization in Figure 3-5. There are many ways that Agent A can solve this problem. The simplest way would be to have Agent A start at the end of the job decomposition tree and work backwards, sending the top-level tasks to the closest available agents that have the ability to complete those tasks. Those agents would then divide up the next level of tasks and decide which other agents to allocate those tasks to in a recursive manner, until the bottom of the precedence tree is reached and the solution is sent back to Agent A. This technique, known as *hierarchical planning*, is useful in cases where jobs in the precedence graph are more general near the top and more specific near the bottom, and is explored in more detail in Section 3.4.

While this method requires the least amount of planning and works well in dynamic agent environments, it makes the assumption that each agent in the network is capable of making the same kinds of task allocation decisions that Agent A can. Also, hierarchical planning may not always result in a schedule with an optimal makespan, because each agent does not see the entire problem horizon and therefore may not know the best way to coordinate with other agents.

By planning "routes" for the various jobs to travel in the agent network in advance, a better schedule can be obtained. For example, Agent A can start at the bottom of the job precedence graph in Figure 3-6(a) and send those tasks out to the appropriate agents along with instructions for those agents to send the results of that computation to another agent, until the final result is sent back to Agent A. MILP techniques can be used to compute this scheduling problem to optimality and are described in detail in Chapter 4. Having an agent plan the schedule completely in advance makes it lose the ability to adapt its plan to dynamic changes in the network during execution of the schedule, but allows for the makespan of the schedule to be minimized. Figure 3-7 shows the optimal schedule for the arithmetic example described in this chapter, computed using XpressMP™ optimization software (see Appendix B). In this Gantt chart, the horizontal arrows represent job execution times, the diagonal arrows represent communication delay between agents, and each job is labeled by its unique identifying number defined in Figure 3-6(a).

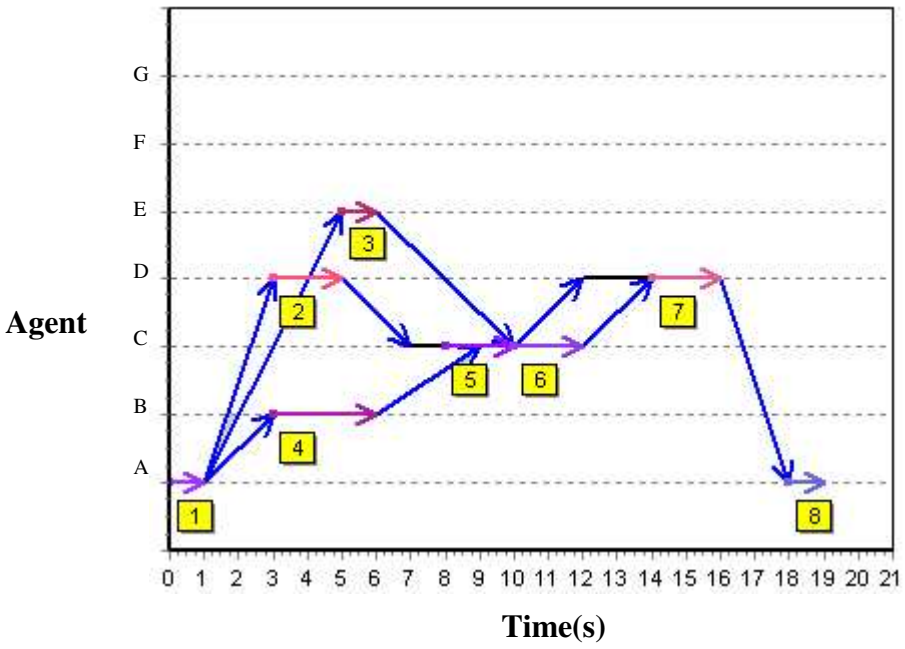

Figure 3-7: A Gantt chart showing the optimal schedule for the arithmetic problem described in this section.

This schedule is very sensitive to changes in both the agent network and the problem tree structure. For example, if the communication delay between Agents D and G is changed

from 5s to 1s (see Figure 3-5), the optimal makespan is reduced from 19s to 15s, and results in a completely different optimal job allocation, as shown in Figure 3-8:

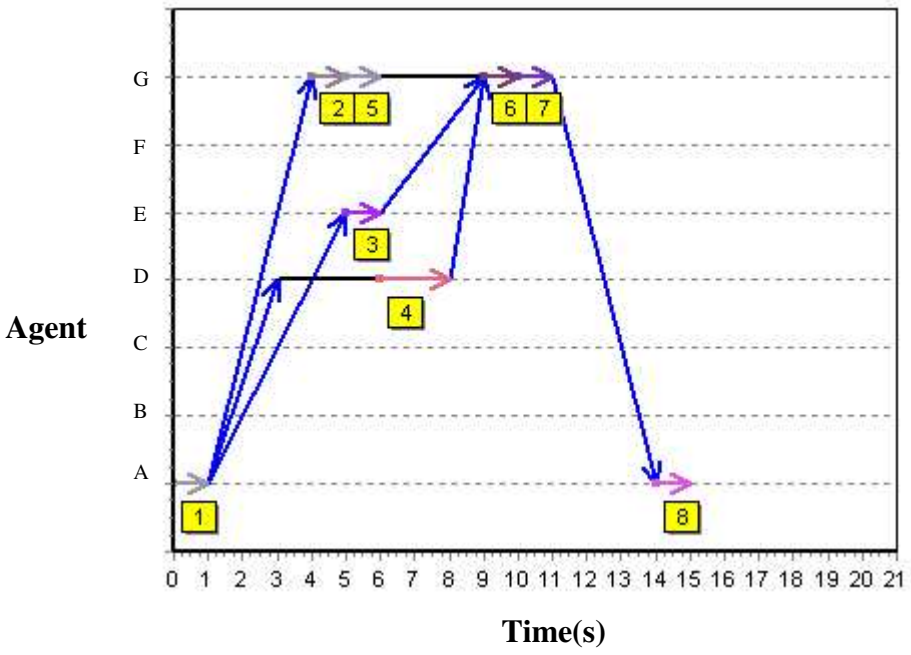

Figure 3-8: A Gantt chart showing the new optimal schedule computed after changing the communication delay between Agent D and Agent G from 5s to 1s.

The above example demonstrates the value of being able to re-compute the optimal makespan of a multi-agent schedule when small changes are made to the network or job structure. If this were a multiprocessor scheduling problem and the same schedule was to run successively with the same inputs each time, this recalculation would result in a 21% increase in efficiency.

3.4 Hierarchical Planning and Scheduling

As mentioned in Chapter 2, using MILP techniques for schedule optimization is an $\mathcal{NP}$-hard problem, and therefore is not feasible for solving large-scale scheduling problems. Fortunately, under certain circumstances these large problems can be broken down into smaller sets of scheduling problems that have a shorter problem horizon and can be solved recursively. This hierarchical planning technique mentioned in the previous

section can be useful in situations where there is a large problem that needs to be broken down in stages, or if agents are organized in a command hierarchy.

3.4.1 Communications Networks vs. Command Hierarchies

In some scenarios, an agent organization may be in the form of a *command hierarchy*. For example, the military uses command hierarchies to efficiently group agents with similar or complementary abilities together, to maintain flexible and robust control over large numbers of units, and to facilitate information flow throughout the organization.

It is important to distinguish between a command hierarchy and a communications network. A command hierarchy does not necessarily show all of the communication paths between agents; the arrows in a command hierarchy instead represent which agents have the *authority to allocate jobs* to other agents. In fact, each agent in this hierarchy may not even be aware of which agents are at different levels in it. For example, in Figure 3-9 the Unit Commander may not know the names and specific abilities of every individual soldier and Unmanned Autonomous Vehicle (UAV) the bottom of the command chain. Agents at each level of the military command hierarchy often only have the ability to share intelligence with other agents in the same functional group; thus each level of the command hierarchy can be seen as a unique communication network.

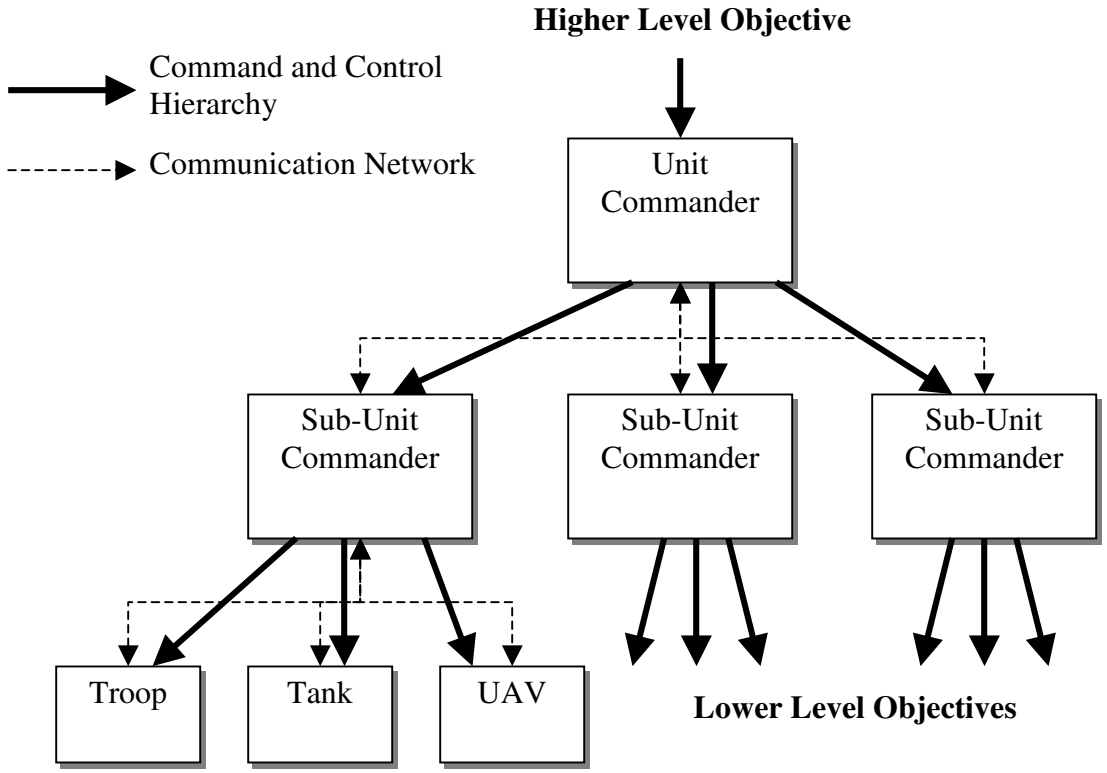

Figure 3-9: An example military command hierarchy, adapted from Baxter and Hepplewhite [1].

3.4.2 Job Precedence Graphs vs. Objective Decomposition Hierarchies

In the diagram in Figure 3-9, a single high-level objective is given to the Unit Commander, who splits the objective up into lower-level objectives and passes them to the Sub-Unit Commanders. The Sub-Unit Commanders then each take their objective, divide it up further, and pass those low-level objectives to their troops, tanks, and UAV's. The Unit Commander's objective is usually more general and is considered over a longer time horizon, while the low-level objectives are more short-term and specific. Higher-level objectives are said to have *large granularity*, while lower-level ones have *small granularity*.

However, in the job precedence graphs used in the MILP scheduling algorithm presented in this thesis, such as the arithmetic example presented earlier, jobs typically have very similar granularities. Each servive (add, subtract, multiply) is functionally independent

of the other services, i.e. no service consists of completing a set of any of the others, and they all have the similar time horizons. How does this hierarchical objective decomposition fit into our problem-solving model then?

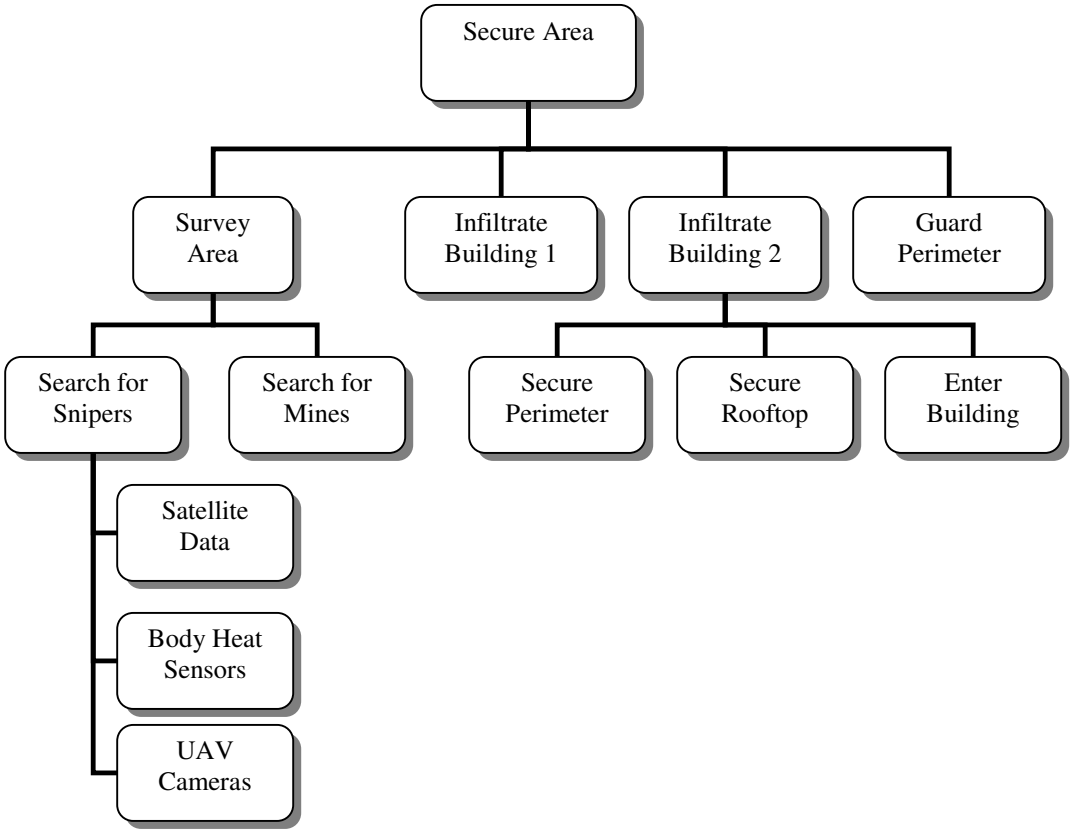

Figure 3-10: Part of an objective decomposition hierarchy for a Military Operations in Urban Terrain (MOUT) scenario.

Figure 3-9 shows an example of an objective decomposition hierarchy in a Military Operations in Urban Terrain (MOUT) scenario. Suppose that the top objective, "Survey Area", is assigned to the Unit Commander in Figure 3-9. The Unit Commander decomposes this general objective into a set of smaller, more specific objectives, and wants to find the optimal Sub-Unit Commanders to assign this next level of objectives to so that the tasks are completed in the shortest amount of time. In order to compute the optimal schedule for these jobs, two inputs are needed: an agent network with known communication and execution delays, and a partially-ordered job precedence graph.

The jobs at *each level* in Figure 3-10 have similar granularities and may have additional precedence constraints relative to each other that aren't shown in the figure. For example, consider the second level of this tree. The four jobs there may have to be completed in a partial order: The area should be surveyed for dangerous elements before entering any of the buildings, and only after both buildings are secure should another set of agents be set to guard the area perimeter. Therefore, these four objectives form their own job precedence graph:

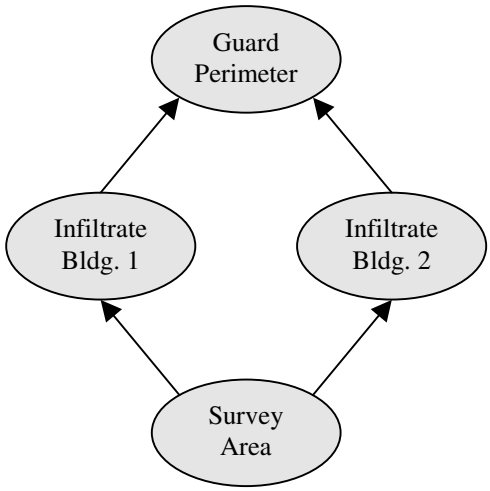

Figure 3-11: A job precedence graph for the second level of tasks from Figure 3-10.

Because the Sub-Unit Commanders belong to the same network, communication delays may be computed between each of them. Execution delays may be harder to calculate though, especially higher up in the decomposition hierarchy, because the jobs are still very high-level and not very specific. For example, if the Unit Commander asks a Sub-Unit Commander how long it would take to "survey the area", the best that the Sub-Unit Commander may be able to do is provide an estimate based on the decomposition and optimal allocation of that sub-objective to other agents. Likewise, the Sub-Unit Commander can only calculate its own optimal schedule after obtaining execution time estimates from the agents directly below it. Eventually, these requests for execution times reach the bottom of the agent hierarchy and the jobs are specific enough that the agents there can determine their own execution times. The data then flows back up the hierarchy, enabling schedules to be optimized at each successive level until the Unit

Commander has enough time estimates to compute its own optimal schedule at the highest level. Figure 3-12 shows how the problem-solving model described in Section 3.1 may be applied recursively in this manner to optimize task allocations at each level of an objective decomposition hierarchy.

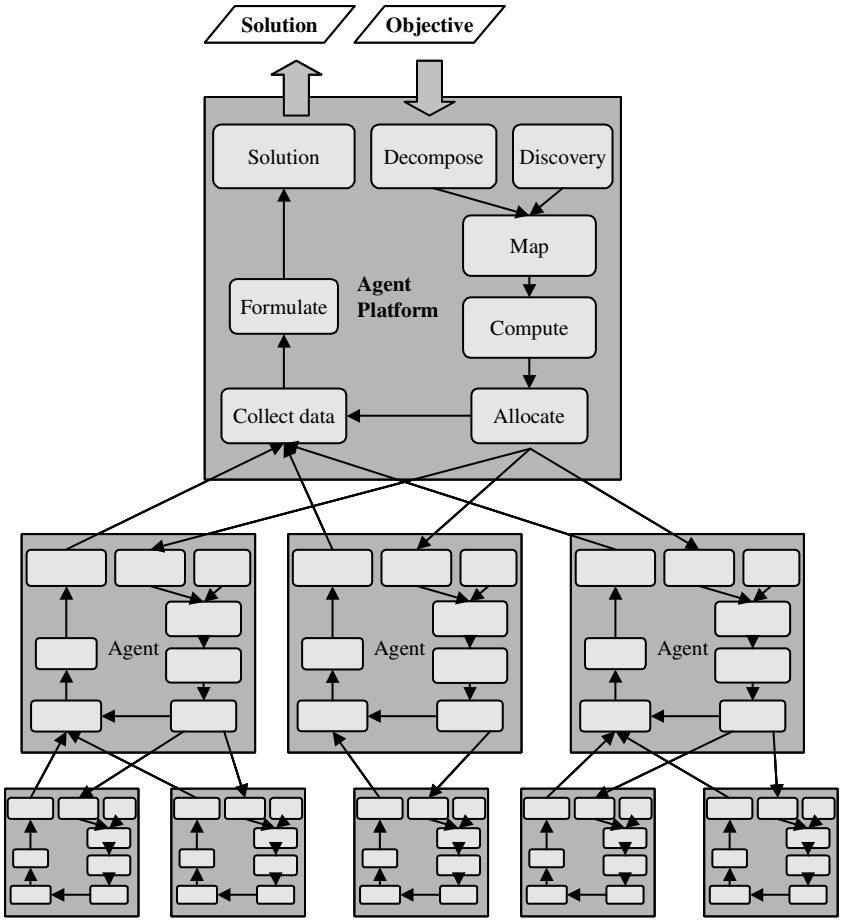

Figure 3-12: The problem model from Figure 3-2 can be applied recursively to compute optimal schedules each level of an objective decomposition hierarchy.

While this technique is limited to problems where the agent organization is a command hierarchy and the objective can be decomposed at each level of this hierarchy, it is nonetheless very useful for breaking up a large scheduling problem into a set of smaller schedules that are more feasible to solve. Exact limits to number of jobs and agents that the MILP algorithm presented in this thesis can solve in a reasonable amount of time are explored in Chapter 5.

-This Page Intentionally Left Blank-

Chapter 4

Schedule Optimization

This chapter presents a Mixed Integer-Linear Programming (MILP) formulation that can be used to optimally solve the types of multi-agent scheduling problems presented in Chapter 3.

4.1 Motivation

The primary motivation behind developing the MILP formulation in this chapter is to provide a scheduling algorithm that not only produces optimal makespan solutions, but also is flexible enough to model real-world problems that adhere to the size limitations defined by the abstract models in Chapter 5. While much research has been done on multi-agent scheduling and many algorithms have been proposed, almost all of them make many simplifying assumptions about the problem model or settle for sub-optimal solutions by using heuristics. These heuristic techniques are useful for simplified theoretical problem domains or for getting solutions quickly for large-scale scheduling problems where it is acceptable for the solution to be within a certain bound of the optimal value. In many cases though an optimal makespan may be sought for a smaller-

scale scheduling problem in business, military, or manufacturing domains, and it is these cases that this thesis seeks to address.

4.1.1 Flexibility

Flexibility is a key issue in developing an algorithm to solve multi-agent scheduling problems that occur in different domains, because each distinct problem may take a different set of parameters into account. Many existing algorithms are designed to solve specific classes of scheduling problems. For example, in some problems communication delay may be so small compared to the execution delay that it can be considered negligible, or jobs may not be precedence-constrained relative to each other. However, the MILP formulation presented in this chapter seeks to solve not only R|PREC,r_j|C_{max}|JP problems, the most complex class of problems defined by Lawler, Lenstra, Kan, and McDowell [2,6], but also any simpler class of problem as well, and the problem may include any of the following parameters:

- Communication delay can exist between agents, and may be a function of both the data being communicated and the available bandwidth between a pair of agents.
- Execution delay for each job may be a function of both the specific job and the agent that it is assigned to.
- Jobs may be precedence-constrained, and the precedence tree can take the form of any directed, acyclic graph (DAG).
- Time in this model is continuous, not discrete-valued. Therefore execution and communication delays may be any positive real-valued number and there is no limit to the time horizon that the schedule must be performed within.
- Jobs may have release times as additional constraints. Release times specify the earliest time that a job's execution can begin.

The tradeoff of having this flexibility and solution optimality is that there is a limit to the problem size that the algorithm can handle, since scheduling is an $\mathcal{NP}$ –hard problem. Therefore, the MILP formulation is most useful for solving schedules where there is

either a small number of agents, a small number of jobs, or where a large problem can be decomposed recursively into a hierarchy of smaller scheduling problems, as described in Section 3.6. Chapter 5 explores the limits to the size of scheduling problems that can be feasibly solved with this algorithm.

4.1.2 Assumptions

While the MILP formulation is designed to be as flexible as possible to take many different parameters into account, the nature of MILP techniques makes it necessary to make a few simplifying assumptions in our problem model.

Assumption 1: Job pre-emption is not allowed

Once an agent begins to execute a job, it must continue to completion without interruption. Also, an agent may process only one job at a time.

Assumption 2: All relevant parameters to the problem are known in advance

To compute the schedule to optimality, information about all available agents including services offered, execution delays, and communication delays are needed. If information about agents is not known *a priori*, those resources will not be taken into account when solving the problem. Methods for obtaining this information about the agent network are reviewed in Appendix C.

Assumption 3: The agent network is static

Because the entire schedule is computed prior to execution, any dynamic changes in the agent organization or job precedence graph during execution of the schedule may result in a sub-optimal or infeasible solution.

Assumption 4: Each individual job can be completed with a single agent

If a job requires a combination of services from multiple agents to be completed, that job must be further decomposed into smaller tasks specific to each service before the optimal schedule is computed, or that group of agents must be modeled as a single agent.

While these assumptions place some limits on the types of problems that can be solved using our technique, the algorithm is still flexible enough to be used for modeling scheduling problems in many different scenarios.

4.2 Implementation

MILP techniques involve defining a set of *constraints, decision variables,* and an *objective function*. The multi-agent scheduling problem can be modeled using the following variables and constraints.

4.2.1 Problem Variables

Input Parameters

Sets

J Set of all jobs $j \in J$ that must be completed to achieve an objective. Each job j is indexed numerically but may have a unique name that describes the service needed to complete it.

A Set of all available agents $a \in A$. Each agent a is indexed numerically but also may have a unique name associated with it.

Indicator Variables

P Immediate job precedence graph.

$$P^{j,k} = \begin{cases} 1, \text{ if job } j \in J \text{ directly precedes job } k \in J \\ 0, \text{ otherwise} \end{cases}$$

Q Full job precedence graph.

$$Q^{j,k} = \begin{cases} 1, \text{ if job } j \in J \text{ comes anytime before job } k \in J \text{ in } P \\ 0, \text{ otherwise} \end{cases}$$

B Agent ability matrix

$$B^{j,a} = \begin{cases} 1, \text{ if agent } a \in A \text{ provides the services needed to complete job } j \in J \\ 0, \text{ otherwise} \end{cases}$$

Data Values

D Execution Delay matrix

$D^{j,a}$ = the execution delay for agent $a \in A$ to process job $j \in J$. $D^{j,a} \in \Re$

C Communication Delay matrix

$C^{j,a,b}$ = the communication delay from sending the results of job $j \in J$ from agent $a \in A$ to agent $b \in A$. To be more specific, we define $C^{j,a,b}$ as a function of the number of bits that are produced by job j, $bits^j$, transmitted from agent a to agent b with bandwidth, $bandwidth^{a,b}$, in bits/second. In this case, $C^{j,a,b} = bits^j / bandwidth^{a,b}$. $C^{j,a,b} \in \Re$

R Job Release Times

R^j = release time for job $j \in J$. $R^j \in \Re$

Decision Variables

Schedule Generation

$X_{j,a}$ Allocation of jobs to agents

$$X_{j,a} = \begin{cases} 1, \text{ if agent } a \in A \text{ is assigned to complete job } j \in J \\ 0, \text{ otherwise} \end{cases}$$

S_j Scheduled start time of job $j \in J$. $S_j \in \Re$

Support Variables

$\theta_{j,k}$ Support variable used to determine whether jobs $j \in J$ and $k \in J$ overlap.

$$\theta_{j,k} = \begin{cases} 1, \text{ if job } k \in J \text{ is started before job } j \in J \text{ is completed} \\ 0, \text{ otherwise} \end{cases}$$

M An upper bound on the makespan set by default to a very large number. $M \in \Re$

Objective

C_{max} The makespan of the schedule. $C_{max} \in \Re$

4.2.2 Constraints

The following linear-integer constraints describe the conditions needed to compute an optimal multi-agent schedule.

Objective function:

Minimize: $C_{\max}$

Subject to:

$$\left.\begin{array}{l} C_{\max} \geq S_j + D^{j,a} * X_{j,a} \\ X_{j,a} \in \{0,1\} \end{array}\right\} \qquad \forall j \in J, a \in A \,\middle|\, B^{j,a} > 0 \qquad (4\text{-}1)$$

The objective function and agent/job allocation decision variables are defined in (4-1). To minimize the number of irrelevant binary variables, $X_{j,a}$ is only defined as binary when $B^{j,a} > 0$, i.e. when agent a offers the services necessary to complete job j.

$$X_{j,a} = 0 \qquad \forall j \in J, a \in A \,\middle|\, B^{j,a} = 0 \qquad (4\text{-}2)$$

Constraint (4-2) states that an agent cannot be assigned a job unless it provides the services necessary to complete that job. This constraint narrows down the total number of binary decision variables that need to be computed so that an optimal solution is reached faster.

$$\sum_{a \in A} X_{j,a} = 1 \qquad \forall j \in J \qquad (4\text{-}3)$$

Constraint (4-3) specifies that each job must be assigned once to exactly one agent.

$$\left.\begin{array}{l} S_k \geq S_j \\ S_k \geq S_j + (D^{j,a} + C^{j,a,b}) * (X_{j,a} + X_{k,b} - 1) \end{array}\right\} \begin{array}{l} \forall (j,k) \in J, (a,b) \in A \\ \quad \left| \, j \neq k, P^{j,k} > 0, B^{j,a} > 0, B^{k,b} > 0 \right. \end{array} \qquad (4\text{-}4)$$

The constraints in (4-4) state that a job cannot start until its predecessors are completed and data has been communicated to it if the preceding jobs were executed on a different agent.

$$S_j \geq R^j \qquad \forall j \in J \,\middle|\, R^j > 0 \qquad (4\text{-}5)$$

Constraint (4-5) specifies that a job cannot start until after its release time.

$$\left.\begin{aligned} & S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j \geq -M * \theta_{j,k} \\ & S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j < M * (1 - \theta_{j,k}) \\ & \theta_{j,k} \in \{0,1\} \end{aligned}\right\} \quad \forall (j,k) \in J \,\middle|\, j \neq k, Q^{j,k} = 0 \qquad (4\text{-}6)$$

$$X_{j,a} + X_{k,a} + \theta_{j,k} + \theta_{k,j} \leq 3 \qquad \forall (j,k) \in J, a \in A \qquad (4\text{-}7)$$

Collectively, (4-6) and (4-7) specify that an agent may process at most one job at a time. Constraint (4-7) states that if two jobs j and k are assigned to the same agent, their execution times may not overlap. Mathematically, jobs j and k overlap if the following two conditions are met:

$$1) \quad S_k < \sum_{a \in A} D^{j,a} * X_{j,a} + S_j \qquad (4\text{-}8)$$

Job k starts sometime before job j finishes execution, and

$$2) \quad S_j < \sum_{a \in A} D^{k,a} * X_{k,a} + S_k \qquad (4\text{-}9)$$

Job j starts sometime before job k finishes execution.

To describe these two conditions, the binary support variable $\theta_{j,k}$ is defined to equal 1 when (4-8) is true, and 0 otherwise. Because (4-9) is equivalent to (4-8) with j and k reversed, we can use the same variable $\theta_{k,j}$ to describe (4-9). Therefore, jobs j and k

overlap when both $\theta_{j,k}$ and $\theta_{k,j}$ are true, or when $\theta_{j,k} + \theta_{k,j} > 1$. Unfortunately, defining $\theta_{j,k}$ as a binary variable in terms of (4-8) and (4-9) is more difficult, because in an MILP problem we cannot use decision variables such as S_j as part of the condition for a constraint to occur. Instead, we must use the linear constraints in (4-6) to define $\theta_{j,k}$. If M is set at a large enough value, $\theta_{j,k}$ is forced to always assume the correct binary value with relation to (4-8) and (4-9).

Together, these constraints define the multi-agent scheduling problem using a small number of binary decision variables, which is an important factor in solving an MILP problem efficiently. XpressMP™ optimization software is used to model these problems and compute optimal schedules based on the above constraints (see Appendix B).

4.3 Multiple Objective Optimization

As shown in Chapter 3, an agent formulating a problem may often have a different objective than minimizing the makespan of the schedule, or may have other objectives in addition to it. These goals, like minimizing the makespan, can also be modeled using mathematical objective functions. After determining the objective functions for each, *Goal Programming* and *Multiple Objective Linear Programming* (MOLP) techniques may be used to find the decision variable values whose solution satisfies the most objectives.

4.3.1 Alternate Objective Functions

1) Minimize the average weighted completion time of all jobs

A common alternative to finding the optimal makespan of a schedule is to minimize the average (weighted) completion time of all of the jobs. For example, in certain instances it may be better to complete 4 out of 5 jobs very early even if the 5th job takes a long time than to complete all of the jobs in less time, but have them all completed near the end of the schedule. In this case we can either try to minimize the average completion time of all jobs (weighing each job equally), or assign a fractional weight to each job signifying

how important that job is compared to other jobs. The latter case requires an additional parameter:

W^j = the fractional weight given to a particular job, where $W^j \in \Re$ and $\sum_{j \in J} W^j = 1$

The new objective function is

$$\text{Minimize:} \quad \sum_{j \in J} \sum_{a \in A} (S_j + D^{j,a}) * X_{j,a} * W^j \tag{4-10}$$

2) Minimize the total number of agents used to solve the problem

In an agent environment it is often useful to allocate jobs to as few agents as possible that can complete an objective in a reasonable amount of time. This is helpful when agents are a scarce resource or are needed to be available to complete additional outside tasks, and it makes the system more fault-tolerant if the structure of the agent network changes dynamically. To model this objective an additional binary decision variable is defined:

$$\mu_a = \begin{cases} 1, & \text{if agent } a \in A \text{ is assigned one or more jobs to complete} \\ 0, & \text{otherwise} \end{cases}$$

μ_a can be defined using integer-linear constraints in a similar fashion to $\theta_{j,k}$, based on the binary decision values of $X_{j,a}$ for each agent:

$$\left.\begin{aligned} \sum_{j \in J} X_{j,a} - 0.5 &\geq -M * (1 - \mu_a) \\ \sum_{j \in J} X_{j,a} - 0.5 &\leq M * \mu_a \end{aligned}\right\} \qquad \forall a \in A \tag{4-11}$$

The objective function then becomes

$$\text{Minimize:} \quad \sum_{a \in A} \mu_a \tag{4-12}$$

3) Minimize the total amount of bandwidth used between agents

Bandwidth is often a scarce resource in an agent network, so it may be beneficial to minimize the total number of bits communicated between agents when computing an optimal schedule. To calculate the number of communications that take place, a composite binary decision variable $Y_{j,a,k,b}$ is defined:

$$Y_{j,a,k,b} = \begin{cases} 1, \text{ if } X_{j,a} = 1 \text{ and } X_{k,b} = 1 \\ 0, \text{ otherwise} \end{cases}$$

Using the following constraint:

$$Y_{j,a,k,b} = X_{j,a} * X_{k,b} \tag{4-13}$$

The objective function then becomes:

$$\text{Minimize:} \quad \sum_{\substack{(a,b)\in A \\ |a\neq b}} \sum_{\substack{(j,k)\in J \\ |P^{j,k}>0}} Y_{j,a,k,b} * bits^{j} \tag{4-14}$$

4) Maximize the aggregate value associated with assigning jobs to particular agents

In many cases there may be a value associated with assigning tasks to particular agents. For example, in a business model where each agent produces a different amount of revenue by completing various tasks assigned to it, a manager may wish to maximize the total revenue produced as an objective in computing the optimal schedule. In this case the model requires an additional set of parameters:

$value^{j,a}$ = the value associated with assigning job $j \in J$ to agent $a \in A$.

$value^{j,a} \in \Re$

The objective function for maximizing the aggregate value is then defined as:

$$\text{Maximize:} \quad \sum_{j\in J} \sum_{a\in A} value^{j,a} * X_{j,a} \tag{4-15}$$

5) Minimize the aggregate cost associated with assigning jobs to particular agents

Minimizing the costs associated with assigning tasks to agents is another common objective that an agent may have when computing an optimal schedule. In a military scenario, mobile autonomous agents use fuel to move around and may have risk levels associated with performing different operations, and it is often beneficial to minimize these values. Cost is defined by the following parameter:

$cost^{j,a}$ = the cost associated with assigning job $j \in J$ to agent $a \in A$.

$cost^{j,a} \in \Re$

The objective function for minimizing cost is analogous to that for maximizing aggregate value (4-15):

$$\text{Minimize:} \quad \sum_{j \in J} \sum_{a \in A} cost^{j,a} * X_{j,a} \tag{4-16}$$

4.3.2 Multiple Objective Linear Programming

To optimize multiple objectives simultaneously using the same integer-linear program, each objective is treated as a *goal* to be achieved. In many cases two objectives may conflict with one another, thus making it impossible to obtain optimal values for both at once. For example, in some problems minimizing the schedule's makespan may involve using a greater number of agents and more bandwidth than desired, which would conflict with any objective that tries to minimize costs associated with these parameters. In these cases there is often no single optimal solution, and a value metric must be used as a means of weighing possible solutions against each other. The problem may then be solved iteratively until a satisfactory solution is produced [19].

To illustrate MOLP techniques, we use the multi-agent arithmetic example from Chapter 3, shown again in Figure 4-1. This time, however, we assume that there is a cost associated with assigning a task to each agent (i.e. each agent charges a fee for every operation that it performs). For simplicity we assume that the cost is specific to each agent and is independent of the job chosen. Therefore, in addition to minimizing the

makespan of the schedule, Agent A has the additional objective of minimizing the total cost across all agents. Suppose that cost is defined as:

COST = {$0, $4, $5, $3, $1, $2, $2} for each agent {A, B, C, D, E, F, G}

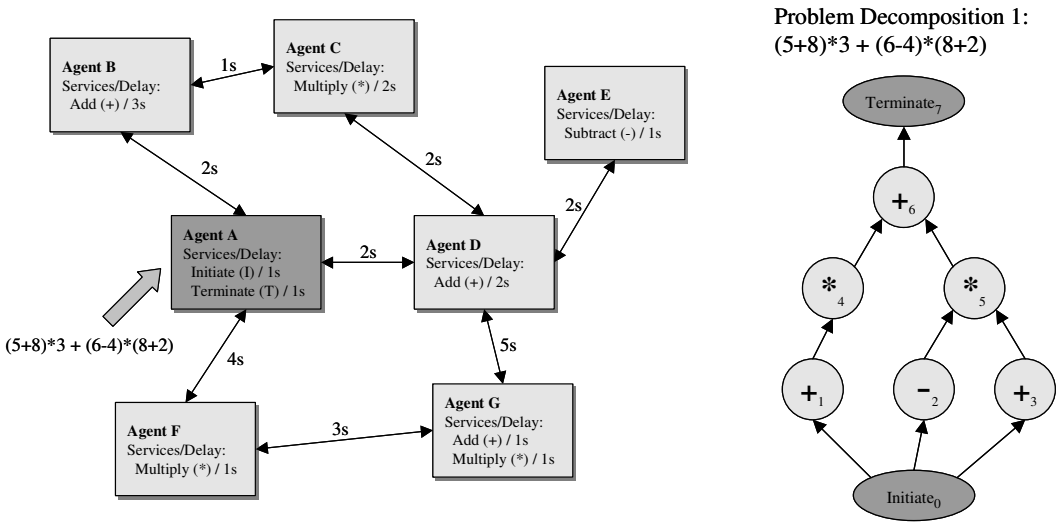

Figure 4-1: The agent organization and problem formulation used for the arithmetic example in Chapter 3

4.3.2.1 Determining Target Objective Values

The first step in solving an MOLP problem is to change each objective function into a *goal* with a target value. The objective then becomes to find the optimal agent/job allocation that achieves the closest approximation of these target values. It seems reasonable that the optimal solution values for each of the individual objective functions would provide good target values for the corresponding goals. Solving the MILP problem twice, once for each individual objective function, produces the following optimal target values:

	Objective Function		*makespan*	*cost*
1)	Minimize:	C_{max}	19s (target)	$20
2)	Minimize:	$\sum_{j \in J} \sum_{a \in A} cost^{j,a} * X_{j,a}$	26s	$11 (target)

The objective now is to find the allocation that produces a makespan as close to 19s as possible, and a total cost as close to $11 as possible.

4.3.2.2 Determining the Goal Programming Objective

The next step is to design a metric that can be used to weigh the two goals against each other so that they are both optimized as much as possible. Because each of the target goal values is measured in different units (dollars vs. time in this case), it is difficult to compare tradeoffs between them directly. The fairest solution is to instead calculate the *percentage deviations* of each goal's actual value from its target value and have the new objective function be to minimize the weighted sum of these deviations:

$$\text{Minimize:} \qquad w_1 * \frac{C_{\max} - 19}{19} + w_2 * \frac{\sum_{j \in J} \sum_{a \in A} (cost^{j,a} * X_{j,a}) - 11}{11} \qquad (4\text{-}17)$$

By varying the values of w_1 and w_2, we can place different emphasis on each goal and achieve different optimal allocations based on the relative importance of these goals to one another. For example, by setting w_1 and w_2 each to 0.5, the schedule shown in Figure 4-2 is produced, which achieves the optimal target cost value ($11), and minimizes the makespan to 23s, which is within 21% of its target value.

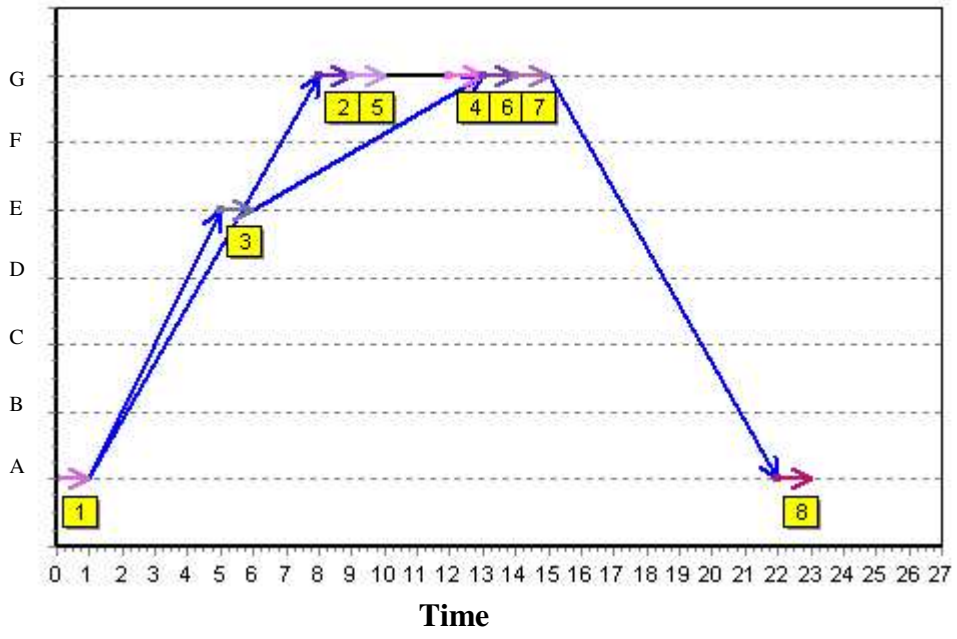

Figure 4-2: An MOLP schedule that tries to minimize both the schedule's makespan and the execution cost on each agent, where w_1 = .5 and w_2 = .5. The makespan is 23s and the total cost is $11.

Varying w_1 and w_2 places different emphasis on each goal relative to the other one, which results in different sets of optimal solutions, as shown in Figure 4-3. In this figure, varying w1 and w2 between 0 and 1 produces three distinct solutions. The agent formulating the problem must decide which of these solutions is best for each particular scenario, and that can vary on a case-by-case basis.

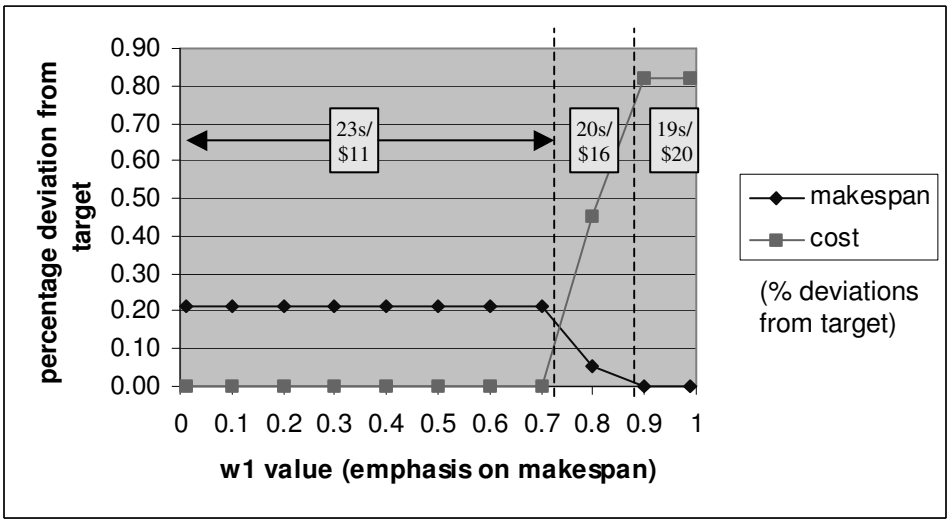

Figure 4-3: A chart showing all of the different optimal solutions for minimizing the weighted sum of percentage deviations, using different values of w_1 and w_2.

Each of the three possible solutions shown in Figure 4-3 represents a different *corner point* of the MILP problem's feasible region. Minimizing the weighted sum of percentage deviations from target values will only produce corner point solutions; in order to explore other non-corner point solutions along the edge of the feasible region, the MINIMAX objective function must be used instead. MINIMAX tries to minimize the maximum percentage deviation of each individual goal from its target value instead of minimizing a weighted sum of these values. A detailed description MINIMAX optimization can be found in *Spreadsheet Modeling and Decision Analysis*, chapter 7 [19].

4.4 Refining the MILP Formulation

The amount of time needed to solve any particular MILP problem can vary significantly depending on how well a problem is formulated. Unnecessary or redundant binary decision variables or loose constraints may result in additional Branch and Bound iterations and much longer computation times. Refining an MILP formulation can be very difficult, and often involves a lot of trial and error due to the fact that there is no guaranteed formula for making every model run faster. As shown in Chapter 2, the best indicator of the quality of an MILP formulation is how well its constraints approximate the *convex hull* of the feasible region. Unfortunately, this is very difficult to predict in advance for all but the most trivial problems, as some problems involving only a hundred decision variables may be very challenging, while others with thousands of variables are sometimes solved with fewer iterations. While there are a few rules of thumb that will generally lead to better problem formulations, the only way to find the best formulation is to have insight into the structure of the system being modeled, and to run numerous computational tests.

4.4.1 Eliminating Binary Decision Variables

The difference between Linear Programming and Mixed Integer-Linear Programming is that for MILP problems the Branch and Bound technique must be used to recursively refine solutions to optimality. This is a much more time-consuming process than just using the Simplex method to solve an LP problem; in fact a new LP formulation must be solved at each node in the Branch and Bound tree. When integer decision variables are restricted to binary values, as they are in the MILP formulation presented in this chapter, one of these variables is set to either 0 or 1 at each node of the Branch and Bound tree as shown in Figure 4-4. Thus, in a worst-case scenario, a problem that has k binary decision variables could have up to 2^k Branch and Bound iterations. Fortunately this is almost never the case because the Branch and Bound is a greedy algorithm that narrows down possible solutions as it runs, but this shows that Branch and Bound problems can take exponentially longer to solve as they use more binary variables.

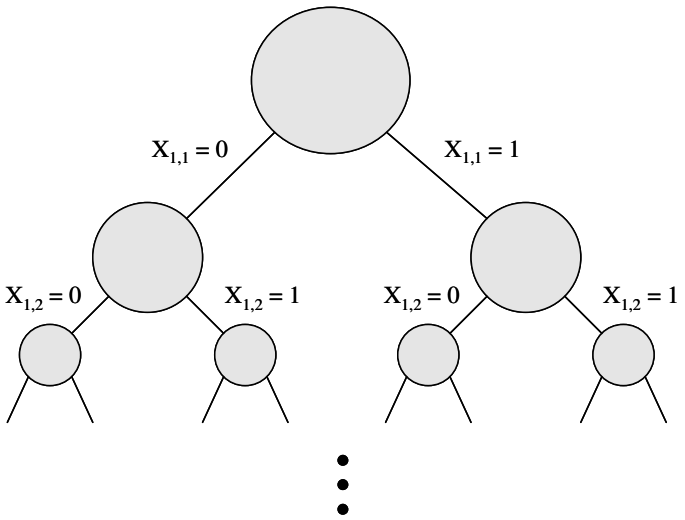

Figure 4-4: A Branch and Bound tree with binary decision variables.

For this reason, reducing the number of binary variables used in an MILP problem is an essential step in shortening computation time. The MILP formulation presented in this chapter contains two binary decision variable types, $X_{j,a}$ and $\theta_{j,k}$. A scheduling problem that has m agents and n jobs contains $(n*m)$ $X_{j,a}$ variables and (n^2) $\theta_{j,k}$ variables. Therefore, this MILP formulation would have $n(n + m)$ total binary variables, and its Branch and Bound tree would have at most $2^{n(n + m)}$ nodes corresponding to these variables.

4.4.1.1 Discrete Time vs. Continuous Time

The original MILP formulation developed for this thesis used a discrete-time model to compute optimal multi-agent schedules. Instead of using the binary variable $X_{j,a}$ and the continuous variable S_j to describe the schedule, an additional parameter T and a binary decision variable $X_{j,a,t}$ were used to represent job execution at discrete time values, where

$$X_{j,a,t} = \begin{cases} 1, \text{ if agent } a \in A \text{ is assigned to complete job } j \in J \text{ at time } t \in T \\ 0, \text{ otherwise} \end{cases}$$

Using this discrete-time model made it much more straightforward to model the linear constraints. For example, instead of having to define the support variable $\theta_{j,k}$ to model

the restriction that each agent can only perform one service at a time (4-6, 4-7), this specification can be modeled by a single linear constraint using discrete-time values:

$$X_{j,a,t} + X_{k,a,s} \leq 2 \qquad \begin{array}{l} \forall (j,k) \in J, (t,s) \in T, a \in A \\ \mid j \neq k, B^{j,a} > 0, B^{k,a} > 0, t \leq s \leq t + D^{j,a} \end{array} \tag{4-18}$$

While using discrete-time variables simplifies the constraints needed to model the problem, it results in a formulation that takes more time to solve and makes the algorithm less flexible. Using discrete-time, all possible time values must be explicitly defined, and therefore an upper bound on the schedule's makespan must be determined before the problem is formulated. While this can be done using common heuristic techniques, it limits the overall flexibility of the algorithm by forcing the time and execution/communication delays to adhere to pre-defined interval lengths. Also, discrete-time variables add another layer of combinatorial complexity to the problem: instead of *n(n+m)* binary variables there are now (*m*n*t*) variables, where *t* is number of distinct time steps in the model. Because of this added complexity, the MILP problem was reformulated to use continuous time instead.

4.4.2 Refining Constraints

In general, adding more constraints to an MILP formulation tightens the bound on the feasible region, thus creating a better approximation of the convex hull. While this tends to result in fewer Branch and Bound iterations, adding more constraints also increases the computational complexity of the LP relaxations produced at each step in the Branch and Bound process. Therefore, it is important to weigh the tradeoffs in computation time whenever a new set of constraints is added. For example, constraint (4-2) and the constraint $S_k \geq S_j$ in (4-4) are not necessary to solve the scheduling problem to optimality, but they help to create a tighter bound on the set of feasible points so that a solution is reached faster.

Another improvement made to the MILP formulation in this chapter was to eliminate unnecessary constraints that do not tighten the bound on the feasible region. The input parameters *B*, *P*, and *Q* defined in Section 4.2.1.1 define each agent's abilities and the structure of the job precedence tree, and are used to selectively apply constraints to only the appropriate jobs or agents. For example, *P* and *B* are used to specify that the constraints in (4-4) only apply to jobs that have immediate precedence constraints relative to each other, and to agents that are capable of completing these jobs. Similarly, the constraints in (4-6) state that two jobs may not be executed simultaneously on the same agent, and *Q* is used to specify that these constraints only apply to agents that have no precedence relations at all relative to each other.

$$X_{j,a} = 0 \qquad \forall j \in J, a \in A \,\middle|\, B^{j,a} = 0 \qquad (4\text{-}2)$$

$$\left.\begin{aligned} & S_k \geq S_j \\ & S_k \geq S_j + (D^{j,a} + C^{j,a,b}) * (X_{j,a} + X_{k,b} - 1) \end{aligned}\right\} \quad \forall (j,k) \in J, (a,b) \in A \;\middle|\; j \neq k, P^{j,k} > 0, B^{j,a} > 0, B^{k,b} > 0 \qquad (4\text{-}4)$$

$$\left.\begin{aligned} & S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j \geq -M * \theta_{j,k} \\ & S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j < M * (1 - \theta_{j,k}) \\ & \theta_{j,k} \in \{0,1\} \end{aligned}\right\} \quad \forall (j,k) \in J \,\middle|\, j \neq k, Q^{j,k} = 0 \qquad (4\text{-}6)$$

Creating tighter bounds on the integer feasible region and eliminating unnecessary constraints and binary variables significantly shortens the time needed to compute an optimal schedule using MILP techniques. For instance, consider the time taken to solve the arithmetic scheduling problem from Chapter 3 using a discrete-time model versus a continuous-time model, shown below in Figure 4-5.

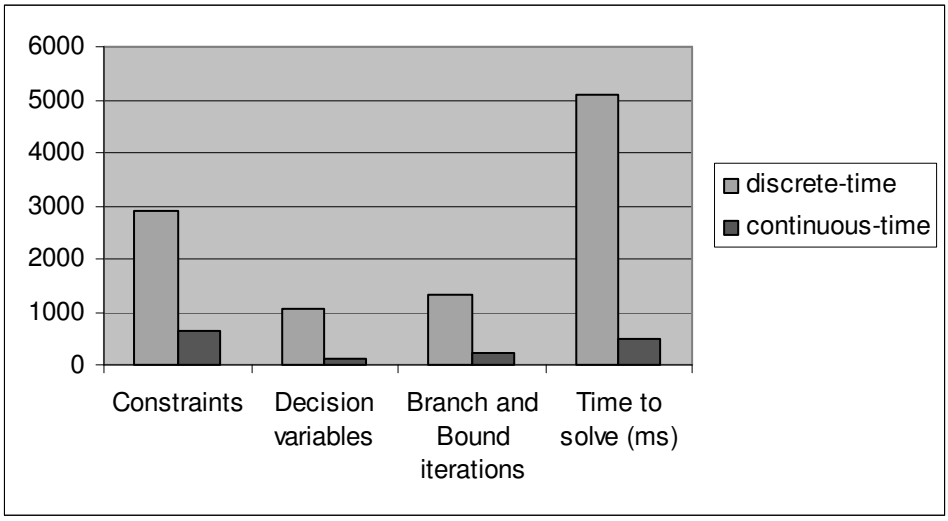

Figure 4-5: Comparison of the number of constraints, decision variables, Branch and Bound iterations, and time to solve a scheduling problem using two different MILP formulations.

This figure shows that reducing the number of binary variables and modeling the MILP formulation in continuous time reduces the problem to approximately 10% of its original complexity. This difference becomes even greater with larger problems, as shown in the next chapter.

-This Page Intentionally Left Blank-

Chapter 5

Simulation and Results

This chapter examines the performance of the MILP formulation presented in Chapter 4 when it is applied to different abstract scheduling applications. As mentioned in Chapter 2, the main drawback of using MILP techniques is that the problem is $\mathcal{NP}$-hard, and takes exponential time to solve. Therefore, while the algorithm in this thesis can model the most complex classifications of scheduling problems, solutions are limited to instances where there is a small number of agents or jobs. By modeling the MILP problem using XpressMP™ optimization software, we can gather data about the complexity of different problem instances and determine the limits to the problem size that the simulation can feasibly handle, and the rate at which complexity grows as more variables are added.

The first half of this chapter studies the complexity of the MILP scheduling formulation with regard to the number of agents and jobs in the problem model. By fixing some parameters and varying others, abstract examples are created that can be used as benchmarks to measure the algorithm's performance relative to different combinations of agents and jobs. The second half of this chapter examines a scheduling application based on the closed-loop command and control architecture presented in Chapter 3 [10], and

studies how changing the agent organization for a given job precedence graph affects the optimal makespan of the schedule as well as the difficulty of computing the makespan.

5.1 Scheduling Problem Complexity

This section explores the complexity of the MILP scheduling problem, based on the number of agents and jobs used in each instance. As mentioned in Chapter 4, using a continuous-time model instead of a discrete-time model allowed us to reduce the number of binary variables to $n(n+m)$ for a problem with n jobs and m agents. Based on this, it would seem reasonable to assume that the complexity of the MILP problem depends more on the number of jobs than on the number of agents. The following benchmarks were used to test this theory and to determine the rate at which the exponential-time algorithm runs relative to each of these two parameters. Each experiment was performed on an Intel Pentium II processor running at 450MHz with 512Mb of available RAM using XpressMP™ optimization software (see Appendix B).

5.1.1 Agent Variation

This benchmark demonstrates how the time to solve the MILP problem increases as the number of agents increases. The number of jobs is fixed at one, and the execution delay and communication delays between all agents are set to constant values (the agents are assumed to be in a fully-connected network). The time to solve the problem as a function of the number of agents is shown in Figure 5-1(a).

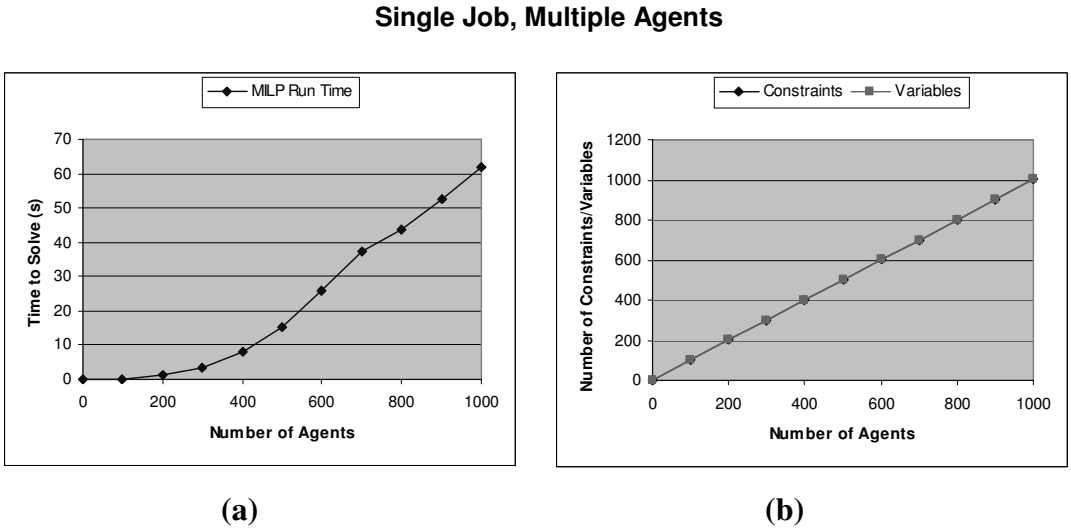

Figure 5-1: Performance test of the time needed to solve the MILP scheduling problem with a single job and multiple agents (a), and the number of constraints and variables used to model the problem, after presolving (b).

This data shows that when the number of jobs is limited to one, the complexity of the problem still increases exponentially, but at a very slow rate. Even with over 1000 agents, an optimal schedule can still be computed in a reasonable amount of time (62s). This makes sense because the problem has essentially been reduced to a resource allocation problem where a single resource (job) must be allocated to the agent that computes it the fastest, as there are no precedence constraints or parallel job executions to take into account. Also, Figure 5-1(b) shows that the number of constraints and decision variables grows linearly as a function of the number of agents.

However, the problem becomes significantly harder to solve once a second job is added, as shown in Figure 5-2(a). With two precedence-constrained jobs, the time to compute the optimal schedule increases at a much faster rate even with a smaller number of agents. This is because with m agents, instead of m possible agent/job allocations, there are now m^2 different schedules. Even with this added complexity though, an optimal schedule can still be computed for up to 90 agents in less than 10 minutes. It should be noted that while the number of decision variables still grows linearly, the number of constraints increases exponentially after the second job is introduced, which adds delay in computing the LP relaxation of the problem at each Branch and Bound step.

Figure 5-2: Performance test of the time needed to solve the MILP scheduling problem with two precedence-constrained jobs and multiple agents (a), and the number of constraints and variables used to model the problem, after presolving (b).

5.1.2 Job Variation

This benchmark examines the scheduling algorithm's performance for different numbers of precedence-constrained jobs, while keeping the number of agents fixed at one. The jobs are ordered in a single precedence chain such that (job 1 $\prec$ job 2 $\prec$ … $\prec$ job n), and the execution and communication delays are constant values. Figure 5-3(a) shows the performance of the MILP algorithm as a function of the number of jobs to be scheduled.

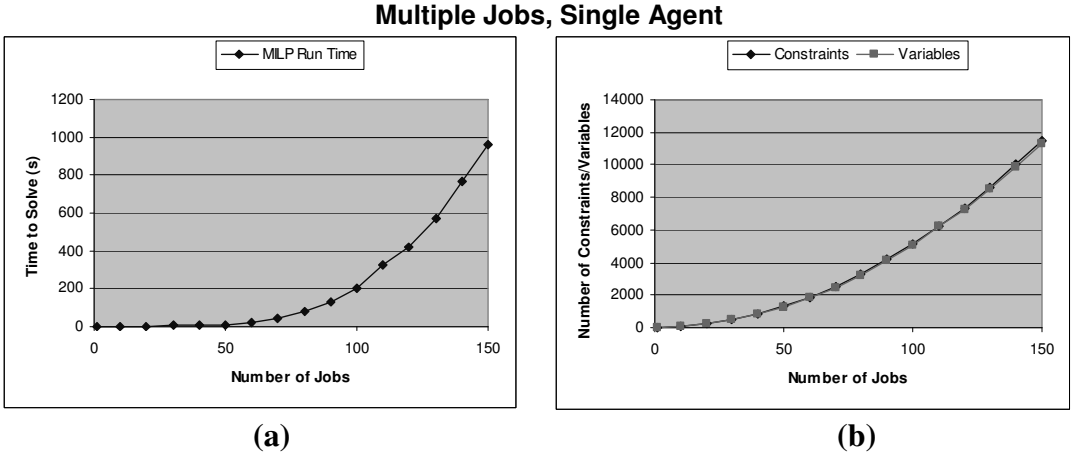

Figure 5-3: Performance test of the time needed to solve the MILP scheduling problem with multiple precedence-constrained jobs and a single agent (a), and the number of constraints and variables used to model the problem, after presolving (b).

As seen by comparing Figures 5-3 and 5-1, the problem complexity increases at a higher rate with the number of jobs than it does with the number of agents. This is because while there is only one agent to schedule all of the jobs on, many more MILP constraints are needed to enforce the precedence relations between the jobs, as shown in Figure 5-3(b). Also, as mentioned earlier in this chapter, the number of binary decision variables depends more on the number of jobs than on the number of agents. Using the MILP scheduling algorithm, it is still possible to compute the schedule in less than 15 minutes for up to 150 jobs with only a single agent.

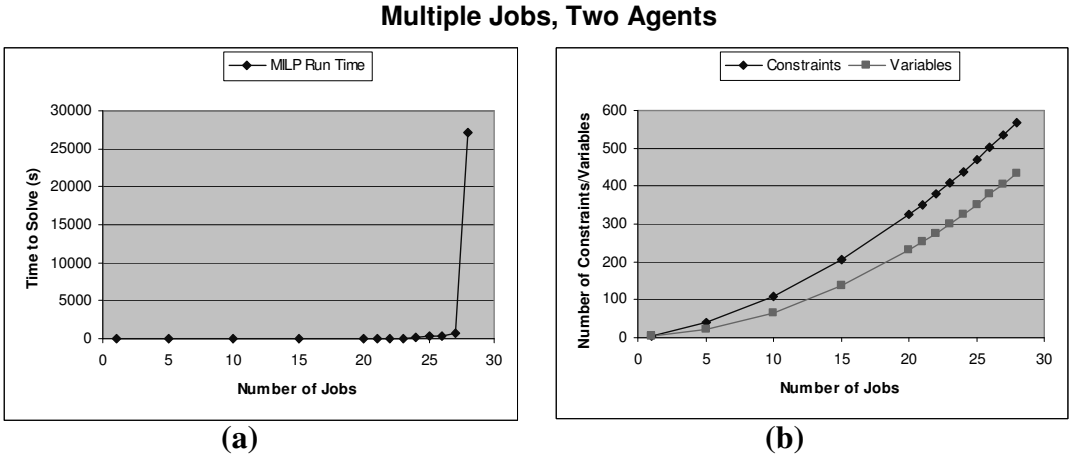

Figure 5-4: Performance test of the time needed to solve the MILP scheduling problem with multiple precedence-constrained jobs and two agents (a), and the number of constraints and variables used to model the problem, after presolving (b).

As with the previous example, the MILP algorithm takes much longer to compute an optimal schedule when another parameter is added to the problem. Adding another agent to the model introduces communication delay, which increases the complexity of the problem significantly. Figure 5-4(a) shows that the problem takes much longer to solve with two agents, with the time to solve increasing exponentially until the number of jobs reaches 27. With the 28^{th} job, however, the algorithm's run time jumps from 691s to 27139.7s, an increase by a factor of almost 40. This sudden jump in computation time may be caused by multiple factors. First, the XpressMP™ optimization software used to solve this model sometimes employs different Branch and Bound node selection strategies for the same problem model, resulting in varying computation times depending on the nodes is chooses (see Appendix B). Also, the MILP Search Chart produced by

XpressIVE™ in Figure 5-5 shows that the lower bound produced by each successive LP relaxation is increasing at a logarithmically when there are 28 jobs, and it will take a long time for the lower bound to meet the upper bound at that rate. By inspection, however, it can be seen in this figure that the optimal solution, 28s, was found in the first minute of the algorithm's run time, so there is no need to let the algorithm run to completion in this case.

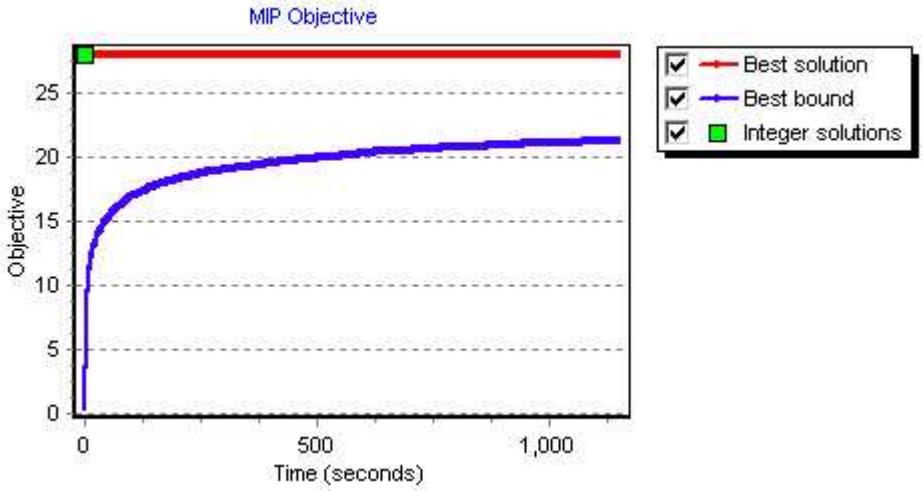

Figure 5-5: An MILP search chart produced using XpressIVE™ software running a scenario with 2 agents and 28 precedence-constrained jobs.

5.1.3 Job Parallelism

In addition to the number of jobs and agents used in the MILP model, the amount of *parallelism* in the job precedence graph can also play a role in the complexity of the scheduling problem. Being able to solve jobs in parallel across multiple agents not only shortens the makespan of a schedule, but reduces the computation time of the optimal schedule as well. Mathematically, the parallelism P of a job precedence graph can be defined as the total execution time for all jobs in the system divided by the execution time of the graph's *critical path*, as defined by Liao [16]:

$$P = \frac{\sum_{j \in J} D^j}{\max_{j \in J} L(j)} \tag{5-1}$$

In this equation *L(j)* is the sum of job execution times on the longest path from job *j* to an *exit task*, a job that has no successors in the job precedence graph. Mathematically *L(j)* is defined by Liao as:

$$L(j) = \max_{\forall \pi_k} \sum_{j \in \pi_k} D^j \tag{5-2}$$

where π_k is the *k*th path from job *j* to an exit task.

To determine how job parallelism affects the complexity of the MILP scheduling problem, we use the example from the previous section, two agents and multiple jobs, but this time the jobs are precedence-constrained such that they form two parallel chains, instead of a single chain. This configuration doubles the parallelism of the schedule, and the optimal makespan can be reduced by 50% by assigning each chain to be executed in parallel on a different agent. The time taken to compute the schedule is reduced as well, as seen by comparing Figure 5-6 to Figure 5-4(a).

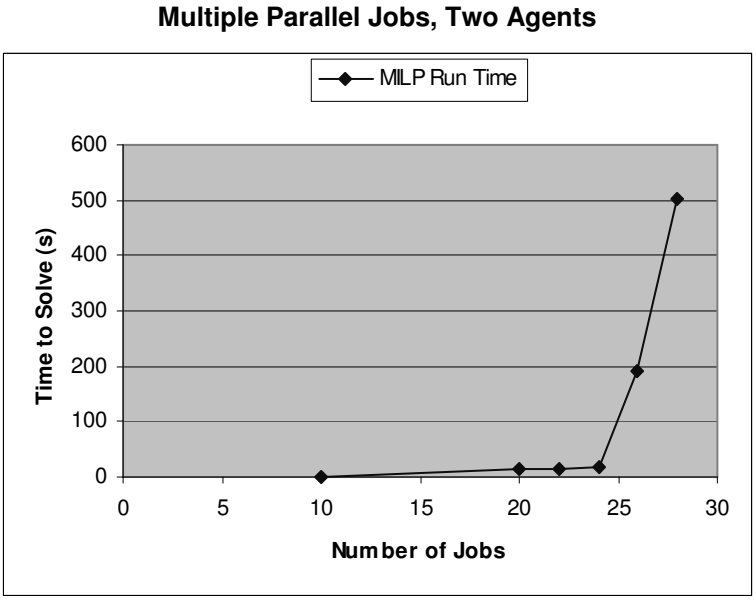

Figure 5-6: Performance test of the time needed to solve the MILP scheduling problem with multiple precedence-constrained jobs and two agents when the jobs form two parallel chains.

5.1.4 Conclusion

While the MILP scheduling algorithm does not perform well with large-scale problems because of its exponential complexity, these results show that the algorithm can be effectively used to compute minimal makespans for scheduling problems where the number of agents and jobs is kept small (<20). Also, by increasing the parallelism of the job precedence graph, the computational time as well as the makespan can be greatly reduced, allowing more agents and jobs to be modeled.

5.2 Simulation

The scenario addressed in this section is based on the closed-loop command and control architecture developed at Draper Laboratory that was introduced in Chapter 3, Figure 3-1. As mentioned in that chapter, this architecture is used to model autonomous systems and can be used recursively to solve hierarchical problem decompositions. While the architecture is used in Chapter 3 to describe the schedule optimization and multi-agent problem-solving processes themselves, it can also be used to model a wide variety of other agent planning and control scenarios, such as the hierarchical control of military air operations [10].

The four sub-functions, Plan, Execute, Monitor, and Diagnose, can themselves be seen as individual jobs that are precedence-constrained to each other by the flow of data between them, as shown in Figure 5-6. For example, in this figure an autonomous system generates a plan, and divides it into two sub-objectives which must themselves be planned and executed, and the results are monitored, diagnosed, and returned to the higher-level process. For simplicity, the feedback mechanism shown at each planning stage in Chapter 3, Figure 3-3 has been eliminated, and it is assumed that the lower-level planning processes simply forward their results back to the higher-level process without immediately re-planning, and that only a single iteration of the closed-loop architecture is performed.

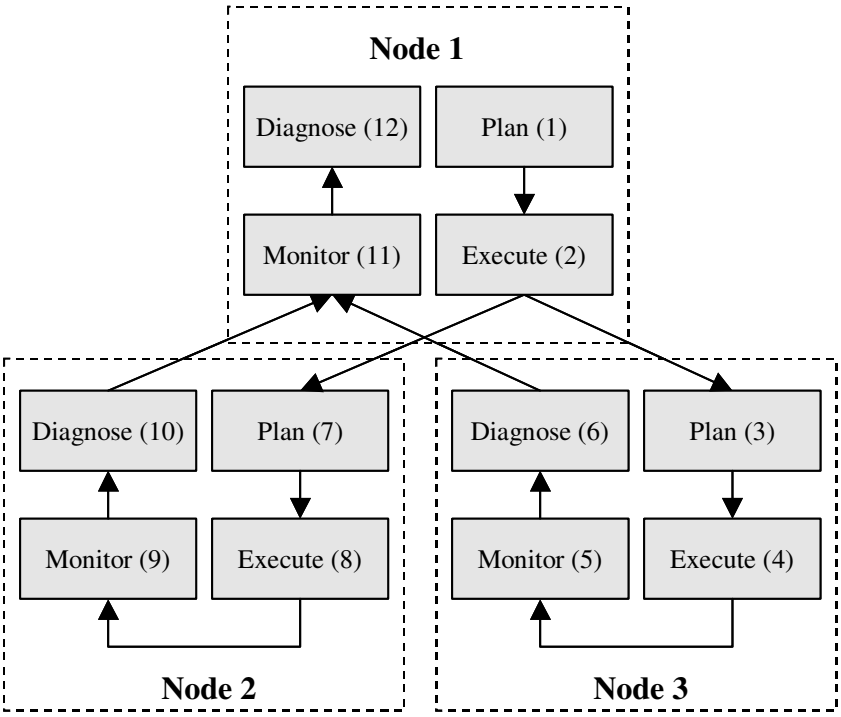

Figure 5-6: Hierarchical Command and Control Architecture.

While it is often assumed that each 4-stage planning process takes place on a single autonomous agent platform, these individual stages can actually be abstracted away and seen as separate entities that can be computed independently by different agents. By assigning each job a unique ID, the hierarchical command and control architecture can be seen as a graph of precedence-constrained jobs that can be completed in parallel by a set of agents that have the ability to perform these four operations.

Suppose that a commander wishes to use this architecture to complete a particular problem, and that it has to choose the set of agents that can complete these ordered tasks in the shortest amount of time. The commander has three sets of agents to choose from, each of which has different execution delays for completing each job, and different communication delays between each pair of agents. The three possible agent organizations are shown in Figures 5-7(a),(b), and (c).

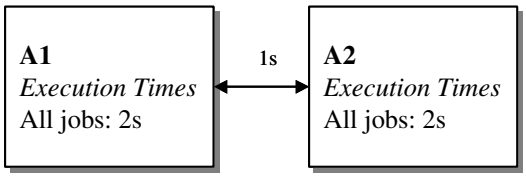

Figure 5-7(a): Organization A. Two agents with equal capabilities at performing each job.

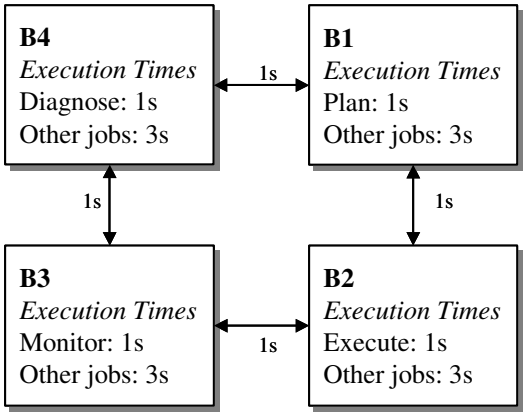

Figure 5-7(b): Organization B. Four agents that each specialize in performing a different job type.

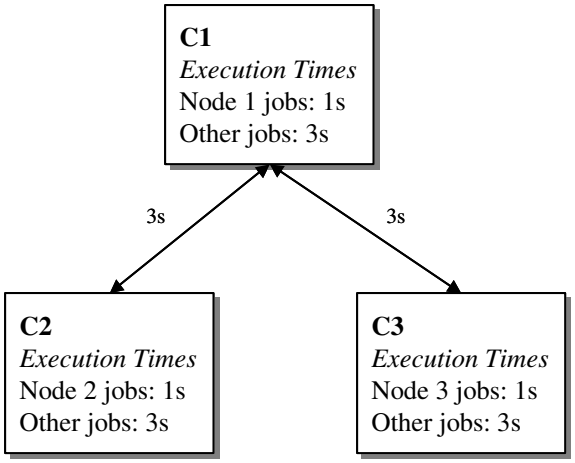

Figure 5-7(c): Organization C. Three agents that specialize in each of the three nodes from Figure 5-6.

Organization A contains two agents with a communication delay of 1s between them. These agents don't specialize in any particular job, but can execute any job in 2s. Using the MILP scheduling algorithm and XpressIVE™ to compute the optimal schedule for this agent organization, the Gantt chart in Figure 5-8 is produced. The schedule is computed in 2.5s, and the optimal makespan is 17s.

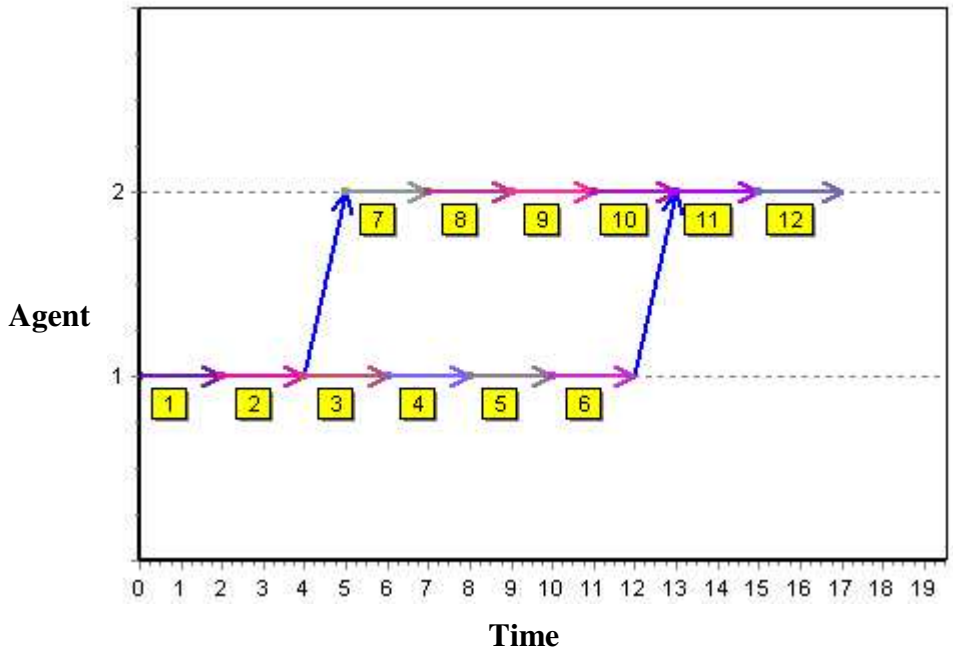

Figure 5-8: The optimal schedule computed for Organization A, with two agents performing 12 jobs.

Organization B, on the other hand, contains 4 agents, each of which is specialized to perform a specific job type. Agent B1 takes 1s to perform the Plan operation, but 3s to perform any other operation. Similarly, B2 specializes in Execution, B3 specializes in Monitoring, and B4 specializes in Diagnosis. The Gantt chart produced by the MILP scheduling algorithm in Figure 5-9 shows that by using this organization, the total makespan can be reduced to 16s. Because there are more agents though, the MILP problem is more complex and now takes 409s to compute the optimal schedule.

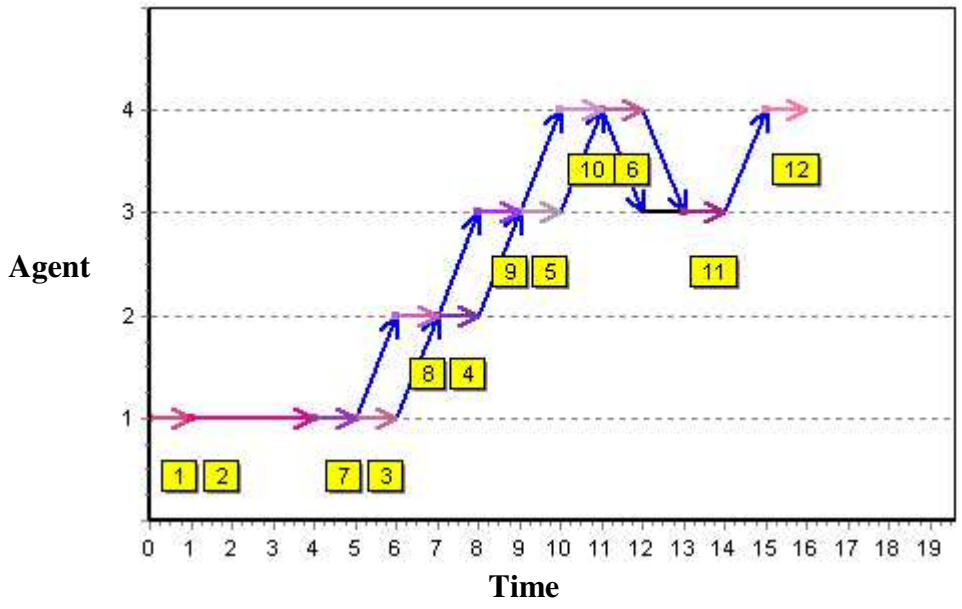

Figure 5-9: The optimal schedule computed for Organization B, with four agents performing 12 jobs.

Organization C is the traditional agent organization that would be used in the command and control hierarchy, where each agent corresponds to one of the 4-stage problem nodes. These agents can perform any job in their assigned node in 1s, but take 3s to perform any other job. However, this agent organization has longer communication delays between agents, with each communication taking 3s. The optimal schedule using this agent organization has an optimal makespan of 14s though, as shown in Figure 5-10, making it the best choice out of the three.

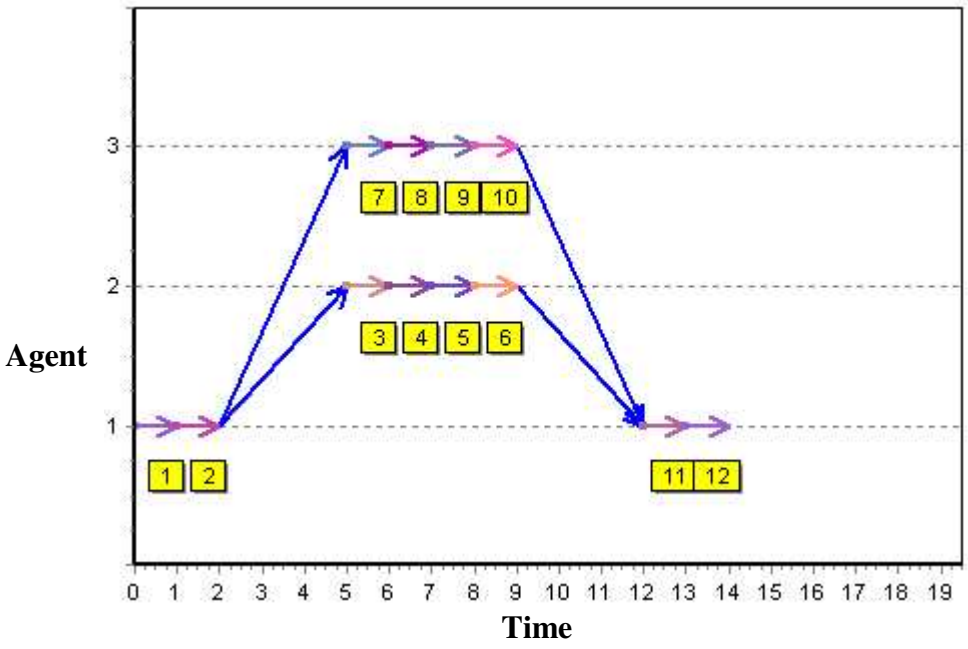

Figure 5-10: The optimal schedule computed for Organization C, with three agents performing 12 jobs.

-This Page Intentionally Left Blank-

Chapter 6

Conclusion

The goal of this thesis has been to explore multi-agent scheduling complexity and to develop a technique for solving the most difficult classes of these problems. A high-level, closed-loop framework is used to model the various stages of multi-agent problem-solving, including problem formulation and decomposition, job/service mapping, schedule computation, and task allocation. A new Mixed-Integer Linear Programming (MILP) formulation is introduced which is capable of computing optimal makespans to schedules in the presence of communication delay, job execution delay, release times, and other parameters, and the algorithm's performance is studied under different conditions. Extensions to the problem are also explored such as using MILP techniques recursively in an agent command hierarchy to solve large scheduling problems, and using Multiple Objective Linear Programming (MOLP) techniques to optimize additional objective functions such as minimizing cost or bandwidth usage.

6.1 Original Contributions

This thesis presents a new, flexible MILP formulation that can be used to solve scheduling problems with a variety of parameters in the context of a multi-agent problem-solving framework. MILP techniques were chosen to model scheduling problems because of the intuitive nature of modeling these problems as a set of constraints with an objective function, and because these techniques produce optimal solutions. A significant amount of work was done developing and optimizing this MILP formulation by reducing the number of binary variables and redundant constraints used in the model.

While the scheduling algorithm presented in this thesis is not suited to solve problems with large numbers of agents and jobs, it can be used as an effective alternative to existing heuristics if exact solutions are desired for complex classes of scheduling problems that have small numbers of agents or jobs. The advantages and disadvantages to using this MILP approach are listed below:

Advantages:

- The MILP approach presented in this thesis offers more flexibility than most existing scheduling algorithms and heuristics. In addition to being able to solve the most complex class of scheduling problems, R|PREC,r_j|C_{max}|JP, it can also solve any simpler class of problems well. By taking into account communication delay based on bandwidth between agents and the amount of data transmitted by each job as well as execution delays on each agent, release times, and being able to model any form of directed, acyclic job precedence graph, the algorithm can be used in a wide variety of problem domains.
- The algorithm achieves completely optimal makespans for multi-agent scheduling problems, unlike many heuristics that produce an approximate solution within a certain bound of the true optimal value.

- In certain instances where there is an agent command hierarchy and a job decomposition tree that has smaller granularity jobs at each successive level, large-scale scheduling problems can be broken down into sets of smaller ones, and the MILP scheduling algorithm can be applied recursively to produce optimal solutions in a shorter amount of time.
- Using an MILP formulation to model the scheduling problem makes it possible to easily change objective functions or to use Multiple Objective Linear Programming (MOLP) techniques to achieve multiple objectives at once. Minimizing the average weighted completion time of jobs, the number of agents or amount of bandwidth used, or the aggregate cost of agent/job allocations may be achieved in addition to minimizing the schedule's makespan.
- The MILP formulation presented in this thesis uses continuous-time instead of discrete-time, thus reducing the number of binary variables needed to model the problem and allowing execution and communication delays to assume any real-number value, as well as creating an unbounded time horizon for the schedule.

Disadvantages:

- The main drawback to using an MILP algorithm is that because it runs in exponential time it is only feasible to use on smaller-scale problems, where there are a limited number of agents and jobs.
- Because the algorithm is static rather than dynamic, all information about the agent network must be known *a priori* to solving the problem. Also, it is unable to adapt the schedule to dynamic changes in the agent network or job precedence graph during execution of the schedule itself; if a change is made the entire schedule must be re-computed based on the new input data.

6.2 Future Work

Extensions to the work done in this thesis may include exploring a dynamic programming approach to the same problem that is capable of taking into account uncertainty levels or

a limited view of the agent environment, and can adapt to unexpected changes in the environment or problem domain during execution of the actual schedule. Other future areas of research related to this thesis may also include dynamically re-organizing teams of agents that respond to changing objectives and environments.

Problem formulation and decomposition techniques are also introduced in this thesis, but no formula is developed to compute optimal job decompositions. Optimization techniques to determine the best way to formulate and decompose a problem for a given agent organization is another promising research avenue that may be pursued based on the work done in this thesis, although in many cases the optimal decomposition seems to vary depending on the nature of each individual problem.

Possible future extensions to the scheduling problem model itself and the MILP formulation include allowing agents the ability to perform multiple jobs simultaneously, and taking into account an agent's location and movement, allowing the scheduling technique to be applied to vehicle scheduling and routing problems.

Additionally, it may be helpful to perform additional studies on the convergence rate of the MILP technique relative to the number of agents and jobs in the problem, so that approximate computation times may be predicted in advance and the Branch and Bound algorithm may be stopped earlier with a near-optimal solution for large-scale problems.

It is important to note that while much research has been done on the complexity of multi-agent scheduling problems and heuristics for solving them, considerable work remains to develop an adaptive, flexible algorithm that is capable of taking into account the numerous intricacies and details of today's real-world scheduling problems and finding their optimal solutions.

Appendix A: MILP Formulation

Minimize: $C_{\max}$

Subject to:

$$\left.\begin{array}{l} C_{\max} \geq S_j + D^{j,a} * X_{j,a} \\ X_{j,a} \in \{0,1\} \end{array}\right\} \qquad \forall j \in J, a \in A \,\middle|\, B^{j,a} > 0$$

$$X_{j,a} = 0 \qquad \forall j \in J, a \in A \,\middle|\, B^{j,a} = 0$$

$$\sum_{a \in A} X_{j,a} = 1 \qquad \forall j \in J$$

$$\left.\begin{array}{l} S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j \geq -M * \theta_{j,k} \\ S_k - \sum_{a \in A} D^{j,a} * X_{j,a} - S_j < M * (1 - \theta_{j,k}) \\ \theta_{j,k} \in \{0,1\} \end{array}\right\} \qquad \forall (j,k) \in J \,\middle|\, j \neq k, Q^{j,k} = 0$$

$$X_{j,a} + X_{k,a} + \theta_{j,k} + \theta_{k,j} \leq 3 \qquad \forall (j,k) \in J, a \in A$$

$$\left.\begin{array}{l} S_k \geq S_j \\ S_k \geq S_j + (D^{j,a} + C^{j,a,b}) * (X_{j,a} + X_{k,b} - 1) \end{array}\right\} \qquad \forall (j,k) \in J, (a,b) \in A \,\middle|\, j \neq k, P^{j,k} > 0, B^{j,a} > 0, B^{k,b} > 0$$

$$S_j \geq R^j \qquad \forall j \in J \,\middle|\, R^j > 0$$

-This Page Intentionally Left Blank-

Appendix B: XpressMP™ Optimization Software

The Mixed Integer-Linear Programming formulations presented in this thesis were modeled and solved using Dash Optimization's XpressMP™ software. The software by default uses Dual Simplex and Branch and Bound techniques to solve MILP problems, and comes with its own proprietary modeling language, MOSEL. Using MOSEL it is possible to model constraints, decision variables, and logical conditions without having to explicitly construct a matrix of constraint coefficients.

B.1 Input File Format

Before the XpressMP™ Optimizer can be used to solve an MILP scheduling problem, input parameters that define the agent organization and job precedence tree must be read from a text file, such as the one shown in Figure B-1. This file contains the values for the variables $B^{j,a}$, $D^{j,a}$, $P^{j,k}$, $Q^{j,k}$, and $C^{j,a,b}$ (or $bandwidth^{a,b}$ and $bits^{j}$ if $C^{j,a,b}$ is defined by these values) and are modeled as matrices. If any of these parameters does not apply to a particular problem, these matrices can be changed to reflect that. For example, if the execution time for a job depended only on the agent performing it and not the job itself, each row of $D^{j,a}$ would be identical. Similarly, if communication delay depended only on the number of bits being transmitted, the $bandwidth^{a,b}$ matrix may contain all entries of 1.

B.2 MOSEL Programming Language

MOSEL is a language designed specifically to solve Mathematical Programming problems. Using MOSEL, MILP problems can be represented using linear constraints and objective functions, just as they are in Chapter 4. The program then takes these constraints, and forms a *problem matrix*, where each row represents a constraint, each column represents a decision variable, and the matrix entries are the coefficients to these variables. If none of the constraints or objective function uses inequalities such as $\leq$ or $\geq$, the entire problem can be solved using basic matrix solution techniques, treating the

```
! total number of jobs  agents  and times
NJ: 4
NT: 12
NA: 3
! Durations of jobs on each agent
! D(i,j) is the execution time of job i on agent j
D:    [1   3.2 2.8
       4   2   3
       2   3.5 0.5
       1.6 2   1.1]
! agents capable of performing each job
! B(i,j) = 1 means Job i can be completed by Agent j
B:    [1 1 0
       1 1 1
       0 1 1
       1 1 0]
! Bandwidth between pairs of agents
BAND: [1000 30  20
       30  1000 50
       20    50  1000]
! Number of bits transmitted as a result of each job
BITS: [10 32 20 0]
! precedence constraints for jobs
! P(i j) means job i comes before job j
P:    [0 1 1 0
       0 0 0 1
       0 0 0 1
       0 0 0 0]
! full prec. tree: Q(i j) means job i comes at some point
before job j
Q:    [0 1 1 1
       0 0 0 1
       0 0 0 1
       0 0 0 0]
```

Figure B-1: An example input file for a scheduling problem with 3 agents and 4 jobs

problem as $Ax = b$, where A is the constraint coefficient matrix, x is a vector of decision variables, and b is the objective function vector. However, the MILP formulation presented in this thesis, along with most other existing LP problems, contains inequality constraints, thus an algorithm such as the Simplex method must be used to obtain the optimal solution of each LP relaxation.

An example of the MOSEL code used to solve the MILP formulation in Chapter 4 is shown in Figure B-2. MOSEL allows the user to define sets of constraints based on certain logical conditions, but it also contains basic programming constructs such as *for* and *while* loops, and *if/then* statements. While most of the constraints listed in Chapter 4 can be translated directly to MOSEL code, there are a few that must be represented differently. For example, the constraint

$$Y_{j,a,k,b} = X_{j,a} * X_{k,b} \tag{4-13}$$

cannot simply be represented by the MOSEL code

```
forall(j in J, k in J, a in A, b in a) do
      Y(j,a,k,b) = X(j,a)*X(k,b)
end-do
```

Multiplying $X_{j,a} * X_{k,b}$ to produce $Y_{j,a,k,b}$ turns the problem into a *quadratic programming* problem that would require more complicated methods to solve, so the following auxiliary constraints are used instead to convey the same meaning.

```
forall(j in J, k in J, a in A, b in a) do
      Y(j,a,k,b) >= X(j,a) + X(k,b) - 1
      Y(j,a,k,b) <= X(j,a)
      Y(j,a,k,b) <= X(k,b)
      Y(j,a,k,b) >= 0
end-do
```

Because having these extra constraints adds to the overhead in solving the problem, it is advantageous to avoid using extra binary decision variables such as $Y_{j,a,k,b}$ if possible.

```
...
...
! define objective function z
forall(j in J, a in A | B(j,a)>0) do
      Cmax >= S(j) + D(j,a)*X(j,a)
      X(j,a) is_binary
end-do

! a job can only be assigned to an agent capable of doing it
forall(j in J, a in A | B(j,a) = 0) do
      X(j,a) = 0
end-do

! each job is done once on one agent
forall(j in J) do
      SUM(a in A) X(j,a) = 1
end-do

! an agent may execute at most one job at a time
forall(j in J, k in J | (j<>k AND Q(j,k)=0)) do

      S(k) - (S(j) + SUM(a in A) (X(j,a)*D(j,a))) >= -M*O(j,k)
      S(k) - (S(j) + SUM(a in A) (X(j,a)*D(j,a))) <= M*(1 - O(j,k)) - R
      O(j,k) is_binary
end-do
forall(j in J, k in J, a in A) do
      O(j,k) + O(k,j) + X(j,a) + X(k,a) <= 3
end-do

! a job cannot start until its predecessors are completed
forall(j in J, k in J, a in A, b in A | (j<>k AND P(j,k)>0 AND B(j,a)>0
AND B(k,b)>0)) do
      (S(k) - S(j))/(D(j,a) + C(j,a,b)) >= (X(j,a) + X(k,b) - 1)
      (S(k) - S(j))/(D(j,a) + C(j,a,b)) >= 0
end-do

! objective function
      minimize(Cmax)
...
...
```

Figure B-2: MOSEL code modeling the MILP constraints for the multi-agent scheduling problem.

B.3 Xpress™ Optimizer

Once a Mathematical Programming formulation is compiled, the XpressMP™ Optimizer is used to compute the optimal solution to the problem. After the problem matrix has been constructed, a *presolve* technique is performed which is used to eliminate redundant constraints and variables. In the case of MILP problems, bound tightening and coefficient tightening are applied to tighten the LP relaxation as part of the presolve. For example, if the matrix has the binary variable x and one of the constraints is $x <= 0.2$, presolve will automatically fix x at 0 because it can never take the value 1. Presolve produces a new, smaller problem matrix which is then used to solve the LP relaxation of the MILP problem.

Using these LP relaxations and the Branch and Bound technique, the XpressMP™ Optimizer solves for the optimal integer solution by tightening its upper bound (the best integer solution found) and the lower bound (found using the LP relaxation) until these bounds meet, as in Figure B-3. It should be noted that while solutions are usually consistent, the Optimizer sometimes chooses to pursue different paths in the Branch and Bound tree for different instances of the same problem model, resulting in different computation times and numbers of nodes traversed when the model is run multiple times.

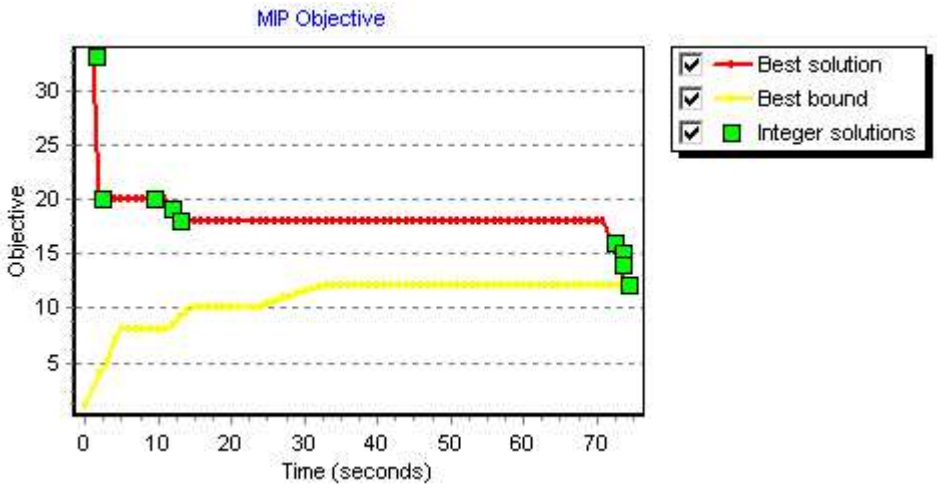

Figure B-3: Output from Xpress-IVE showing the upper and lower bounds of the optimal integer solution to an example MILP scheduling problem and the time taken to find this solution.

B.4 Graphing Functionality

XpressIVE™, the programming interface used to model problems in MOSEL and display solutions, features graphing functionality which was very useful for displaying optimally computed schedules in the form of a Gantt chart. In these charts, the x-axis is used to represent time in the schedule, and the y-axis represents each individual agent. Each horizontal numbered arrow in the graph represents a different job and the time needed to execute that job, and the arrows that connect these jobs to each other represent the communication of job data between agents, and the delay caused by this. Figure B-4 shows an example of a schedule computed in a scenario where there are 4 agents and 16 jobs that are executed in parallel.

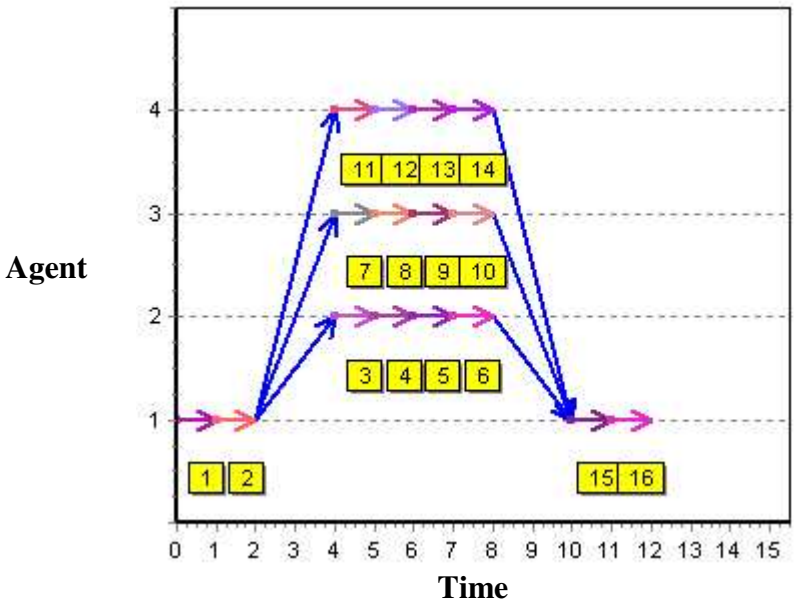

Figure B-4: A Gantt chart produced using the graphing function in Xpress-IVE™

In addition to this graphical solution, Xpress-IVE™ also displays the total number of constraints and decision variables before and after presolve, the number of Simplex iterations needed to solve the LP relaxation, the number of nodes traversed during the Branch and Bound process, and the time needed to solve the problem to optimality. This data is useful for determining the limits to the problem size that could be solved in a reasonable amount of time, and for comparing different agent and job configurations to each other.

Appendix C: Agent/Service Discovery

In order to optimally solve a multi-agent scheduling problem using the algorithm presented in this thesis, it is necessary to first have full information about the agent environment and the resources that are available to solve the problem. Therefore, a planning agent must discover the structure of the physical agent network, including all services offered by various agents, their execution times, and communication delays before determining how to allocate jobs to agents. Considerable research has been done on inter-agent communication and agent discovery techniques that can help an agent to formulate a scheduling problem [6].

C.1 Agent Directories

Because the infrastructure of an agent environment is often dynamic, with agents entering and leaving at random intervals, it is helpful to have a central system set up that keeps track of which agents are available at any given time. This data repository may be used as a reference for agents entering the network that wish to collect data about other available agents and services, or to provide updates for agents already in the network. One such method developed by Decker, Sycara, and Williamson [6] is to use *middle agents*, whose sole purpose is to collect data about available resources in the network and advertise this data to any agent that requests it. This data consists of a directory of *white pages* and *yellow pages*. White pages are simply a listing of all agents currently in the network, and the address used to identify and communicate with that agent, such as an IP address. White pages may contain additional information such as available bandwidth between each pair of connected agents. Yellow pages are a directory of services provided by all agents in the network, and may contain a reference to the corresponding white pages entry for each agent, and possibly a listing of execution delays for each service. White pages and yellow pages may be kept up-to-date by having the middle agent send multicast requests and periodic pings to agents in the network. By using both white page and yellow page services, agents can keep track of the services offered by other agents and communication and execution delays for each of these services.

C.2 Finding the Shortest Path Between Agents

An agent may be able to determine the available bandwidth and communication delay between agents that are directly connected to it in the network simply by querying these agents directly. However, in order to compute an optimal schedule it is necessary to know shortest or least-costly path between all pairs of agents in the network, including those not directly connected to each other. In other words, given an *adjacency matrix* of communication delays between connected agents, an agent needs to compute a *shortest-path matrix* between all pairs of agents in the network. Figure C-1 shows the adjacency and shortest-path matrices for the network described in Chapter 3, Figure 3-5 below:

	A	*B*	*C*	*D*	*E*	*F*	*G*
A	0	2		2		4	
B	2	0	3				
C		3	0	2			
D	2		2	0	2		5
E				2	0		
F	4					0	3
G				5		3	0

(a)

$\Rightarrow$

	A	*B*	*C*	*D*	*E*	*F*	*G*
A	0	2	3	2	4	4	7
B	2	0	3	3	5	6	8
C	3	3	0	2	4	7	7
D	2	3	2	0	2	6	5
E	4	5	4	2	0	7	7
F	4	6	7	6	7	0	3
G	7	8	7	5	7	3	0

(b)

Figure C-1: Adjacency matrix (a) and shortest-path matrix (b) showing the communication delay between agents for the network in Chapter 3, Figure 3-5.

Fortunately, efficient algorithms are available for computing the shortest-path matrix from an adjacency matrix. The most widely known of these is Dijkstra's Algorithm [18]. Given a graph G with a set of nodes *N* and a set of edges *E*, each edge with a cost w_e, Dijkstra's Algorithm computes the shortest path (the path with lowest total cost) from a starting node $s \in N$ to every other node in *N*.

Dijkstra's algorithm works by keeping track of an array *A* of best-known estimates for each node's distance from the starting node, and a *priority queue* of these nodes ordered by this estimated distance. The algorithm starts by looking at the starting node and all of

the nodes directly adjacent to it. The adjacent node closest to the starting node is removed from the front of the priority queue, and the nodes adjacent to that node are examined and *A* is updated to reflect any shorter paths that are found. This process is repeated, each time removing the node with the closest estimated distance to the starting node, until there are no nodes left. The distance calculated in *A* for each node that is removed from the priority queue is the shortest path between it and the starting node. A pseudo-code representation of Dijkstra's Algorithm is shown below, derived from Morris [18]:

```
initialize_single_source(Graph g, Node s) {
      for each node n in Nodes(g)
            g.distance[n] = infinity
            g.pi[n] = nil
      g.distance[s] = 0;
}

relax(Node u, Node v, cost[][]) {
      if distance[v] > distance[u] + cost[u][v] then
            distance[v] = distance[u] + cost[u][v]
            pi[v] = u

}

dijkstra(Graph g, Node s) {

      initialize_single_source(g,s)
      S = {0}
      Q = Vertices(g) // priority queue
      while Q is not empty
            u = ExtractCheapest(Q);
            add node u to S;
            for each vertex v adjacent to u
                  relax(u,v,cost[u][v])

}
```

The most efficient way to implement Dijkstra's Algorithm is to use adjacency lists to store the graph and to use a heap as a priority queue. If the graph has m edges and n nodes, then the algorithm runs $O((m+n)\log n)$ time. Because for the agent networks presented in this thesis we want to find the distance from every node to every other node though, this algorithm must be run n times and therefore the total running time is $O((m+n)*n\log n)$. Once the complete shortest-path matrix has been computed, these communication delays may be used to compute the optimal job allocation among agents in the network.

-This Page Intentionally Left Blank-

References

[1] Baxter, J.W., Hepplewhite, R.T. "A Hierarchical Distributed Planning Framework for Simulated Battlefield Entities," Defense Evaluation and Research Agency, Worcestershire, UK, 2000.
URL: http://mcs.open.ac.uk/plansig2000/Papers/P7.pdf [4.11.2003]

[2] Bellman, R. *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[3] Bradley, S., Hax, A., Magnanti, T. *Applied Mathematical Programming*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1977.

[4] Chekuri, C., Bender, M. "An Efficient Approximation Algorithm for Minimizing Makespan on Uniformly Related Machines". *In Proceedings of the 6th Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 383-393. 1998.
URL: http://cm.bell-labs.com/cm/cs/who/chekuri/postscript/related_sched.ps [10.15.2002]

[5] Cook, S. A. "The complexity of theorem proving procedures," *In Proceedings, Third Annual ACM Symposium on the Theory of Computing*, ACM, New York, pp.151-158, 1971.

[6] Decker, K., Sycara, K., and Williamson, M., "Middle Agents for the Internet," *In Proceedings of IJCAI97,* Nagoya, Japan, 1997.
URL: http://citeseer.nj.nec.com/decker97middleagents.html [8.22.2002]

[7] Engels, D., Feldman, J., Karger, D., Ruhl, M. "Parallel Processor Scheduling with Delay Constraints," MIT Laboratory for Computer Science, 2001.
URL: http://theory.lcs.mit.edu/~karger/Papers/2001-soda-1.ps [11.13.2002]

[8] Goldwasser, S., Bellare, M. "The Complexity of Decision Versus Search," *In SIAM Journal on Computing*, Vol. 23, No. 1, 1994.
URL: http://citeseer.nj.nec.com/523838.html [5.4.2003]

[9] Graham, R.L. "Bounds for Certain Multiprocessing Anomalies," *Bell System Tech. J* 45 pp.1563-1581, 1966.
URL: http://citeseer.nj.nec.com/context/55748/0 [2.12.2003]

[10] Hall, W., Adams, M. "Closed Loop, Hierarchical Control of Military Air Operations," DARPA JFACC Program, CSDL-R-2914, Charles Stark Draper Laboratory, Cambridge, MA, 2001.

[11] Kan, A.H.G. Rinooy. "Machine Scheduling Problems (Classification, complexity and computations)". Nartinus Nifhoff, The Hague, 1976.
URL: http://citeseer.nj.nec.com/context/734382/0 [2.12.2003]

[12] Karp, R.M. "Reducibility Among Combinatorial Problems," in R.E. Miller and J.W. Thatcher, eds., *Complexity of Computer Computations*, Plenum Press, New York, 1972.
URL: http://citeseer.nj.nec.com/context/22534/0 [10.11.2002]

[13] Kohler, Walter H. "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems". *IEEE Trans. On Computers*, 24(12):1235-1238, 1975

[14] Kwok, Yu-Kwong and Ishfaq, Ahmad. "Dynamic Critical-Path Scheduling: An Effectuve Technique for Allocating Task graphs to Multiprocessors". *IEEE Trans. On Parallel and Distributed Systems*, Vol. 7, p. 506, 1996.
URL:
http://citeseer.nj.nec.com/cache/papers/cs/11712/http:zSzzSzwww.eee.hku.hkzSz~ykwokzSzpaperszSzdcp-tpds.pdf/kwok96dynamic.pdf

[15] Lawler, E.L., Lenstra, J.K., Kan, A.H.G. Rinnooy. "Recent Developments in Deterministic Sequencing and Scheduling: A Survey", Reprinted from M.A.H. Dempster (eds.), *Deterministic and Stochastic Scheduling*, 35-73, D. Reidel Publishing Co., 1982.

[16] Liao, Guoning. "A Comparative Study of DSP Multiprocessor List Scheduling Heuristics", ACAPS Laboratory, McGill School of Computer Science, 1993.
URL:
http://citeseer.nj.nec.com/cache/papers/cs/754/ftp:zSzzSzftp.capsl.udel.eduzSzpubzSzdoczSzacapszSzmemoszSzmemo63.pdf/liao93comparative.pdf [2.5.2003]

[17] McDowell, Mark E. "Multiprocessor Scheduling in the Presence of Communication Delay". Master of Science Thesis. MIT, Dept. of Elec. Engineering and Comp. Science, Boston, 1989.

[18] Morris, John. *Data Structures and Algorithms*, Chapter 10.2, 1998
URL: http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/dijkstra.html
[8.20.2002]

[19] Ragsdale, Cliff T. *Spreadsheet Modeling and Decision Analysis*, Chapter 7, South-Western College Publishing, Cincinnati, Ohio, 1998.

[20] Rifkin, S., "When the project absolutely must get done: Marrying the organization chart with the precedence diagram". *Master Systems, Inc.,* McLean, VA.
URL: http://www.vite.com/products/Rifkin.pdf [4.15.2003]

[21] Sih, Gilbert C. and Lee, Edward A. "A Compile-Time Scheduling Heuristic for Interconnection-constrained Heterogeneous Processor Architectures". *IEEE Trans. On Parallel and Distributed Systems*, 4(2):175-187, 1993.
URL: http://www.computer.org/tpds/td1993/l0175abs.htm [3.22.2003]

[22] Yu, M.Y., Gajski, D.D. "Hypertool: A Programming Aid for Message-Passing Systems", *IEEE Trans. On Parallel and Distributed Systems*, Vol. 1, No. 3, p. 330-343, 1990.
URL: http://www.computer.org/tpds/td1990/l0330abs.htm [3.22.2003]