Coordinating Autonomous Planning Agents

MASTER'S THESIS — ADRIAAN TER MORS

Coordinating Autonomous Planning Agents

master's thesis — Adriaan ter Mors

Committee:

Prof. dr. ir. H.J. Sips
dr. C. Witteveen
ir. H.J.A.M Geers
ir. J.M. Valk

Preface

Sooner or later, but probably sooner, every computer scientist must face the question: *When will the project be finished?* More often than not, the answer to that question, though sincerely given, will turn out to be a lie. My intial answer to the question when I thought to finish my graduation project was September — of 2003, that is.

Surely, I thought when I was ready to start writing this report in the early summer of 2003 — about a year after I had approached Cees Witteveen with the question whether there were any graduation projects within the Collective Agent Based Systems (CABS) research group — *I won't need more than a month or two to write my report?* Since then, however, I have discovered that most of the actual research work is done while writing the report.

Still, the ten months it finally took is a long time to spend on a single report. One excuse is that in the meantime, me and my supervisors have written a number of research papers on the subject, at least one of which has been accepted for publication. A second reason why it took me so long to finish is that the size of the report has gotten rather out of hand, even though I endeavored to write concisely and (mostly) to the point. Fortunately, to gain an understanding of the research presented in this report, it is not necessary to read the entire report page-to-page. Chapters 3 and 4, and the second half of Chapter 5 — together comprising the more technical parts of this report, containing e.g. details of algorithms — require only a glance to keep with the main account (reading only introductory sections of those chapters should suffice).

This report was not written by me locking myself in my room and reappearing 20 months later with the finished report. Rather, it has been a collaboration between me and my supervisors. Therefore I would like to thank Jeroen Valk for his boundless enthusiasm on multi-modal logistic problems and autonomous planning agents, and Cees Witteveen for general guidance and support, and also for furthering the research by presenting familiar problems from a different point of view.

Delft, April 2004

Abstract

We consider the problem of coordinating autonomous agents that have to achieve a *joint task*, consisting of a set of *(elementary) tasks*, partially ordered by a set of precedence constraints. Each agent is allocated a non-overlapping subset of tasks, for which it needs to make a plan. Precedences may exist between tasks allocated to the same agent (*intra-agent* precedences), or between tasks allocated to different agents (*inter-agent* precedences). Because of the latter set of constraints, agents are dependent on each other, and coordination is required for successful joint operation.

We assume that an agent needs to make a *plan* to execute its subset of tasks. To guarantee an agent full autonomy in the planning of its tasks, we require that planning and coordination be separated. In the literature on multi-agent coordination, no approach yet exists where agents can (*i*) work together on a joint task, and (*ii*) where planning and coordination are separated. Therefore, we present a new *pre-planning* coordination framework in which agents receive a set of additional constraints prior to planning, such that subsequently the joint plan will always be feasible, regardless of the plans produced by the individual agents.

The *coordination problem*, which is to find a minimal set of additional constraints such that a feasible joint plan is guaranteed, is computationally hard. The problem of verifying whether a given set of additional constraints is a coordination solution (the *coordination verification problem*) is co-NP-complete. The coordination problem itself is Σ_2^p-complete. Finally, we can show that it is highly unlikely that constant-ratio approximation algorithms exist for the coordination problem.

Nevertheless, we have designed approximation algorithms that return very efficient solutions when applied to the multi-modal logistics problem (which consists of delivering packages between and within cities). In fact, our pre-planning coordination algorithms (*i*) outperform state-of-the-art (multi-agent) planning systems on a set of benchmark logistic problems, and (*ii*) show that by efficiently separating planning from coordination, existing (single-agent) planning tools can be reused to solve multi-agent planning problems.

List of Tables

5.1 Transformation of a logistics instance into a composite task. 61
5.2 For every possible coordination set an optimal visiting sequence and the cost of the coordination set. 65
5.3 Tasks and visiting sequences for a one-order n-city instance. 66
5.4 The location of the agents at times $time = 0$ and $time = 1$. 66
5.5 The location of the agents at times $time = 0$ up to $time = 3$. 67

6.1 AIPS dataset: number of splits generated by each policy. 80
6.2 AIPS dataset: number of moves generated by each policy. 81
6.3 A dataset of random logistic instances. 83
6.4 Random dataset: number of splits generated by each policy. 86
6.5 Random dataset: number of moves generated by each policy. 87
6.6 Comparing the output of planning systems with and without making use of pre-planning coordination. 91

List of Figures

1.1 Different vehicles and their capabilities to transport packages across Europe. 4

2.1 Parcels must be delivered between locations; a package can be transferred from one truck to another at the depot. 12
2.2 For both agents, executing the post-depot task before the pre-depot task is a feasible refinement with regard to their local goals. 13

3.1 Only tasks t_1 and t_2 are in T_{inter}. 17
3.2 (a): There may be an inter-agent cycle if $t_1 \prec t_2$. (b): If $t_2 \prec t_1$, then no cycle can pass through the agent. 18
3.3 Despite the presence of an inter-agent cycle, this instance is coordinated. . . 20
3.4 (a): A refinement cycle in $G_{\mathcal{T}}$. (b): If r_2 and r_3 are added to E, then $r_1^{-1} \in \mathit{INTRA}$. 23
3.5 The subgraph constructed for a forbidden pair $\{(x, y), (u, v)\} \in C$. 25
3.6 (a): A PWFP instance with a single forbidden pair and (b): the corresponding coordination instance. 26
3.7 The subgraph containing a forced-choice gadget, constructed for a forbidden pair $c_j = \{(x^j, y^j), (u^j, v^j)\} \in C_1$. 28
3.8 (a) an FVS instance and (b) its corresponding CP instance, coordinated by adding constraint (d_2, d_1). 33

4.1 (a): Unconstrained, (b): for $\Delta = \{\delta_1, \delta_2\}$, $E \cup \Delta$ is cyclic. 36
4.2 (a): Unconstrained, (b): with new constraint set $[E \cup (t_2, t_7)]^+$. 36
4.3 (a): Unconstrained, (b): the set REF is empty after adding $t_2 \prec t_7$. 39
4.4 (a): Uncoordinated instance and (b): after agent A_2 adds constraint δ. . . . 40
4.5 The outer cycle is not covered in the feedback arc set. 45
4.6 Despite the presence of an inter-agent cycle, this instance is coordinated. . . 45
4.7 If task t is used to form to distinct cycles, then t can be reached from itself. 48
4.8 (a): $G_{\mathcal{T}}$ and (b): D_A for a the same (coordinated) instance. 49
4.9 (a): $G_{\mathcal{T}}$, (b): coordinated D_A and (c): corresponding $G_{\mathcal{T}}$. 50

5.1 Infrastructure of a two-city logistics instance. 60
5.2 There is refinement cycle in the coordination graph of Example 5.2.1. . . . 61
5.3 An order-graph: three locations and the orders between them. 64
5.4 Four tasks and the possible constraints between them. 64

6.1 The computational steps performed to solve a logistics instance. 72
6.2 The joint-plan cost, in terms of the number of moves, produced by each of the three policies. 75
6.3 The joint-plan cost in terms of the number of actions (move, load and unload actions), produced by Policy 1 and by TALplanner. 76
6.4 The number of truck splits generated per instance. 77
6.5 The number of plane splits generated per instance. 77
6.6 The amount of overhead generated by Policy 3 per agent: data points are pairs < number splits, overhead > per agent per instance. 79
6.7 The amount of overhead generated by Policy 3 for the *plane agent*: data points are pairs < number splits, overhead > for the plane agent per instance. 79
6.8 A one-city truck goal for which Arbiter0 is inefficient. 81
6.9 The number of truck splits generated per instance. 84
6.10 The number of plane splits generated per instance. 84
6.11 The overhead generated per instance. 85
6.12 The amount of overhead generated by Policy 3 per agent: data points are pairs < number splits, overhead > per agent per instance. 85
6.13 The amount of overhead generated by Policy 3 for the *plane agent*: data points are pairs < number splits, overhead > for the plane agent per instance. 86
6.14 CPU times: coordinated STAN vs. uncoordinated STAN, for track 1 (additional instances) of the AIPS logistics dataset. 89
6.15 Number of actions produced by coordinated HSP vs. uncoordinated HSP, for track 1 (additional instances) of the AIPS logistics dataset. 89
6.16 Percentage of time or actions saved by applying pre-planning coordination to STAN and HSP. 90

A.1 A set of orders and the order-graph 96
A.2 (a): Directed graph G, (b): directed graph G/F, i.e., the graph for which each vertex in F is split in two. 96

C.1 The graph G associated with a three-clause QSAT_2 instance. 116

Contents

1 Introduction **1**
- 1.1 Autonomous Agents . . . 1
- 1.2 Autonomy and the Need for Coordination . . . 2
- 1.3 Coordinating Multiple Agents . . . 3
- 1.4 Contributions of this Research . . . 5

2 A Task-Oriented Coordination Framework **7**
- 2.1 Problem Statement . . . 8
- 2.2 A Task Framework . . . 9
  - 2.2.1 Task allocation and agent goals . . . 9
  - 2.2.2 Plans, goals, and refinements . . . 10
  - 2.2.3 The joint plan . . . 11
- 2.3 The Coordination Problem . . . 12

3 Analyzing the Coordination Problem **15**
- 3.1 The Coordination Graph . . . 16
  - 3.1.1 Tasks and the coordination problem . . . 16
  - 3.1.2 Precedences and the coordination problem . . . 17
  - 3.1.3 Refinements and the coordination problem . . . 18
  - 3.1.4 Definition of the coordination graph . . . 19
- 3.2 The Coordination Graph and the Coordination Problem . . . 20
  - 3.2.1 Inter-agent cycles . . . 20
- 3.3 Constructing a Coordination Set . . . 21
- 3.4 Complexity of the Coordination Problem . . . 23
  - 3.4.1 Complexity of coordination verification . . . 24
  - 3.4.2 Complexity of the coordination problem . . . 26
  - 3.4.3 Subclasses in NP . . . 30
  - 3.4.4 Approximability of the coordination problem . . . 32

4 Approximation Techniques **35**
- 4.1 Iteratively Constructing a Coordination Set . . . 36
  - 4.1.1 Adapting iterative coordination to enable negotiation . . . 39
- 4.2 Coordination and Feedback Arc Sets . . . 42
  - 4.2.1 Negotiation and BSFAS coordination . . . 46
- 4.3 Partitioning Local Goals . . . 48
  - 4.3.1 Agent dependencies . . . 49

4.3.2 A coordination algorithm . . . 50
4.3.3 Partitioning strategies . . . 53
4.3.4 Negotiation in partitioning coordination . . . 54
4.4 Concluding Remarks . . . 56

5 Applying the Framework to a Planning Problem **57**
5.1 The Logistics Problem . . . 58
5.2 Identifying Tasks for a Logistics Problem . . . 59
5.3 A Model for Plan Cost . . . 61
5.3.1 Concrete plans in the logistics domain . . . 62
5.3.2 The Cost of Precedence Constraints . . . 63
5.3.3 Coordinating with minimal cost . . . 65
5.4 The Joint Plan . . . 66
5.4.1 Autonomy-preserving scheduling . . . 67

6 Empirical Results **71**
6.1 Test Setup . . . 71
6.1.1 Hypotheses . . . 74
6.2 Running the AIPS Dataset . . . 74
6.2.1 Results . . . 74
6.2.2 Discussion of AIPS results . . . 80
6.3 A Random Dataset . . . 82
6.3.1 Interpretation of random dataset results . . . 86
6.4 Reuse of Existing Planners using Coordination . . . 88
6.5 Concluding Remarks . . . 92

7 Conclusions and Future Work **93**

A Solving the One-City Problem **95**

B Source Code for Partitioning **99**
B.1 The Protocol . . . 99
B.2 Agent Strategy . . . 107

C Complexity of Quantified PWFP **115**

Chapter 1

Introduction

> *Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.* — Isaac Asimov

1.1 Autonomous Agents

Foolishly, I thought my curt reply would be self-explanatory. However, when I answered: "multi-agent coordination" in response a friend's question "So what do you do at the university?", I only received a blank look in return; perhaps a courteous nod, at the most. Neither did any further words from me seem to enlighten, until I gave in to the temptation to illustrate the concept of an intelligent agent by mentioning the Terminator movies. Finally, recognition dawned.

Of course, typical multi-agent research and multi-agent applications are much more mundane than huge robots stomping around in a post-apocalyptic wasteland. Yet this is the kind of image that clings to the research area of artificial intelligence. Even in 1971, as Fikes and Nilsson [10] presented their STRIPS system that would shape the minds of AI researchers, Stanley Kubric's *2001: A Space Odyssey* had already shaped the public image of artificial intelligence three years earlier: that of advanced, untrustworthy, or even malicious computers.

In *A Space Odyssey*, HAL is the board computer of a spaceship. Designed to assist the astronauts, it have been given intelligence — and a mind of its own — to be able to figure out what its duties are and how they should be performed. However, the thinking computer finds a goal of its own, namely to dispose of the crew.

Perhaps surprisingly, the above mentioned science fiction movies have a lot in common with the current research into 'intelligent agents'. Like HAL, an intelligent software agent is designed to assist humans in difficult or tedious tasks. Also, an agent has a certain amount of autonomy that allows it to operate without constant input from the user.

Broadly speaking, there are two differences between the above-mentioned science fiction movies and multi-agent research. First, and unsurprisingly, the visions of Hollywood producers outrageously exceed current (and near-future) capabilities of artificial intelligence technology. The second difference is more subtle and has to do with agent autonomy.

Literally, the word 'autonomous' means self-ruling; this we interpret as the freedom to choose your own goals. Kubric's HAL takes this freedom very far. Although it as been

given goals during design (to assist the astronauts), it rejects these goals and forms goals of its own. This situation is similar to the relation between man and God. Although God created man, and gave him a set of rules to live by, he also gave him free will, allowing man to do all that God has forbidden.

An agent designer does not bestow his creation, the agent, with free will. An autonomous agent may formulate its goals, but these goals will always contribute to the purpose for which the agent was designed.[1] Thus, we claim that the autonomy of an agent is the freedom to refine its goals. Not giving agents free will makes economic sense: why put a lot of effort into creating something that doesn't do what you want?[2]

An interesting question is whether it is at all possible to create an agent with a free will. In other words, can we create a *conscious* agent (or computer, or other artifact)? This question is a matter of considerable philosophical debate (e.g.[13, 19, 20, 26]). Opponents point out that computers merely manipulate strings of ones and zeros: given a sequence of bits (program plus data), a microprocessor produces some other sequence of bits (output). No conscious understanding takes place at any point in this procedure [19]. Proponents argue that if we were capable of manufacturing a brain, or if we were capable of replacing key components in a living brain with artificial ones, would we not have created artificial consciousness?[3]

Whatever the outcome of this debate (if ever there will be one), for the foreseeable future, computers and agents will do what they are told (programmed) — and *only* what they are told.

1.2 Autonomy and the Need for Coordination

Living in a present-day Western society, we enjoy a fair deal of autonomy in our day-to-day activities: we can go out for a stroll, we can play the music we like. However, living in a society with equally autonomous individuals places restrictions on our autonomy; we can't take a stroll down the fast lane of a highway, and we can't play our favorite music at maximum volume at 2:00 AM in the morning. Thus, the freedom of an individual must be restricted; *coordination* is the process in which the appropriate restrictions are formulated.

In a society of any size, it is e.g. infeasible to negotiate with every car owner over whether or not I should be allowed to walk down the highway this afternoon. Instead, coordinated behaviour is ensured (or at least attempted) by a set of rules, laws and social conventions. These rules aim to achieve coordination while at the same time maximizing the autonomy of the individual.

There is actually some relevance to the above discussion (in case you were wondering whether I was just suffering from a particularly bad case of writer's block): the gist of the above is also applicable to multi-agent systems if we replace the words 'society' and 'individual' with respectively 'multi-agent system' and 'agent'. In particular, the above means to convey the fact that if agents are autonomous in deciding which actions to take,

[1] If the software (the agent) contains bugs, then we take the bugged code as the 'purpose' of the agent, and, furthermore, if a programmer would design an agent specifically to have free will, or intelligence, then we take this to be the agent's purpose.

[2] "Yes alright, no need to rub it in!" – God

[3] Alternatively, we could ask ourselves if we humans are truly conscious [26]. Indeed, does it at times not seem that free will is merely an illusion?

then this autonomy might need to be restricted (i.e., through coordination), if agents are to co-exist if not in harmony, then at least without interfering with each other.

1.3 Coordinating Multiple Agents

There are many reasons why the actions of multiple agents need to be coordinated. This ground has not so much been covered as trampled by introductory chapters of earlier publications in the Distributed Artificial Intelligence (DAI) community. To summarize these reasons: either an agent finds himself in the situation where other agents can hinder him in the accomplishments of his goals (e.g. by physically getting in the way), or an agent may need other agents to be able to perform his tasks (e.g., some tasks may require the capabilities of more than one agent). In either case, the behaviour of other agents must be taken into account; or, in other words, agents need to coordinate their activities. How other-agent behaviour must be taken into account depends on the type of agent application.

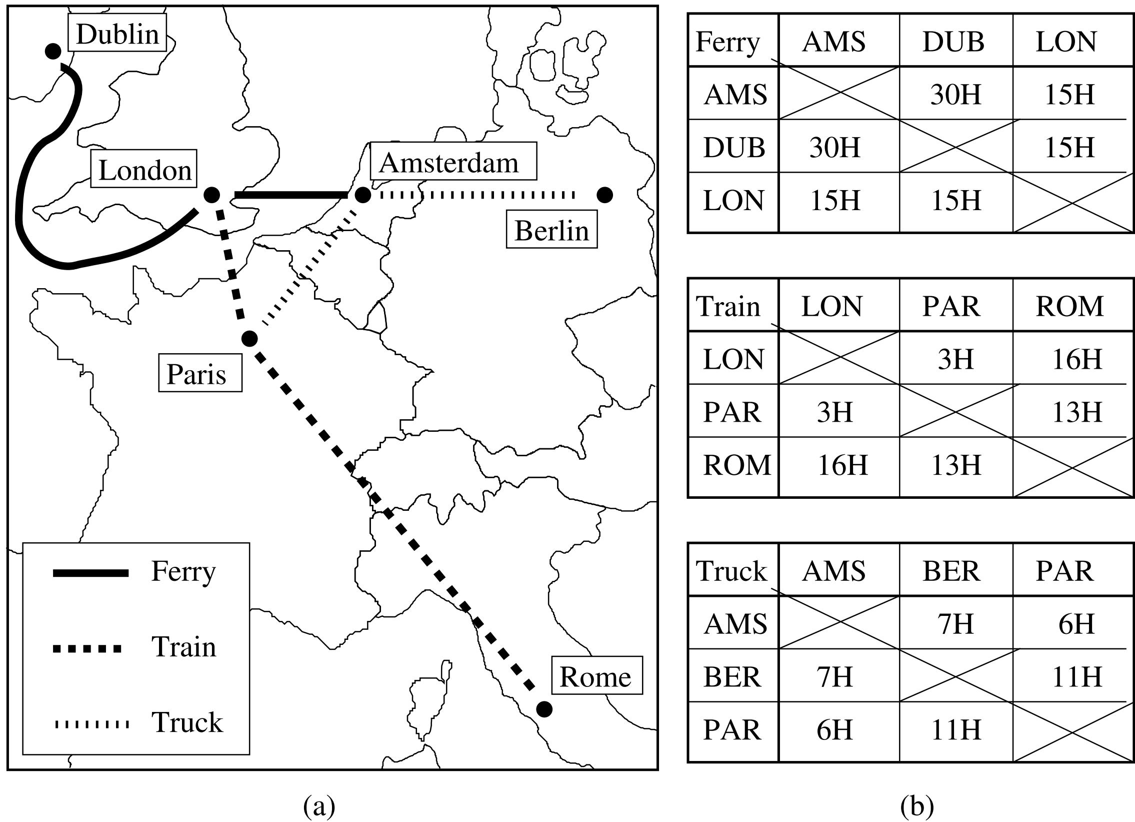
Keeping with the sci-fi theme, consider the example multi-agent system of a number of Mars exploration robots (similar to the work by Steels [23]). Suppose that there are a number of homogeneous explorer robots and a base (agent). The explorer robots collect precious rocks from the surface and return them to the base agent, where they are analyzed. The explorer robots have a certain degree of autonomy: they need to figure out for themselves how to best achieve their goal of bringing back high-quality rocks to the base.

In this example, we can identify several coordination issues. First, the mining robots must avoid looking for rocks in exactly the same spot. Second, they must avoid colliding into each other while moving around on the planet surface. Third, only one robot at a time can unload his rocks at the base.

The Mars explorer robots form what we term an *intra-organizational* multi-agent system. Although all agents are designed to perform their goals as best they can, they will not try to achieve their goals at the expense of other agents, since all agents have been designed by one organization with the purpose of achieving the best result for this organization. This means that certain coordination issues are not relevant for this multi-agent system. For instance, there should be no contention among the explorer robots about who may explore the most promising spot; neither does a rock-laden agent, on his way back to base, have to worry about being ambushed by other agents who are out to steal his precious stones.

In *inter-organizational* multi-agent systems, there are different coordination issues. Suppose that in the above example, each agent works for a different organization. Although agents will probably not be programmed to steal, there certainly will be negotiation about who gets the best spots (assuming of course that certain areas can be identified as promising). In other words, the devision of the ‘spoils’ over the agents is important in inter-organizational multi-agent systems, while it is not of particular interest in intra-organizational ones. In general, the difference between inter- and intra-organizational multi-agent systems is not just limited to whether agents are *self-interested* or not. Agents in different organizations are also less likely to operate in the same space or make use of the same resources.

As an example of an inter-organizational multi-agent system, consider a multi-modal transportation problem where packages must be delivered between locations. In Figure 1.1,



Ferry	AMS	DUB	LON
AMS		30H	15H
DUB	30H		15H
LON	15H	15H	

Train	LON	PAR	ROM
LON		3H	16H
PAR	3H		13H
ROM	16H	13H	

Truck	AMS	BER	PAR
AMS		7H	6H
BER	7H		11H
PAR	6H	11H	

Figure 1.1: *Different vehicles and their capabilities to transport packages across Europe.*

we have three transportation companies, each of which can travel only certain routes. To deliver a package from London to Berlin, two transportation companies need to cooperate, e.g, the Ferry and the Truck. First, the ferry must take the package from London to Amsterdam, then a truck can drive the package from Amsterdam to Berlin.

In this example, we can associate an agent with each transportation company. The goal of an agent is to find a favorable schedule of activities for its transportation company. Coordinating with other agents can include agreeing when a package will be delivered. For instance, for the London-Berlin package, the agent of the trucking company can try to negotiate with the agent responsible for the boat to deliver the package in Amsterdam at 12:00 the next day.

We believe that in the future, inter-organizational multi-agent systems will become more important. Since the second world war, trade and globalization have steadily increased. Inevitably, this leads to increased specialization [3] as companies focus on what they do best and outsource the rest [18]. Outsourcing activities to other companies, however, introduces the need to coordinate with these companies. Efficiency considerations will lead to automation of (some of) the coordination activities. In case the coordination activities require a certain amount of autonomy from the coordinators, i.e., the coordinators must be able to independently decide *how* to coordinate, then the coordination activities can be performed by agents.

1.4 Contributions of this Research

Our research differs from other multi-agent research in that we study inter-organizational multi-agent systems in which agents wish to perform their own tasks independently, yet they must perform a joint task that creates dependencies between the agents.[4] Mainly, we focus on the following coordination problem, presented in Chapter 2: how to guarantee that agents can successfully work together, if we grant agents full autonomy in their planning activity, that is, if agents decide how to perform their tasks independently of other agents? The solution we promote is to sufficiently constrain agents *prior to planning*.

In Chapter 3, we analyze the computational complexity of this coordination problem. It turns out that the problem is too hard to allow fast algorithms that always return the optimal solution. Therefore, the best we can hope for is to find efficient heuristic approximation algorithms. In Chapter 4, we present a number of approximation techniques, and we identify how these approximation techniques can be extended to enable agents to be autonomous also during the coordination phase.

We have tested some of these approximation techniques on a benchmark set of multi-modal logistics planning problems from the AIPS[5] 2000 competition, and we present the results in Chapter 6. It turns out that our pre-planning-coordination approach outperforms all competitors of the AIPS. Also, it shows that by separating planning from coordination, difficult multi-agent planning problems can be solved using very simple single-agent planning tools, in fact, using planning tools we have written ourselves — need we say more?

[4] An earlier study [25] showed that for other research, agents *either* share a joint task, *or* they work independently.

[5] Artificial Intelligence Planning and Scheduling competition.

Chapter 2

A Task-Oriented Coordination Framework

Everything that is really great and inspiring is created by the individual who can labor in freedom — Albert Einstein

What sets a software agent apart from any garden-variety piece of software is the agent's ability — and its responsibility — to decide on its own the best course of action. That is, instead of having all its actions pre-programmed, the agent can decide for itself how to accomplish its goals. Other than this attribute of agency, the agent's *autonomy*, researchers are not in agreement on the definition of an agent.[1]

Clearly though, the characteristics of the agent, and of the environment in which it is designed to operate, determine if and how agents need to be coordinated[2]. For example, in an intra-organizational multi-agent system (i.e., in which all agents serve the same organization), with agents operating in close physical proximity, the location of each agent is relevant to the other agents. If agent A_1 is currently at location loc_3, then coordination must ensure that other agents do not try to enter loc_3 while A_1 is still there, or alternatively, that agent A_1 moves out of the way in case they do try.

We on the other hand study agents in an inter-organizational setting, i.e., where agents may belong to different organizations. Of course, *inter-organizational multi-agent system* could still mean almost anything, so in Section 2.1 we define (*i*) what we mean by an agent, (*ii*) the type of problems the agents must solve, and (*iii*) the coordination problem that arises in the thus-defined setting.

An earlier study [25] showed that no existing coordination mechanisms are suited for the type of multi-agent systems we study. In Section 2.2 we therefore present a framework for modeling (inter-organizational) multi-agent systems. In Section 2.3, we show how we can express the *coordination problem* in terms of this framework.

[1]Several papers that deal with the definition of an agent are [4, 12, 15].

[2]Recall from Chapter 1 that coordination is the process of placing restrictions on the autonomy of individual agents, to ensure that every agent can pursue his intended goals, in spite of or thanks to the actions of other agents

2.1 Problem Statement

We assume that as agents can belong to different organizations, they are self-interested.[3] Agents have their own private goals to achieve, and we assume that they make *plans* to accomplish these goals. If agents belong to different organizations, they probably have competitive relations of some sort. Consequently, agents do not wish to share details of their plans with other agents.

We further assume that the agents' private goals contribute to a *joint task* (or global goal), which is a set of interdependent tasks. Specifically, an agent's private goal is to perform the subset of tasks assigned to it. Because of the dependencies between the tasks, the agents become dependent on each other for the completion of their private goals.

The above interpretation of an inter-organizational multi-agent system has two consequences for the nature of the coordination process. First, it restricts the type of dependencies that are likely to exist between the agents; second, it places additional constraints on the mechanisms used to coordinate the agents.

With regard to the dependencies that can arise between agents, if agents represent different organizations, then *resource conflicts* are less likely to occur. For instance, if agents operate apart (or are not physical entities at all), then they need not fear occupying the same location; also, all resources required to accomplish its goals (e.g tools, money) are likely owned by the organization it works for. Consequently, an agent does not need to share resources with agents from other organizations. Instead, one agent depends on another agent in case the latter must perform some tasks before the former can start on some of his.

With regard to the coordination mechanisms that are applicable, if agents cannot achieve coordination by exchanging details of plans, then planning and coordination must be separated. It is not hard to see that the only way to coordinate the agents is to restrict their (planning) autonomy *before* they commit to a certain plan. Thus, we are faced with the following coordination problem:

> how to ensure that, whatever plans the individual agents come up with, these plans can always be combined into a feasible joint plan?

A coordination method to solve this problem must (*i*) allow separation of the planning and the coordination process, and (*ii*) must coordinate the dependencies arising from the joint task. In [25], we have shown that no method in the multi-agent coordination literature meets both these requirements. It turns out that if planning and coordination are separated, then the goals of the agents are unrelated (cf. [22]); if a method coordinates the dependencies arising from a joint task, then the coordination process is intertwined with the planning process (cf. [8, 6]). Thus, a new framework for inter-organizational multi-agent coordination is required.

[3] That is, they care only about the *utility* of their own organization, not about the utility of agents belonging to other organizations.

2.2 A Task Framework

An *elementary task* or simply a *task* is a unit of work that can be performed by single agent working alone. In our discussion of coordination, we do not distinguish any lower levels of abstraction than a task. This means that we do not decompose tasks into subtasks (cf. [5]), for instance. Note that this does not imply that elementary tasks are necessarily low-level concepts; a task may correspond to a single action, but it may also correspond to e.g. a sequence of actions. Typically, we view a task as an assignment that an agent has to perform. In the examples we will give, we will for instance associate a task with the assignment of delivering a package from one location to another by some transportation agent such as a truck.

A task can depend on other tasks if there exists a *precedence constraint* between two tasks. If a task t_1 is preceded by task t_2, denoted by $t_1 \prec t_2$, then execution of t_1 may not start until t_2 has finished.

Definition 2.2.1 (Composite Task). *A composite task $\mathcal{T} = (T, \prec)$ is a partial order where T is a non-empty set of tasks, and $\prec$ is a set of precedence constraints over T, i.e., $\prec \subseteq T \times T$.*

We will use the concept of a composite task to formalize the joint task.

A note on terminology: we often speak of the set of precedence constraints E, by which we mean the transitive reduction of the precedence relation $\prec$.[4] A composite task can then be represent as a directed acyclic graph $G = (T, E)$. We will mostly use the set E when referring to the set of precedence constraints, while we will use $\prec$ if we want to refer to the *transitive* precedence relation.

The final component of the framework is the set of agents $A = \{A_1, \ldots, A_n\}$. Agents in A are assumed to be autonomous planning[5] agents. We will not associate any properties with an agent, such as its capabilities[6]. The relevance of distinguishing agents lies in the fact that (*i*) the elementary tasks are allocated to agents and that (*ii*) the autonomy of the agents introduces the need to coordinate.

2.2.1 Task allocation and agent goals

Given a composite task $\mathcal{T} = (T, E)$ and a set of agents A, the tasks in T must be distributed among the agents. How tasks can or should be allocated to agents depends on the specific planning domain. We shall see in our treatment of the logistics domain (Chapter 5), that for certain planning problems, there is only one allocation of tasks to agents possible. For other domains, many task allocations are possible. In our research, we do not investigate how to best allocate tasks to agents. Instead, we assume that task allocation occurs *a priori*. The allocation of tasks to agents can be specified by a function:

$$f : T \rightarrow A$$

[4]The transitive reduction of a directed graph G is the directed graph G' with the smallest number of edges such that for every path between vertices in G, G' has a path between those vertices.

[5]Agents that do not make plans are purely reactive.

[6]In [27], the capabilities of the agents are part of the framework.

The function f associates with each task exactly one agent to accomplish it. Consequently, the set of tasks T is partitioned into $\mathbf{T} = \{T_1, \ldots, T_n\}$, with T_i the set of tasks allocated to agent A_i.

During task allocation, agent A_i not only receives a set of tasks T_i, but also inherits the precedence constraints that apply to T_i. In this respect, task allocation can be thought of as giving each agent a goal.

Definition 2.2.2 (Local Goal). *Given a composite task $\mathcal{T} = (T, \prec)$ and a task allocation function f, the local goal $G_i = (T_i \prec_i)$ of agent A_i is given by:*

$$\begin{aligned} T_i &= \{t \in T | f(t) = i\} \\ \prec_i &= \prec \cap (T_i \times T_i) \end{aligned}$$

Thus, an agent's local goal consists of a set of tasks T_i that need to be accomplished and an ordering relation $\prec_i$ that must be adhered to during execution.

2.2.2 Plans, goals, and refinements

Conceptually, plans and goals are the same. This is not only true in our framework, but it also holds in general. A goal specifies agent activity at a certain level of detail. A plan also specifies agent behaviour, but at a higher level of detail. A plan, in turn, can be viewed as the goal for a plan of even finer detail.

As an example, suppose that Bob intends to buy Alice a Valentine's Day gift. To achieve this goal, he decides to buy perfume at the local perfumery. This plan specifies his intent of buying Alice a gift at a greater level of detail. However, it does not specify yet where Bob will park his car, or whether he will talk to salesperson Charles or salesperson Trudy.

Perhaps there *is* a conceptual difference between goals and plans in case we can identify the highest level of detail. In the AI planning community, it is commonplace to identify a set of *elementary* or *atomic* actions or operators. Obviously, the existence of elementary actions is a simplifying assumption that enables modeling of the problem at hand. In our framework, it is neither necessary nor useful to make this assumption.

If plans and goals are equal in our framework, then the question is why we introduced the notion of a plan in the first place. Recall that we assume our agents to be planning agents, i.e., we assume agents to plan their actions. If there is a planning activity, then it is only natural to talk about plans. To illustrate the planning activity, we return to the example of Bob's Valentine's Day gift.

Suppose that apart from the perfume for Alice, Bob also needs to go to the library to pick up a book on cryptography. We can model Bob's intentions with two tasks:

$$\begin{aligned} t_1 &: \text{library book on cryptography} \\ t_2 &: \text{scent for a woman} \end{aligned}$$

These two tasks are unrelated; Bob does not, for instance, need the book on cryptography to decide which perfume to buy for Alice. Thus, Bob has the goal $G = (\{t_1, t_2\}, \emptyset)$. To accomplish these tasks, Bob plans his actions. The library closes early, so he will pick up the book first and then get the perfume for Alice. This plan can be represented as $G' = (\{t_1, t_2\}, \{t_1 \prec t_2\})$.

During the planning activity, Bob has created a more detailed goal. In the above example we have thus distinguished two goals G and G'. The relation between these two goals is that G' *refines* G:

Definition 2.2.3 (Refinement). *Let $G = (T, \prec)$ and $G' = (T', \prec')$ be two goals. G' refines G, denoted $G' \vdash G$, iff:*

$$T' = T \tag{2.1}$$
$$\prec' \supseteq \prec \tag{2.2}$$

We will refer to G' as the *refined goal* or simply as the *plan*, in case the refinement of the original goal is the result of planning activity. Note that in our framework, the planning activity *only* specifies a refined ordering relation $\prec'$. The actual planning software an agent uses to plan his activities will produce a plan with all kinds of details such as resource usage that are not relevant in our current discussion. Such an overly-detailed plan can be viewed as a *concrete plan*; the plan represented as a composite task (goal) is the *abstract plan*. The relation between the concrete plan and the abstract plan need not be trivial. Nevertheless, we will assume that given a concrete plan, we can always extract the abstract plan. That is, whatever information is stored in the concrete plan, we can always deduce (i) the set of tasks accomplished by the plan, and (ii) the ordering relation of the tasks in the plan.

We require that agents only make acyclic plans. With regard to goal refinement, this means that if G' refines G, then the relation $\prec'$ may add extra precedences to $\prec$, but not any precedences that make $\prec'$ cyclic. If a cyclic ordering relation $\prec'$ were created, then we would call the pair $(T', \prec')$ an *infeasible plan*.

2.2.3 The joint plan

The plan each agent makes for his local goal should contribute to the global goal, which is executing the composite task $\mathcal{T}$. All local plans must therefore be combined into a joint plan. With regard to a joint plan, we can again make the distinction between the *concrete plan* and the *abstract plan*. A concrete joint plan will include e.g. timing information: if one agent A_i wants to start on a task t that is preceded by a task t' from a different agent A_j, then A_i needs to know the time at which time t' will be finished. The abstract plan only contains information about the order in which tasks are executed. Again, we only consider the abstract joint plan. In that case, we can simply form the joint plan by uniting the plans of the individual agents, along with the set of precedence constraints not contained within any agent's goal:

Definition 2.2.4 (Joint Plan). *Given a composite task $\mathcal{T} = (T, \prec)$ and a partitioning* $\mathbf{T}$ *of T, resulting in a set of agent goals $G_i = (T_i \prec_i)$, $i = 1, \ldots, n$, and set of goal refinements $p_i = (T_{p_i}, \prec_{p_i})$, the joint plan $J = (T_J, E_J)$ is given by*

$$E_J = [\prec \cup \prec_{p_1} \cup \cdots \cup \prec_{p_n}]^-$$
$$T_J = T$$

In case the relation E_J is cyclic, we say that the joint plan is *infeasible*.[7]

[7]Note that R^- is the transitive reduction of relation R.

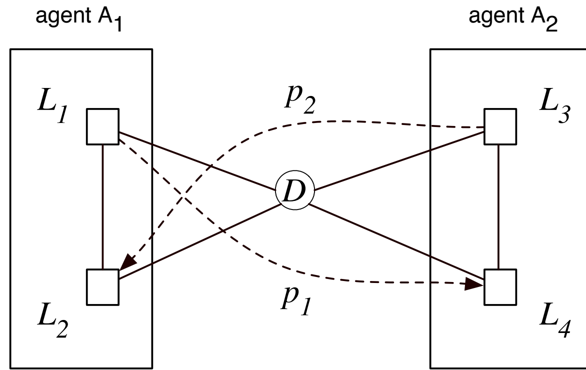


Figure 2.1: *Parcels must be delivered between locations; a package can be transferred from one truck to another at the depot.*

Definition 2.2.4, and with it our discussion of multi-agent coordination, ignores timing issues. In Chapter 5, we discuss why and under which conditions timing is not an issue.[8]

2.3 The Coordination Problem

The coordination problem we study is how to keep agents autonomous in their planning activity, while still having the guarantee of a feasible joint plan. That is, we want to ensure that whatever local plans the agents come up with, these plans can be combined unchanged into a joint plan.

The following example shows how an infeasible (cyclic) joint plan may be created, in case no coordination is performed.

Example 2.3.1. *We have a multi-agent planning problem where parcels have to be transported between locations: a package p_1 must be transported from location L_1 to L_4 and a package p_2 that must be transported from L_3 to L_2 (see Figure 2.1). We have two transportation agents: agent A_1 handles locations L_1, L_2 and the depot D, while agent A_2 serves locations L_3, L_4 and D. Both agents have to start and finish at the depot D. Transportation of package p_1 requires agent A_1 to pick up the package at L_1, but as location L_4 is out of A_1's region, he will only transport p_1 as far as the depot D. From there, agent A_2 will drive the package to L_4. Similarly, agent A_2 will bring p_2 to the depot, but A_1 must perform the final trip to L_2.*

Now agent A_1 has to make a plan for carrying out the tasks $t_1 = (L_1, D)$ and $t_2 = (D, L_2)$, while A_2 has to make a plan for $t_3 = (L_3, D)$ and $t_4 = (D, L_4)$. These tasks are interrelated: t_1 has to be completed before t_4 can start and likewise t_3 has to precede t_2. If we would allow both agents to plan independently, then the two plans could easily become incompatible: for example, if agent A_1 would aim for the plan (visiting sequence) $D - L_2 - L_1 - D$ to achieve his tasks t_2 and t_1 and A_2 would aim for the visiting sequence $D - L_4 - L_3 - D$, these plans cannot be combined in a multi-agent plan achieving T: Clearly,

[8]Basically, timing is of no importance for coordination if agents are concerned only about the amount of work they must do, not the times at which they must do it.

to start its plan, agent A_1 has to wait in D until t_3 has been accomplished by agent A_2, but agent A_2 cannot complete this task, because it intends to wait until t_1 has been completed by A_1. Hence, trying to execute the agent plans would result in a deadlock.

In Example 2.3.1, no additional constraints were placed on the agents before they started making their plans. As a result, each agent makes a plan that from his local point of view is correct, but it results in an infeasible joint plan, since E_J contains the cycle: $(t_1, t_4, t_3, t_2, t_1)$.

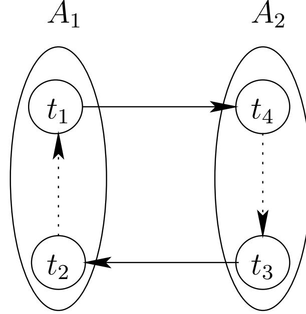


Figure 2.2: *For both agents, executing the post-depot task before the pre-depot task is a feasible refinement with regard to their local goals.*

In Figure 2.2, the situation before planning is depicted. A solid arrow represents a precedence constraint that is part of the precedence relation $\prec$ and must therefore be adhered to by any correct (joint) plan. Dashed arrows represent possible goal refinements. For instance, for agent A_1, a feasible refinement would be to add precedence constraint $t_2 \prec t_1$ (i.e., to execute his post-depot task before his pre-depot task) to his local goal, which has no precedence constraints yet. From a global point of view, however, not both agents should be allowed to add their 'dashed' precedence, otherwise the joint plan will become cyclic. To decide which agents should be free to add which constraints, coordination is required.

A simpler problem is to verify whether any coordination is needed, that is, the *coordination verification* problem is to decide whether it is *im*possible for the agents to make refinements resulting in an infeasible joint plan:

Definition 2.3.2 (Coordination Verification Problem). *The coordination verification problem (CVP) is: given a coordination instance $(\mathbf{T}, G)$ with $G = (T, E)$ a dag and $\mathbf{T} = \{T_1, \ldots, T_n\}$ a partition of T, does it hold that, for all sets of refinements $R \subseteq T \times T$ satisfying:*

1. *$R = R_1 \cup \cdots \cup R_n$ where $R_i \subseteq T_i \times T_i$ for $i = 1, \ldots, n$, and*
2. *for each i, the graph $(T_i, E_i \cup R_i)$ is acyclic,*

the graph $(T, E \cup R)$ is acyclic?

Note that in a yes-instance of CVP, it holds that whatever locally acyclic plans the agents come up with, the result can be combined in a feasible joint plan.

In the above depot example, we obviously have a no-instance of the coordination verification problem: if we allow both agents to plan independently, then an infeasible joint plan might result. It is not hard to see how the agents can be coordinated: either agent A_1 must perform its pre-depot task t_1 before its post-depot task t_2, or agent A_2 must perform its pre-depot task before starting on its post-depot task. In general, the coordination problem is to find a set of additional constraints such that, whatever refinements the individual agents make, the joint plan will always be feasible. The coordination problem is a minimization problem: in order to ensure coordination, we wish to add as little constraints as possible, because additional constraints reduce the planning autonomy of an agent. Also, if an agent is severely constrained in the planning process, then it might not be possible to find a low-cost plan.[9]

Definition 2.3.3 (Coordination Problem). *The coordination problem (CP) is: given a coordination instance* $(\mathbf{T}, G)$ *find a set of precedence constraints* $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$ *with* $\Delta_i \subseteq T_i \times T_i$ *such that:*

1. $E \cup \Delta$ *is acyclic,*
2. $(T, E \cup \Delta)$ *is a yes-instance of CVP, and*
3. $|\Delta|$ *is minimal,*

A note on terminology: we will call a set of constraints Δ a *coordination set* if it meets requirements 1 and 2 of Definition 2.3.3. If a set Δ meets requirements 1, 2 and 3, then Δ is an *optimal coordination set*. We say that a coordination instance $I = (\mathbf{T}, G)$ is *coordinated*, if the empty set $\emptyset$ is a coordination set, i.e., in case I is a yes-instance for CVP.

Definition 2.3.3 reflects our choice to coordinate agents in the pre-planning phase: the coordination problem asks to find a set of constraints, so that agents can subsequently refine their goals plan independently of each other.

[9] We will discuss *plan cost*, and the relation between plan cost and additional precedence constraints, in Chapter 5.

Chapter 3

Analyzing the Coordination Problem

Personally, I never take a peek at the last page of a book. If I must know the number of pages, I am ever careful not to inadvertently read the author's final few lines. Some people, however, yield almost instantly to this low form of cheating. If such people were to read this thesis, then chances are that they would skip this chapter and proceed directly to the approximation algorithms in Chapter 4. Indeed, with the definition of the coordination problem in the previous chapter and its solutions in the next, what remains to be said in this chapter?

If you are still reading, you will find in this chapter an analysis of the coordination problem that bridges the gap between the problem definition and the solution methods by presenting the theory (i.e., definitions, propositions, etc.) on which the approximation algorithms rely. Specifically, we will associate a graph structure, which we name the *coordination graph*, with a coordination instance, and then show that the coordination problem can be reduced to a kind of cycle breaking in the coordination graph.

In Section 3.3, we briefly investigate the composition of a coordination set Δ. In the previous chapter, we have defined a solution for a coordination instance to consists of a set of additional intra-agent constraints. We discuss which types of arcs qualify for placement in a coordination set.

At the end of this chapter, in Section 3.4, we analyze the complexity of the coordination problem. For the benefit of the reader who feels complexity analysis is about as exciting as watching paint dry, we here summarize the results: the coordination problem is not in P[1], that is, there do not exist algorithms that always return the optimal solution, while requiring an amount of time polynomial in the size of the coordination instance. A problem in NP requires that a solution can be *verified* in polynomial time. However, the problem of verifying whether a coordination set indeed coordinates an instance is co-NP-complete. The coordination problem itself is Σ_2^p-complete. Finally, the coordination problem is also hard to approximate. Even for near-trivial restricted classes of the coordination problem, it is highly unlikely that there exist constant-ratio approximation algorithms.

[1]unless P = NP, which is very unlikely

3.1 The Coordination Graph

We already have a graph notation for a complex task — we can think of the pair (T, E) as a directed acyclic graph — so why do we further attempt to confuse the reader with yet another graph representation? Primarily, because we also want to be able to represent the set of possible refinements in a graph, since these refinements — representing agent planning autonomy — are at the heart of the coordination problem.

A second reason is that a coordination instance can contain information that is not relevant from the point of view of coordination. Clearly, culling irrelevant tasks, precedences and refinements from the coordination graph does not change the coordination problem, but it does simplify proof-construction. Once we have proven a certain case irrelevant for coordination, we do not have to take this case into account in the proofs of all subsequents propositions.

3.1.1 Tasks and the coordination problem

Suppose an agent is allocated a single task t. Also, suppose there are no tasks preceding or succeeding t. Clearly, the time at which t is executed is irrelevant with regard to coordination. To study the coordination problem, we can concentrate on the set of tasks that *might* be important for coordination.

In the planning activity, agents refine their goals. Thus, for any plan $p_i = (T_i, \prec_{p_i})$, the relation $\prec_{p_i}$ is acyclic. Recall that the coordination problem is to ensure that for any set of plans, the relation $E_J = [\prec \cup \prec_{p_1} \cup \cdots \cup \prec_{p_n}]^-$ will be acyclic. Since both the agent plans and the relation $\prec$ are acyclic, a cycle in E_J must involve the plan of more than one agent. Hence, we are interested in (potential) inter-agent cycles in the relation E_J. Consequently, a cycle in E_J involves tasks that are related to 'other-agent' tasks; a task t is related to a task t' if there is a precedence constraint between t and t'. We define the set *INTER* of inter-agent precedences constraints:

Definition 3.1.1. *The set of inter-agent precedence constraints is given by*

$$INTER = E \setminus \bigcup_{i=1}^{n} (T_i \times T_i)$$

We say that a task t is related to a task from another agent if t is incident on an *INTER* arc.

Definition 3.1.2. *The set of tasks $T_{inter} \subseteq T$ is the set of tasks that are directly preceded or succeeded by a task allocated to another agent:*

$$T_{inter} = T^{in} \cup T^{out}$$

where $T^{in} = ran(INTER)$ and $T^{out} = dom(INTER)$.

To avoid confusion, note that T^{in} is the set of tasks that have an *in*coming *INTER* arc, and T^{out} is the set of tasks that have an *out*going *INTER* arc.

Note that we have defined the set *INTER* as $E \setminus \bigcup_{i=1}^{n}(T_i \times T_i)$. By using E instead of $\prec$ in Definition 3.1.1, we avoid including a lot of irrelevant tasks in that T_{inter}. Consider

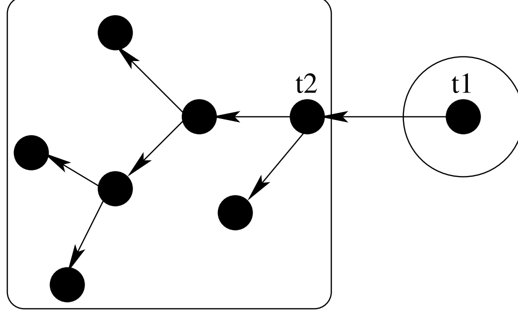


Figure 3.1: *Only tasks t_1 and t_2 are in T_{inter}.*

Figure 3.1, where t_1 and t_2 are in T_{inter}. If we had defined *INTER* as $\prec \setminus \bigcup_{i=1}^{n}(T_i \times T_i)$, then all tasks in the figure would be in T_{inter}, because for all $t' \in T : t_1 \prec t'$.

Proposition 3.1.3 shows why it is sufficient to restrict our attention to tasks in T_{inter}.

Proposition 3.1.3. *Given a coordination instance $I = (\mathbf{T}, G)$ and a set of plans $\{p_1, \ldots, p_n\}$, if there is an inter-agent cycle C in $\prec \cup \prec_{p_1} \cup \cdots \cup \prec_{p_n}$, then there is an inter-agent cycle C' in $\prec \cup \prec_{p_1} \cup \cdots \cup \prec_{p_n}$ such that $C' \subseteq T_{inter}$.*

Proof. The inter-agent cycle $C = (e_1, \ldots, e_m)$ consists of two types of arcs: arcs in *INTER* and arcs in $\prec_{p_i}$. *INTER* arcs use only tasks in T_{inter}, arcs in $\prec_{p_i}$ may use any type of tasks. We can construct C' by replacing the latter type of arcs with arcs that use only T_{inter} tasks.

Let e_h and e_k be two *INTER* arcs in C, such that for all j, $h \leq j \leq k$, $e_j \in \prec_{p_i}$ for *one* agent A_i. Let $t = \text{ran}(e_h)$ and $t' = \text{dom}(e_k)$. Note that $(t, t') \in \prec_{p_i}$, because there is a path in $\prec_{p_i}$ from t to t', and $\prec_{p_i}$ is transitively closed. Also, $\{(t, t')\} \subset T_{inter} \times T_{inter}$. Thus, to construct C', we can replace all arcs e_j between e_h and e_k with the single arc (t, t'). □

3.1.2 Precedences and the coordination problem

If a task is not interesting for coordination— it is not in T_{inter} — then any precedence constraints involving that task are not of interest either. Thus, we focus on the set of precedence constraints that is a subset of $T_{inter} \times T_{inter}$.

We can divide the set of interesting precedences into precedences *between* agents (i.e, between tasks allocated to different agents) and precedences *within* agents. We have already defined the set of inter-agent precedences *INTER*. We will now define the set of *intra-agent* precedences *INTRA*.

Definition 3.1.4. *The set INTRA represents the precedence relation within agents of tasks in T_{inter}:*

$$INTRA = \prec \cap \bigcup_{i=1}^{n} (T_{inter_i} \times T_{inter_i})$$

where $T_{inter_i} = T_{inter} \cap T_i$.

The set *INTRA* captures the precedence relation within agents. This information is useful if we want to know how an agent can contribute to an inter-agent cycle. For instance,

suppose an agent has two *INTER* tasks t_1 and t_2, with $t_1 \in T^{in}$ and $t_2 \in T^{out}$ (see Figure 3.2). If $t_1 \prec t_2$, then this agent can possibly contribute to an inter-agent cycle (Figure 3.2a). If, however, $t_2 \prec t_1$, then no cycle can pass through the agent (Figure 3.2b).

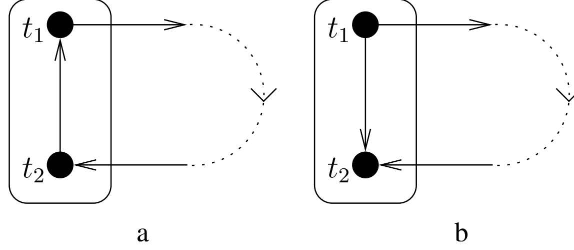


Figure 3.2: *(a): There may be an inter-agent cycle if $t_1 \prec t_2$. (b): If $t_2 \prec t_1$, then no cycle can pass through the agent.*

3.1.3 Refinements and the coordination problem

The agents' freedom to refine their goals is at the heart of the coordination problem. Without coordination, agents might make locally valid plans that result in an infeasible joint plan. However, not all goal refinements are relevant for coordination. For instance, if an agent is allocated (only) two tasks, both of which are not in T_{inter}, then any goal refinement will be compatible with the plans of other agents.

We can specify the set of coordination relevant refinements[2] more accurately than considering a subset of $T_{inter} \times T_{inter}$, though. In fact, if (not iff) a refinement δ is relevant for coordination, then it must satisfy the following four conditions:

1. δ refines a local goal and is thus contained within an agent: $\delta \in \bigcup_{i=1}^{n}(T_i \times T_i)$,
2. $\delta = (t_1, t_2)$ is not in $INTRA^{-1}$: if *INTRA* contains $t_2 \prec t_1$, then adding $t_1 \prec t_2$ creates a cycle in $\prec_i \cup \delta$
3. $\delta = (t_1, t_2)$ is not in *INTRA*: if the constraint $t_1 \prec t_2$ is already in the local goal, then it is not a refinement.
4. The arc is in $T^{in} \times T^{out}$: all possible inter-agent cycles that intersect an agent A_i will 'enter' A_i at some $t \in T^{in}$ and 'exit' A_i at some $t' \in T^{out}$.

This leads us to the definition of the set of *refinement arcs REF*.

Definition 3.1.5. *The set of coordination-relevant refinement arcs REF is given by:*

$$REF = \bigcup_{i=1}^{n}(T_i^{in} \times T_i^{out}) \setminus \left(INTRA^{-1} \cup INTRA\right)$$

where $T_i^{in} = T^{in} \cap T_i$ and $T_i^{out} = T^{out} \cap T_i$.

[2]We will often call a set of constraints a refinement if the goal, resulting from adding the constraints, is a refinement.

In Proposition 3.1.7, we prove that our claim that if a refinement is relevant for coordination, then it must be in *REF*. For Proposition 3.1.7, we need to introduce the concept of a *pass*.

Definition 3.1.6 (Pass). *Given an inter-agent cycle* $C = (t_1, \ldots, t_n, t_1)$ *in* $\prec \cup \prec_{p_1} \cup \cdots \cup \prec_{p_n}$, *a pass* p *of length* m *of* C *through* A_i *is a series of tasks* $(v_j, \ldots, v_{j+m}) \subset C$, *such that (i)* $\{v_j, \ldots, v_{j+m}\} \subseteq T_i$, *and (ii)* v_{j-1} *and* v_{j+m+1} *are not in* T_i.

Proposition 3.1.7 (Coordination-Relevant Refinements). *Given a coordination instance* $I = (\mathbf{T}, G)$ *and a set* $R = \{R_1, \ldots, R_n\}$ *of local-goal refinements, such that* $\prec \cup R$ *contains an inter-agent cycle* C, *then there exists a set* $R' \subseteq REF$ *such that:*

1. R' *is present under the transitive closure of* $\prec$ *and* R*:*

$$R' \subseteq \left([\prec \cup R]^+ - \prec\right)$$

2. $\prec \cup R'$ *contains an inter-agent cycle* C'

Proof. Note that by Proposition 3.1.3, we can assume that R involves only tasks in T_{inter}.

The cycle C consists of, alternately, passes of C through an agent and arcs in *INTER*. To construct C', we might need to replace some of the passes in C. We distinguish two cases, for all passes $p = (t_k, \ldots, t_m)$ in C:

case 1: $(t_k, t_m) \in \prec_i$**:** We can use p for C', and choose $R'_i = \emptyset$.

case 2: $(t_k, t_m) \notin \prec_i$**:** We can choose $R'_i = \{(t_k, t_m)\}$ (regardless of the composition of R_i). The set R'_i satisfies the requirements of Proposition 3.1.7:

1. $(t_k, t_m) \in REF$: first, note that $t_k \in T^{in}$ and $t_m \in T^{out}$; second, $(t_k, t_m) \notin \prec_i$, by definition of case 2; third, $(t_k, t_m) \notin \prec_i^{-1}$ because R_i is a refinement: there is a path from t_k to t_m in $\prec_i \cup R_i$, so there can be no path in $\prec_i$ from t_m to t_k.
2. $(t_k, t_m) \in [\prec_i \cup R_i]^+ - \prec_i$. There is no path from t_k to t_m in $\prec_i$, by definition of case 2, but there is a path from t_k to t_m in $\prec_i \cup R_i$.

□

3.1.4 Definition of the coordination graph

The coordination graph represents the set of tasks, the precedence relation between those tasks, and the refinements that the agents are allowed to make during planning. We define the coordination graph as follows:

Definition 3.1.8 (Coordination Graph). *Given a coordination instance* $I = (\mathbf{T}, G)$, *the directed graph* $G_{\mathcal{T}} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, *with*

$$\begin{aligned} V_{\mathcal{T}} &= T_{inter} \\ E_{\mathcal{T}} &= INTRA \cup INTER \cup REF \end{aligned}$$

is called the coordination graph.

In depicting the coordination graph, we represent arcs in *INTRA* and *INTER* as solid arcs, arcs in *REF* as dotted or dashed arcs, and we delineate the tasks belonging to one agent by a convex area.

3.2 The Coordination Graph and the Coordination Problem

When is a coordination instance coordinated? According to Definition 2.3.3 of the coordination problem, an instance is coordinated if $\Delta = \emptyset$ is a coordination set. In that case, for any set of local-goal refinements, the relation E_J is acyclic, so the joint plan refines the complex task. To check whether a coordination instance is coordinated, we do not want to enumerate the exponential number of local-goal-refinement sets, though.

In this section, we will prove that an instance is coordinated if and only if the coordination graph contains a certain type of cycle, which we will call a *refinement cycle*. In Section 3.4, however, we will show that detection of a refinement cycle cannot be done in polynomial time.

3.2.1 Inter-agent cycles

If a certain set of goal refinements $R = \{R_1, \ldots, R_n\}$ creates a cycle in $E_J = [\prec \cup R]^-$, then this cycle will be visible in the coordination graph. After all, the coordination graph contains all tasks, precedences and refinements that are relevant for coordination, as shown in Section 3.1.

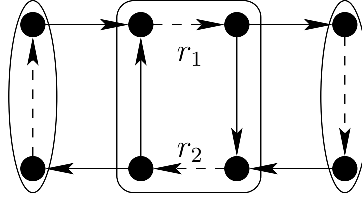


Figure 3.3: *Despite the presence of an inter-agent cycle, this instance is coordinated.*

Thus, we are looking for inter-agent cycles in the coordination graph. However, not all inter-agent cycles are interesting from the point of view of coordination, as Figure 3.3 shows. In Figure 3.3, there is an inter-agent cycle in the coordination graph, but the instance is coordinated; the only way that a cycle in E_J can be created is if the middle agent chooses $R = \{r_1, r_2\}$ to refine his goal. However, $\{r_1, r_2\}$ is not a refinement, since it creates a local cycle.

Thus, we are interested in inter-agent cycles that are *locally acyclic*, that is, the refinement arcs contributing to the cycle should not form a cycle with the local precedence constraints in $\prec_i$.

Definition 3.2.1 (refinement cycle). *An inter-agent cycle C is a refinement cycle iff, for all $i = 1, \ldots, n$, $(REF \cap C) \cup \prec_i$ is acyclic.*

We introduced the term 'refinement cycle' to avoid writing 'locally-acyclic inter-agent cycle' every time.

Theorem 1. *Given a coordination instance $I = (\mathbf{T}, G)$, the following assertions are equivalent.*

1. *There exists a refinement cycle in the coordination graph.*
2. *The coordination instance is not coordinated.*

Proof.

(1: refinement cycle $\rightarrow$ not coordinated) Let $C = (e_1, \ldots, e_m)$ be a refinement cycle in the coordination graph. To prove that the coordination instance is not coordinated, we need to show that there exists a set of goal refinements $R = \{R_i, \ldots, R_n\}$ that leads to a cycle in $E_J = [\prec \cup R]^-$.

We choose $R = REF \cap C$. Clearly,

1. $\prec_i \cup R_i$ is acyclic, because C is locally acyclic by Definition 3.2.1,
2. $\prec \cup R$ is cyclic, since C is in $INTER \cup INTRA \cup R$, which is a subset of $\prec \cup R$.

(2: not coordinated $\rightarrow$ refinement cycle) Assume that agents make refinements to their goals in such a way that $E_J = [\prec \cup R_1 \cup \cdots \cup R_n]^-$ contains an inter-agent cycle $C = (e_1, \ldots, e_m)$. By Proposition 3.1.7, we may assume w.l.o.g. that $R \subseteq REF$.

The cycle C maps to an inter-agent cycle C' in the coordination graph: If $e_i \in C \cap R$, then $e_i \in REF$ and since $REF \subseteq E_{\mathcal{T}}$, e_i is in the coordination graph. If $e_i = (t, t') \in C \cap \prec$, then there is a path in $INTRA \cup INTER$ from t to t'.

It is easily understood that C is locally acyclic. Since R_i refines $(T_i, \prec_i)$, $\prec_i \cup R_i$ is acyclic, for all $i = 1, \ldots, n$.

□

3.3 Constructing a Coordination Set

A coordination set is a set of additional intra-agent constraints Δ. Since $E \cup \Delta$ must be acyclic, the set Δ refines $\mathcal{T}$, by Definition 2.2.3. But which refinements should we make in order to ensure coordination? Note that because of Theorem 1, the coordination problem can be rephrased in terms of the coordination graph: to coordinate the agents, we must ensure that the coordination graph contains no refinement cycles.

The precedence relation $\prec$ is acyclic; consequently, any cycle in the coordination graph must include refinement arcs. More specifically, every cycle in the coordination graph must contain at least two arcs from REF.

Proposition 3.3.1. *For any cycle C in $G_{\mathcal{T}}$ it holds that $|C \cap REF| \geq 2$.*

Proof. Let $C = (v_1, v_2, \ldots, v_n, v_1)$ be a cycle in $G_{\mathcal{T}}$. Since $INTRA \cup INTER$ is acyclic, C contains at least one refinement arc in C. Without loss of generality, assume $(v_1, v_2) \in REF$. Suppose for the sake of contradiction that the remaining arcs in C are *not* in REF, i.e., $(C \setminus \{(v_1, v_2)\}) \subseteq (INTRA \cup INTER)$. But then the path $(v_2, \ldots, v_n, v_1)$ is in $INTER \cup INTRA$, and therefore, since $INTRA$ is transitively closed, $(v_2, v_1) \in INTRA$.

Due to the fact that $INTRA^{-1} \cap REF = \emptyset$, by Definition 3.1.5, we have $(v_1, v_2) \notin REF$, which is a contradiction. Hence, not all remaining arcs can be in $INTER \cup INTRA$, which means that there must be at least one more arc that is in REF. It follows that $|C \cap REF| \geq 2$. □

To remove all refinement cycles from the coordination graph, we need to remove arcs from the coordination graph. Since we assume that the allocation of tasks and the relation $\prec$ cannot be altered during coordination, we cannot remove arcs from $INTRA$ or $INTER$. Hence, we are only allowed to remove arcs from the set REF. Note that arcs from REF can be removed by adding arcs to the set $INTRA$ (Definition 3.1.5). Hence, we can remove an arc $r \in REF$ by adding either r or r^{-1} to $INTRA$.

Refinement arcs can be viewed as *cycle-enabling* arcs. If we add an arc $r \in REF$, this cycle-enabling arc will still be in the coordination graph. An arc in REF^{-1} can be viewed as a *cycle-breaking* arc. If $r \in REF$, then adding r^{-1} to $INTRA$ ensures that the cycle-enabling r is no longer in the coordination graph.

Coordination entails constraining agent freedom; in case of our coordination problem, this means that not all agents may be allowed to add all possible refinements. Thus, coordination must specify which refinements are not allowed. To disallow a refinement $r = (t_1, t_2)$, we can add $r^{-1} = (t_2, t_1)$ to $INTRA$. Hence, a coordination set is specified by a set of refinements $\Delta \subseteq REF^{-1}$.

Proposition 3.3.2. *Given a coordination instance $I = (\mathbf{T}, G)$ and an arbitrary coordination set Δ for I. Then the set*

$$\Delta' = [E \cup \Delta]^{+} \cap REF^{-1}$$

that is, the set of constraints in REF^{-1} that are added as a consequence of Δ (possibly under transitive closure), is also a coordination set.

Proof. Let $G_{\mathcal{T}}$ be the coordination graph associated with $\mathcal{T} = (T, E)$ and let $G_{\mathcal{T}'}$ be the coordination graph associated with $\mathcal{T}' = (T, E \cup \Delta)$.

For every refinement cycle C in $G_{\mathcal{T}}$, let $REF_{\mathcal{T}}(C) = REF_{\mathcal{T}} \cap C$. In $G_{\mathcal{T}'}$, C no longer exists, because not all arcs in $REF_{\mathcal{T}}(C)$ are in $REF_{\mathcal{T}'}$ (or in $INTRA_{\mathcal{T}'}$). For every C in $G_{\mathcal{T}}$, there must be at least one $r \in REF_{\mathcal{T}}(C)$ that has been removed because of the fact that $r^{-1} \in [E \cup \Delta]^{+}$. Otherwise, if all $r \in REF_{\mathcal{T}}(C)$ had been removed because $r \in [E \cup \Delta]^{+}$, then the cycle C would still be present in $E \cup \Delta$, which contradicts the fact that Δ is a coordination set.

We can form Δ' by putting, for all C, the arcs $r^{-1} \in [E \cup \Delta]^{+}$, such that $r \in REF_{\mathcal{T}}(C)$, into Δ'. Note that Δ' coordinates I:

- all refinement cycles in $G_{\mathcal{T}}$ are broken by addition of Δ', since every refinement cycle in $G_{\mathcal{T}}$ has at least one arc in Δ'^{-1}.
- $E \cup \Delta'$ is acyclic, because $E \cup \Delta' \subseteq [E \cup \Delta]^{+}$, and $E \cup \Delta$ is acyclic.

Also, it is clear that $\Delta' \subseteq REF_{\mathcal{T}}^{-1}$. □

It follows from Proposition 3.3.2 that, although a coordination set is specified by a subset of REF^{-1}, we do not have to coordinate by directly adding arcs from REF^{-1} to $INTRA$.

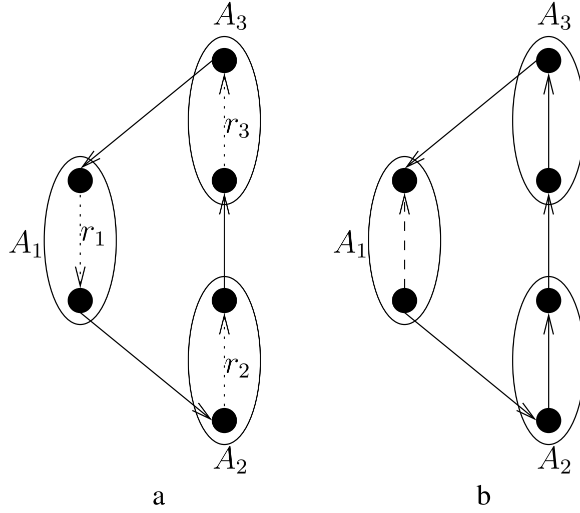


Figure 3.4: *(a): A refinement cycle in $G_{\mathcal{T}}$. (b): If r_2 and r_3 are added to E, then $r_1^{-1} \in INTRA$.*

To coordinate their activities, some agents must forego the right to make a refinement so that other agents are free to make the refinements they want. This understanding can be reached in two ways. First, an agent can explicitly relinquish the right to choose a certain refinement $r \in REF$ by adding r^{-1} to his set of local precedences $\prec_i$. This can be interpreted as the *cooperative* mode of coordination: one agents gives way, so that other agents can act freely.

A second way to coordinate is that an agent claims the right to refine his goal in a certain way: given a refinement $r \in REF$, he adds r to $\prec_i$. Subsequently, the set of allowed refinements for the other agents might be reduced, because some arcs that were in the set REF (defined in terms of the original precedence relation $\prec$), may no longer be in the set of refinement arcs associated with $E \cup \{r\}$. This form of coordination can be interpreted as *selfish*.

In Figure 3.4, the selfish form of coordination is illustrated. In Figure 3.4a, we see that there is a refinement cycle in the coordination graph. In Figure 3.4b, agents A_2 and A_3 have refined their goals by r_2 and r_3, respectively. For agent A_1, this means that refinement r_1 is no longer allowed. Indeed, from the definition of *INTRA*, it is easy to see that r_1^{-1} has been added to $\prec_{A_1}$ as a result of the additions of r_2 and r_3 to E. Also, note that even though agents A_2 and A_3 initially added $\Delta = \{r_2, r_3\}$, the coordination set is specified by $\Delta' = \{r_1^{-1}\}$.

3.4 Complexity of the Coordination Problem

Due to Theorem 1, we can view the coordination problem as the problem of removing all refinement cycles from the coordination graph. A similar problem is the Feedback Arc Set problem (FAS), which is to find set of arcs F such that every cycle in the graph has at least one arc in F; removing a feedback arc set from the graph results in all cycles being broken. The feedback arc set problem is an optimization problem in NP(O). The

coordination problem turns out to be harder than that, though.

For problems in NP[3], a solution must be verifiable in polynomial time. For instance, for the feedback arc set problem, we can verify in polynomial time that for a given feedback arc set F, every cycle indeed intersects F. For the coordination problem, however, it is not possible to verify in polynomial time that a coordinated instance contains no refinement cycle. That is, the coordination verification problem is co-NP-complete, as we will see in Section 3.4.1. In Section 3.4.2, we will prove that the coordination problem itself is Σ_2^p-complete, which means that it is solvable in non-deterministic polynomial time, if we have an NP-oracle for the coordination verification problem.

Given the complexity of the coordination problem, we have tried to identify in Section 3.4.3 subcases of the coordination problem that are easier to solve. We have not found any subcases in P, but we have been able to identify structural characteristics without which a coordination instance is in NP.

Finally, in Section 3.4.4, we analyze the approximability of the coordination problem. We shall see that even for severely restricted coordination instances, it is unlikely that there exist constant-ratio approximations.

3.4.1 Complexity of coordination verification

The coordination-verification problem is to verify whether there *cannot* exist a set of refinement arcs that create a refinement cycle. It is easy to see that CVP is in co-NP: a certificate for a no-instance of CVP is a set of refinements satisfying all properties of Definition 2.3.2. In other words, a no-certificate consists of a set of refinement arcs that constitutes a refinement cycle.

To prove that this problem is co-NP-complete, we will present a reduction from the Path With Forbidden Pairs problem (PWFP).

Definition 3.4.1. *The path with forbidden pairs problem is: given a tuple (G_0, C, s, t) where $G_0 = (V, E_0)$ is a directed acyclic graph, $C = \{c_1, c_2, \ldots, c_n\}$ is a set of pairs of arcs in E_0, and s and t are two distinct nodes in V, does there exist a path from s to t using at most one arc from every $c_j \in C$.*

Specifically, we will reduce PWFP to the complement of the coordination-verification problem. The complement of CVP, which we will call the Coordination-Failure Detection problem (CFD) asks whether the coordination graph contains a refinement cycle.

To reduce PWFP to CFD, we need to transform the graph $G_0 = (V, E_0)$ (the PWFP instance) to a tuple $(\mathbf{T}, G = (T, E))$. Intuitively, this transformation consists of two parts: an interesting part and a not-so-interesting part. The not-so-interesting part entails (i) creating a single-task agent for every vertex in V, and (ii) replicating those arcs that are not involved in any forbidden pair to the set of precedence constraints E.

The interesting part regards the transformation of forbidden pairs. We encode each forbidden pair as a so-called *path-blocking gadget*, depicted in Figure 3.5. For a certain forbidden pair $c_j = \{(x, y), (u, v)\}$ in the PWFP problem, we may choose either, but not both, of these arcs for creating a path from s to t. Similarly, in creating an inter-agent cycle in CFD, we may add either refinement arc (x^j, y^j), or (u^j, v^j), but not both. This

[3] Although the coordination problem is an optimization problem, we will discuss the complexity of the coordination problem in terms of its decision variant.

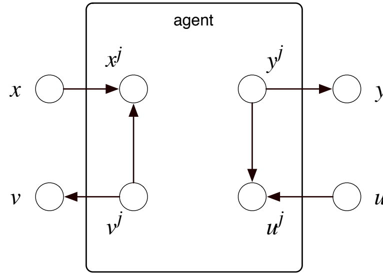


Figure 3.5: *The subgraph constructed for a forbidden pair* $\{(x, y), (u, v)\} \in C$.

mutual exclusion is enforced by the arcs in (v^j, x^j) and (y^j, u^j): addition of e.g. refinement arc (x^j, y^j) creates a path $v^j - x^j - y^j - u^j$ in $\prec_i$. Subsequently, (u^j, v^j) may no longer be added, as it would create a local cycle.

To complete the transformation, we need to connect t to s. In this way, we can equate the existence of an $s - t$ path in PWFP to the existence of a refinement cycle in CFD. However, we do not directly connect t to s, to avoid instances with a trivial $s - t$ path (using no arcs from forbidden pairs) to result in a cyclic coordination instance. Instead, we place an agent between s and t, that can connect t to s by adding a single refinement arc. An example of a transformation from PWFP to CFD is given in Figure 3.6.

Formally, the reduction of PWFP to CFD is defined as follows:

1. For $i = 1, \ldots n$: $T_i = \{v_i\}$; for $j = 1, \ldots, k$ (with $k = |C|$):
   $T_{n+j} = \{x^j, y^j, u^j, v^j \mid \{(x, y), (u, v)\} \in C\}$ and $T_{n+k+1} = \{s_0, t_0\}$, where both s_0 and t_0 do not occur in V. Obviously, $T = \bigcup_{i=1}^{n+k+1} T_i$.

2. E is the smallest set of arcs satisfying the following conditions:

   (a) For every arc $e = \{u, v\} \in E_0$ not occurring in a pair of arcs in C, e occurs in E.

   (b) For every constraint-pair of arcs $c_j = \{(x, y), (u, v)\} \in C$, E contains the following arcs
   $$(x, x^j), (y^j, y), (u, u^j), (v^j, v), (y^j, u^j), (v^j, x^j)$$
   (See Figure 3.5 for an illustration).

   (c) Finally, E contains the arcs (t, t_0) and (s_0, s).

Proposition 3.4.2. *The coordination verification problem is co-NP-complete.*

The proof of Proposition 3.4.2 consists of proving that PWFP reduces to CFD. In particular, we need to prove:

1. The transformation results in a correct instance of the CFD problem; this means that (T, E) must be a dag.

2. A yes-instance of the PWFP problem maps to a yes-instance of the CFD problem.

3. A yes-instance of the CFD problem maps to a yes-instance of the PWFP problem.

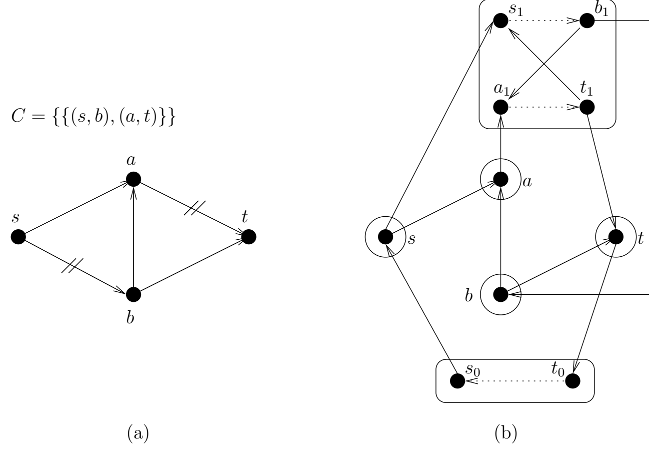


Figure 3.6: *(a): A PWFP instance with a single forbidden pair and (b): the corresponding coordination instance.*

Proof. The graph $G = (\{T_i\}_{i=1}^{i=n+k+1}, E)$ is acyclic: the subgraph of G restricted to nodes in V is clearly acyclic. The additional nodes $\{x^j, u^j\}_{j=1}^{k}$ and the node t_0 are clearly endpoints of paths; the additional nodes $\{y^j, v^j\}_{j=1}^{k}$ and the node s_0 are clearly starting points of paths. Hence, the additional nodes cannot contribute to a cycle in E.

A yes-certificate for a PWFP instance is given by a set of arcs $E' = \{e_1, \dots, e_m\}$ such that each e_j corresponds to one arc from the constraint-pair $c_j \in C$. Clearly, E' contains at most one arc from every pair in C. We map the set E' to a yes-certificate R for CFD in the following manner: if (x, y) from c_j in E', then $(x^j, y^j) \in R$. We complete the set R by adding arc (t_0, s_0). Clearly, R creates an inter-agent cycle in G: R creates a path from s to t, and a path from t to s, so we have an inter-agent cycle C'. C' is a refinement cycle, since each agent contributes at most one refinement arc to C', and it requires two refinement arcs to create a local cycle, according to Proposition 3.3.1.

A yes-certificate R creates an inter-agent cycle in CFD. It is easily verified that any inter-agent cycle must include the arc (t_0, s_0) from agent A_{n+k+1}. Thus, $R' = R \setminus \{(t_0, s_0)\}$ creates a path from s to t in the CFD-graph. The set R' can only map to a yes-certificate for PWFP. First, note that only path-blocking-gadget agents are capable of adding refinement arcs (apart from agent A_{n+k+1}, of course). Second, each such an agent can add at most one refinement arc: either (x^j, y^j), or (u^j, v^j) (these are the only refinement arcs for one agent). Adding both would result in a cyclic local plan, which is not allowed. Thus, R' maps to a yes-certificate for the PWFP instance, since an $s - t$ path can be created using at most one arc from every forbidden path. □

3.4.2 Complexity of the coordination problem

A solution Δ of a CP instance $(\mathbf{T}, G)$ is a cardinal-minimal set Δ of additional arcs that is sufficient to guarantee feasibility of the joint plan given arbitrary, individually feasible plans. Given a coordination instance I and an integer $K \geq 0$, the *decision variant* of the

coordination problem asks for the existence of a solution Δ of size at most K.[4] In this section, we will show that for arbitrary values of $K > 0$, we are faced with a Σ_2^p-complete problem. To prove this, we need to introduce the following quantified version of the PWFP problem.

Definition 3.4.3 ($\exists\forall\neg$PWFP). *Given a* PWFP *instance* $(G_0 = (V, E_0), C, s, t)$*, and a partitioning* $\{C_1, C_2\}$ *of* C*, the* $\exists\forall\neg$PWFP *problem is to find an* exclusive choice *from* C_1*, i.e., a set* X_1 *that contains exactly one arc from every pair of forbidden pairs in* C_1*, such that for every exclusive choice* X_2 *from* C_2*, there does* not *exist a path from* s *to* t *in the set of arcs* $E_0' = (E_0 \setminus C) \cup X_1 \cup X_2$.[5]

We use the complement of the PWFP problem, in order to be able to relate a coordination set — ensuring that no refinement cycles can exist — to a solution for $\exists\forall\neg$PWFP that ensures that no $s - t$ path can be formed.

Showing that $\exists\forall\neg$PWFP is Σ_2^p-complete is straightforward using a reduction from the quantified satisfiability problem QSAT$_2$ that slightly adapts a standard reduction from 3-SAT to PWFP (cf. [24]).

Proposition 3.4.4. *The coordination problem is in* Σ_2^p.

Proof. To see that the coordination problem is in Σ_2^p, take a coordination instance $(\mathbf{T}, G)$ and a $K > 0$. Nondeterministically, guess a set of arcs $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$ and verify whether: (i) $|\Delta| \leq K$, (ii) each $(T_i, \prec_i \cup \Delta_i)$ is acyclic, and (iii) $(\mathbf{T}, (T, E \cup \Delta))$ is a yes-instance of the CVP problem. The first two verifications can be done in polynomial time, while the last verification requires the consultation of an NP-oracle. Hence, the problem is in Σ_2^p. □

To prove that coordination is also Σ_2^p-hard, we will reduce $\exists\forall\neg$PWFP to the coordination problem. Intuitively, the reduction consists of three parts, two of which are interesting and one that is not. The non-interesting part is the same as the trivial part for the previous reduction, i.e., it consists of replicating vertices and non-interesting arcs. The first interesting part is also the same as the interesting part of the previous reduction: for every forbidden pair in C_2, we create the path-blocking gadget of Figure 3.5. The second interesting part is the transformation of forbidden pairs in C_1. For this, we introduce the so-called *forced-choice gadget*, which extends the path-blocking gadget.

The idea behind the reduction is to link the finding of a coordination set to the finding of an exclusive choice for C_1. That is, if we have found a coordination set — which means that whatever refinement arcs the agents might add, no refinement cycle can be created — then we should automatically have an exclusive choice for C_1 such that no path from s to t can be found — whatever exclusive choices we come up with for C_2.

The forced-choice gadget (Figure 3.7) forces a coordination set to be constructed by adding only constraints within the forced-choice-gadget agents ('C_1-agents'). This is accomplished by introducing two potential paths from s to t through each C_1-agent: if the agent adds either refinement arc (a^j, b^j) or refinement arc (c^j, d^j), a path from s to t is

[4]Note that in case $K = 0$, this problem equals the CVP problem.

[5]Here, $E_0 \setminus C$ is a shorthand for the set of arcs from E_0 that do not occur in any forbidden pair.

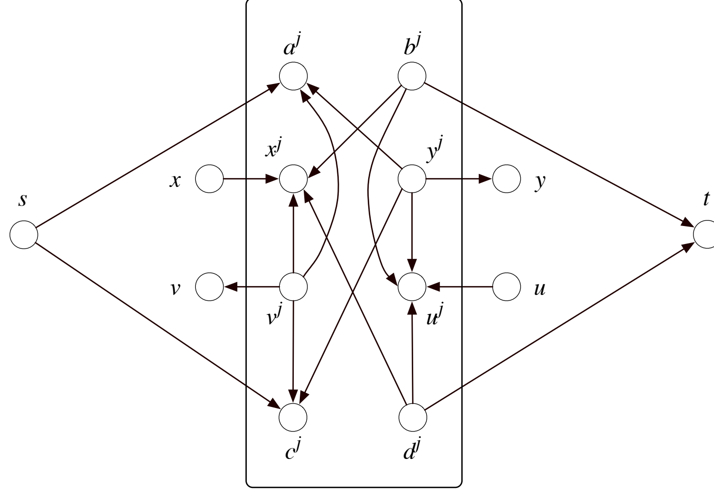


Figure 3.7: *The subgraph containing a forced-choice gadget, constructed for a forbidden pair* $c_j = \{(x^j, y^j), (u^j, v^j)\} \in C_1$.

created, and, since we connect t to s, a refinement cycle is created. The forced-choice gadget allows both these potential paths to be broken by adding a single constraint: if either (x^j, y^j) or (u^j, v^j) is added, then neither (a^j, b^j) nor (c^j, d^j) can be added without creating a local cycle.

Finally, we have to slightly alter the agent connecting t to s for this reduction. Since all inter-agent cycles pass through the agent connecting t to s, a coordination set can be found simply by 'blocking' this agent. Therefore, we ensure that this agent has $K+1$ refinement arcs to choose from in connecting t to s; breaking them all would require $K+1$ constraints, while we are only allowed to find a coordination set of size K.

Specifically, the reduction is specified as follows:

1. For every $v_i \in V$, $T_i = \{v_i\}$. For every pair $c_j = (\{x, y\}, \{u, v\})$ occurring in C, T_{n+j} contains the additional nodes x^j, y^j, u^j, v^j. Moreover, for every pair $c_j = (\{x, y\}, \{u, v\})$ occurring in C_1, T_{n+j} contains four additional nodes (tasks): a^j, b^j, c^j and d^j. Finally, T_{n+m+1} ($m = |C|$) contains the node s_0, and the $K+1$ ($K = |C_1|$) nodes $t_0, \ldots, t_k$.

2. The set of arcs E contains the following elements:

    (a) For every arc $e = \{u, v\} \in E_0$ not occurring in a pair of arcs in C, e occurs in E;

    (b) For every constraint-pair of arcs $c_j = \{(x, y), (u, v)\} \in C$, E contains the following arcs
    $$(x, x^j), (y^j, y), (u, u^j), (v^j, v), (y^j, u^j), (v^j, x^j)$$

    (c) For every pair of arcs $\{(x, y), (u, v)\} \in C_1$, E contains the following additional arcs:

$$(v^j, a^j), (v^j, c^j), (y^j, a^j), (y^j, c^j), (b^j, x^j), (b^j, u^j)$$
$$(d^j, x^j), (d^j, u^j), (s, a^j), (s, c^j), (b^j, t), (d^j, t)$$

(d) Finally, E contains the arcs $\{(t, t_0), \ldots, (t, t_k)\}$ and (s_0, s).

Proposition 3.4.5. *The coordination problem is Σ_2^p-complete*

Again, we will prove this proposition by showing that the reduction from ∃∀¬PWFP to CP is correct, i.e., we must prove that:

1. The transformation results in a correct instance of the CP problem; this means that $G = (T, E)$ must be a dag.
2. A yes-instance of the ∃∀¬PWFP problem maps to a yes-instance of the CP problem.
3. A yes-instance of the CP problem maps to a yes-instance of the ∃∀¬PWFP problem.

Proof. To understand that the transformation results in an acyclic graph $G = (T, E)$, note that all additional nodes in the CP instance either have only incoming arcs, or only outgoing arcs. Hence, these new nodes cannot introduce a cycle.

A yes-certificate for ∃∀¬PWFP consists of an exclusive choice X_1 for C_1. We can directly map X_1 to a yes-certificate Δ for CP: $(x, y) \in X_1 \rightarrow (x^j, y^j) \in \Delta$. To verify that Δ coordinates the CP instance, first note that any possible inter-agent cycle must include a path from s to t, because the original PWFP-graph is acyclic. Second, note that after adding Δ to the CP-instance, the following agents still have the capability to add refinement arcs: agent A_{n+m+1} ($m = |C|$) can add the arcs $\{(t_0, s_0), \ldots, (t_k, s_0)\}$ (each of which effectively creates a path from t to s), and the agents corresponding to a C_2-constraint-pair can add exactly one refinement arc, from a choice of two: agent A_{n+j} ($c_j \in C_2$) can either add (x^j, y^j), or (u^j, v^j).

Clearly, the set of possible combinations of refinement arcs for the C_2-agents corresponds to the set of all possible exclusive choices X_2 for C_2. Consequently, since in the ∃∀¬PWFP instance there does not exist an exclusive choice X_2 that creates a path from s to t (given X_1), there does not exist a set of refinement arcs in the CP instance connecting s to t. Hence, Δ is a coordination set.

We must show that *any* coordination set Δ directly maps to an exclusive choice X_1 for C_1. There are three types of agents that are capable of adding constraints to Δ:

1. The agent A_{n+m+1}: every inter-agent cycle in the coordination graph must pass through this agent, since it connects t to s. Hence, we can coordinate the instance by 'blocking' this agent. However, that would take $K + 1$ constraints, since agent A_{n+m+1} can add $K + 1$ refinement arcs that would connect s to t.
2. The C_1-agents: the forced-choice gadget enables two direct paths from s to t, either by adding refinement arc (a^j, b^j) or by adding (c^j, d^j). Adding either (x^j, y^j) or (u^j, v^j) blocks both these direct paths.

3. The C_2-agents: clearly though, the C_2-agents cannot break the direct paths from s to t created by the C_1-agents.

Hence, a coordination set must consist of arcs from C_1-agents, exactly one per agent, since we have K C_1-agents, and we are allowed to use K constraints.

It is easy to see that a coordination set indeed maps to an exclusive choice X_1 solving the PWFP instance: note that in the coordinated CP instance, agent A_{n+m+1} is unconstrained, allowing it to add refinement arcs connecting t to s. If other agents (i.e, C_2-agents, since these are the only other agents capable of adding refinement arcs) would be able to add refinement arcs connecting s to t, then it would be possible to create an refinement cycle. This is not possible, however, since Δ is a coordination set. Hence, no path can be created from s to t, and consequently, X_1 is a yes-certificate for the $\exists\forall\neg$PWFP instance. □

3.4.3 Subclasses in NP

We have not been able to find any non-trivial coordination instances in P, but we can identify structural characteristics of a coordination instance without which the coordination problem is in NP, and coordination verification in P.

Note that, intuitively, the difficulty with verifying a coordination set Δ is that we have to verify, for a possibly exponential number of sets $R = \bigcup_{i=1}^{n} R_i$ of refinement arcs, whether $E \cup \Delta \cup R$ is acyclic if, for all $i = 1, \ldots, n$, $\prec_i \cup \Delta_i \cup R_i$ is acyclic.

In this section, we identify a class of coordination instances for which, in order to solve the coordination verification problem, we merely have to check whether $E \cup \Delta \cup REF$ contains no inter-agent cycles. We will show that this check can be performed in polynomial time, so CVP is in P for these instances and, consequently, coordination itself is in NP.

Definition 3.4.6 (Local Planning Cycle). *A Local Planning Cycle, abbreviated as locplan-cycle, for agent A_i is an intra-agent cycle $C \subseteq (REF_i \cup E_i)$, such that $C \cap REF_i \neq \emptyset$ and $C \cap E_i \neq \emptyset$.*

Proposition 3.4.7. *Let $I = (\mathbf{T}, G = (T, E))$ be a coordination instance such that its coordination graph contains no locplan-cycles, then the following two assertions are equivalent:*

1. *The instance I is coordinated.*

2. *The coordination graph contains no inter-agent cycles.*

Proof.

(1: instance coordinated → no inter-agent cycles)

Suppose on the contrary that the instance is coordinated, yet the coordination graph still contains an inter-agent cycle C.

The fact that I is coordinated means that there does *not* exist a set $R = \bigcup_{i=1}^{n} R_i$, $R \subseteq REF$, such that (i) for all $i = 1, \ldots, n$ $: \prec_i \cup R_i$ is acyclic, while (ii) $(E \cup R)$ contains a cycle. Hence, the presence of the inter-agent cycle C implies that if we choose $R = REF \cap C$, then there must be at least one agent for which $\prec_i \cup R_i$ contains a cycle C'.

It follows almost immediately that C' is a locplan-cycle:

- C' cannot consist exclusively of arcs in $\prec_i$, because $\prec$ is acyclic;
- C' cannot consist exclusively of arcs from R_i, otherwise C would not be an elementary cycle: if all arcs in C' are also in C, then C is not elementary.

The presence of the locplan-cycle C' is a contradiction.

(2: no inter-agent cycle → instance coordinated)
If the coordination graph contains no inter-agent cycle, then it does not contain a refinement cycle. Hence, by Theorem 1, the instance is coordinated.

□

We cannot directly claim that if an instance contains no locplan-cycle, then it is in NP. We must also require that it is not possible that after adding some arcs Δ to $\prec_i$ (e.g., for coordination purposes), $\prec_i \cup \Delta \cup REF_i$ contains a locplan-cycle (REF defined with regard to the precedence relation $\prec \cup \Delta$). In other words, the coordination instance must not be *refinable* to a coordination instance that contains a locplan-cycle.

Proposition 3.4.8. *Let $I = (\mathbf{T}, (T, E)\ )$ be a coordination instance that cannot be refined to contain a locplan-cycle, i.e., there exists no set of additional constraints $\Delta = \bigcup_{i=1}^{n} \Delta_i$, such that:*

1. $\Delta_i \subseteq (T_i \times T_i)$,
2. $\prec_i \cup \Delta_i$ *is acyclic, and*
3. *the coordination graph contains a locplan-cycle.*

Then I is in NP.

Proof. Let Δ be a solution for I, i.e., $I' = (\mathbf{T}, (T, E \cup \Delta)\ )$ is a coordinated instance. Verifying that Δ is solution for I can be done by verifying that the coordination graph contains no inter-agent cycle, according to Proposition 3.4.7. To prove that I is in NP, we need to show that this check can be performed in polynomial time.

Consider the subset feedback arc set problem, which is defined as follows:

> Let $I_{SFAS} = (V, E_0, X)$, with (V, E_0) a directed graph and $X \subseteq E_0$, find a minimum subset $F \subseteq E_0$, such that F contains at least one arc from every directed cycle in (V, E_0) that also intersects X.

The set of inter-agent cycles in the coordination graph is the set of cycles intersecting $INTER$, so we can choose $X = INTER$.

As SFAS is in NP[6], we can verify in polynomial time that $F = \emptyset$ is a subset feedback arc set for the coordination graph of $I' = (T, E \cup \Delta)$. This implies that we can verify in polynomial time that the coordination graph of I' contains no inter-agent cycles. □

[6]The decision variant of SFAS is in NP, SFAS itself is in NPO.

At the moment, we do not yet know if we can verify, in polynomial time, whether an arbitrary coordination instance can contain locplan-cycles, i.e., whether Proposition 3.4.8 applies. However, even if such a check would require exponential time, Proposition 3.4.8 can still be of value if we can prove that a certain class of instances cannot contain locplan-cycles. For instance, in the following two corollaries, we identify classes of coordination instances that are in NP, on account of the fact that these instances cannot contain a locplan-cycle.

Corollary 3.4.9. *The set of coordination instances for which each agent has either at most one task in T^{in}, or at most one task in T^{out}, is in NP.*

Proof. We will prove that we cannot form a locplan-cycle. Let C be an inter-agent cycle, let $R_i = REF \cap (T_i \times T_i)$ for all $i = 1, \ldots, n$, and let $R_i(C)$ be the intersection of C with R_i.

For this type of coordination instance, all arcs in R_i have precisely one task (node) in common: if, for instance, an agent A_i has exactly one task $t \in T^{in}$, then every arc in R_i has starting vertex t. Hence, C cannot be an elementary inter-agent cycle in case $R_i(C)$ contains more than one arc.

Due to Proposition 3.3.1, we can rule out the case where $R_i(C)$ is a single arc e: any cycle in the coordination graph — including a locplan-cycle — must contain at least two arcs from REF. □

Corollary 3.4.10. *The set of coordination instances in which for all agents A_i, the sets $T^{in} \cap T_i$ and $T^{out} \cap T_i$ are totally ordered, is in NP.*

Proof. Let C be an inter-agent cycle, and suppose on the contrary that it is possible that C induces a locplan-cycle C' in some agent A_i. Let $R_i = REF \cap (T_i \times T_i)$, and let $R_i(C)$ be the intersection of C (and C') with R_i.

From Proposition 3.3.1, we know that C' must contain at least two refinement arcs. If we denote $C' = (e_1, \ldots, e_m)$, then let e_j be the first arc in $R_i(C)$ and let e_k be the last arc in $R_i(C)$ (first and last defined in terms of the indices i of arcs e_i in C').

We have $(\mathrm{ran}(e_k), \mathrm{dom}(e_j)) \in \prec_i$, since all arcs from $e_k \ldots e_m$ to $e_1 \ldots e_j$ are in E. However, we also have $(\mathrm{dom}(e_j), \mathrm{dom}(e_k)) \in \prec_i$, because all arcs in T^{in} are totally ordered. Together, this implies $e_k^{-1} = (\mathrm{ran}(e_k), \mathrm{dom}(e_k)) \in \prec_i$. This, however, means that e_k cannot be in R_i. □

3.4.4 Approximability of the coordination problem

In approximation theory a commonly used measure for approximation quality consists of the quotient of the costs of an approximated versus the cost of an optimal solution. This is the so-called *approximation ratio*. An approximation ratio of 1 indicates optimality, while large approximation ratios indicate poor approximations. The approximation ratio plays an important role in the definition of a well-known complexity class in approximation theory: the class APX. An optimization problem (such as CP) is said to be in APX, if a polynomial time approximation algorithm exists such that for every instance (and particularly for very large ones) the approximation ratio in bounded from above by a constant.

Unfortunately, it is very unlikely that we are able to find constant-ratio approximations for the coordination problem, since we will show in this section that the APX-hard Feedback

Vertex Set problem (FVS) reduces to the coordination problem. This means that if we are able to find constant-ratio approximations for the coordination problem, then the reduction below would immediately yield a constant-ratio approximation for FVS. However, despite much research effort, the best known approximations for FVS are $\mathcal{O}(\log|V|\log\log|V|)$ [21, 9].

Although slightly disappointing, it is of course not surprising that coordination is APX-hard, as it is outside NPO. However, the reduction will show that even for severely restricted coordination instances, the APX-hardness holds. In fact, we reduce the FVS problem to coordination instances where agents have at most two tasks. Due to Propostion 3.4.8, these instances are clearly in NP, yet they are still APX-hard.

The feedback vertex set problem is defined as follows:

Definition 3.4.11. *The Feedback Vertex Set problem is: given a directed $G_0 = (V, E_0)$ and integer K, find a subset of vertices $F \subseteq V$ such that F contains at least one vertex for every directed cycle in G, and $|F| \leq K$.*

To reduce FVS to CP, we split each vertex up into two vertices — one vertex incident on all incoming arcs, the other incident on all outgoing arcs — and create an agent for those two tasks. The reduction is illustrated in Figure 3.8.

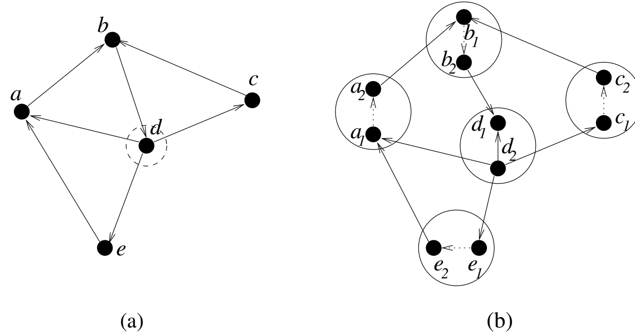


Figure 3.8: *(a) an FVS instance and (b) its corresponding CP instance, coordinated by adding constraint (d_2, d_1).*

More formally, given a FVS instance $G_0 = (V, E_0)$, we obtain a coordination instance $(\mathbf{T}, G = (T, E))$ by the following transformation:

1. For every $v_j \in V$, we construct the agent A_j, having set of tasks $T_j = \{t^i_j, t^o_j\}$.
2. For every $(v_j, v_k) \in E_0$, E contains the arc (t^o_j, t^i_k).

The correspondence between a feedback arc set and a coordination set is given by:

$$v^j \in F \leftrightarrow (t^o_j, t^i_j) \in \Delta$$

Proposition 3.4.12. *The coordination problem, restricted to instances where is each agent is allowed at most two tasks, is APX-hard.*

The correctness of Proposition 3.4.12 can be understood by noting that a cycle in the FVS-instance maps to a refinement cycle in the coordination graph: that is, if all agents A_j along the cycle were to add the refinement arc (t_j^i, t_j^o), then a cyclic joint plan would result. The transformed CP instance is solved if and only if, for every refinement cycle, at least one agent A_j intersected by the cycle adds the constraint (t_j^o, t_j^i). Consequently, a coordination set of cardinality K maps to a feedback vertex set of cardinality K — and vice versa.

Chapter 4

Approximation Techniques

> *Focus on the solution, not on the problem.* — Terry Goodkind

> *Your mind can only hold one thought at a time; make it a positive and constructive one.* — H. Jackson Brown Jr.

Quotations like the above are readily associated with happiness gurus, psychiatrists, and fantasy novelists. Disregarding their advice, we have found that by focusing on the problem in the previous chapter, we are now able to narrow the focus of our 'constructive thoughts'. In fact, by focusing on the problem in the previous chapter, in particular on the complexity of the coordination problem, we now know that we need not seek for either optimal algorithms, or for algorithms that approximate the optimal solution to within a constant factor — no tractable algorithm meeting either of these criteria exists. Hence, the best we can hope for is to find heuristic approximation techniques that turn out to be efficient.

If it is difficult to compare algorithms with regard to their (worst-case) efficiency, then how are we supposed to judge if one approximation technique is better than another? For one, by noting that we are not simply dealing with some problem from graph theory, but rather with a problem regarding the coordination of autonomous agents. Definition 2.3.3 of the coordination problem seeks only guarantee a feasible solution while maximizing *planning autonomy*. An autonomous agent, however, will also want to maximize its *coordination autonomy*, that is, to be able to have some influence on the process that additionally constraints its goal and the goals of other agents.

Ideally, this chapter would have presented a number of negotiation protocols, compared to each other using a set of criteria appropriate for evaluating multi-agent negotiation (cf. [17]). Unfortunately, this part of our research has not yet reached that level of maturity. Instead, we will present three different kinds of approximation techniques, and identify how each can be applied in, or adapted to, a multi-agent negotiation protocol.

In Section 4.1, we present the most naive of the three approaches: we simply add constraints one by one until we are certain that the composite task has been coordinated. In Section 4.2, we take the approach of reducing the coordination problem to a problem from graph theory, of the family of feedback arc set problems. Due to the fact that the feedback arc set problems are in NP, while coordination is outside NP, this reduction is — for general coordination instances — not weight-preserving, but at least it is always sound.

Finally, in Section 4.3, we present an approach to coordination that is based on the idea of partitioning the agents' goals into smaller sets of tasks. This is an inherently distributed approach to coordination, and it was first presented in [28].

4.1 Iteratively Constructing a Coordination Set

A coordination set is a subset of REF^{-1}. An arbitrary set $\Delta \subseteq REF^{-1}$ may not be a coordination set for two reasons:

1. $E \cup \Delta$ is cyclic,
2. $E \cup \Delta$ is not cyclic, but the coordination graph still contains a refinement cycle.

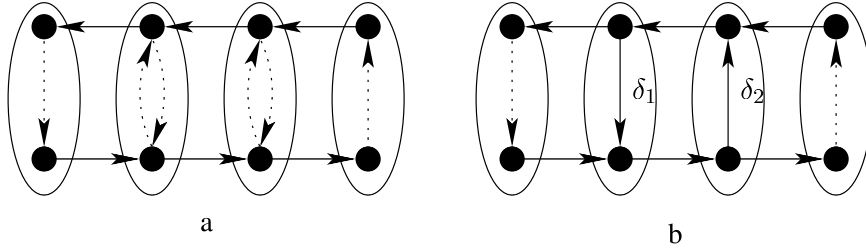


Figure 4.1: *(a): Unconstrained, (b): for $\Delta = \{\delta_1, \delta_2\}$, $E \cup \Delta$ is cyclic.*

In Case 2, agents are left with too much planning freedom, and, to coordinate this composite task, we must simply add more constraints. Case 1 is depicted in Figure 4.1. Before choosing a set Δ, in Figure 4.1a, there are no intra-agent constraints. After choosing $\Delta = \{1 \cup 2\}$, $E \cup \Delta$ contains a cycle. Thus, we must build a coordination set without making the precedence relation $E \cup \Delta$ cyclic.

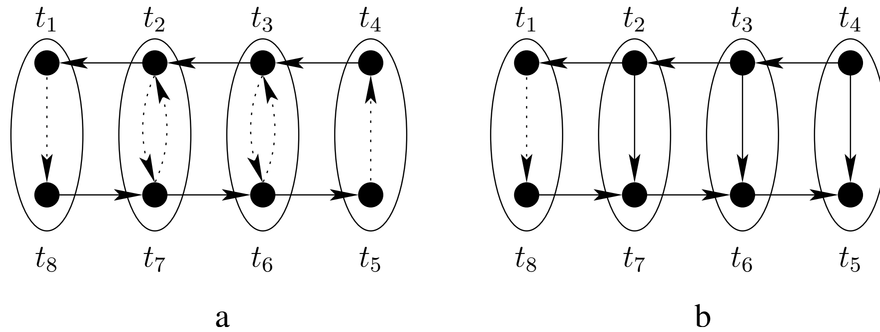


Figure 4.2: *(a): Unconstrained, (b): with new constraint set $[E \cup (t_2, t_7)]^+$.*

Figure 4.2 illustrates what happens if we add a single constraint. In Figure 4.2a (the same as Figure 4.1a), there are no intra-agent precedence constraints. By adding constraint $t_2 \prec t_7$, constraints $t_3 \prec t_6$ and $t_4 \prec t_5$ are also added to $\prec$. This is because $\prec$ is a

transitive relation. If $t_2 \prec t_7$ is added to E, then a path is created from t_3 to t_6 in E. Thus, $(t_3, t_6) \in \prec = [E]^+$. The same holds for (t_4, t_5).

Figure 4.2 illustrates that adding a single constraint from REF^{-1} doesn't make the precedence relation cyclic. The fact that an arc in REF^{-1} doesn't create a cycle is easy to understand, since the inverse of a refinement arc can be interpreted as a cycle-breaking arc. Some arcs in REF^{-1} are also in REF, though, the set of cycle-enabling arcs. A single refinement arc cannot make the precedence relation cyclic, however, since any cycle in the coordination graph contains at least two refinement arcs.

Lemma 4.1.1. *Let $\mathcal{T} = (T, E)$ be a composite task. For all $\delta \in (REF \cup REF^{-1})$, $(T, E \cup \{\delta\})$ is a composite task.*

Proof. The definition of a composite task imposes two conditions on the pair $(T, E \cup \{\delta\})$:

1. $E \cup \{\delta\} \subseteq T \times T$,
2. $E \cup \{\delta\}$ is acyclic.

The first condition is met since $E \subseteq T \times T$ and $\delta \in T \times T$. To prove the second condition, suppose on the contrary that $E \cup \{\delta\}$ is cyclic, and let $\delta = (t_1, t_2)$. Since $(C \setminus \{\delta\}) \subseteq \prec$, we can conclude $t_2 \prec t_1$.

case 1 $\delta \in REF$**:** According to Definition 3.1.5, $REF \cap INTRA^{-1} = \emptyset$. This contradicts $\delta = (t_1, t_2) \in INTRA^{-1}$.

case 2 $\delta \in REF^{-1}$**:** Then $\delta^{-1} = (t_2, t_1) \in REF$. According to Definition 3.1.5, $REF \cap INTRA = \emptyset$. This contradicts $(t_2, t_1) \in INTRA$.

□

Thus, we can add a single constraint from either REF or REF^{-1} without making the precedence relation cyclic. We will now show that adding such a constraint brings us closer to a coordinated composite task.

Lemma 4.1.2. *Let $\mathcal{T} = (T, E)$ be a composite task. If $REF_{\mathcal{T}} \neq \emptyset$, then if $\delta \in (REF_{\mathcal{T}}^{-1} \cup REF_{\mathcal{T}}^{-1})$ and $\mathcal{T}' = (T, E \cup \{\delta\})$, then $REF_{\mathcal{T}'} \subset REF_{\mathcal{T}}$.*

Proof. Recall the definition of REF:

$$REF = \bigcup_{i=1}^{n} (T_i^{in} \times T_i^{out}) \setminus \left(INTRA^{-1} \cup INTRA\right)$$

The set $\bigcup_{i=1}^{n} (T_i^{in} \times T_i^{out})$ is unchanged by adding δ to E. Furthermore,

$$INTRA_{\mathcal{T}'} \supseteq INTRA_{\mathcal{T}} \cup \{\delta\} \tag{4.1}$$

$$INTRA_{\mathcal{T}'}^{-1} \supseteq INTRA_{\mathcal{T}}^{-1} \cup \{\delta^{-1}\} \tag{4.2}$$

Hence,

$$REF_{\mathcal{T}'} \subseteq REF_{\mathcal{T}}$$

Also, since $\delta \in REF_{\mathcal{T}} \vee \delta^{-1} \in REF_{\mathcal{T}}$, but not $(\delta \in REF_{\mathcal{T}'} \vee \delta^{-1} \in REF_{\mathcal{T}'})$, we have that

$$REF_{\mathcal{T}'} \subset REF_{\mathcal{T}}$$

□

The set REF is finite, so after adding a finite number of additional constraints $\delta \in REF$, the set REF will be empty. If that is the case, the composite task has been coordinated, by corollary 4.1.3.

Corollary 4.1.3. *Let $\mathcal{T} = (T, E)$ be a composite task, such that $REF = \emptyset$. Then $G_{\mathcal{T}}$ is acyclic.*

Proof. By proposition 3.3.1, any cycle in the coordination graph of a composite task contains at least two arcs from the set REF. Consequently, if REF is empty, then the coordination graph is acyclic. □

Using Lemmata 4.1.1 and 4.1.2, and Corollary 4.1.3, we can define an algorithm to find an approximate coordination set. Lemma 4.1.1 can be used to show how we can iteratively add constraints from REF^{-1} to E. Corollary 4.1.3 provides the stop condition for the algorithm. Finally, Lemma 4.1.2 tells us that this stop condition will be reached using the iteration specified by lemma 4.1.1.

Algorithm 1 ITERATIVE COORDINATION

1: **Input:** composite task $\mathcal{T} = (T, E)$
2: **Output:** coordination set Δ
3: $\Delta = \emptyset$
4: **while** $REF_{(E \cup \Delta)} \neq \emptyset$ **do**
5: Choose $\delta \in REF^{-1}_{(E \cup \Delta)}$
6: $\Delta = \Delta \cup \{\delta\}$
7: **end while**
8: **return** Δ

In lines 4 and 5 of Algorithm 1, we denote the set REF by $REF_{(E \cup \Delta)}$ to emphasize that REF changes (is reduced) as a result of adding constraints to the precedence relation $[E \cup \Delta]^+$. In every iteration, a constraint $\delta \in REF^{-1}$ is selected for addition to Δ. We pick δ arbitrarily, though obviously the algorithm can be extended at this point to let a heuristic function guide the selection of the next constraint δ.

Theorem 2. *Given a composite task $\mathcal{T}$, Algorithm 1 returns a coordination set Δ.*

Proof. We need to prove that

1. for every iteration, $E \cup \Delta$ is acyclic,

2. on termination, $G_{\mathcal{T}'}$, $\mathcal{T}' = (T, E \cup \Delta)$, contains no refinement cycles.

ad 1 Initially, $\Delta = \emptyset$ and E is acyclic. Hence, $E \cup \Delta$ is acyclic initially.

In each iteration, Algorithm 1 adds constraints $\delta \in REF^{-1}_{E \cup \Delta}$ to Δ. By Lemma 4.1.1, adding a single $\delta \in REF^{-1}_{E \cup \Delta}$ to Δ will not make $E \cup \Delta \cup \{\delta\}$ cyclic.

ad 2 The algorithm stops when $REF_{E \cup \Delta} = \emptyset$. Lemma 4.1.2 tells us that this stop condition will be reached: the initial set REF is a finite set and in each iteration

$$REF_{E \cup \Delta \cup \{\delta\}} \subset REF_{E \cup \Delta}$$

Once $REF_{E \cup \Delta} = \emptyset$ — with $E \cup \Delta$ still acyclic — $G_{\mathcal{T}'}$ is acyclic according to Corollary 4.1.3. □

Algorithm 1 constrains agents until the set REF is empty. This means that all planning autonomy — with regard to goal refinements that are relevant for coordination — is taken away from the agents. An obvious enhancement of Algorithm 1 would be to terminate the algorithm as soon as the composite task has been coordinated. However, in Chapter 3 we have shown that the problem of checking whether a composite task is coordinated is co-NP-complete.

Even if it were possible to polynomially verify coordination, this would not ensure that an adapted version of Algorithm 1 would coordinate while leaving the agents some planning autonomy with regard to arcs in REF, as is illustrated in Figure 4.3. In Figure 4.3, it is shown that, by adding a single constraint (and at least one constraint must be added to coordinate this instance), the set REF might become empty.

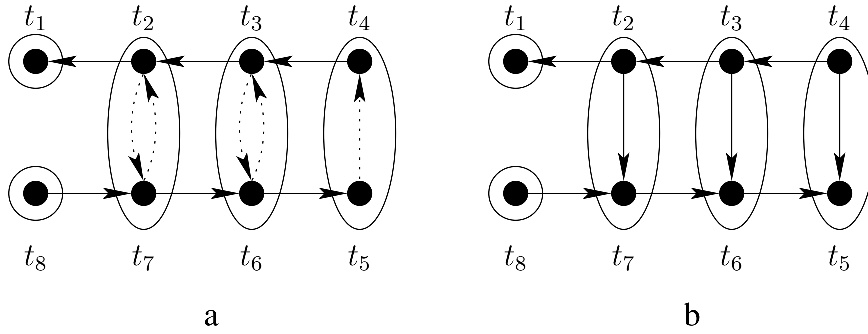


Figure 4.3: *(a): Unconstrained, (b): the set REF is empty after adding $t_2 \prec t_7$.*

Nevertheless, we might improve the average efficiency if we adapt Algorithm 1 to stop as soon as the coordination graph contains no more *inter-agent* cycles, a property which can be verified in polynomial time (see Section 4.2).

4.1.1 Adapting iterative coordination to enable negotiation

In Line 5 of Algorithm 1, an arbitrary $\delta \in REF^{-1}$ is chosen for placement in Δ. To enable agents to choose which constraints are added, we can replace this line with an auction, in which all agents submit bids for the addition of constraints. The lowest bid wins the auction, the winning agent A_i adds the corresponding constraint to Δ (and $\prec_i$) and he is paid the amount his bid for compensation.

The downside of this approach is illustrated in Figure 4.4. Suppose agent A_2 wins the auction. He agrees to add constraint δ (Figure 4.4b) and is paid the amount of his bid. In this case we see that agents A_3 and A_4 also receive a constraint. However, they have

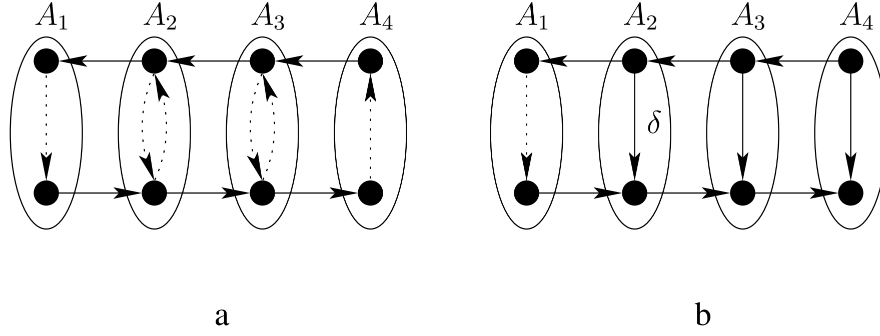


Figure 4.4: *(a): Uncoordinated instance and (b): after agent A_2 adds constraint δ.*

not agreed to accept these constraints and, furthermore, they receive no compensation for receiving these constraints.

The addition of a single constraint δ actually results in a set ϕ_δ of additional constraints:

$$\phi_\delta = [\prec \cup \{\delta\}]^+ - \prec$$

To ensure that agents only receive constraints for which they have entered a bid, all agents that have a constraint in ϕ_δ should be allowed to bid for δ. If the resulting aggregate bid for δ wins the auction round, all agents will be paid the amount of their bid.

Algorithm 2 is a coordination algorithm that starts an auction, in every iteration of the algorithm, to decide which constraint to add next. In one auction round, aggregate bids b_δ are collected for all $\delta \in REF^{-1}$. An aggregate bid b_δ is a set of bids b_{δ,A_i}, one bid for each agent A_i for which ϕ_δ intersects $T_i \times T_i$.

The aggregate bid that has the lowest cost (which is the sum of all the prices of the agent bids) wins the auction. To conclude the iteration, the winning δ (if there are more aggregate bids with the same total price, one is chosen at random) is added to the current Δ and the agents are paid the amount they specified in their bid.

The function MAKEBID in Line 12 represents the agent strategy. In the above negotiation protocol, the strategy of an agent should determine the price of a bid. If agents are autonomous, they can choose any strategy they like. In Algorithm 3, we give an example of an agent strategy.

In Algorithm 3, an agent calculates the cost (to him) of the set ϕ_δ. He makes one plan for his local goal where ϕ_δ does not need to be adhered to, and one plan where the constraints in ϕ_δ do need to be adhered to.[1] The difference in cost between the two is the price an agent assigns to the bid for constraint δ.

Algorithm 2 maintains coordination autonomy for all agents: an agent receives only constraints he has agreed to receive and he is compensated for these constraints by the amount he has specified in his bid.

[1] Note that since planning is NP-hard in general, the agents may need to use a heuristic function instead of actually constructing a plan.

Algorithm 2 ITERATIVECOORDINATIONAUCTION

1: **Input:** coordination instance $I = (\mathbf{T}, G = (T, E))$
2: **Output:** coordination set $\Delta \subseteq REF^{-1}$
3: $\Delta = \emptyset$
4: $\mathcal{T}' = (T, E \cup \Delta)$
5: **while** $G_{\mathcal{T}'}$ contains inter-agent cycles **do**
6: {Collect agent bids}
7: $B = \emptyset$
8: **for all** $\delta \in REF^{-1}$ **do**
9: $b_\delta = \emptyset$
10: **for all** $A_i \in A$ **do**
11: **if** $\phi_\delta \cap T_i \times T_i \neq \emptyset$ **then**
12: b_{δ,A_i} = invoke agent function MAKEBID(A_i, ϕ_δ)
13: $b_\delta = b_\delta \cup \{b_{\delta,A_i}\}$
14: **end if**
15: **end for**
16: $B = B \cup \{b_\delta\}$
17: **end for**
18: {Select cheapest constraint δ as bid winner}
19: $B_{min} = \{b \in B | \forall 1 \leq j \leq |REF| : \text{ totalPrice}(b) \leq \text{totalPrice}(b_j)\}$
20: Choose $b_{min} \in B_{min}$
21: $\delta_{min} = \text{constraint}(b_{min})$
22: {Increment coordination set and notify winners}
23: $\Delta = \Delta \cup \delta_{min}$
24: $W = \{A_i \in A | \phi_{\delta_{min}} \cap T_i \times T_i \neq \emptyset\}$
25: **for all** $i \in W$ **do**
26: price = price(b_{δ_{min},A_i})
27: pay(A_i, price)
28: **end for**
29: **end while**

Algorithm 3 MAKEBID

1: **Input:** Set of constraints ϕ_δ, agent A_i
2: **Output:** b_{δ,A_i}: agent A_i's bid for the addition of δ
3: $p_1 = \text{makePlan}(T_i, \prec_i)$
4: $p_2 = \text{makePlan}(T_i, \prec_i \cup \phi_\delta)$
5: price $= c(p_2) - c(p_1)$
6: $b_{\delta,i} = (\delta, \text{price})$
7: **Return** $b_{\delta,i}$

Proposition 4.1.4. *Let Δ be a coordination set resulting from Algorithm 2. For agent A_i, let*

$$\phi_{\Delta,A_i} = ([\prec \cup \Delta]^+ - \prec) \cap (T_i \times T_i)$$

For all $e \in \phi_{\Delta,A_i}$, agent A_i has received the payment it has specified.

Proof. Suppose on the contrary that there exists an $e \in \phi_{\Delta,A_i}$ for which agent A_i has not specified or received payment.

Algorithm 2 chooses one constraint δ per iteration. Hence, there must be an iteration n such that

$$e \in \phi_\delta = [\prec \cup \Delta_{n-1} \cup \{\delta\}]^+ - [\prec \cup \Delta_{n-1}]^+$$

Since $\phi_\delta \cap (T_i \times T_i \neq \emptyset)$, agent A_i has been asked (Line 12) to submit a bid for δ. In the bid for δ, agent A_i has specified the aggregate price for all constraints in $\phi_\delta \cap T_i \times T_i$, including e. Since δ is added to Δ in iteration n, agent A_i has been paid the amount of his bid (Line 27), contrary to the assumption. □

4.2 Coordination and Feedback Arc Sets

The coordination graph contains all cycles that can be created through refinement. If we remove a feedback arc set F, i.e., a set of arcs such that every cycle in the graph has an arc in F, then the coordination graph is acyclic. Thus, there exists an intuitive transformation from the coordination problem to the feedback arc set problem. All of the variations of the feedback arc set problem discussed in this section are NP-complete [9]. As the coordination problem is outside NP[2], the transformation specified in this section is not a weight-preserving Turing reduction.[3]

Definition 4.2.1 (FAS). *Given a directed graph $G = (V, E_0)$, find a minimum feedback arc set, i.e., a subset of arcs $F \subseteq E_0$ such that F contains at least one arc from every directed cycle in G.*

Removing a feedback arc set from a graph makes that graph acyclic. For coordination purposes, however, we do not need to remove *all* cycles from the coordination graph; we are only interested in removing refinement cycles. The *Subset* Feedback Arc Set problem (SFAS) is used for finding a feedback arc set such that only *interesting cycles* are covered.

Definition 4.2.2 (SFAS). *Given an instance $I = (V, E_0, X)$, with (V, E_0) a directed graph and $X \subseteq E_0$, find a minimum subset $F \subseteq E_0$, such that F contains at least one arc from every directed cycle in (V, E_0) that also intersects X.*

From definition 4.2.2, we see that the set of interesting cycles is characterized by a subset $X \subseteq E_0$ of arcs. Thus, we need to find a set of arcs that characterizes the set of refinement cycles in the coordination set. Due to the fact that the coordination verification problem is co-NP-complete, there does not exist a set X of arcs (specifiable beforehand) that characterizes the set of refinement cycles, since the existence of interesting cycles in an SFAS instance is polynomially verifiable. We can, however, characterize a superset of

[2]unless the polynomial hierarchy collapses

[3]See for instance [27, 14] for the definition of a Turing reduction.

the set of refinement cycles; the set of all *inter-agent* cycles is the set of cycles intersecting *INTER*.

Choosing $X = INTER$, removing a subset feedback arc set from the coordination graph will remove all inter-agent cycles and thus all refinement cycles from the coordination graph. However, note that we are not at liberty to remove any arbitrary set of arcs from the the coordination graph. Recall that $E_{\mathcal{T}} = INTER \cup INTRA \cup REF$. The sets *INTER* and *INTRA* are part of the precedence relation $\prec$. Constraints in $\prec$ must be adhered to at all times, so these arcs may not be removed. From Proposition 3.3.1, we know that *every* cycle in the coordination graph must contain at least two arcs from *REF*. This means that we can construct a feedback arc set for the coordination graph using only arcs in *REF*.

To direct a feedback arc set solver to consider only refinement arcs for placement in a feedback arc set F, we solve the Blackout Subset Feedback Arc Set problem instead. In the BSFAS problem, arcs that are marked as blackout arcs may not be placed in the feedback arc set.

Definition 4.2.3 (BSFAS). *Given an instance $I = (V, E_0, X, B)$, with (V, E_0) a directed graph, $X \subseteq E_0$ and $B \subset E_0$, find a minimum subset $F \subseteq (E_0 \setminus B)$, such that F contains at least one arc from every directed cycle in (V, E_0) that also intersects X.*

If we have a subset feedback arc set $F \subseteq REF$, we can remove F from the coordination graph (thereby removing all refinement cycles, coordinating the instance) by adding its inverse F^{-1} to the set of precedence constraints E. However, as exemplified by Figure 4.1, adding an arbitrary set of constraints can create a cyclic precedence relation. We will show that if the feedback arc set F contains no *redundant* arcs[4], then adding $\Delta = F^{-1}$ to E will not introduce any cycles. Of course, for $E \cup F^{-1}$ to be acyclic, the feedback arc set itself must be acyclic. In [27], it is proved that if a feedback arc set F is cyclic, then it contains redundant arcs.

Proposition 4.2.4 (From [27]). *For every BSFAS instance $I = (V, E_0, X, B)$ with solution F such that $X \cap F = \emptyset$, there exists at least one redundant arc in every cycle in F.*

Before Proposition 4.2.4 we said that adding $\Delta = F^{-1}$ will not introduce cycles if F contains no redundant arcs. Proposition 4.2.4 states the extra condition that $X \cap F$ must be empty. With $X = INTER \subseteq B$, i.e., all arcs in X are blackout arcs, this condition is always met in our setting. Using Algorithm 4, we can inspect each of the arcs in F and remove an arc if it is redundant.

In two steps, we will now prove that we can find a coordination set Δ by inverting a feedback arc set F. We will prove that (i) adding Δ to the precedence relation will not make $E \cup \Delta$ cyclic, and that (ii) after adding Δ to E, the coordination graph contains no refinement cycles. However, we will first specify the reduction from the coordination problem to BSFAS.

[4]Given a feedback arc set F, an arc $e \in F$ is *redundant* if $F - e$ is still a feedback arc set.

Algorithm 4 REDUCEFAS

Input: feedback arc set $F = \{e_1, \ldots, e_n\}$
Output: feedback arc set containing no redundant arcs
for all $1 \leq i \leq n$ **do**
  if ISFAS$(F - e_i)$ **then**
    $F = F - e_i$
  end if
end for
return F

Algorithm 5 CP $\preceq$ BSFAS

Input: coordination instance $I_c = (\mathbf{T}, G)$
Output: coordination set Δ
$I_{fas} = (V_{\mathcal{T}}, E_{\mathcal{T}}, INTER, INTER \cup INTRA)$
$F = \text{BSFAS}(I_{fas})$
$F' = \text{REDUCEFAS}(F)$
return $\Delta = F'^{-1}$

Lemma 4.2.5. *Let $I = (\mathbf{T}, G)$ be a coordination instance, and let F be a feedback arc set, free of redundant arcs, for the BSFAS instance $I_{fas} = (V_{\mathcal{T}}, E_{\mathcal{T}}, INTER, INTER \cup INTRA)$, and let $\Delta = F^{-1}$. Then $E \cup \Delta$ is acyclic.*

Proof. Let $\mathcal{T} = (T, E)$ be the composite task prior to adding Δ, and let $G_{\mathcal{T}}$ be its corresponding coordination graph.

Suppose on the contrary that $E \cup \Delta$ contains a cycle C. Since both E and Δ are acyclic, C must contain arcs from both E and Δ; let $\Delta_C = \{\delta_1, \ldots, \delta_k\} = C \cap \Delta$, and let $E_C = E \cap C$. Note that two consecutive arcs from Δ_C are connected by a path in E, i.e., $\text{ran}(\delta_i) \prec \text{dom}(\delta_{i+1})$.

Since δ_i^{-1} is non-redundant ($i = 1, \ldots, k$), there is at least one cycle C_i in $G_{\mathcal{T}}$ that is only covered in F by δ_i^{-1}.

We can construct the cycle $C' = \left(\bigcup_{i=1}^{k} C_i \setminus \delta_i^{-1} \right) \cup E_C$ in $G_{\mathcal{T}}$ (illustrated in Figure 4.5), i.e., C' is the cycle constructed from the paths (provided by $C_i \setminus \delta_i^{-1}$) from $\text{dom}(\delta_i)$ to $\text{ran}(\delta_i)$, for all $i = 1, \ldots, k$, united with the paths in E_C connecting consecutive δ_i.

The cycle C' is not covered in F: for no i does the path from $\text{dom}(\delta_i)$ to $\text{ran}(\delta_i)$ contain an arc in F, since δ_i^{-1} is the only arc covering C_i. Also, since $E_C \cap F = \emptyset$, the set E_C contains no arc that covers C' in F. But if C' is not covered in F, then F is not a feedback arc set, which is a contradiction. □

Theorem 3. *Let $I = (\mathbf{T}, G)$ be a coordination instance and let $\Delta = Algorithm5(I)$. Then Δ is a coordination set for I.*

Proof. First, note that $\mathcal{T}' = (T, E \cup \Delta)$ is a composite task: $E \cup \Delta$ is acyclic, due to Lemma 4.2.5, and also $\Delta \subseteq T \times T$.

Second, it is not difficult to see that $G_{\mathcal{T}'}$ contains no refinement cycles. In fact, $G_{\mathcal{T}'}$ doesn't contain any inter-agent cycles: every inter-agent cycle C in $G_{\mathcal{T}}$ ($\mathcal{T} = (T, E)$, i.e.,

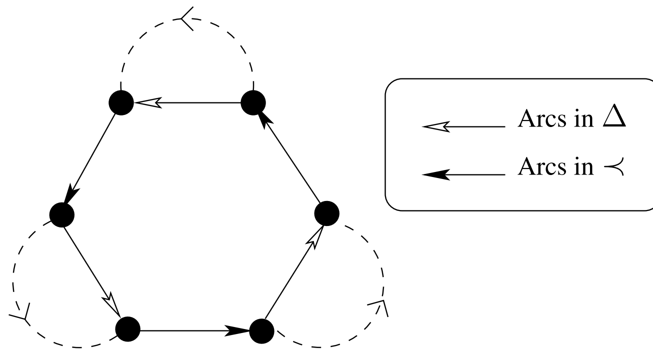


Figure 4.5: *The outer cycle is not covered in the feedback arc set.*

the uncoordinated composite task) has at least one arc $e \in F = \Delta^{-1}$. Inverting this arc breaks C. □

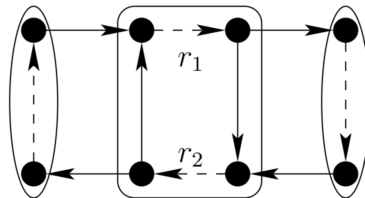


Figure 4.6: *Despite the presence of an inter-agent cycle, this instance is coordinated.*

For the family of feedback arc set problems, $\mathcal{O}(\log |V| \log \log |V|)$-approximations have been developed [21, 9]. However, as the coordination problem is outside NP, these approximation ratios do not apply to general coordination instances. In Figure 4.6, which repeats Figure 3.3, we show a coordination instance for which an optimal feedback arc set translates to a sub-optimal coordination set: the coordination instance in Figure 4.6 is coordinated, as a cycle in E_J can only be created if agent A_2 makes a locally cyclic plan, by adding both refinements r_1 and r_2. Thus, the empty set is a solution for the coordination instance. A solution for the corresponding BSFAS instance, is not empty, however, as there exists an inter-agent cycle that must be covered. Consequently, the approximation ratio of the BSFAS transformation is not bounded by a constant: a non-empty coordination set is infinitely poorer than an empty coordination set.

However, for coordination instances that cannot contain locplan-cycles (Definition 3.4.6), the above BSFAS transformation is a weight-preserving Turing reduction, because of the fact that for such instances, the instance is coordinated iff the coordination graph contains no inter-agent cycles. This means that for those coordination instances, the approximation ratios for the FAS problem also apply to the coordination problem.

4.2.1 Negotiation and BSFAS coordination

Algorithm 5 is an inherently centralized approach to approximating the coordination problem. However, we can use the idea of solving the coordination problem with BSFAS to formulate an approximation technique that gives the agents some control over the distribution of constraints. Effectively, the algorithm we present in this section takes an approach opposite to Algorithm 2 of Section 4.1: instead of incrementally adding constraints to an initially empty coordination set, we remove constraints from an initially maximal coordination set. In fact, we remove *redundant arcs*, as long as the set of constraints is a feedback arc set, and consequently, a coordination set.

As an initial feedback arc set, we can choose $F = REF$. Thus, initially, agents have no planning autonomy (with regard to coordination-relevant refinements, at least). By repeatedly removing redundant arcs from F (and $F = REF$ always contains redundant arcs — see Proposition 4.2.6), agents are returned planning autonomy. Agents negotiate over who may remove which redundant arc next. Coordination autonomy in this algorithm is thus the freedom to negotiate over who will regain the most planning autonomy.

Algorithm 6 SFASCoordinationAuction

1: **Input:** coordination instance $I = (\mathbf{T}, G)$, (blackout subset) feedback arc set F
2: **Output:** coordination set $\Delta \subseteq REF^{-1}$
3: **while** F contains redundant arcs **do**
4: $\quad\Delta = F^{-1}$
5: $\quad\quad$ {Find redundant arcs}
6: $\quad R = \{e \in F | F - e$ is a feedback arc set $\}$
7: $\quad\quad$ {Invite agents to enter bids}
8: $\quad B = \emptyset$
9: $\quad$ **for all** $\delta \in R^{-1}$ **do**
10: $\quad\quad b_\delta =$ invoke makeBidFAS(agent(δ), Δ, δ)
11: $\quad\quad B = B \cup \{b_\delta\}$
12: $\quad$ **end for**
13: $\quad\quad$ {Find highest bid}
14: $\quad\Delta_{max} = \{b \in B | \forall 1 \leq i \leq n : \text{price}(b) \geq \text{price}(b_i)\}$
15: $\quad$ Choose $\delta_{max} \in \Delta_{max}$
16: $\quad\quad$ {Decrement feedback arc set and bill the winner}
17: $\quad F = F - \delta_{max}^{-1}$
18: $\quad$ bill(agent(δ_{max}), price($b_{\delta_{max}}$))
19: **end while**

In a single iteration of Algorithm 6, the set of redundant arcs is found first. Then, for each redundant arc, the corresponding agent (i.e., agent A_i if arc $\delta \in T_i \times T_i$) is asked to submit a bid. The highest bid (ties are broken randomly) wins the auction. The winning agent may (and must) remove δ from Δ, and in return he must pay the amount of his bid.

The function makeBidFAS represents the agent strategy. In Algorithm 7, we have given an example of an agent strategy, and it is similar to Algorithm 3. An agent makes two plans: one for which the set of (additional) constraints is Δ, one for which the set of (additional) constraints is $\Delta - \delta$. The difference in cost between both plans represents the

Algorithm 7 MAKEBIDFAS

1: **Input:** Agent A_i, current set of constraints Δ, redundant constraint δ
2: **Output:** Bid b_δ
3: $p_1 = \text{plan}(T_i, \prec_i \cup \Delta)$
4: $p_2 = \text{plan}(T_i, \prec_i \cup (\Delta - \delta))$
5: $\text{savings} = c(p_1) - c(p_2)$
6: $b_\delta = (\delta, \text{savings})$
7: **Return** b_δ

amount the agent saves by not having to adhere to δ; the agent assigns this amount to the price of his bid.

We would like to have some guarantee that removing redundant arcs in the fashion of Algorithm 6 will result in a coordination set of low cardinality. This means that we want to be able to remove a lot of arcs. Initially at least, a there are a lot of redundant arcs.

Proposition 4.2.6. *Let $F = REF$ be a feedback arc set for the coordination graph, then for all $e \in F$, e is redundant.*

Proof. From proposition 3.3.1, we know that any cycle in the coordination graph contains at least two arcs in REF. Therefore, *any single* $e \in F$ may be removed from F, such that $F - e$ is still a feedback arc set: Every cycle in the coordination graph that involves e is covered by at least one more refinement arc in F. □

Unfortunately, removing a redundant arc from the feedback arc set does not always result in more planning freedom for an agent. That is, the removal of a constraint $e^{-1} \in F^{-1}$ does not necessarily result in e being in REF afterwards, in case

$$e^{-1} \in [E \cup (F - e)^{-1}]^+ \tag{4.3}$$

We have not been able to find satisfyingly good lowerbounds on the performance of Algorithm 6, but we can prove that at least one agent will regain a refinement arc.

Proposition 4.2.7. *Let Δ be a coordination set returned by Algorithm 6, $\mathcal{T} = (T, E)$ and $\mathcal{T}' = (T, E \cup \Delta)$. $REF_{\mathcal{T}'}$ is non-empty if $REF_{\mathcal{T}}$ is non-empty.*

Proof. In case $G_{\mathcal{T}}$ contains no inter-agent cycles, then $\Delta = \emptyset$ and $REF_{\mathcal{T}'} = REF_{\mathcal{T}}$. In case $G_{\mathcal{T}}$ does contain inter-agent cycles: suppose on the contrary that $REF_{\mathcal{T}'} = \emptyset$. Let $\delta \in \Delta$; since all arcs in Δ are not redundant, there must be at least one inter-agent cycle C in $G_{\mathcal{T}}$ that is only covered (in the feedback arc set $F = \Delta^{-1}$) by δ^{-1}. A cycle in the coordination graph contains at least two refinement arcs (Proposition 3.3.1), so all other arcs in $REF_{\mathcal{T}} \cap C$ must be in $INTRA_{\mathcal{T}'}$: they are not in $REF_{\mathcal{T}'}$, because $REF_{\mathcal{T}'}$ is assumed empty, and they are not in $INTRA^{-1}_{\mathcal{T}'}$ because their inverses do not cover C in the feedback arc set.

If an arc e is in $REF_{\mathcal{T}}$ but also in $INTRA_{\mathcal{T}'}$, then we know from Lemma 4.2.8 (below) that there exists an e' (possibly e itself), such that $e' \in \Delta$ and $(\text{dom}(e), \text{ran}(e)) \in [E \cup \{e'\}]^+$.

Again, since e' is not redundant, there must be a cycle C' (in $G_{\mathcal{T}}$) such that only e'^{-1} is the only arc in F covering C'. Thus, the other arcs in $REF_{\mathcal{T}} \cap C'$ must be in $INTRA_{\mathcal{T}'}$.

For any of these arcs in $INTRA_{\mathcal{T}'}$, we know there must be an arc $e'' \in \Delta$, such that $(\text{dom}(e'), \text{ran}(e')) \in [E \cup \{e''\}]^+$.

We can continue this line of reasoning forever if either there is an unlimited number of vertices in the coordination graph (which obviously there is not), or, in order to 'construct' the next cycle C''', we may reuse vertices. It is easily verified that the for the latter option, a reused vertex can be reached from itself in $E \cup \Delta$ (illustrated in Figure 4.7). This means that there is a cycle in $E \cup \Delta$, contradicting the fact that Δ is a coordination set. □

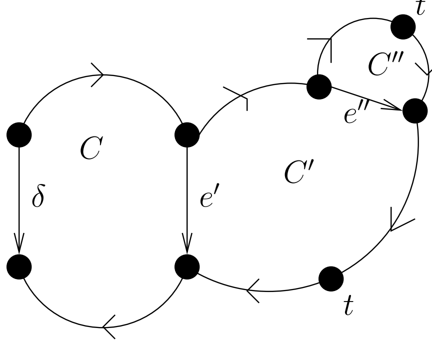


Figure 4.7: *If task t is used to form to distinct cycles, then t can be reached from itself.*

Lemma 4.2.8. *Let Δ be a coordination set returned by Algorithm 6, $\mathcal{T} = (T, E)$ and $\mathcal{T}' = (T, E \cup \Delta)$ and assume that $REF_{\mathcal{T}'} = \emptyset$. If there is an arc $e \in INTRA_{\mathcal{T}'}$ but also $e \in REF_{\mathcal{T}}$, then there is an arc $e' \in \Delta$, such that $(dom(e), ran(e)) \in [E \cup \{e'\}]^+$.*

Proof. We will show that there exists an e' with the required properties. Since we have only added arcs in Δ to E, there must be a minimum set $\Delta' \subseteq \Delta$ such that $e \in [E \cup \Delta']^+$, i.e., there is a path in $E \cup \delta$ from $\text{dom}(e)$ to $\text{ran}(e)$ that uses all arcs from Δ'.

Let $\Delta' = \{\delta_1, \ldots, \delta_n\}$. For every δ_i, we know that there must be at least one cycle C_i in $G_{\mathcal{T}}$ that is only covered, in the feedback arc set $F = \Delta^{-1}$, by δ_i^{-1}, since all δ_i are non-redundant. Also, since $REF_{\mathcal{T}'} = \emptyset$, we know that in $G_{\mathcal{T}'}$, C_i is a path in $E \cup \Delta$ from $\text{dom}(\delta_i)$ to $\text{ran}(\delta_i)$.

It is easily verified that there exists a path in $E \cup \Delta$ from $\text{dom}(\delta_1)$ to $\text{ran}(\delta_n)$ using the 'cycle-paths' C_i'. In $G_{\mathcal{T}}$, this path forms an inter-agent cycle that is broken, in $G_{\mathcal{T}'}$, *only* by $e' = (\text{dom}(\delta_1), \text{ran}(\delta_n))$. Hence, e' must be in Δ. Also, e' forms a path from $\text{dom}(e)$ to $\text{ran}(e)$ in $E \cup \{e'\}$. Hence, e' is the arc we seek. □

4.3 Partitioning Local Goals

In the previous sections, we approximated the coordination problem by directly constructing a coordination set Δ. In Algorithm 1, we added constraints to Δ one by one; in Section 4.2, we found a coordination set by inverting an acyclic feedback arc set for the coordination graph. In this section, we do not concentrate on constructing a coordination set —

although after coordination, a coordination set can be distinguished — but instead we coordinate by partitioning agent task sets ($T_i, i = 1, \dots, n$) into *segments*, and by constraining the agents to execute their segments in a linear order.

4.3.1 Agent dependencies

Suppose agent A_i has a task t that is preceded by a task t', $t' \prec t$, and $t' \in T_j, j \neq i$. In other words, the task t from agent A_i is preceded by task t' from agent A_j. Agent A_i is dependent on agent A_j, as A_i cannot execute t until agent A_j has finished t'. We can visualize these dependencies in the agent dependency graph D_A, which was first defined in [28].

Definition 4.3.1 (Agent Dependency Graph). *Let* $(\mathbf{T}, G)$ *be a coordination instance. The agent dependency graph* D_A *is the graph* $D_A = (A, \prec_A)$*, where* $(A_i \prec_A A_j)$ *iff there exist tasks* $t \in T_i$ *and* $t' \in T_j$ *such that* $t \prec t'$*.*

To avoid confusion, note that the direction of the arcs in D_A is in the direction of the precedence relations that exist between tasks; so $A_i \prec_A A_j$ means that A_j is dependent on A_i because A_i has a task that precedes one of A_j's tasks.

If we restrict the agent dependency graph to being constructed from only T_{inter} tasks (which we can do according to Proposition 3.1.3), then then the agent dependency graph is a graph contraction of the coordination graph: all tasks belonging to one agent are grouped into a single vertex, with the same inter-agent adjacencies as the tasks that make up the agent vertex. This means that all intra-agent precedences are ignored, leaving only inter-agent precedences.

Proposition 4.3.2. *If* D_A *is acyclic, then* $G_{\mathcal{T}}$ *contains no refinement cycles.*

Proof. Trivial; any two consecutive agents in an inter-agent cycle in $G_{\mathcal{T}}$ are connected in D_A. □

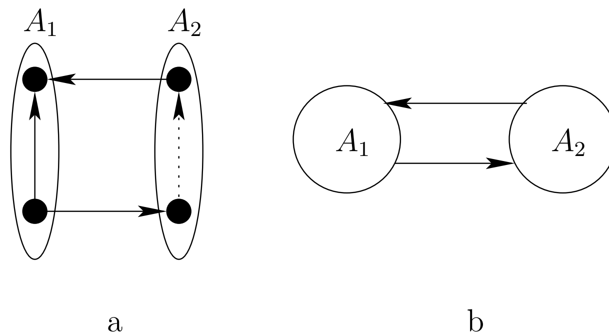


Figure 4.8: *(a):* $G_{\mathcal{T}}$ *and (b):* D_A *for a the same (coordinated) instance.*

The requirement that the agent dependency graph may not contain a cycle is stronger than the requirement that the coordination graph may not contain refinement cycles. That is, if the agent dependency graph is acyclic, then the coordination graph contains no inter-agent cycles, but the implication in the other direction does not hold. The coordination

graph in figure 4.8 contains no refinement cycles and the instance is, by definition, coordinated. The agent dependency graph is still cyclic, however.

To coordinate an instance, it would suffice to make the agent dependency graph acyclic. However, it is not obvious how this should be accomplished. The set $\prec_A$ is the result of both the original composite task (T, E), which may not be altered, and of task allocation, which, in the logistics domain for instance, is also fixed. We will not change the actual task allocation, but we will artificially refine it. Consider the situation where each task is allocated to a different agent. Then $\prec_A = \prec$, so D_A is acyclic because $\prec$ is. D_A becomes cyclic when tasks are grouped into agents. We will partially undo this by dividing agents into sub-agents. More specifically, we will partition agent A_i's set of tasks T_i into $\{T_{i_1}, \ldots, T_{i_k}\}$ and associate a sub-agent A_{i_j} with each *segment* T_{i_j}. In D_A, agent A_i is replaced by a chain of sub-agents $i_1, \ldots, i_k$. Figure 4.9 shows a composite task and an agent dependency graph where agent A_1's task set has been partitioned. Note that the segments of agent A_1 are ordered in such a way that D_A is made acyclic. In Figure4.9(c), the corresponding coordination graph is shown.

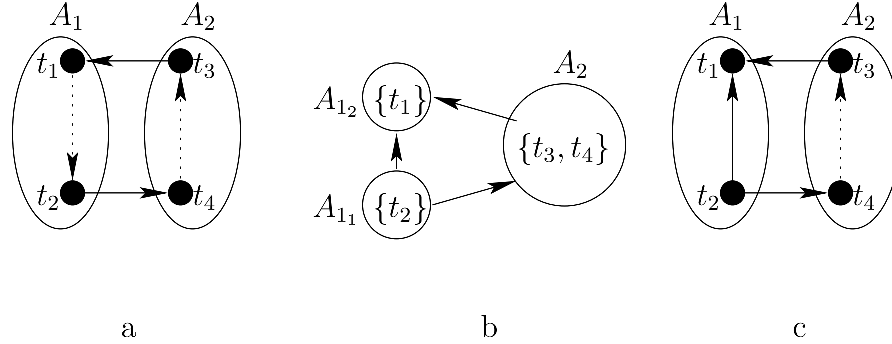


Figure 4.9: *(a): $G_{\mathcal{T}}$, (b): coordinated D_A and (c): corresponding $G_{\mathcal{T}}$.*

4.3.2 A coordination algorithm

We will now present an algorithm to partition the set of tasks T_i for agent A_i, which is also described in [27]. The intuitive idea behind the algorithm is that each agent tries to schedule a task t as soon as all tasks preceding it have already been scheduled by other agents. With scheduling a task t we mean that an agent commits to execute t in a certain order with regard to his other tasks: namely, before tasks that have not yet been scheduled (i.e., tasks that will be scheduled in a subsequent segment) and after tasks that have been scheduled earlier (i.e., in earlier segments). The agent creates a segment of all tasks that can be scheduled at that moment, separating it from the remaining, unscheduled tasks. Tasks in the scheduled segment are added to a global store **done**, that is distributed among all agents.

The result of Algorithm 8 is that task sets are partitioned, or equivalently, that agents

Algorithm 8 distributed partitioning

1: **Input:** set of local tasks T_i, and for each $t \in T_i$ the set $T_t = \{t' \in T | t' \notin T_i \wedge t' \prec t\}$ of prerequisites assigned to other agents.
2: **Output:** a positive integer k and a partition $\{T_{i_1}, \ldots, T_{i_k}\}$
3: $k := 1$
4: **while** $T_i \neq \emptyset$ **do**
5: **repeat**
6: $T_{i_k} := \{t \in T_i | T_t \subseteq \text{done}\}$
7: **until** $T_{i_k} \neq \emptyset$
8: done := done $\cup T_{i_k}$
9: $T_i := T_i \setminus T_{i_k}$
10: $k := k + 1$
11: **end while**

are divided into subagents. The partitioning results in a set of precedence constraints Δ:

$$\Delta = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} T_{i_j} \times T_{i_{j+1}} \tag{4.4}$$

In words, all tasks in segment j must precede all tasks in segment $j + 1$. This implies the following lemma, which states that there can be no precedence constraint $t \prec t'$ if t is in a 'higher' segment that t'.

Lemma 4.3.3. *Let $\mathcal{T} = (T, E)$ be a composite task and let $\bigcup_i \bigcup_j T_{i_j}$ be the segments resulting from Algorithm 8. Then,*

$$\forall i, t_1, t_2 \in T_i [t_1 \prec t_2 \rightarrow (t_1 \in T_{i_j} \wedge t_2 \in T_{i_k} \wedge j \leq k)]$$

Proof. Suppose on the contrary that there is a pair $\{t_1, t_2\} \subseteq T_i$ for some agent A_i such that $t_1 \prec t_2$ and t_1 is in a higher segment, i.e.,

$$t_1 \in T_{i_j} \wedge t_2 \in T_{i_k} \wedge j > k$$

From Line 6 of Algorithm 8, we deduce that if t_2 is in a lower segment, then at some point all of t_2's inter-agent prerequisites are in **done**, whereas for t_1 there is at least one task $t_3 \in T_m, m \neq i$ such that $t_3 \prec t_1$. However,

$$t_3 \prec t_1 \quad \wedge \quad t_1 \prec t_2$$

implies

$$t_3 \quad \prec \quad t_2$$

Hence, t_2 also has an inter-agent prerequisite that has not been scheduled yet, which is a contradiction. □

The precedence constraints that result when an agent schedules a segment affect only the scheduling agent: no other agents receive any additional constraints.

Proposition 4.3.4. *If an agent A_i schedules a segment in Algorithm 8, then no agent other than A_i receives any additional constraints.*

Proof. Consider the kth segment scheduled by agent A_i, and let

$$\Delta_k = T_{i_k} \times (T_i \setminus [T_{i_1} \cup \cdots \cup T_{i_{k-1}}])$$

be the set of corresponding local precedences.

Suppose on the contrary that another agent A_j receives an additional constraint δ because of Δ_k. In that case, there is a path in $E \cup \Delta_k$ from $\text{dom}(\delta)$ to $\text{ran}(\delta)$, such that $\text{dom}(\delta) \not\prec \text{ran}(\delta)$ (with $\prec$ the precedence relation prior to adding Δ_k).

However, if $\text{dom}(\delta) \prec t$ for some task $t \in T_{i_k}$, then $\text{dom}(\delta)$ must already have been scheduled by agent A_j. Also, if $t' \prec \text{ran}(\delta)$, then $\text{ran}(\delta)$ cannot have been scheduled yet by agent A_j. Thus, $\text{ran}(\delta)$ will be scheduled in a later segment than $\text{dom}(\delta)$. Hence, $\text{dom}(\delta) \prec \text{ran}(\delta)$, contrary to the assumption that δ is a result of Δ_k. □

We can interpret the partitioning as the creation of sub-agents. We denote the sub-agent of agent A_i, segment j by A_{i_j}. Now we can define the sub-agent dependency graph $D^*_A = (\bigcup_i \bigcup_j A_{i_j}, \prec^*_A)$, where

$$\prec^*_A = \bigcup \prec_{seg} \cup \{(A_{i_j}, A_{k_m}) | i \neq k, \exists t \in T_{i_j}, \exists t' \in T_{k_m} : t \prec t'\} \tag{4.5}$$

and

$$\prec_{seg} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{k_i} A_{i_j} \times A_{i_{j+1}} \tag{4.6}$$

The following proposition states that the set of constraints defined in Equation 4.4 is a coordination set.

Proposition 4.3.5. *If D^*_A is an acyclic graph, then $G_\mathcal{T}$ contains no refinement cycles.*

Proof. Because D^*_A is also a graph contraction of the coordination graph, it is easy to understand that if there is a refinement cycle in the coordination graph, then there must be a cycle in $\mathcal{D}^*_A$.

The difference between $\mathcal{D}_A$ and $\mathcal{D}^*_A$ is that in the sub-agent dependency graph, tasks belonging to one agent may be stored in a different segment.

Let C be a refinement cycle in $G_\mathcal{T}$ and let $p = (t_1, \ldots, t_m)$ be a pass of C through an agent. If $t_1 \not\prec t_m$, then t_1 and t_m are tasks in the same segment, so, in D^*_A, the pass is replaced by the single vertex corresponding to that segment.

If $t_1 \prec t_m$, then t_1 and t_m may either be in the same segment, or t_m may be in a higher segment. In the latter case, the pass p in $G_\mathcal{T}$ is replaced in D^*_A by the chain of sub-agents $\text{subagent}(t_1) - \cdots - \text{subagent}(t_m)$.

Because C is a refinement cycle, we do not have to consider the case $t_m \prec t_1$; the precedence $t_m \prec t_1$ would make a local cycle with the pass p. □

4.3.3 Partitioning strategies

In Chapter 5, where we examine the cost of precedence constraints, we will discuss how coordinating using partitioning affects cost. For now, we only point out that (*i*) additional precedence constraints are in general undesirable and that (*ii*) the composition of the coordination set depends on the way task sets are partitioned. From Equation 4.4, we can see that partitioning into a higher number of segments likely increases the number of additional precedence constraints.

To understand how a set of tasks is partitioned and into how many segments, we will take a closer look at Algorithm 8. In Line 6, the kth segment is constructed; all tasks that have not been scheduled yet, but which can be scheduled now because other-agent prerequisites are in **done**, will be in segment k.

Algorithm 8 is a distributed algorithm which is executed simultaneously by all agents. Therefore, we do not know exactly when the "now" is in the phrase "all tasks that can be scheduled now", and, consequently, we cannot (in general) predict which partitions will result.

To be able to analyze partitioning strategies deterministically, we present a centralized variation on Algorithm 8. Two parameters that can vary are:

1. When to schedule. In Algorithm 8, the segment is scheduled as soon as there is a non-empty set of tasks to schedule.
2. Which tasks to schedule. In Algorithm 8, the maximum segment is always scheduled: all tasks that can be scheduled, are scheduled now.

A heuristic for parameter 1 might take into account that waiting a while before scheduling might result in a larger segment, if more agents have scheduled their tasks first. Of course, if every agent tries to wait, nothing will happen. A heuristic for parameter 2 might be to schedule less than the maximum segment. A certain task may be postponed if the agent prefers the task to be in the same segment as some as yet unschedulable task.

In Algorithm 9, we will vary only the first parameter, for the second parameter we will only choose the maximum-segment strategy.

In Algorithm 9 the 'real time' of Algorithm 8 is replaced by steps. In one step all agents construct their maximum segment (Algorithm 10) given the tasks that are in **done** after the previous step. Then, the centralized component uses some heuristic function SCHEDULEPOLICY to determine which agent must schedule his segment this step. All other agents do not schedule. We have experimented (Chapter 6) with three different scheduling policies.

Policy 1: at the start of the algorithm, a random ordering of the set of agents is made. In each step, the next agent in the order will be instructed to schedule its segment.

Policy 2: after all agents have built their segment T_{i_k}, all agents evaluate the following term:

$$\frac{|T_{i_k}|}{|T_i|}$$

In words, the size in tasks of the current segment relative to the size of the entire local set of tasks. The agent with the highest value is instructed to schedule his segment.

Algorithm 9 Centralized Partitioning

1: **Input:** composite task $\mathcal{T} = (T, E)$
2: **Output:** $P = \{T_{1_1}, \dots, T_{1_{k_1}}, \dots, T_{n_1}, \dots, T_{n_{k_n}}\}$: For all agents, the segments corresponding to the partitioning of their set of tasks.
3: $P = \emptyset$
4: **done** $= \emptyset$
5: **while done** $\neq T$ **do**
6: {Build the set of schedulable segments}
7: $S = \emptyset$
8: **for all** $A_i \in A$ **do**
9: $s_i =$ buildSegment(A_i, **done**)
10: $S = S \cup s_i$
11: **end for**
12: {Pick a segment to schedule}
13: $s =$ schedulePolicy(S)
14: **done** = **done** $\cup s$
15: $P = P \cup s$
16: **end while**
17: **Return** P

Algorithm 10 buildSegment

1: **Input:** set **done** of scheduled tasks
2: **Output:** segment $s_i \subseteq T_i$
3: **for all** $t \in T_i$ **do**
4: $T_t = \{t' \in T | t' \prec t \wedge t' \notin T_i\}$
5: **end for**
6: $s_i = \{t \in T_i | T_t \subseteq$ **done**$\}$
7: **Return** s_i

Policy 3: similar to 2, but now all agents must evaluate the term:

$$\frac{|(\bigcup_{j \neq i}\{T_j \times T_{i_k}\}) \cap \prec|}{|(\bigcup_{j \neq i}\{T_j \times T_i\}) \cap \prec|}$$

in words, the number of inter-agent precedences satisfied by segment k divided by the total number of inter-agent precedences.

Heuristic 1 is similar to Algorithm 8, since agents schedule in a random order. Heuristic 2 is based on the idea that if large segments, relative to the size of T_i, are scheduled, then a small number of segments will be needed. Heuristic 3 is based on the idea that if a large number of precedences are satisfied for this segment, then there will be few precedences left to cause partitioning of the remaining set of tasks.

4.3.4 Negotiation in partitioning coordination

Adapting Algorithm 9 to include negotiation, there are two obvious changes to make. First, agents can negotiate over who must schedule the next segment. Second, agents can offer

to schedule a smaller segment than the maximum segment. In Algorithm 11, we have assumed that agent autonomy only pertains to the freedom to negotiate *when* to schedule. With regard to the composition of the segment to schedule, it is convenient to assume that agents still schedule the maximum segment.

Algorithm 11 PARTITIONING NEGOTIATION

1: **Input:** composite task $\mathcal{T} = (T, E)$
2: **Output:** $P = \{T_{1_1}, \ldots, T_{1_{k_1}}, \ldots, T_{n_1}, \ldots, T_{n_{k_n}}\}$: For all agents, the segments corresponding to the partitioning of their set of tasks.
3: $P = \emptyset$
4: **done** $= \emptyset$
5: **while done** $\neq T$ **do**
6: {collect bids}
7: $B = \emptyset$
8: **for all** $A_i \in A$ **do**
9: s_i = invoke BUILDSEGMENT(A_i,**done**)
10: b_i = invoke MAKEBIDSEGMENT(A_i, s_i)
11: $B = B \cup b_i$
12: **end for**
13: {Lowest bid wins}
14: $B_{min} = \{b \in B | \forall 1 \leq i \leq n : \text{price}(b) \leq \text{price}(b_i)\}$
15: Let $b_{min} \in B_{min}$
16: $s = \text{segment}(b_{min})$
17: **done** = **done** $\cup\, s$
18: pay(agent(b_{min}), price(b_{min}))
19: **end while**

Algorithm 12 MAKEBIDSEGMENT

1: **Input:** agent A_i, segment $s \subseteq T_i$
2: **Output:** bid b for scheduling s
3: $p_{s_1} = \text{plan}(s)$
4: $p_{s_2} = \text{plan}(T_i \setminus s)$
5: $\text{cost}_1 = c(p_{s_1}) + c(p_{s_2})$
6: $p = \text{plan}(T_i)$
7: $\text{cost}_2 = c(p)$
8: $\text{price} = \text{cost}_1 - \text{cost}_2$
9: $b = (s, \text{price})$
10: **Return** b

In Algorithm 11, a round of negotiation is started to determine who schedules the next segment. All agents make a bid to schedule a certain segment for a certain price. The lowest bid wins the round[5]; the winning agent is paid the amount of his bid, and he schedules his segment.

[5]Here it is convenient to assume that agents schedule their maximum segment, in order to avoid agents submitting singleton segments to try to win the auction.

We have given an example of an agent strategy in Algorithm 12, in which the agent determines the height of his bid by calculating the cost of scheduling the current candidate segment. First, he makes a plan to execute the candidate segment and a plan to execute the remainder of the unscheduled tasks. He also makes a single plan for all the as yet unscheduled tasks (so including the tasks of the candidate segment). The difference in cost between the cost of the latter plan and the sum of the costs of the former two plans, is the cost of scheduling the candidate segment. This cost he assigns to the price of his bid.

Coordination autonomy in Algorithm 12 is ensured by the fact that an agent can specify the compensation it wants for every constraint it receives. By Proposition 4.3.4, we know that an agent only receives constraints when he schedules his segment. In Algorithm 12, an agent only schedules a segment if he wins the auction, thereby receiving compensation for the cost he incurs.

4.4 Concluding Remarks

In this chapter, we have addressed a number of issues that must be tackled if the approximation techniques presented in this chapter are to be extended to negotiation protocols. In particular, we have tried to adapt the algorithms in such a way that agents only receive the constraints they have agreed to receive. However, other issues remain.

Rosenschein and Zlotkin [17] have identified a number of criteria to judge negotiation mechanisms (mechanism = protocol + strategy), among others:

- Efficiency: a solution should contain few constraints; both globally, and for individual agents.
- Stability: agents should have no incentive to deviate from the presented strategies.
- Distribution: ideally, the protocols should not require a centralized component.

For the moment, the analysis of our mechanisms in terms of these criteria, and the improvement of our mechanisms based on the results of such an analysis, remain future work.

Even if our negotiation mechanisms may not be ready to be deployed in a commercial multi-agent system, they can still serve as cost-based heuristic approximation functions. In the standard coordination algorithms, we arbitrarily pick e.g. the next constraint to add or the next segment to schedule, but instead of making arbitrary selections, we can guide the selection process by making use of heuristic functions. The agent strategies we have defined in this chapter can be viewed as heuristics in which the next step taken either has the lowest immediate cost (incremental coordination and coordination by partitioning) or the highest immediate gain (FAS-based coordination).

Chapter 5

Applying the Framework to a Planning Problem

Do you remember chalk hearts melting on a playground wall
Do you remember dawn escapes from moon-washed college halls
Do you remember the cherry blossoms in the market square
Do you remember I thought it was confetti in our hair
By the way, didn't I break your heart?
— *Kayleigh* by Marillion

This is an oh-by-the-way kind of chapter. In the next chapter, we will present some empirical results, by solving multi-modal logistics problems using our pre-planning coordination approach. To judge the viability of this approach, we will compare the quality of our solutions to solutions produced by more traditional multi-agent planning systems. Traditionally, plans are compared on the basis of the cost of the (joint) plan, not on the basis of the degree of autonomy for the agents, on which we have concentrated so far.

In this chapter, we need to address three issues before we can define the cost of a joint plan:

1. We need to encode the logistics problem in terms of the task-framework[1] of Chapter 2. In particular, this means that, given a logistics instance, we need to (*i*) identify tasks, (*ii*) identify precedence constraints between tasks, and (*iii*) allocate tasks to agents (Section 5.2).

2. We need to define the cost of a plan. It turns out that representing a plan as a composite task (i.e., as an *abstract* plan), is insufficient for this purpose. Thus, we need to choose a suitable representation for *concrete* plans. If we have a concrete-plan representation, we can not only reason about the cost of a plan, but also about the cost of precedence constraints (Section 5.3).

3. For a concrete joint plan, timing issues become relevant, since the concrete plans of the individual agents must be synchronized. Fortunately, we can show that — under

[1] Other, more traditional techniques to encode (multi-agent) planning problems include STRIPS [10] and PDDL [11].

certain assumptions — the concrete plans of the agents can be *scheduled* without fundamentally changing these plans, that is, without affecting the cost of these plans (Section 5.4).

Oh, by the way, we must also define the multi-modal logistics problem.

5.1 The Logistics Problem

The logistics problem is about delivering packages. There are packages that need to be transported between locations, and there are vehicles (trucks and planes) that can transport the packages. We wish to deliver all packages at minimum cost. There are a number of locations spread over a number of cities. *Within* cities, trucks carry out deliveries; *between* cities, planes transport the packages. There are many variations of the logistics problem and the one we will treat is a fairly simple one. Different variations of the logistics problem vary in (among others) these parameters:

1. The set of available actions: For instance, is the loading and unloading of packages considered an action or are they assumed to happen automatically, e.g. always unload when a package has reached its destination?

2. The cost function: For instance, is the cost of the action to move from location A to location B dependent on the distance between A and B, or is the cost a constant?

3. The transportation infrastructure: Within a city, which locations are connected to which other locations and between which cities exists a flight corridor?

4. Capabilities of the agents (vehicles) / the assignment of regions to vehicles: What is the set of locations that each vehicle can reach? Can there be more than one agent performing deliveries in a single region (set of locations), or is there e.g. only one truck per city?

5. The capacity (for packages) of the vehicles: Unlimited capacity, capacity for k packages or unit capacity.

6. Order characteristics: For instance, are there deadlines associated with an order?

7. The objective function: Do we minimize workload (the cost of all actions), the total time taken to execute all plans, or a combination of both?

Although this list of parameters is not exhaustive, it is sufficient for our purposes, since we reduce the problem to the bare basics. The problem we will study, which we will call the *n-city* problem, has the following characteristics:

1. Agents can perform one type of action: *move(x,y)* to move from location x to location y.

2. Each move action costs 1, both for trucks and planes.

3. There are n cities, each with m locations. That is, the infrastructure is fully connected. Within a city, any location is reachable from any other location. In every city, there is one location that is also the airport. Every airport is directly reachable from every other airport.

4. In each city, there is one truck and there is one plane flying between the cities.

5. Every vehicle has unlimited capacity.

6. There are no deadlines or time windows associated with a package. Each order is fully specified by a pair of locations (l_1, l_2).

7. We minimize the total move cost, which equals the total number of moves using the above cost function.

Given the above characteristics, an n-city instance is completely specified by:

- the number of cities n,
- the number of locations per city m,
- the set of orders O, each $o \in O$ consisting of a pair of locations.

Hence, $I = (O, n, m)$.

In case n equals 1 (i.e., only one city) we will call the problem the *one-city* problem. We can represent a one-city instance by a pair (L, O) where L is a set of locations and $O \subseteq L \times L$ is a set of orders. Since every location can be reached from any other location in the one-city problem, locations that do not need to be visited for any order in O have no use. Thus, we assume that

$$L = \text{ran}(O) \cup \text{dom}(O)$$

Consequently, the set L becomes superfluous, i.e., a one-city instance is fully specified by the set O. For clarity, however, we will represent a one-city instance as (L, O).

5.2 Identifying Tasks for a Logistics Problem

In this section, we transform a logistics instance I into a composite task $\mathcal{T} = (T, E)$. We will call this process *task identification*. Due to the characteristics of the n-city problem, *task allocation* is a by-product of task identification. If we define the set of capabilities of an agent as the set of distinct move actions he can perform, then the intersection of the capabilities of any two agents is empty. Therefore, each task that is found during task identification can be performed by exactly one agent.

Tasks are identified on the basis of orders; each order results in a set of tasks. Which tasks are identified, depends on the nature of the order. For order $o = (l_1, l_2)$, if location l_1 and location l_2 are in the same city, then the local truck can perform the order without the help of other agents and we therefore associate a single task with o. On the other hand, if l_1 and l_2 are in different cities, then the plane will have to transport the package. Depending on the pickup and destination locations of the package, a truck might have to

bring the package from l_1 to the airport and the truck in the destination city might have to pick it up from the airport in order to drive the package to its final location l_2. We assign a task to each stage in the journey.

We can thus distinguish two types of orders:

1. intra-city orders
2. inter-city orders, for which we distinguish three cases depending on the pickup and destination locations:
   - non-airport to non-airport
   - non-airport to airport or vice versa
   - airport to airport

We create a task for each stage in the journey of a package. Intra-city orders have only one stage, so one task suffices. Inter-city orders can consist of one to three stages; three stages for non-airport to non-airport, one for airport to airport and two stages are required in case exactly one of the locations is an airport. Example 5.2.1 illustrates the identification of tasks.

Example 5.2.1. *Figure 5.1 shows the infrastructure of a simple logistics problem, with $n = 2$ and $m = 2$. In Table 5.1, the orders and the resulting tasks are shown.*

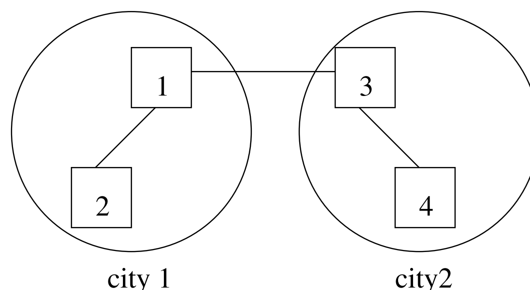


Figure 5.1: *Infrastructure of a two-city logistics instance.*

To conclude task identification, we must introduce precedence constraints between tasks where necessary. In a multiple-stage order, stages two and three cannot start until respectively stages one and two have been completed. Thus, we insert a precedence constraint between the task associated with stage one and the task associated with stage two and we insert a constraint between the task associated with stage two and the task associated with stage three. Table 5.1 gives the result of task identification for Example 5.2.1.

Constructing the coordination graph (Figure 5.2) for Example 5.2.1 illustrates the need for multi-agent coordination in the logistics domain. There are three agents, the truck in city one, the plane and the truck in city two. Note that task t_1 does not appear in the coordination graph, since t_1 is not connected to any other-agent task (i.e., $t_1 \notin T_{inter}$). Between the plane and Truck 2 a refinement cycle exists in the coordination graph. This means that the agents need to coordinate: When left to their own devices, it is possible that, if the agents refine their goals by ($t_6 \prec t_3$) (the plane) and ($t_4 \prec t_5$) (Truck 2), then the cycle $t_3 - t_4 - t_5 - t_6 - t_3$ is created in the joint plan.

Order	Tasks	Precedences
$(2,1)$	$t_1 = (2,1)$	
$(2,4)$	$t_2 = (2,1)$	
	$t_3 = (1,3)$	$t_2 \prec t_3$
	$t_4 = (3,4)$	$t_3 \prec t_4$
$(4,1)$	$t_5 = (4,3)$	
	$t_6 = (3,1)$	$t_5 \prec t_6$

Table 5.1: *Transformation of a logistics instance into a composite task.*

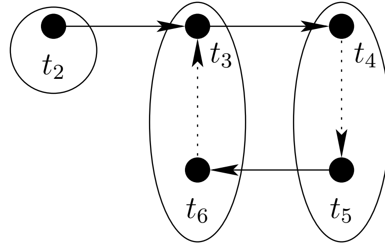


Figure 5.2: *There is refinement cycle in the coordination graph of Example 5.2.1.*

5.3 A Model for Plan Cost

In Chapter 2, we represent a plan as a composite task $(T_{p_i}, \prec_{p_i})$, that is, as a set of tasks T_{p_i} which will be executed in the (partial) order $\prec_{p_i}$. This representation was chosen because (i) it contains all the information needed to tackle the coordination problem and (ii) it leaves agents free to choose their internal plan representation.

Depending on the definition of plan cost, the above plan representation is not always sufficient. The representation $(T_{p_i}, \prec_{p_i})$ can be sufficient in case we define the cost of a plan as the total execution time. Then, we can associate a duration $d(t)$ with every $t \in T_{p_i}$ and determine the longest path in $\prec_{p_i}$ (measured as the sum of durations of tasks in the path), which gives the minimal execution time that can be achieved given $\prec_{p_i}$.

In our research, however, we want to determine the *workload* of a plan. For this definition of cost, the representation $p = (T_{p_i}, \prec_{p_i})$ contains too little information. If we would associate a cost $c(t)$ with every task $t \in T_{p_i}$, then the cost of a plan would be the same for all possible plans; whatever (correct) plan an agent comes up with, the set of tasks in the plan is always T_{p_i}, which is equal to T_i. The cost of all possible joint plans would also be the same, since the set of tasks in J always equals T.

The fact that a plan p for a goal $G_i = (T_i, \prec_i)$ always contains exactly the set of tasks T_i is a result of our idea of what a task is. We view a task as a goal, an assignment, a state that has to be reached. Given a goal G_i, an agent should make a plan to perform all assignments in T_i. This set T_i does not change; no other assignments are somehow invented or allocated. However, an agent is free to choose the set of *actions* that will be executed to accomplish all tasks in T_i. Thus, tasks specify *what* needs to be done, actions specify *how* it will be done.

A task is performed by executing actions. In STRIPS-like fashion, we think of a task as a goal that can be reached by executing a *sequence* of actions (A more general approach would have been to consider partially ordered sets of actions.). We will call the sequence of actions to perform a goal $G_i = (T_i, \prec_i)$ a *concrete plan.*

Definition 5.3.1 (Concrete Plan). *A concrete plan p is a linear order of actions:* $p = (o_1, \ldots, o_m)$.

We identify an action in a concrete plan by e.g. 'operator o_i' instead of 'action a_i' to avoid confusion with *agent* A_i. We associate a cost $c(o)$ with each action o. The cost of a concrete plan is the sum of the costs of all actions in the (concrete) plan.

Definition 5.3.2 (Plan Cost). *Given a concrete plan* $p = (o_1, \ldots, o_n)$ *, the cost of p is:*

$$c(p) = \sum_{i=1}^{n} c(o_i)$$

Recall from Section 5.1 that, in the logistics domain, we assign a cost of 1 to every move-action. Thus, the cost of a concrete plan is equal to the number of actions or moves in the concrete (logistics) plan.

5.3.1 Concrete plans in the logistics domain

For the logistic one-city problem, we will now introduce a slightly different representation of a concrete plan that is more convenient and more readable. Using this alternative representation, we can give a clear definition of the one-city problem. Instead of representing a concrete plan by the actions in the plan, we can also represent the plan by listing the sequence of locations an agent visits as a result of the (move-)actions he performs.

We divide time into steps: In each step, an agent can perform a single action (a move). If the agent performs the action move(x, y) at time t, then he must be at location x at time t, and he will be at location y at time $t+1$. Hence, an agent's plan is fully specified by the sequence of locations that indicate where the agent is at each time step. We will call this sequence of locations the *visiting sequence*. We can now formulate the one-city problem in terms of a visiting sequence. (A note on notation: In the following definitions, if we have a visiting sequence $vs = (s_1, \ldots, s_n)$, we will use the notation $i <_{vs} j$, to indicate that i and j are both indices in vs and that i is the smaller index, i.e., s_i occurs before s_j in vs.)

Definition 5.3.3 (One-City Problem). *Given a one-city instance* (L, O)*, find a minimum length visiting sequence* $vs = (s_1, \ldots, s_n)$*,* $\forall 1 \leq i \leq n : [s_i \in L]$*, such that*

$$\forall o = (l, l') \in O : \exists i \exists j [i <_{vs} j \wedge s_i = l \wedge s_j = l']$$

During coordination, we impose precedence constraints on the order of execution of an agent's tasks. Each task corresponds to one (local) order, so a precedence constraint between two orders means that an agent must first finish (=deliver) one order, before he can start (=pick up) the other order. Since Definition 5.3.3 does not support the notion of 'precedence constraint', we will give another definition of the one-city problem, especially for use in our task-framework.

Definition 5.3.4 (Constrained One-City Problem). *Let* $(L, O, \prec_o)$ *be a one-city instance, augmented with a partial order* $\prec_o \subseteq O \times O$. *Find a minimum-length visiting sequence* $vs = (s_1, \ldots, s_n)$, $\forall 1 \leq i \leq n : [s_i \in L]$, *such that:*

$$\forall o = (l, l') \in O \quad : \quad \exists i \exists j [i <_{vs} j \wedge s_i = l \wedge s_j = l'] \tag{5.1}$$

$$o_1 = (l_1, l_2) \prec_o o_2 = (l_3, l_4) \quad \rightarrow \quad \exists i \exists j \exists k \exists l [i <_{vs} j <_{vs} k <_{vs} l \wedge l_1 = s_i \wedge l_2 = s_j \wedge l_3 = s_k \wedge l_4 = s_l \tag{5.2}$$

Similar to Definition 5.3.4, we could give a definition for the n-city problem. Such a definition would specify that, for every agent, we need to find a visiting sequence satisfying such a set of conditions, that (*i*) agent behaviour is coordinated and that (*ii*) all packages can be delivered. This is not, however, the approach we take in our research. We do not present algorithms to solve the entire composite task, thereby specifying the behaviour of all agents. Instead, we first use a coordination algorithm and then we let agents find solutions for their local subproblems. Thus, our approach to approximating the n-city problem is as follows:

1. Transform orders into tasks (and precedences), allocating goals to agents.
2. Coordinate agents, restricting agent goals.
3. All agents make plans (visiting sequences) for their local goals, which are instances of the constrained one-city problem.

5.3.2 The Cost of Precedence Constraints

Suppose we have the local goal $G_i = (\{t_1, t_2\}, \emptyset)$, for some agent A_i, with t_1 the task to transport a package from location A to location B, and t_2 the task to transport another package from A to B. The visiting sequence A - B enables delivery of both packages. If we introduce precedence constraint $t_1 \prec t_2$, resulting in the goal $(\{t_1, t_2\}, \{t_1 \prec t_2\})$, then the minimum-length visiting sequence becomes A - B - A - B. Thus, because of the constraint $t_1 \prec t_2$, two extra moves are required: one from B back to A to pick up the second package, one from A to B to deliver the second package. We define the cost of constraint $t_2 \prec t_1$ as the difference in cost between the first and the second plan, so in this case the cost is 2 (as each additional move costs 1).

Definition 5.3.5. *Let* $G = (T, \prec)$ *be a goal,* $\Delta \subseteq T \times T$ *a set of precedences,* p_1^* *an optimal plan for* G *and* p_2^* *an optimal plan for* $G' = (T, \prec \cup \Delta)$. *Then the cost of* Δ *is:*

$$c(\Delta) = c(p_2^*) - c(p_1^*)$$

We define the cost of a set of precedences relative to a goal, because it is not a constant, but instead it depends on the set of constraints already present.

Example 5.3.6. *Consider the constrained one-city instance with three locations in Figure 5.3, where each arrow represents a task. Thus, the set of tasks is* $T = \{t_1, t_2, t_3\}$, *with*

$$\begin{aligned} t_1 &= (1, 2) \\ t_2 &= (2, 3) \\ t_3 &= (3, 1) \end{aligned}$$

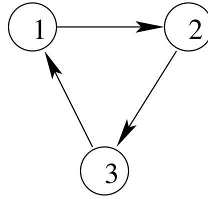


Figure 5.3: *An order-graph: three locations and the orders between them.*

Starting from goal $(T, \emptyset)$, we can calculate the cost of adding $\Delta = \{t_2 \prec t_1\}$.

$$\begin{aligned} p_1^* &= (1,2,3,1) \\ p_2^* &= (2,3,1,2) \end{aligned}$$

$c(p_2^*) = c(p_1^*)$, so the cost of $\{t_1 \prec t_2\}$ is 0.

Now consider the goal $(T, \{t_3 \prec t_2\})$ and again $\Delta = \{t_2 \prec t_1\}$.

$$\begin{aligned} p_1^* &= (3,1,2,3) \\ p_2^* &= (3,1,2,3,1,2) \end{aligned}$$

Two extra moves are required for p_2^*, so now the cost of Δ is 2. (Note that, due to transitive closure, the set $\{t_3 \prec t_2\} \cup \{t_2 \prec t_1\}$ also contains $t_3 \prec t_1$).

Example 5.3.6 shows that a single constraint $t_2 \prec t_1$ can separate more than just the execution of tasks t_2 and t_1. We have encountered this problem before when discussing negotiation algorithms. Adding a single constraint δ to $\prec$ results in the addition of the set ϕ_δ to the precedence relation:

$$\phi_\delta = [\prec \cup\, \delta]^+ - \prec$$

Clearly, ϕ_δ depends on the set of constraints $\prec$ that is already 'present' at the time δ is added.

During coordination, we usually do not add a single constraint, but a set of constraints Δ (although Δ can be constructed by adding one constraint at a time). Example 5.3.7 shows a constrained one-city instance in which the effect is listed of every possible set of constraints Δ on the optimal visiting sequence.

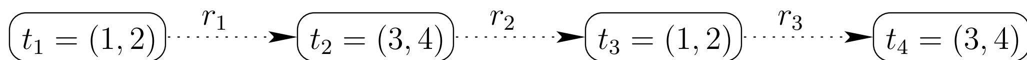


Figure 5.4: *Four tasks and the possible constraints between them.*

Example 5.3.7. *Figure 5.4 depicts a constrained one-city instance with four tasks. In case $\Delta = \emptyset$, an optimal visiting sequence can be found by executing t_1 and t_3 in parallel, resulting in subsequence 1,2, and by executing t_2 and t_4 in parallel, resulting in subsequence 3,4. Optimal visiting sequences for all tasks are e.g. 3,4,1,2; 1,2,3,4; 3,1,2,4; etc.*

Table 5.2 lists optimal visiting sequences for all possible constraint sets.

Δ	Visiting sequence	$c(\Delta)$
$\emptyset$	1,2,3,4	0
$\{r_1\}$	1,2,3,4	0
$\{r_1, r_2\}$	1,2,3,4,1,2	2
$\{r_1, r_2, r_3\}$	1,2,3,4,1,2,3,4	4
$\{r_2\}$	1,2,3,4	0
$\{r_2, r_3\}$	3,4,1,2,3,4	2
$\{r_3\}$	1,2,3,4	0

Table 5.2: *For every possible coordination set an optimal visiting sequence and the cost of the coordination set.*

In general, adding constraints to an agent's goal reduces the set of correct plans (i.e., the set of possible goal refinements) that are available to the agent. If we denote the set of correct plans for a goal $G = (T, E)$ by $\mathcal{P}(G)$, then for any $\Delta \subseteq T \times T$

$$\mathcal{P}(G') \subseteq \mathcal{P}(G)$$

where $G' = (T, E \cup \Delta)$. Furthermore, if all minimum cost plans for G are in $\mathcal{P}(G) - \mathcal{P}(G')$, then Δ has non-zero cost, according to definition 5.3.5.

In case of the logistics domain, we can see that the cost of precedences is due to the loss of positive relationships (cf. Von Martial's research [30]). For instance, in Example 5.3.7, if $\Delta = \{d_1, d_2\}$, then the equality relationship between t_1 and t_3 can no longer be utilized. Example 5.3.7 also shows that coordination cost can be zero even if the set of possible optimal plans reduces. For instance, in case $\Delta = \{d_2\}$, $c(\Delta) = 0$, but the visiting sequence 3,1,2,4 is no longer a correct plan, yet it is a correct plan in case $\Delta = \emptyset$.

5.3.3 Coordinating with minimal cost

Having defined the cost of precedence constraints in Definition 5.3.5, we can give an alternative definition of the coordination problem. In Definition 2.3.3, the objective is to coordinate using a minimal number of constraints. Fewer constraints equals more autonomy for the agents and *probably* results in lower plan cost.

Using Definition 5.3.5, we can now formulate the coordination problem in such a way that the objective is to minimize the cost of the coordination set Δ (equivalently, to minimize the cost of the joint plan). If we find a coordination set Δ such that costs are minimized, then $|\Delta|$ will *probably* be small, meaning agent autonomy is expected to be high. Even if $|\Delta|$ is not small for a minimum-cost coordination set, autonomous, rational agents will want to use the freedom they are given to be able to construct a plan of low cost. So from this point of view, autonomy is of secondary importance when compared to plan cost.

An intuitive way to minimize cost is to associate a weight $w(\delta)$ with each $\delta \in REF^{-1}$ and then find a coordination set $\Delta = \{\delta_1, \ldots, \delta_n\}$ such that $\sum_{i=1}^{n} \delta_i$ is minimal. However, as we have seen in Examples 5.3.6 and 5.3.7, the cost of a single constraint is dependent on which other constraints are in Δ (and $\prec$). Thus, we define a weight function that associates

a value (cost) with each subset of REF^{-1}:

$$w : 2^{REF} \rightarrow \mathbb{N}$$

The cost-based definition of the coordination problem is:

Definition 5.3.8 (Coordination Problem). *The coordination problem (CP) is: given a coordination instance* $(\mathbf{T}, G)$ *find a set of precedence constraints* $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$ *with* $\Delta_i \subseteq T_i \times T_i$ *such that:*

1. $E \cup \Delta$ *is acyclic,*
2. $(T, E \cup \Delta)$ *is a yes-instance of CVP, and*
3. $w(\Delta)$ *is minimal,*

Note that the original Definition 2.3.3 is a special case of Definition 5.3.8, where $w(\Delta) = |\Delta|$, for all $\Delta \subseteq REF^{-1}$.

5.4 The Joint Plan

Consider an n-city instance with two cities and two locations per city (Figure 5.1) and a single order from location 2 to location 4. Using the task identification method from Section 5.2, we identify three tasks, which are listed in Table 5.3 along with visiting sequences to perform the tasks.

Vehicle	Task	Visiting sequence
Truck 1	$t_1 = (2, 1)$	2,1
Plane	$t_2 = (1, 3)$	1,3
Truck 2	$t_3 = (3, 4)$	3,4

Table 5.3: *Tasks and visiting sequences for a one-order n-city instance.*

To know how to act, it is not sufficient for an agent to know which actions to perform and in which order; an agent also needs to know when to perform them. Thus, at some point, agents must determine a schedule for their concrete plan (i.e., start and finish times for all actions). Suppose the agents of the above example make the schedules listed in Table 5.4 for the visiting sequences listed in Table 5.3.

Vehicle	$time = 0$	$time = 1$
Truck 1	2	1
Plane	1	3
Truck 2	3	4

Table 5.4: *The location of the agents at times* $time = 0$ *and* $time = 1$.

This example shows that agents are not free to choose their schedules even if the joint plan J is feasible, because global (inter-agent) precedence constraints can be violated. For

instance, between tasks t_1 and t_2 of Table 5.3 exist the precedence constraint $t_1 \prec t_2$. This means that t_2 may not start until t_1 has been completed. Table 5.4 shows that t_1 has been completed at $time = 1$, but t_2 starts at $time = 0$.

A valid schedule for the tasks in Table 5.3 is given in Table 5.5. The '-' entries indicate that we do not care where the agents are at that time; since we minimize workload instead of total time taken, the fact that e.g. Truck 2 is idle for the first two time steps does not affect plan quality.

Vehicle	$time = 0$	$time = 1$	$time = 2$	$time = 3$
Truck 1	2	1	-	-
Plane	-	1	3	-
Truck 2	-	-	3	4

Table 5.5: *The location of the agents at times $time = 0$ up to $time = 3$.*

A scheduling algorithm must ensure that agents perform their tasks at such times, that global precedence constraints are adhered to. It is easy to construct such a schedule for a feasible joint plan J. If we find a topological sort s for the precedence relation E_J, we can schedule tasks in order s. This sequentializing of tasks is not a good scheduling algorithm, however, because (i) it takes away planning autonomy from the agents and (ii) by additionally constraining the order of task execution, it may increase plan cost.

5.4.1 Autonomy-preserving scheduling

After coordination, agents should be free to find plans independently of each other. The result will be a set of concrete plans. The scheduling activity that follows must not alter these concrete plans.[2] Otherwise, agents would not have been autonomous in their planning activity after all. We must show that if all agents make valid goal refinements, then there exists a schedule that does not change the concrete plans of the agents. Note that for such a schedule, the cost of the joint plan — if we define plan cost according to Definition 5.3.2 — is not affected by scheduling: the cost of the joint plan is determined by the set of actions in the joint plan. If that set of actions remains unchanged by scheduling, then so does plan cost.

To show that we can find an autonomy-preserving schedule, we must schedule the concrete plans themselves. A schedule is a function

$$s : O \rightarrow \mathbb{N}$$

that maps actions (from the set of 'operators' O) to points in time. Since we assume that each action requires one time unit, we can fully specify the timing of the joint plan by mapping all actions to their start times. Any *feasible schedule*, whether for tasks or for actions, must respect the precedence constraints of the (refined) goals. However, precedences are defined between tasks, not between actions. Thus, we must specify the relation between a task and the actions that perform it.

[2] Assuming we define a concrete plan according to Definition 5.3.1.

Assumption 5.4.1. *Given a local goal $G_i = (T_i, \prec_i)$ and a concrete plan $p_i = (o_1, \ldots, o_m)$ for G_i, we assume that if p_i performs all tasks in T_i, then, for every $t \in T_i$, we can identify in p_i the actions that start and finish t. Thus:*

$$\forall t \in T_i : \exists j, k : j \leq k \wedge \mathit{start}(t) = o_j \wedge \mathit{finish}(t) = o_k$$

Using Assumption 5.4.1, we can associate an abstract plan with a concrete plan.

Definition 5.4.2. *Let $G_i = (T_i, \prec_i)$ be a local goal and $p_i = (o_1, \ldots, o_m)$ be a concrete plan for G_i, then the abstract plan implied by p_i is given by $G'_i = (T_i, \prec'_i)$, where*

$$(t_1, t_2) \in \prec'_i \leftrightarrow \exists j, k : j < k \wedge o_j = \mathit{finish}(t_1) \wedge o_k = \mathit{start}(t_2)$$

The refinement relation is defined (Definition 2.2.3) between goals, not between a goal and a concrete plan. However, if p' is the abstract plan corresponding to concrete plan p, and $p' \vdash G$, then we say that the concrete plan p refines G, $p \vdash G$.

Using Assumption 5.4.1, we can also state the requirements that a feasible schedule must satisfy.

Definition 5.4.3 (Feasible Schedule). *Let P be a set of concrete plans for a set of local goals G. A schedule s is feasible iff:*

1. *if actions o_i and o_j are both in the same concrete plan p, and $i < j$, then $s(o_i) < s(o_j)$*
2. *if o_i finishes task t and o_j starts task t', and $t \prec t'$, then $s(o_i) < s(o_j)$*

A feasible schedule imposes a partial order on the set of all actions. To show that a feasible schedule exists for a given set of plans $P = \{p_1, \ldots, p_n\}$, we need to show that the set of actions from all concrete plans can be partially ordered, while respecting global precedences and the ordering of actions within each concrete plan. We now introduce a set of precedence constraints E_P between the actions of all concrete plans, and show that if the concrete plans refine a set of coordinated goals, then the set E_P is acyclic and thus a partial order.

Definition 5.4.4. *Let $\mathcal{T}$ be a composite task and let P be a set of concrete plans. We define the set E_P of precedence constraints between actions in P. We have $(o_i, o_j) \in E_P$ iff:*

$$\exists p \in P : (o_i \in p \wedge o_j \in p) \wedge (i < j) \vee$$
$$\exists t_1, t_2 : \mathit{finish}(t_1) = o_i \wedge \mathit{start}(t_2) = o_j \wedge (t_1 \prec t_2)$$

Proposition 5.4.5. *Let $\mathcal{T}$ be a composite task, $G = (G_1, \ldots, G_n)$ a set of goals such that (i) $J_G \vdash \mathcal{T}$ and (ii) J is coordinated, and let $P = (p_1, \ldots, p_n)$ be a set of concrete plans such that for all $i = 1, \ldots, n$ $p_i \vdash G_i$, then the set E_P is acyclic.*

Proof. Suppose on the contrary that there is a cycle $C = (o_1, o_2, \ldots, o_m, o_1)$ in E_P.

case 1: C contains only actions from one concrete plan (i.e., C is a local cycle). However, a concrete plan cannot contain a cycle, because by Definition 5.3.1, a concrete plan is a linear order.

case 2: C contains actions from more than one concrete plan. Similar to Definition 3.1.6, we can distinguish *passes* of C through the concrete plans. For every pass $p = (o_i, \ldots, o_k)$, we have: $\exists t_1, t_2 : \text{start}(t_1) = o_i \wedge \text{finish}(t_2) = o_k$.

Consider the precedence relation between t_1 and t_2 in J. It is possible that either $t_1 \prec t_2$, or $t_1 \not\prec t_2$, or $t_1 = t_2$, but not $t_2 \prec t_1$. In the last case, (o_k, o_i) would be in E_P, and the corresponding concrete plan would be cyclic on account of the pass p; this is a contradiction, since a concrete plan cannot be cyclic. This leaves two case to consider:

case a: For all passes: for the pair t_1, t_2 corresponding to this pass, we have: $t_1 \prec t_2$. But then, E_J would be cyclic, which contradicts the fact that $J_G \vdash \mathcal{T}$.

case b: There are passes for which $t_1 \not\prec t_2$. Since (t_1, t_2) is in $T_{in} \times T_{out}$ and not in $\mathit{INTRA} \cup \mathit{INTRA}^{-1}$, (t_1, t_2) must be in REF (by Proposition 3.3.1, there must be at least two such passes).

It is intuitively clear that the set of passes forms an inter-agent cycle in the coordination graph of J. Furthermore, it forms a locally acyclic inter-agent cycle: a local cycle would require a cyclic concrete plan. However, the existence of a locally acyclic inter-agent cycle in the coordination graph of J contradicts the fact that J is coordinated.

case c: The case that for all agents, $t_1 = t_2$, is similar to case a: it implies that E_J would be cyclic.

□

Thus, if agents make concrete plans that refine their coordinated goals, then the set E_P of precedences between actions is acyclic. If E_P is an acyclic relation, then the relation $\prec_P = E_P^+$ is a partial order.

Corollary 5.4.6. *Let $P = (p_1, \ldots, p_n)$ be a set of concrete plans that refine a set of coordinated goals $G = (G_1, \ldots, G_n)$, then there exists a feasible schedule for P.*

Proof. From Proposition 5.4.5, we know that the relation E_P is acyclic, so we have a partial order $\prec_P$ of precedences between actions of all concrete plans. By making a topological sort from $\prec_P$, we totally order the set of actions. We can form a feasible schedule by assigning successive times to successive actions in the total order. □

Chapter 6

Empirical Results

The tragedy of science is the heartless murder of beautiful theories by ugly facts.
— Gregory Benford[1]

To test the viability of our pre-planning coordination approach in general, and of our partitioning algorithms in particular, we have solved a set of multi-modal logistics instances. To judge the quality of our solutions, we have recorded the cost of the joint plan for each instance, and compared our plan cost to the cost of plans produced using state-of-the-art (multi-agent) planning systems.

Of course, the objective of the coordination problem (defined in Chapter 2) is to maximize individual-agent planning autonomy (apart from ensuring coordinated plans, of course), not to minimize plan cost. Our justification for maximizing autonomy — apart from the fact that agents might want to act independently of other agents — is that we expect that, if agents have a lot of planning autonomy, then they will be able to construct cheap plans; restrict the agents' planning autonomy, and these cheap plans may become infeasible, as we have argued in Chapter 5.

In this chapter we put the above hypothesis to the test. In particular, we investigate whether, when using the partitioning strategies of the previous chapter, more plan splits (resulting in more precedence constraints) increases the cost of the resulting plan.

We also compare our pre-planning coordination approach with state-of-the-art centralized planning systems. We will show (*i*) that our coordination approach is competitive with the best centralized planners, (*ii*) that the performance of less efficient planning systems can be improved by making use of pre-planning coordination, and (*iii*) that single-agent planning tools can be 'upgraded' to solve multi-agent planning problems.

The comparisons in this chapter are carried out on the basis of a benchmark set of logistics instances taken from the AIPS[2] 2000 competition (the AIPS competition is described in [1]).

6.1 Test Setup

We will judge the quality of an approximate coordination solution in terms of the quality of the joint plan. Figure 6.1 illustrates that, in order to find the joint plan, we need more

[1] The original quotation is from Thomas H. Huxley.
[2] Artificial Intelligence Planning and Scheduling

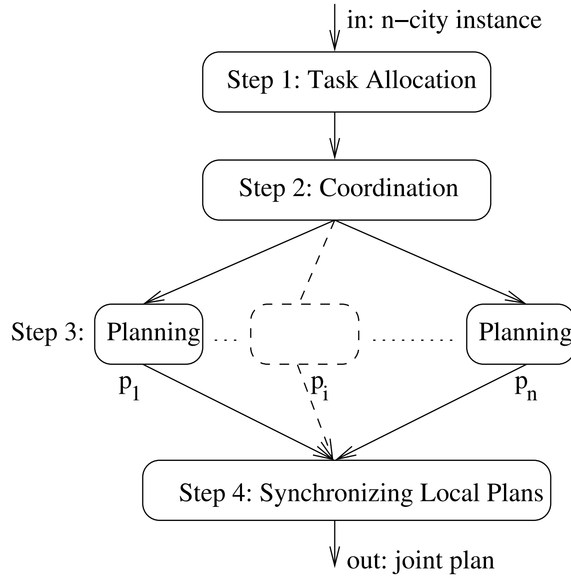


Figure 6.1: *The computational steps performed to solve a logistics instance.*

than a coordination algorithm: we need to allocate tasks, construct local plans and merge the local plans into a joint plan.

The input for Step 1, task allocation, is a logistic n-city instance from the AIPS dataset. Given a n-city instance, we can identify and allocate tasks as described in Chapter 5. Since there is only one allocation of tasks to agents possible for the n-city problem, this step can't influence the quality of the joint plan. This means that we do not need to find a 'best' task allocation.

The coordination step, Step 2, is accomplished by using Algorithm 9, instantiated with one of the three *policies* of Section 4.3.3. Recall that Algorithm 9 coordinates by partitioning agent task sets T_i into *segments* $\{T_{i_1}, \ldots, T_{i_m}\}$, such that all tasks in T_{i_j} are executed prior to tasks in $T_{i_{j+1}}$. In sections to follow, we will often refer to the policies of Section 4.3.3, so we list them again below for reference.

Recall that in Algorithm 9, one agent schedules a (maximal) segment of tasks in each iteration of the algorithm. To determine which agent will schedule the next segment, we can use one of these three policies:

Policy 1: before the start of the algorithm, a random ordering of agents is made. In each iteration of the algorithm, the next agent in the ordering is asked to schedule his next segment. After the last agent in the ordering has scheduled his segment, the next agent to schedule is again the first in the ordering.

Policy 2: after all agents have built their segment T_{i_k}, all agents evaluate the following term:

$$\frac{|T_{i_k}|}{|T_i|}$$

In words, the size in tasks of the current segment relative to the size of the entire local set of tasks. The agent with the highest value is instructed to schedule his segment.

Policy 3: similar to 2, but now all agents must evaluate the term:

$$\frac{|(\bigcup_{j\neq i}\{T_j \times T_{i_k}\})\cap \prec |}{|(\bigcup_{j\neq i}\{T_j \times T_i\})\cap \prec |}$$

in words, the number of inter-agent precedences satisfied by segment k divided by the total number of inter-agent precedences.

Once coordination is done, each agent can find a plan for his local goal, independently of the other agents, in Step 3. After coordination, an agent's goal is given by a sequence of segments. Each of the segments, containing a set of tasks, is an (unconstrained) one-city instance (as defined in Definition 5.3.3). An agent makes a concrete plan (a visiting sequence) for each segment. The entire local (concrete) plan for the agent is formed simply by concatenating the segment plans. For example, if an agent makes segment plans $(1,2)$, $(2,4)$ and $(3,1)$, then the agent's entire plan becomes $(1,2,4,3,1)$. Note that segment plans 1 and 2 'fit': the end location of the first segment, location 2, is the same as the start location of the second segment. Segments 2 and 3 do not fit; the agent needs to perform one extra action to move from location 4 to location 3. In Appendix A, we have described how we can find a visiting sequence for the one-city problem of Definition 5.3.3.

The local segment plans we calculate are optimal. In this way, we know that any extra cost in the joint plan can be attributed to the inefficiency of the coordination solution.[3]

In Step 4, we form the joint plan. The cost of the joint plan is determined by the cost of the local plans. Using Definition 5.3.2 of the cost of a plan, we say that the cost of a plan is the sum of the costs of the actions in that plan. The set of actions in the joint plan is simply the union of the actions of the local plans. Therefore, we do not need Step 4 to determine the cost of the joint plan, which is why we dashed Step 4 in Figure 6.1. Section 5.4 describes how we can find a feasible schedule for the set of concrete plans.

If we want to know how far the cost of an approximate joint plan is removed from the optimal joint plan, we need to know the cost of the optimal joint plan. Unfortunately, to know the cost of the optimal plan, we need to find the optimal plan. Finding optimal solutions for all AIPS instances proved too time-consuming. However, we have been able to find a lowerbound on the cost of an optimal solution: if we let all agents find optimal plans for their (unconstrained) initial goal, then the cost of the resulting (probably infeasible) joint plan is at least as low as the cost of the optimal solution.

For all problem instances, we now have both the cost of the actual solution and a lowerbound on the cost of the optimal solution, so we can define a ratio that indicates how far the actual solution is removed from the optimal solution. This *relative performance ratio* r, which we will call *overhead* or *inefficiency*, is given by:

$$r = \frac{c - c^*}{c^*}$$

with c the cost of the actual solution and c^* the cost of an optimal solution for the unconstrained goal.

[3]Calculating optimal segment plans is NP-hard, but it is just doable using the solvers specified in Appendix A.

6.1.1 Hypotheses

For the relative performance ratio r to be a good indicator of the distance between the optimal solution and the actual solution, it is required that the ratio between the optimal solution and the lowerbound is more or less the same for all instances. However, we suspect that for instances with a (relatively) high number of orders (packages to be delivered) the values of the optimal solution and the lowerbound diverge slightly, since for such instances, more coordination (i.e., more constraining) is required for the optimal solution, but not for the lowerbound on the optimal solution. Thus, we formulate the following hypothesis:

Hypothesis 1: If, for a given infrastructure, the number of inter-city orders increases, then the cost of the optimal solution diverges from the cost of the solution for the unconstrained instance.

We will not actively investigate Hypothesis 1. Rather, it is something to keep in mind when we discuss the results.

The focus of this chapter is the study of the relationship between plan cost and number of splits. As described in Chapter 5, partitioning a goal into segments can increase the cost of the optimal plan. If a set of tasks is partitioned into d segments, then the cost of the plan can be as much as d times the cost of the optimal plan [27]. This ratio can be reached if all work done for the unsplit plan is repeated for the plans for each of the segments.

We expect, however, that only a portion of the work must be redone for each additional segment. Also, note that if there are currently k segments, and we split off segment $k + 1$, then we can only redo work that was done for segment k. We thus expect that for each additional segment, the amount of work that must be redone becomes less. This leads us to the following hypothesis:

Hypothesis 1: The cost of the plan increases sublinearly with the number of segments.

6.2 Running the AIPS Dataset

The AIPS2000 data contains different sets of logistic instances. We have tested our coordination policies on the set of the largest instances, which consists of 170 instances. All instances are very similar and can be characterized by the following:

- the smallest instance has 6 cities, the largest 34;
- every city has two locations, one of which is an airport;
- in every city, there are three packages, all of which are initially at the non-airport location

6.2.1 Results

We have summarized the results of applying the setup of Figure 6.1 to the AIPS dataset in three types of graphs. In Figures 6.2 and 6.3, we show joint-plan cost; in Figures 6.4 and 6.5, we show how many times the agents' task sets have been split up; in Figures 6.6 and 6.7, we analyze the plane agent.

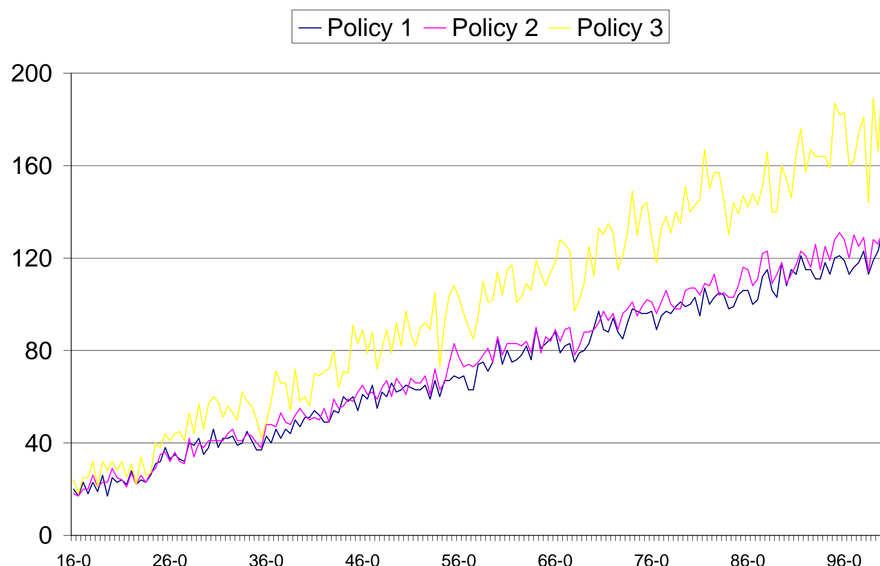


Figure 6.2: *The joint-plan cost, in terms of the number of moves, produced by each of the three policies.*

Figure 6.2 compares the joint-plan cost produced by each of the three policies (actually, produced by the setup of Figure 6.1 instantiated with one of the three policies, but we will sacrifice correctness in favor of clarity). For the majority of the 170 problem instances, Policy 1 just outperforms Policy 2, while both are significantly more efficient than Policy 3.

In Figure 6.3, we compare the joint-plan cost produced by Policy 1 to that of TALplanner, (generally) the most efficient of the planning systems competing in the AIPS 2000 competition. On all but one or two instances, Policy 1 outperforms TALplanner. Note that in Figure 6.3, we measure joint-plan cost in terms of the number of actions — consisting of move, load, and unload actions — instead of in the number of moves. Although we haven't actually generated plans containing load and unload actions, we have calculated the number of load/unload actions required by our plans: one load and one unload action for each task. This doesn't make our comparison with TALplanner unfair, because (*i*) loading and unloading does not make the problem harder given the fact that vehicles have unlimited capacity, and because (*ii*) the plans produced by TALplanner do not contain more load and unload actions than necessary.

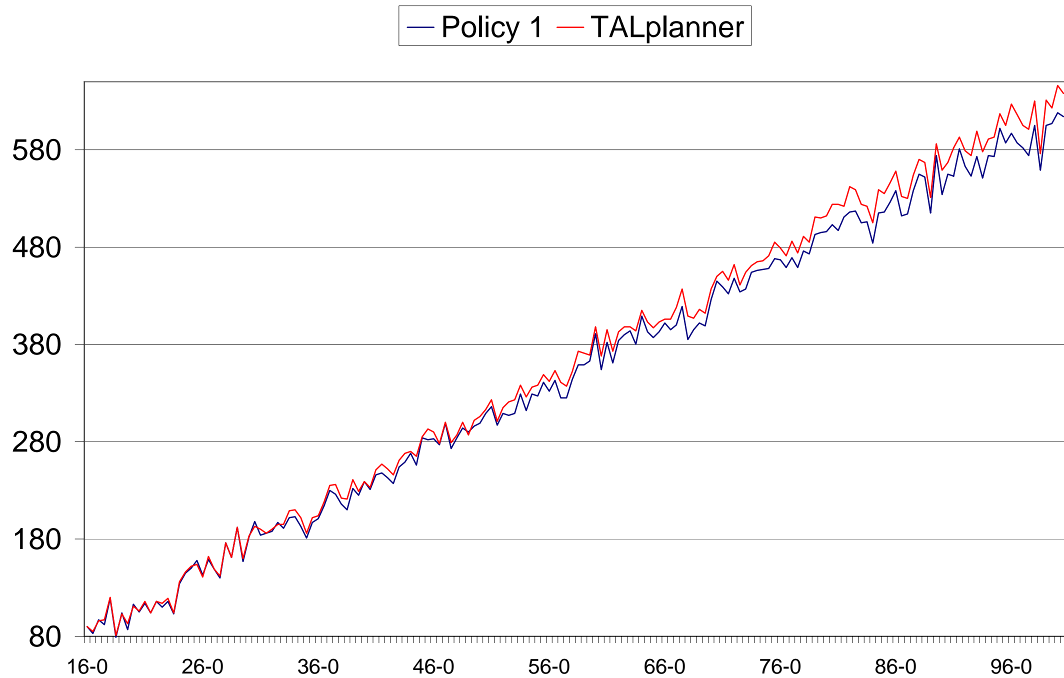


Figure 6.3: *The joint-plan cost in terms of the number of actions (move, load and unload actions), produced by Policy 1 and by TALplanner.*

In Figure 6.4, we compare the number of truck splits each policy generates per instance. For one agent, we define the number of truck splits as the number of segments minus 1, i.e., if $T_i = \{T_{i_1}, \ldots, T_{i_k}\}$, then agent A_i has made $k-1$ splits. It turns out that for most instances, Policy 1 requires the most truck splits, Policy 3 the least, with Policy 2 exactly between them. In Figure 6.5, it may seem as if only two policies are compared, but Policy 1 and Policy 2 overlap, each requiring exactly one plane split for every instance.

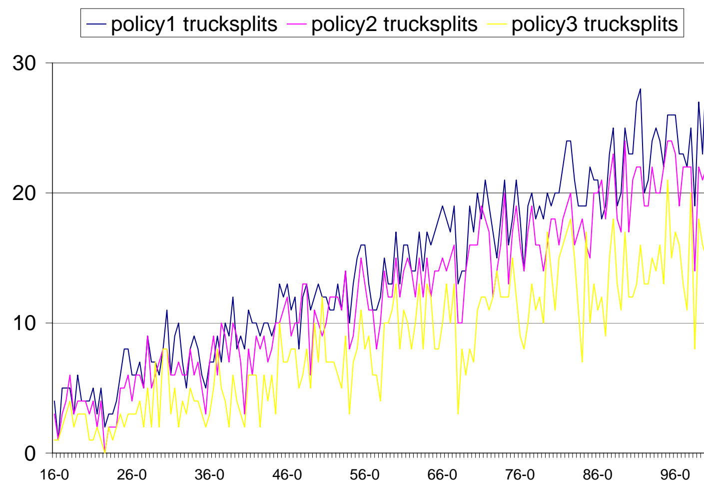


Figure 6.4: *The number of truck splits generated per instance.*

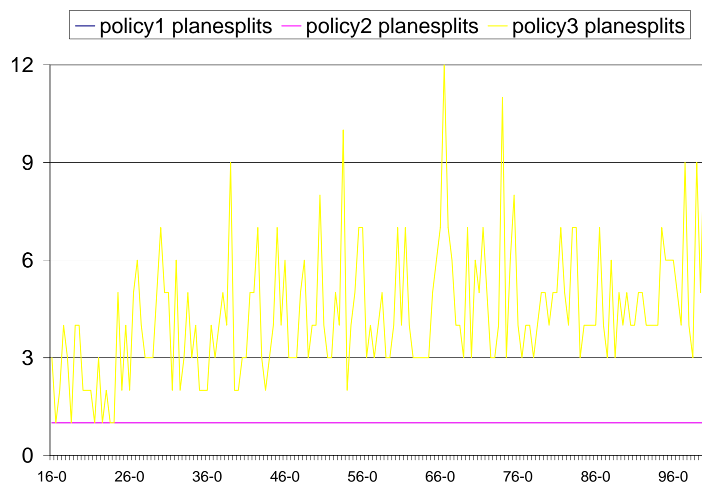


Figure 6.5: *The number of plane splits generated per instance.*

In Figures 6.6 and 6.7 we show the relation between the number of splits and the inefficiency of the policy. For Policies 1 and 2, no relation between overhead and number of splits could be discerned whatsoever; the results was only a cloud of datapoints. For Policy 3, however, a sublinear relation emerges. In Figure 6.6, where data from both the plane agent and from all truck agents are shown, there are a number of outliers with regard to this pattern. Separating the plane agent in Figure 6.7, the spreadsheet program used is able to fit a logarithmic function to the data.

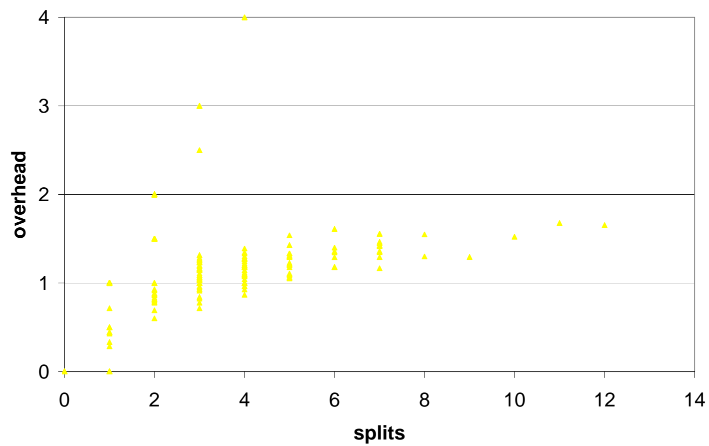


Figure 6.6: *The amount of overhead generated by Policy 3 per agent: data points are pairs < number splits, overhead > per agent per instance.*

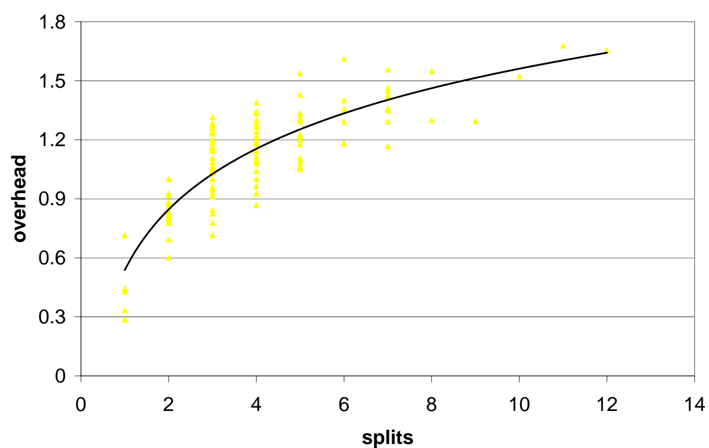


Figure 6.7: *The amount of overhead generated by Policy 3 for the* plane agent*: data points are pairs < number splits, overhead > for the plane agent per instance.*

6.2.2 Discussion of AIPS results

Figure 6.2 shows that Policy 1 performs marginally better than Policy 2, while Policy 3 performs significantly worse than either. In Figure 6.3 we can see that Policy 1 also just outperforms TALplanner. For us, however, the most interesting thing is to find out whether we can discern some relation between the number of segments needed and the overhead generated.

Number of segments

	Truck splits	Plane splits	Total splits
Policy 1	2413	170	2583
Policy 2	2096	170	2266
Policy 3	1447	742	2189

Table 6.1: *AIPS dataset: number of splits generated by each policy.*

The total number of splits required by policies 1, 2 and 3 for the entire dataset are respectively 2583, 2266 and 2189. Policies 2 and 3 thus achieve their intermediate objective of requiring less splits than Policy 1.

Not only do Policies 2 and 3 require less splits, the way in which goals are partitioned also differs. Policies 1 and 2 are similar: Both policies partition the goals of most agents into two segments. Policy 3 meanwhile splits up the plane agent's goal into many more segments, but needs less splits, on average, for the truck agents. The partitioning behaviour of Policy 3 can be explained as follows. For each agent the ratio r,

$$r = \text{global precs in candidate segment} \;/\; \text{total number of global precs}$$

is evaluated to determine which agent will schedule his segment. Suppose that at the start of coordination only one truck agent has scheduled a segment containing one pre-flight order. In the next round of coordination, the ratio r will be 0 for the truck agents (since they can schedule no task with a scheduled preceding task) and $\frac{1}{x}$ (for some x) for the plane agent. $\frac{1}{x} > 0$, so the plane agent will schedule a segment in which he performs one delivery. In general, when coordinating using Policy 3, the plane agent and the truck agents will schedule more or less alternately.

The number of splits required by the truck agents can also be explained from the alternating pattern of Policy 3. Because the plane agent starts delivering packages (actually, scheduling segments for the delivery of packages) when only a few trucks have scheduled a segment, some trucks with post-flight orders will find that all their packages are ready to be picked up from the airport when they first want to schedule a segment. This means that if a truck agent is lucky, he can schedule his entire goal in one segment.

Plan cost

Table 6.2 contains the total number of moves each policy needed for the entire dataset.

We expected that, by using less segments, Policies 2 and 3 would outperform Policy 1, i.e., we expected that Policies 2 and 3 would result in plans with lower cost. The reverse is

	Truck moves	Plane moves	Total moves
Policy 1	6759	5487	12246
Policy 2	6907	5905	12812
Policy 3	8181	8917	17098

Table 6.2: *AIPS dataset: number of moves generated by each policy.*

true: Policy 1 with the most splits has the lowest number of moves, whereas Policy 3 with the least splits has the highest number of moves.

The performance of policies 2 and 3 can be explained by looking at the optimal solution. The structure of the AIPS dataset is such that we can specify a simple coordination protocol (the arbiter0 protocol described in [29]) that always yields the optimal solution. The arbiter0 policy coordinates in three steps:

1. All trucks schedule their pre-flight and intra-city tasks.
2. The plane agent schedules his tasks.
3. All trucks schedule their post-flight tasks.

When coordinating general logistic instances with the arbiter0 protocol, inefficiency can only occur if (truck) agents could have combined intra-city or pre-flight orders with post-flight orders (Example 6.2.1). For the AIPS dataset, this situation never occurs, since all intra-city and pre-flight orders go from the non-airport location to the airport location and post-flight orders go in the opposite direction.

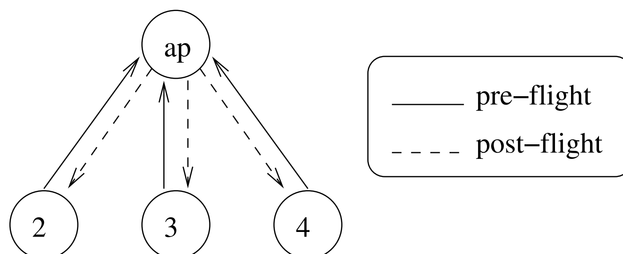


Figure 6.8: *A one-city truck goal for which Arbiter0 is inefficient.*

Example 6.2.1. *In Figure 6.8, one city is depicted with four locations: the airport and locations 2, 3 and 4. The truck agent has three pre-flight orders and three post-flight orders. An optimal local plan, using arbiter0 coordination, is the visiting sequence* $(2, 3, 4, 1, 2, 3, 4)$*. However, if all post-flight packages can be immediately picked up, then it is possible for the truck to be unconstrained, which results in an optimal visiting sequence* $(1, 2, 3, 4, 1)$*.*

The behaviour of Policy 1 closely resembles the optimal protocol. Recall that at the start of coordination, Policy 1 determines a random scheduling order; in our implementation we have simply used the order in which agents appear in the AIPS datafile. If the plane agent is either at the start or the end of the scheduling order, then Policy 1 implements

the arbiter0 protocol. For instance, if the plane has to schedule last, then all trucks will have scheduled their pre-flight orders when the plane has to schedule its segment. Thus all precedences for the plane have been scheduled, so he schedules his entire set of tasks at once. In the subsequent rounds of coordination, the trucks can schedule all their post-flight tasks in the same segment, since all flight tasks have been scheduled.

If the plane agent appears in the middle of the scheduling order, the chance of inefficiency is the highest: For the plane, half of the packages can't be picked up yet, so locations might have to be revisited later. For the trucks, not all post-flight orders can be delivered the first time, and, since trucks will schedule these packages before the other packages can be picked up, this results in extra trips to and from the airport.

In the AIPS dataset, the plane has to schedule when about 20 percent of the trucks have scheduled their segment. Policy 2's behaviour is, for the logistics domain, similar to that of Policy 1, only for Policy 2 the plane agent usually schedules the first segment when about 50 percent of the trucks have scheduled their segment. This explains why Policy 2 performs marginally poorer than Policy 1.

Policy 3 partitions the goal of the plane-agent into many segments — sometimes as much as twelve. Consequently, the quality of the plane's plan is poor. This results in very poor overall performance, because the quality of the plane-plan significantly affects the quality of the joint plan for two reasons. First, the plane's goal is by far the largest; for the AIPS data, the plane has nearly half the number of tasks of all trucks together. Second, a poor plane plan affects truck plans. If a truck has three post-flight packages and the plane delivers them one by one, then, using our policies, the truck will also deliver these packages one by one. Because of this, Policy 3 needs more truck-moves than Policies 1 and 2, despite the fact that Policy 3 produces significantly fewer truck-splits. The smaller number of truck-splits means that many truck agents have no splits at all for Policy 3, against 1 for Policies 1 and 2. However, this does not make these trucks' plans cheaper, because Policies 1 and 2 split their pre-flight from their post-flight tasks, which does not result in overhead.

So, in general, our hypothesis that fewer splits means lower cost plans has not been confirmed. However, looking at Policy 3 alone, we do see the desired pattern emerging. In Figure 6.6, the overhead per agent is plotted against the number of splits per agent for Policy 3. Figure 6.7 shows that the sub-linear pattern is due to the plane agent. The handful of outliers in Figure 6.6 that show a linear pattern are due to truck agents with separated post-flight deliveries. The plan cost of these outliers sometimes reaches the factor d (d segments) times the optimal plan cost. For instance, Figure 6.6 shows that there is one agent with 4 splits and an overhead of 4, meaning that in addition to the optimal plan cost, this agent needs another 4 times the optimal plan cost. This level of inefficiency can be reached if the truck has a sufficient number of post-flight packages that are delivered one by one. The plane agent does not show this linear relation, because for the plane it is unlikely that in each segment all previous work will be repeated.

6.3 A Random Dataset

The results so far are not entirely what we expected, but we feel the results are influenced by the rigid structure of the AIPS data. Therefore, we have created a new set of random

n-city instances. The sort of instances that are contained in this new dataset are given in Table 6.3. In Table 6.3, problem classes are characterized x y z, with x the number of locations per city, y the number of orders and z the number of cities.

	From			To		
	x	y	z	x	y	z
strand 1	10	25	5	10	49	5
strand 2	11	27	5	11	54	5
strand 3	12	30	5	12	59	5
strand 4	20	70	7	20	139	7
strand 5	21	73	7	21	146	7
strand 6	7	14	4	7	27	4
strand 7	8	24	6	8	47	6

Table 6.3: *A dataset of random logistic instances.*

For Policy 1 we again use as random scheduling order the order of the agents in the data files. For this dataset the plane is at the back of the order (a coincidence), which means that Policy 1 implements the arbiter0 protocol.

We show more or less the same graphs for the random dataset as we did for the AIPS dataset. Figures 6.9 and 6.10 show the number of splits required by respectively the trucks and the the plane. In Figure 6.11 we show the overhead generated by each of the policies, for all instances in the dataset. In Figures 6.12 and 6.13, we show the relation between overhead and number of splits for Policy 3.

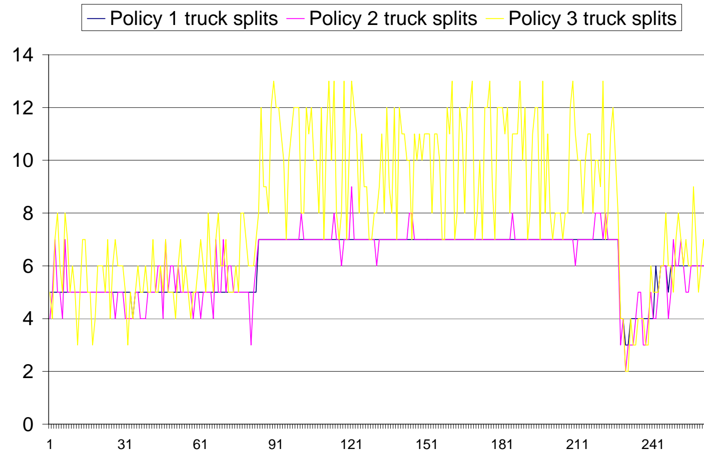


Figure 6.9: *The number of truck splits generated per instance.*

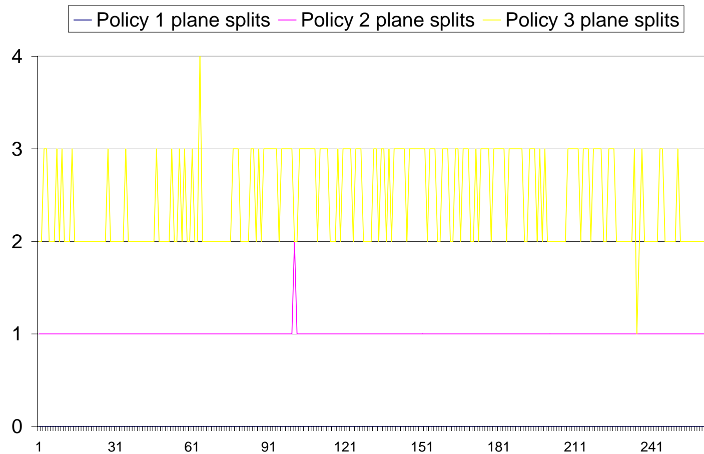


Figure 6.10: *The number of plane splits generated per instance.*

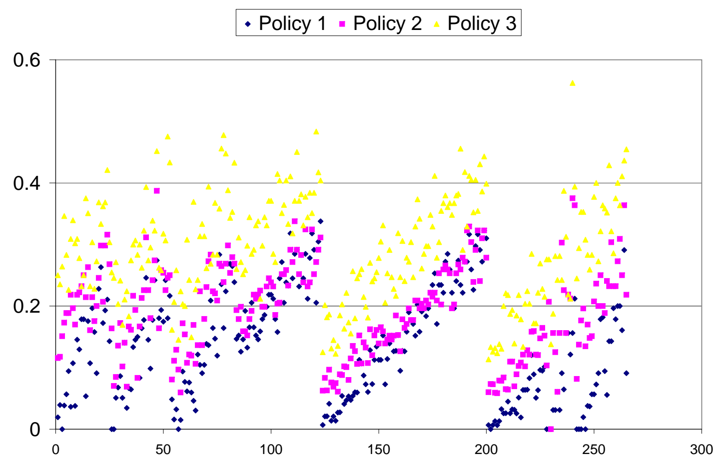


Figure 6.11: *The overhead generated per instance.*

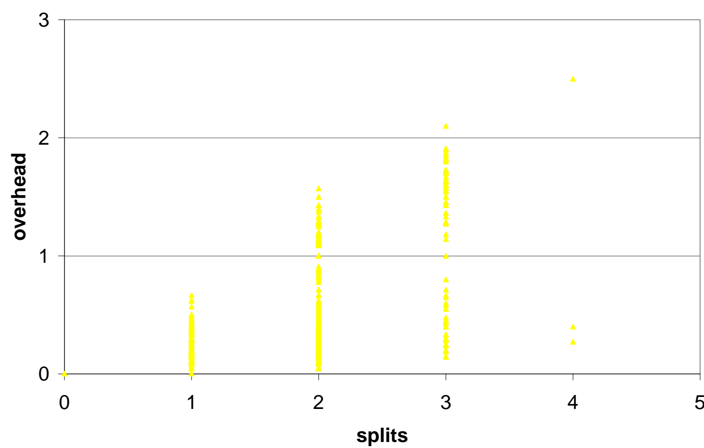


Figure 6.12: *The amount of overhead generated by Policy 3 per agent: data points are pairs* < *number splits, overhead* > *per agent per instance.*

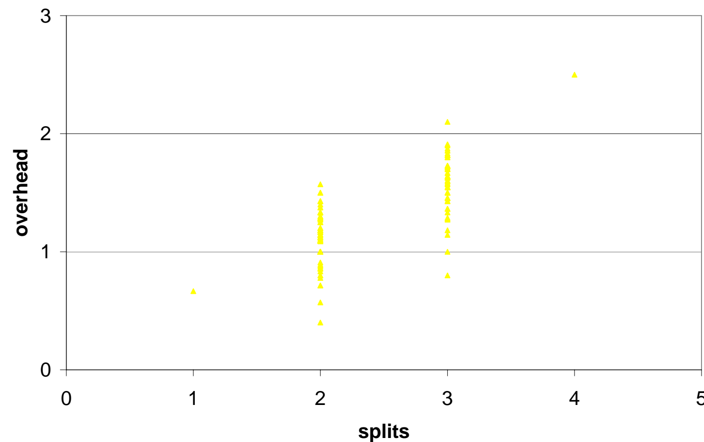


Figure 6.13: *The amount of overhead generated by Policy 3 for the* plane agent*: data points are pairs < number splits, overhead > for the plane agent per instance.*

6.3.1 Interpretation of random dataset results

The relative performance of the three policies is the same for the random dataset as it is for the AIPS dataset. That is, Policy 1 is slightly better than Policy 2, while both are superior to Policy 3. The expected relation between the number of splits and the overhead generated is even less apparent than it was for the AIPS results, however.

Number of segments

The total number of splits required for the policies is given in table 6.4.

	Truck splits	Plane splits	Total splits
Policy 1	1618	0	1618
Policy 2	1622	266	1888
Policy 3	2115	649	2764

Table 6.4: *Random dataset: number of splits generated by each policy.*

Now Policy 1 needs the least number of splits and Policy 3 the most, which is opposite to the situation for the AIPS data. Similar to the AIPS data, Policy 3 needs the most splits for the plane agent, but now Policy 3 also needs the most splits for the trucks; on average that is, because for a few instances Policy 3 needs less.

Note that Policy 1 does not require any plane splits because it implements the arbiter0 protocol. Policy 1 therefore also uses exactly one split for every truck agent. On average,

and for most instances, Policy 2 also uses one split per truck, occasionally using two or none.

Plan cost

In Table 6.5 are the total number of moves each policy needed for the entire random dataset.

	Truck moves	Plane moves	Total moves
Policy 1	30731	2386	33117
Policy 2	30570	3719	34289
Policy 3	31589	5643	37232

Table 6.5: *Random dataset: number of moves generated by each policy.*

Perhaps surprisingly, the number of truck moves required by the three policies does not differ much; most of the difference in performance comes from the plane plan. Policy 3 has about 30 percent more truck splits than policies 1 and 2, but only about 3 percent more truck-moves. Policy three has 140 percent more plane splits than Policy 2 and about fifty percent more plane moves.

The fact that a plane split seems to induce much more overhead than a truck split can be explained by comparing the local goals of a plane agent to a local goal of a truck agent. In the random dataset, the truck agents always cover more locations than the plane agent (our one-city solver was not up to larger plane regions, i.e., more cities), because there are always fewer cities than there are locations per city. Also, most orders are inter-city orders. If there are x cities, then the probability that an order is an intra-city order is roughly $\frac{1}{x}$ (depending on whether we count the plane's region as a city). This means that $1 - \frac{1}{x}$ part of the orders results in a task for the plane, against $\frac{2}{x}$ for a truck agent. So, the plane agent has more orders than the truck agents, spread over fewer locations. This means that there is much scope for executing tasks in parallel, i.e., many deliveries can be made with a short visiting sequence. This also means that if the goal is segmented, then much parallelism will be lost as the plane agent repeats actions in separate segments.

From Figure 6.11, we can conclude that an increasing number of orders for an unchanged infrastructure leads to increased overhead, for all policies. It is possible in Figure 6.11 to distinguish the several strands of instances as clustered sets of points. Not all strands are distinguishable, since a couple of strands are very similar and partially overlap in the graph. Every instance in a strand has one order more than the previous instance (of course, the orders are also different (random) from the previous instance). Along the horizontal axis, the instances in a strand are ordered from left to right as the number of orders increases.

We can conclude that the overhead increases linearly with the number of orders, for all strands and for all policies. This does not necessarily imply that the performance of the policies degrades with the number of orders. At this point, we need to recall that the 'overhead' is defined on the basis of the lowerbound on cost, not on the cost of the optimal solution. As the number of (inter-city) orders increases, the need for coordination also increases. It is thus to be expected that an optimal, coordinated solution will also generate more 'overhead' as the number of orders increases.

The relation between the number of splits and the overhead for Policy 3 is less clear in figures 6.12 and 6.13 than it was for the AIPS dataset, because there is too little spread on the x-axis, i.e., there are too few splits. Each city is a source of precedence constraints for the plane, since each city can produce packages that need to be transported by the plane. Once all trucks have brought these packages to the airport, there are no precedences unscheduled and so no more splits to make for the plane. The second dataset has instances with at most seven cities, whereas the largest instances in the AIPS data has thirty-four cities. Therefore, it is unsurprising that the plane agent doesn't need as much splits for the random instances when compared to the AIPS instances.

6.4 Reuse of Existing Planners using Coordination

Of all the competing planning systems of the AIPS, only TALplanner found low-cost solutions for all problem instances, while needing less than one second of cpu-time. All other planning systems produced plans of higher cost, and/or needed more cpu-time. Two of these other planning systems are STAN and HSP [2].[4]

In this section, we will show that we can improve the performance of both STAN and HSP by using pre-planning coordination. Without coordination, we run the planner once on the entire problem instance; effectively, this is a *centralized* approach to coordination. Using coordination, we apply the planner to each of the agents' local problems in turn.

The particular coordination algorithm we apply is partitioning using the arbiter0 policy. This means that the trucks split their set of tasks into pre- and post-flight orders, whereas the plane remains unconstrained.

Table 6.6 lists the CPU times and the number of actions produced to create the joint plan for centralized and distributed STAN, and for centralized and distributed HSP. For STAN, we see that the already efficient centralized solution is not improved upon by distributed STAN. Still, a significant reduction in solution time[5] can be made (see also Figure 6.14).

Centralized HSP is too slow to find solutions for the ten largest instances (we imposed a time limit of ten minutes), indicated by the entries marked '-'.[6] Distributed HSP, on the other hand, is significantly faster, and quite capable of solving all instances. Furthermore, distributed HSP produces much cheaper plans (Figure 6.15). The cost reduction is possible because the truck problems are now much less complex, so the planner is able to find the optimal solution quickly.

In Figure 6.16, it becomes clear how much time is saved by using pre-planning coordination. For both STAN and HSP, between eighty and ninety-nine percent of solve time can be saved by coordinating the agents prior to planning.

[4]Current research on STAN and HSP can be found at http://planning.cis.strath.ac.uk/STAN/ and http://www.cs.ucla.edu/~bonet/, respectively.

[5]Note that cpu-times for coordinated STAN/HSP are obtained by summing the solution times of all agents, not by recording the finish time of the slowest agent.

[6]Note that HSP, both centralized and distributed, also failed to solve a number of smaller instances. This is due to the fact that the current version of HSP is not yet entirely stable, and produced error messages on those instances.

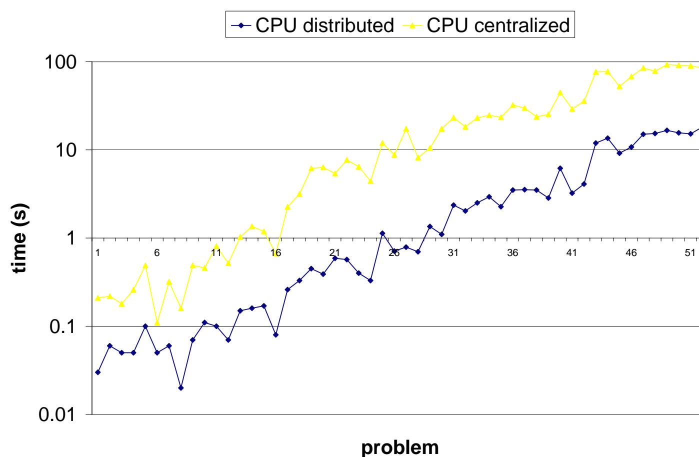


Figure 6.14: *CPU times: coordinated STAN vs. uncoordinated STAN, for track 1 (additional instances) of the AIPS logistics dataset.*

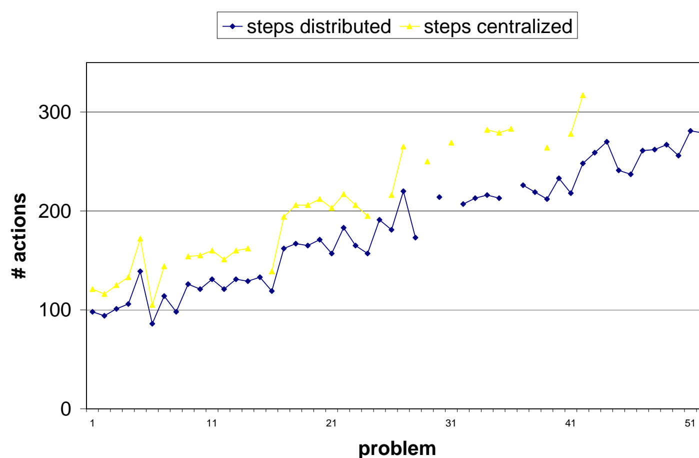


Figure 6.15: *Number of actions produced by coordinated HSP vs. uncoordinated HSP, for track 1 (additional instances) of the AIPS logistics dataset.*

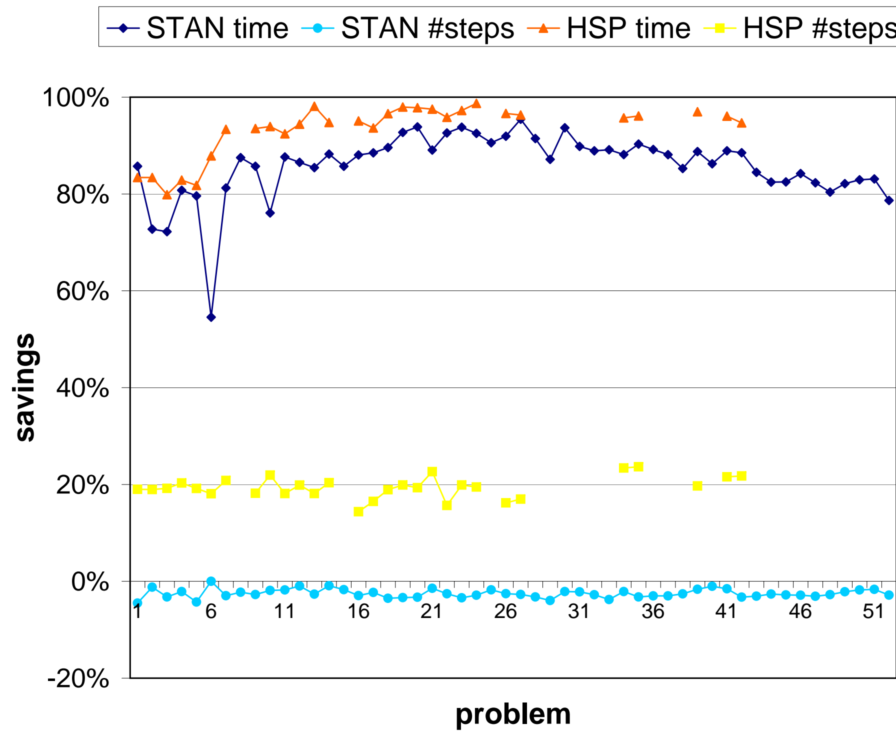


Figure 6.16: *Percentage of time or actions saved by applying pre-planning coordination to STAN and HSP.*

problem	uncoordinated STAN		coordinated STAN		uncoordinated HSP		coordinated HSP	
	cpu-time	plan cost	cpu-time	plan cost	cpu-time	plan cost	cpu-time	plan cost
log-16-0	0.21	89	0.03	93	4.52	121	0.75	98
log-16-1	0.22	83	0.06	84	4.64	116	0.77	94
log-17-0	0.18	93	0.05	96	4.02	125	0.81	101
log-17-1	0.26	95	0.05	97	4.61	133	0.79	106
log-18-0	0.49	117	0.1	122	7.96	172	1.45	139
log-18-1	0.11	78	0.05	78	5.77	105	0.7	86
log-19-0	0.32	101	0.06	104	12.8	144	0.85	114
log-19-1	0.16	89	0.02	91	-	-	0.6	98
log-20-0	0.49	110	0.07	113	19.28	154	1.25	126
log-20-1	0.46	106	0.11	108	14.84	155	0.9	121
log-21-0	0.81	113	0.1	115	14.67	160	1.11	131
log-21-1	0.52	103	0.07	104	18.23	151	1.02	121
log-22-0	1.03	113	0.15	116	69.85	160	1.31	131
log-22-1	1.36	111	0.16	112	27.91	162	1.46	129
log-23-0	1.19	116	0.17	118	-	-	1.28	133
log-23-1	0.67	102	0.08	105	22.74	139	1.12	119
log-24-0	2.26	131	0.26	134	33.89	194	2.16	162
log-24-1	3.17	143	0.33	148	55	206	1.88	167
log-25-0	6.18	149	0.45	154	113.92	206	2.33	165
log-25-1	6.34	152	0.39	157	138.88	212	3.04	171
log-26-0	5.4	138	0.59	140	146.34	203	3.62	157
log-26-1	7.7	156	0.57	160	124.92	217	5.21	183
log-27-0	6.45	147	0.4	152	137	206	3.78	165
log-27-1	4.42	139	0.33	143	248.42	195	3.19	157
log-28-0	11.99	171	1.13	174	-	-	9.46	191
log-28-1	8.8	157	0.71	161	197.31	216	6.66	181
log-29-0	17.35	184	0.79	189	273.46	265	10.08	220
log-29-1	8.19	155	0.7	160	-	-	5.63	173
log-30-0	10.51	177	1.35	184	260.65	250	-	-
log-30-1	17.39	189	1.1	193	-	-	10.38	214
log-31-0	23.22	184	2.36	188	401.79	269	-	-
log-31-1	18.31	181	2.03	186	-	-	13.44	207
log-32-0	23.12	186	2.51	193	-	-	21.71	213
log-32-1	24.71	190	2.93	194	463.14	282	19.79	216
log-33-0	23.43	186	2.27	192	495.4	279	19.29	213
log-33-1	32.36	200	3.5	206	364.35	283	-	-
log-34-0	29.82	199	3.54	205	-	-	32.22	226
log-34-1	23.77	193	3.5	198	-	-	19.74	219
log-35-0	25.35	182	2.85	185	565.41	264	17.05	212
log-35-1	44.89	195	6.18	197	-	-	36.46	233
log-36-0	29.16	196	3.23	199	608.58	278	23.93	218
log-36-1	35.69	215	4.1	222	607.09	317	32.22	248
log-37-0	76.83	228	11.93	235	-	-	101.63	259
log-37-1	77.25	228	13.55	234	-	-	114.3	270
log-38-0	52.41	214	9.18	220	-	-	80.41	241
log-38-1	68.09	209	10.75	215	-	-	92.73	237
log-39-0	84.9	227	15.01	234	-	-	99.01	261
log-39-1	78.24	219	15.35	225	-	-	118.57	262
log-40-0	93.06	232	16.63	237	-	-	160.54	267
log-40-1	91.21	225	15.58	229	-	-	118.24	256
log-41-0	89.85	242	15.18	246	-	-	164.73	281
log-41-1	85.18	245	18.17	252	-	-	174.18	279

Table 6.6: *Comparing the output of planning systems with and without making use of pre-planning coordination.*

6.5 Concluding Remarks

The relation between planning autonomy and resulting plan cost is not as general as we anticipated in Section 6.1.1. For the plane agent, a higher number of constraints (resulting from a finer partitioning of the agent's set of tasks) clearly results in poorer plans, but for the truck agents this relation is not so strong. Especially, as coordination often results in trucks having to perform their pre-flight tasks prior to their post-flight tasks, plan cost is hardly affected at all. In hindsight, this is not that surprising after all: for the AIPS data, the strategy of separating pre-flight from post-flight tasks actually allows optimal truck plans, while for general logistic instances, this is still a very efficient strategy, as shown in [29].

We can conclude that, in designing efficient partitioning strategies, it is not sufficient to concentrate on finding strategies that require a low number of splits. Instead, the specific problem at hand largely determines whether an additional precedence constraint is harmful to plan cost or not.

A second conclusion we can draw is that pre-planning coordination is a viable method to solve multi-agent planning problems. Pre-planning coordination allows us to (re)use single-agent planning tools on the sub-problems allocated to the agents. The fact that these sub-problems are typically less complex than the multi-agent planning problem means that total solve time is reduced (cf. [8]). Also, the cost of the resulting joint plan is typically lower than the cost of the joint plan produced by a single monolithic planning system. In theory, applying one centralized planner — corresponding to the agents forming the grand coalition — could result in the optimal plan being found. In practice, however, the computational complexity of the multi-agent planning problem is such that only a sub-optimal plan can be found in reasonable time.

Chapter 7

Conclusions and Future Work

In this thesis, we have studied the coordination of autonomous agents that together have to complete a joint task, while they wish to keep details of their plans, and of the planning activity itself, to themselves. The solution method we have proposed is to find, prior to planning, a minimum set of additional constraints on the agents' goals.

Curiously, though, this coordination problem has received little or no attention in the multi-agent coordination literature, even after nigh on thirty years of distributed artificial intelligence. Naturally, this raises the question whether our research is of any significance at all. Certainly, if I were a Reader, and I discovered, after almost a hundred pages of mostly prose as dry as ancient parchment, that the report I have struggled through is about as relevant as last week's weather report, I would be positively miffed.

Seriously though, we feel that there are a number of reasons why our research has relevance after all. First, the separation of coordination and planning our approach proposes immediately yields results, as it enables us to reuse existing single-agent planning tools to solve difficult multi-agent planning problems. In fact, in Chapter 6, we have shown that our pre-planning coordination algorithms, combined with our own simple single-agent planning tools, were capable of outperforming the state-of-the-art (albeit of the year 2000) multi-agent planning systems.

A second — and more distant — application area of our research is the area of 'inter-organizational' multi-agent systems. Currently, the vast majority of multi-agent systems (and they are not that prevalent) is located within a single organization. In the future, however, we can expect a greater part of business operations — including dealings with other companies — to be automated. Also, if globalization, and consequently specialization, continue to increase, then we can expect there to be an increased demand of inter-organizational multi-agent systems — multi-agent systems in which agents need to cooperate with other agents for e.g. the production of some good, while remaining autonomous in the planning activity.

To increase the applicability of our framework, we can identify several areas for future work, of which we will now discuss two. First, there is the issue of timing. Although we briefly touched on the need to synchronize agent plans in Chapter 5, we were able to dismiss synchronization and timing as a coordination *problem* under the assumption that agents only care about the amount of work they must perform, not about the times at which to perform it. Obviously, this assumption is unrealistic in many settings; in our multi-modal

logistics domain, it is for instance unlikely that a plane will delay its departure time with an hour or so to enable a truck to pick up some packages first. A natural extension to our framework is then to associate a *deadline* with the completion of each task, or perhaps a *time window*, such that for each task, there is an earliest start-time and a latest completion-time. Of course, scheduling tasks with deadlines in itself is not a new problem, but it is if we combine it with planning-autonomy-maximizing coordination.

A second possible extension to our framework is to examine the relation between task allocation and coordination. Clearly, if we can allocate tasks in such a way that there are no dependencies between agents, then there is no need to coordinate, either. However, if inter-agent dependencies are unavoidable, we can formulate the following problem: how to allocate tasks to agents, such that the subsequent coordination process requires a minimum number of additional constraints?

Of course, it is possible to identify other areas of future work; every multi-agent approach ever published is based upon a number of simplifying assumptions, such as deterministic outcome of actions, and environments that are restricted in the way they are 'allowed' to change. Thus, to fill the future-work paragraph, it is only too easy to coin a phrase like 'dynamic environment', and to point out that, in real life, circumstances change. Circumstances do change, of course, but even though some degree of dynamism might well be taken into account in our framework, we must guard against trying to fit all of reality into one model, for otherwise coordination would become even more complex than it already is. Even now, with a framework that some may consider too simplistic, the computational complexity of coordination is Σ_2^p-complete (see Chapter 3). Also, as Durfee [7] points out, no coordination algorithm can be all things to all men; there is simply a limit to what one model of reality can contain.

> *As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.* — Albert Einstein

Appendix A

Solving the One-City Problem

Presenting the empirical results in Chapter 6, we assumed that agents make optimal *local* plans. The local problem (for all agents, including the plane agent) is the one-city problem (Definition 5.3.3) which we discussed in Chapter 5. Recall that the one-city problem is to pick up and deliver packages to and from locations, where any location can be reached from any other location. Also, we assume that the transporting vehicle can carry an unlimited number of packages at the same time.

A one-city instance $I = (L, O)$ is characterized by a set of locations L and a set of orders $O \subseteq L \times L$. The set L is actually superfluous, since we assume that L does not contain any unused locations:

$$\forall l \in L : l \in \text{ran}(O) \vee l \in \text{dom}(O)$$

An approximate solution that is valid for all one-city instances is the following [29]: First all locations are visited once in an arbitary order to enable pickup of all packages. Second, all locations are visited again in an arbitrary order, which allows delivery of all packages. For most instances, however, we can find a solution that does not visit all locations twice.

For any solution, all locations must be visited at least once (since there are no unused locations in L). Visiting all locations once, in whatever order, is clearly not sufficient though for some instances. If we have two orders (A,B) and (B,A), then either location A or location B must be visited twice. Thus, some locations must be visited twice. An optimal solution (a minimum length visiting sequence) minimizes the number of locations that needs to be visited twice.

Given the set of orders O and the set of locations L, we can define the directed *order-graph* $G_O = (L, O)$. Thus, vertices in G_O are the locations, the arcs are the orders between those locations. An example order-graph is shown in Figure A.1.

Gven a cycle $C = l_1 - \cdots - l_n - l_1$ in G_O, every location in C is both a pickup-location and a delivery-location. In a visiting sequence VS enabling execution of all orders, one location $l_i \in C, i = 1, \ldots, n$ must appear before all other locations in C. For this l_i, this means that the order from l_{i-1} to l_i cannot be delivered on the first visit to l_i. Thus, l_i must appear once more in VS. Hence, for every cycle in the order-graph, at least one location must be visited twice.

To determine which locations need to be visited twice, we need to find a set $L' \subseteq L$ of locations such that for every directed cycle C in G_O, L' contains at least one location in C.

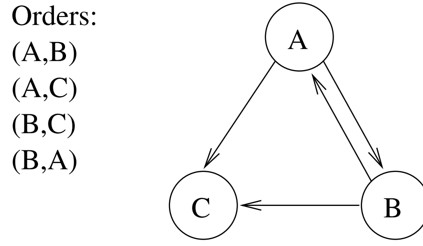


Figure A.1: *A set of orders and the order-graph*

In other words, we need to find a feedback vertex set for the order-graph (see Chapter 3 for the definition of the feedback vertex set problem).

Given a feedback vertex set for the order-graph, finding a visiting sequence is easy. First, we construct the graph G/F: Let G/F be the directed acyclic graph (dag) that results when each vertex v in F is split into v^i and v^o, such that all edges $(v, j) \in O$ are replaced by edges of the form (v^o, j) and all edges $(j, v) \in O$ are replaced by edges of the form (j, v^i) (cf. [16]). Figure A.2 shows an example of a graph G/F.

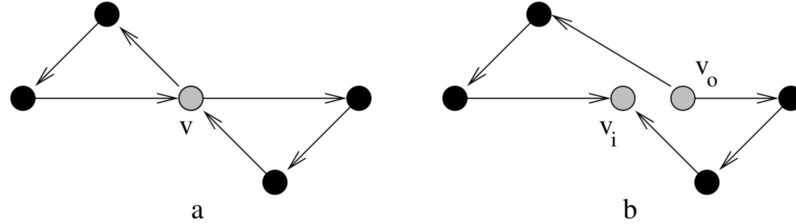


Figure A.2: *(a): Directed graph G, (b): directed graph G/F, i.e., the graph for which each vertex in F is split in two.*

Since G/F is a dag, the set of arcs in G/F induces a partial order on the set of vertices. By making a *topological sort*, a partial order can be extended to a total order. Any total order for the vertices in G/F corresponds to a valid visiting sequence: Locations appear in the ordering in such a way that if $(i, j) \in O$, then there is an appearance of i before an appearance of j in the ordering.

In the above we have described the *Turing-reduction* from the one-city problem to the feedback vertex set problem. Algorithm 13 gives a more concise description of the reduction.

Algorithm 13 One-City $\leq_T$ FVS

1: **Input:** $G_O = (L, O)$, a directed graph.
2: **Output:** minimum-length visiting sequence VS
3: $F = \text{FVS}(G_O)$
4: construct $G/F = (L', O')$
5: construct topological sort s for (L', O')
6: **Return** s

To implement Algorithm 13, we need a solver for the feedback vertex set problem (which is NP-hard). Our choice was to encode the FVS problem as a set of Integer Linear Programming (ILP) constraints, in the following way:

1. Find the set $\mathcal{C}$ of all cycles in G_O.
2. For every $C = l_1 - \cdots - l_n - l_1 \in \mathcal{C}$, encode the constraint:
$$x_{l_1} + \cdots + x_{l_n} \geq 1$$
3. All x_{l_i} can take the values 0 or 1.
4. The objective function is: Minimize $\sum_{i=1}^{|L|} x_{l_i}$

The constraints in 2 signify that for each cycle, there must be at least one variable that is assigned the value 1. The feedback vertex set is given by those variables that have value 1. The locations in the feedback arc set must be visited twice, all other locations can be visited once.

In step 1 of our ILP encoding, we find all cycles in G_O. Since a graph may contain an exponential number of cycles, our encoding might take exponential time. Also, because we encode a constraint for every cycle, we create an exponential-size ILP instance. Since polynomial-size ILP instances can already require exponential solve time, using our ILP encoding we no longer have a Turing reduction, which requires that, apart from the finding of the feedback vertex set (in our case, using ILP), the reduction algorithm must run in polynomial time.

To construct a visiting sequence, given a feedback vertex set F, our implementation differs slightly from the specification of Algorithm 13.[1] Instead of constructing the graph G/F and creating a topological sort, we build a visiting sequence using Algorithm 14.

Input for Algorithm 14 are the sets O (orders), L (locations) and F (locations that need to be visited twice). Starting from an empty visiting sequence VS, each iteration of the algorithm adds one location from either F or L and removes this location from the respective set. Before the algorithm starts iterating, we first construct the *inverse order matrix* M. An entry $M[i][j]$ is *true* if $(i, j) \in O^{-1}$, otherwise $M[i][j]$ is *false*. The matrix M is used to check, for a certain location l_i, if there are locations l_j $(i, j \in \{1, \ldots, n\})$ that must appear in the visiting sequence VS before an occurence of l_i. If $M[i][j]$ is *true*, then there is an order from l_j to l_i, and l_j is not yet in VS. Thus, before the final visit to l_i, we must visit l_j to pick up the package.

At the start of each iteration, we find the set L' of locations. L' contains those locations $l_i \in L$ for which all locations $l_j, (l_j, l_i) \in O$ are already in VS. This means that a visit to a location in L' need not be delayed (any longer), because all packages that must be delivered to locations in L' have been picked up. If L' is non-empty, we can extend the VS with any of the locations in L'. If L' is empty, on the other hand, then we must increment the visiting sequence with an arbitrary element from F. A visiting sequence has been found once all locations in F and L have been added to VS.

[1] The circumstances leading to the discrepancy between the specification and the implementation of Algorithm 13 are comparable to — perhaps I should say: exactly like — a student writing code specification only after the code has been written.

Algorithm 14 buildVisitingSequence

1: **Input:** Set of locations L, set of orders O, feedback vertex set F
2: **Output:** Visiting sequence VS
3: $M[][] = false$
4: **for all** $(l_1, l_2) \in L \times L$ **do**
5:     **if** $(l_2, l_1) \in O$ **then**
6:         $M[l_1][l_2] = \text{true}$
7:     **end if**
8: **end for**
9: $VS = \emptyset$
10: $n = |L|$
11: **while** $F \cup L \neq \emptyset$ **do**
12:     $L' = \{l \in L | \forall 1 \leq i \leq n : M[l][i] == \mathit{false}\}$
13:     **if** $L' \neq \emptyset$ **then**
14:         Let nextloc $\in L'$
15:         $L = L \setminus \{\text{nextloc}\}$
16:     **else**
17:         Let nextloc $\in F$
18:         $F = F \setminus \{\text{nextloc}\}$
19:     **end if**
20:     $VS = VS+$ nextloc
21:     $\forall 1 \leq i \leq n$: $M[i][\text{nextloc}] = \mathit{false}$
22: **end while**
23: **Return** VS

Appendix B

Source Code for Partitioning

In Chapter 4, Section 4.3.3, we presented the coordination-by-partitioning approach as a centralized algorithm, in which input from a number of (distributed) agents is requested. Our aim was to empirically determine the efficiency of various partitioning strategies the agents might follow. Of course, to test a strategy, we need not actually implement a distributed system. In fact, we have implemented the whole coordination algorithm using a single thread of control: a *main* function initializes the agents, starts the the protocol (in which agents are activated one at a time), and handles the result.

Our choice of programming language was C++, having the power of C, yet enabling reuse through object-orientation (OO) techniques. In particular, we had in mind a system in which a new agent strategy could be created simply by subclassing the main agent class. Admittedly, though, the final implementation has turned out to be a great deal more ad hoc than that, as we will explain in Section B.2.

B.1 The Protocol

The code for the coordination protocol is contained within the same file as the *main* subroutine. A large part of this file — the methods *writeLine*, *writeOutput*, *read_number_agents*, and *readInput* are all concerned with either reading the problem instance (consisting of complex task plus task allocation), or writing the results (the partitioning $\mathbf{T}$ of T) respectively from and to file. In addition, the main-file contains one additional auxilary method, *growTempStore*, which is used to inelegantly manage memory, within the method *readInput*.

The routine *main* itself contains the code for the protocol. First, however, it creates the agents, instances of the class *HeuristicAgent*. Agents are initialized with (i) their part of the complex task, i.e., the tasks they have been allocated and, for each task, the tasks on which that particular task is dependent (including tasks allocated to other agents), and (ii) a pointer to the global store **done**.

The global store is implemented as a bitvector, in which each bit corresponds to a task in T. If a bit is set, then this means that the corresponding task has been scheduled in some segment by the agent to which the task has been allocated. Even though all agents have unrestricted access to the global store, mutually exclusive access is guaranteed because at most one agent is busy at a time.

The coordination protocol itself is implemented in a while-loop, which mirrors the specification given in Algorithm 9: until all tasks have been scheduled, an iteration is started in which all agents are first invited to build a candidate segment, after which a sub-routine *pickSegment* is called to determine which agent will schedule next.

```
/* FILE: main.cpp

main reads input from a file, runs the agents sequentially
and stepwise, and outputs the results

input:
    number_agents
    local_task1 prec1,1 ... precm,1    0    (0 is end of line symbol)
            ..
    local_taskn prec1,n ... precl,n    0
            -1                              (end of agent symbol)

            ..                              (following agents)

    file ends with -1


output: same as input, except:
    - local tasks are clusterd, from top to bottom, by segment:

    (first segment, first agent)
    local_task1,1 prec1,1,1 ... precm,1,1    0
              ..
    local_taskn,1 prec1,n,1 ... precl,n,1    0
            -2                                   (segment separator)
    ..                                      (next segment..)


*/
//DOESN'T WORK WITH EMPTY AGENTS
//FUNCTION GROW TEMPSTORE STILL BUGGED

//#define DEBUG
#define MEMCHECK

#include <iostream>
#include <fstream>
#include <string>
#include <time.h>

#ifndef INTREEKS_H
#include "Intreeks.h"
#include "Intreeks.cpp"
#endif

#ifndef BITVECTOR_H
#include "BitVector.h"
#include "BitVector.cpp"
#endif

#ifndef HEURISTICAGENT_H
```

```
#include "HeuristicAgent.h"
#include "HeuristicAgent.cpp"
#endif

#define TASKS_PER_AGENT 20

using namespace std;

typedef IntArray* p;

int readInput(ifstream &file, int num_agents);
int read_number_agents(ifstream &file);
void writeOutput(IntArray*, int, char*);
void writeLine(ofstream&, int, int);
void displayMatrix(int);
void displayResults(IntArray*);
void displayTempStore(int, IntArray*);
int growTempStore(int);

//global variable, modifiable by readInput
//containing the task matrix for all agents
IntArray **ptasks;

//this IntArray is used for memory management:
//when deleting ptasks, it has to be known
//how many IntArrays each agent used
IntArray num_tasks_per_agent;

//array of agents
HeuristicAgent *pagent;

int number_agents;
int heuristicmode;

int growTempStore(int oldsize, int index, IntArray *&temp_store)
//copy temp_store to a new, larger IntArray
{
  IntArray *tmp = temp_store;

  int i;
  int size = index * 2;
  temp_store = new IntArray[size];
  //copy array elements
  for(i = 0; i < oldsize; i++)
  {
    int lengte = tmp[i].getElementCount();
    for(int j=0; j < lengte; j++)
    {
      temp_store[i][j] = tmp[i][j];
    }
  }
  //delete the array pointed to by tmp
  delete [] tmp;
  return size;
}
```

```
void writeLine(ofstream &out, int agent_index, int task_index)
//write a single line of output to file
{
  int linelen = ptasks[agent_index][task_index].getElementCount();
  for(int i=0; i < linelen; i++)
  {
    out << ptasks[agent_index][task_index][i] << ' ';
  }
  out << "0\n";
}

void writeOutput(IntArray *res, int num_agents, char* filename_out)
//write the results to file
{
  int agent, seg, task, segfound, tasknum, number_tasks_written;
  ofstream out;

  //open files
  out.open(filename_out);
  out << num_agents << '\n';

  for(agent = 0; agent < num_agents; agent++)
  {
    number_tasks_written = 0;
    segfound = 1;
    for(seg = 1; segfound == 1; seg++)
    {
      segfound = 0;
      //find the number of tasks this agent has to perform
      tasknum = num_tasks_per_agent[agent];
      for(task = 0; task < tasknum; task++)
      {
        if(seg == res[agent][task])
        {
          number_tasks_written++;
          writeLine(out, agent, task);
          segfound = 1;
        }
      }

      //write segment separator

      //to avoid writing a segment separator just before
      //the agent separator
      if(number_tasks_written < tasknum)
        out << "-2\n";
    }

    //write agent separator
    out << "-1\n";

  }
  out.close();
}

int read_number_agents(ifstream &file)
```

```
//the file is assumed to be already open
//with the cursor at the beginning
{
  int num_agents;
  file >> num_agents;
  return num_agents;
}

int readInput(ifstream &file)
//read the composite task, store it in memory, and
//return the highest taskid that was encountered in the file
{
  int num_agents, temp, maxtaskid, i;
  int tasks_per_agent;

  //lines are first read into temp_store, to
  //determine the number of tasks a line contains
  //and the number of tasks an agent has to perform
  IntArray *temp_store;
  int line_counter;
  //agent_index keeps tracks of the first dimension of ptasks (agents)
  int agent_index;
  //reset the file
  file.seekg(0);
  file >> num_agents;
  //allocate memory
  ptasks = new p[num_agents];
  tasks_per_agent = TASKS_PER_AGENT;
  temp_store = new IntArray[tasks_per_agent];
#ifdef MEMCHECK
    if(temp_store == NULL)
      cout << "out of memory at read_input\n";
#endif

  agent_index = 0;
  line_counter = 0;
  maxtaskid = 0;
  while(agent_index < num_agents)
  {
    //read a number into a temporary variable
    file >> temp;
    if(temp > maxtaskid)
      maxtaskid = temp;
    switch(temp)
    {
      case 0:
      {
        //end of line (no more precedences for this task)
        line_counter++;
      } break;
      case -1:      //end of agent
      {
        //copy tempstore into ptasks
        //if an agent has no tasks, create one empty
        //IntArray for him, instead of null
        int tmp = (line_counter > 0) ? line_counter : 1;
```

```
        ptasks[agent_index] = new IntArray[tmp];
        num_tasks_per_agent.addInt(line_counter);
        for(i = 0; i < line_counter; i++)
        {
          int line_length = temp_store[i].getElementCount();
          for(int j = 0; j < line_length; j++)
          {
            ptasks[agent_index][i][j] = temp_store[i][j];
          }
        }

        //update agent count
        agent_index++;
        //clean up the IntArrays
        for(i=0; i < line_counter; i++)
        {
          int elemcount = temp_store[i].getElementCount();
          for(int j=0; j < elemcount; j++)
          {
            //remove the first element,
            //because IntArray moves elements to the
            //front when items are removed
            temp_store[i].removeIntAt(j);
          }
        }

        //reset the line_counter
        line_counter = 0;
        break;
      }
      default:
      {
        //store the task temp_store
        if(line_counter >= TASKS_PER_AGENT)
        tasks_per_agent = growTempStore(tasks_per_agent, line_counter, temp_store);
        temp_store[line_counter].addInt(temp);
        break;
      }
    }
  }
  delete [] temp_store;
  return maxtaskid;

}

void pickSegment()
{
  int i, index;
  double cost, mincost;
  //splitCost returns a value between 0 and 1
  mincost = 1.1;
  //determine which agent has the lowest splitting cost
  for(i=0; i < number_agents; i++)
  {
    if(!pagent[i].isDone() && pagent[i].madeProgress())
    {
```

```
      cost = pagent[i].splitCost(heuristicmode);
      if(cost < mincost)
      {
        index = i;
        mincost = cost;
      }
    }
  }
  //schedule that agent's segment
  pagent[index].scheduleSegment();
}

int main(int argc, char** argv)
{
  char *filename_in;
  char *filename_out;
  int i;
  int maxtaskid;

  //used to collect results from the agents,
  //once a partitioning has been made
  IntArray *results;

  IntArray tmp;

  //input stream
  ifstream input_file;
  //output stream
  ofstream output_file;
  heuristicmode = 0;

  //handle the command line parameters:
  switch(argc)
  {
    case 1:
    {
      filename_in = "input.txt";
      filename_out = "output.txt";
    }
    break;
    case 2:
    {
      filename_in = argv[1];
      filename_out = "output.txt";
    }
    break;
    case 3:
    {
      filename_in = argv[1];
      filename_out = argv[2];
    }
    break;
    case 4:
    {
      filename_in = argv[1];
      filename_out = argv[2];
```

```
    heuristicmode = atoi(argv[3]);
  }
  break;
  default:
  {
    filename_in = "input.txt";
    filename_out = "output.txt";
  }
  break;
}

//read input
input_file.open(filename_in);
number_agents = read_number_agents(input_file);
maxtaskid = readInput(input_file);
input_file.close();

//initialize the global store
//size = maxtaskid + 1, because bitvector
//starts counting at 0, whereas tasks start at one
BitVector global_store(maxtaskid + 1);

results = new IntArray[number_agents];

pagent = new HeuristicAgent[number_agents];

//start measuring execution time
long t0 = clock();

//initialize the agents: pass their tasks
for(i=0; i < number_agents; i++)
{
  // the pointer to the datastructure
  // of the composite-task object is passed to the agent
  int num_tasks = num_tasks_per_agent[i];
  pagent[i].initialize(num_tasks, ptasks[i], &global_store);
}

int alldone = 0;
//run the algorithm
while(!alldone)
{
  //every agent is assumed ready
  alldone = 1;
  for(i=0; i < number_agents; i++)
  {
    if(!pagent[i].isDone())
    {
      //until proven otherwise
      alldone = 0;
      pagent[i].buildSegment();
    }
  }
  if(!alldone)
    pickSegment();
}
```

```
  //handle the results of the agents:
  //copy the arrays of the agents into results-variable
  for(i=0; i < number_agents; i++)
  {
    pagent[i].getResults(results[i]);
  }

  //finish measuring execution time
  long t1 = clock();
  long diff = t1 - t0;
  double timeinsec = (double)diff / CLOCKS_PER_SEC;
  cout << "coordination algorithm execution time: " << timeinsec << "\n";

  //now write the results back to a file
  writeOutput(results, number_agents, filename_out);

  //free the memory: first the agents
  delete [] pagent;

  //delete the task matrix
  for(i=0; i < number_agents; i++)
  {
    int lengte = num_tasks_per_agent[i];
    delete [] ptasks[i];
  }

  delete [] ptasks;
  delete [] results;

  return 0;
}
```

B.2 Agent Strategy

The following source code is the implementation for partitioning-policies 2 and 3 (of Chapter 4), in which the source code for policy 1 is more or less embedded — namely, in the method *buildSegment*. This method builds a (candidate) segment consisting of all tasks that can be scheduled now. If an agent is indeed chosen to schedule his segment in this iteration, then the tasks in this segment are 'set' on the global store, and the segment-index is incremented. If another agent is chosen to schedule his segment, then the current segment will be used as the basis for the next call of *buildSegment*.

The method *buildSegment* works as follows: for every task not yet scheduled, it is checked whether it still has any unscheduled prerequisites (i.e., preceding tasks) left. If not, then the task can be added to the current segment, and it is *locally* marked as scheduled, so that other local tasks can consider this task as scheduled. Also, this means that a task t that was previously considered unschedulable must be reconsidered, if a newly scheduled task was a local prerequisite for t.

The method *splitCost* implements the agent strategy: it returns the heuristically determined cost of scheduling the current segment, on the basis of which the method *pickSegment* (from the main-file) will determine which agent gets to schedule his segment. Method

splitCost is little more than a switch-statement, however, as it calls either *costSegmentSize* or *costPrecsSatisfied*, depending on whether respectively policy 2 or policy 3 is being implemented.

Now follows first the code of the header file, followed by the code of the .cpp file.

```
/* FILE: HeuristicAgent.h

Documentation:
local_precs[][]: a square matrix where the rows and colums both
represent tasks this agent needs to perform. The matrix is
filled with ones and zeros. A 1 in row i, col j means task i
depends on task j, i.e., task j precedes i

global_precs: a matrix where the rows represent the tasks the agent
has to perform, the colums are tasks outside the agent's control on
which its tasks depend. The matrix is filled with the ids of the
outside tasks.

The task ids of the local tasks are only used for updating the global store,
so they are kept in a separate array, int *taskids. In other cases, the taskid
can be inferred from the row-number in the other datastructures

The array number_segments keeps track of the segment_number a task has been
assigned to. The int segment_number keeps track of the segment that is
currently being scheduled
*/

#ifndef HEURISTICAGENT_H
#define HEURISTICAGENT_H
#define GROOTTE 10
#endif

#ifndef BITVECTOR_H
#include "BitVector.h"
#include "BitVector.cpp"
#endif

#ifndef INTREEKS_H
#include "Intreeks.h"
#include "Intreeks.cpp"
#endif

#include <iostream>

using namespace std;

class HeuristicAgent
{
public:
    HeuristicAgent(BitVector *store = 0)
    {
      done = 0;
      //all agents have a pointer to the globalstore
      global_store = store;
    }
```

```
    ~HeuristicAgent()
    {
      //delete the dynamic structures:
      delete [] global_precs;
      delete [] local_precs;
      delete num_global_precs;
      //invalidate the pointer to the global_store
      global_store = 0;
    }

    int isDone();
    void getResults(IntArray&);
    void buildSegment();
    void scheduleSegment();
    double splitCost(int);
    int madeProgress();
    //the int (num_tasks) is needed to know
    //how many IntArrays are passed
    void initialize(int, IntArray*, BitVector*);

protected:
    int done;
    int number_tasks, segment_number;
    IntArray segment_numbers;
    IntArray taskids;

    //two variables for use in heuristic functions
    int total_num_global_precs;
    int *num_global_precs;

    //and another one, to ensure progress
    int tasks_added_to_segment;

    IntArray *local_precs;
    BitVector *global_store;
    IntArray *global_precs;
    double costSegmentSize();
    double costPrecsSatisfied();
};

//FILE: HeuristicAgent.cpp

#ifndef HEURISTICAGENT_H
#include "HeuristicAgent.h"
#endif

inline int HeuristicAgent::isDone()
{
  return done;
}

inline void HeuristicAgent::getResults(IntArray& array)
{
  int size = segment_numbers.getElementCount();
  for(int i=0; i < size; i++)
  {
    array[i] = segment_numbers[i];
```

```
  }
}

inline int HeuristicAgent::madeProgress()
{
  return tasks_added_to_segment;
}

inline void HeuristicAgent::scheduleSegment()
{
  int i;
  tasks_added_to_segment = 0;
  //schedule the current segment
  for(i=0; i < number_tasks; i++)
  {
    if(segment_numbers[i] == segment_number)
    {
      //put it on the global store
      int taskid = taskids[i];
      (*global_store).set(taskid);
    }
  }

  //increment the segment number
  segment_number++;

  //check to see if there is any work left
  int unfinishedtask = 0;
  for(i=0; !unfinishedtask && i < number_tasks; i++)
  {
    if(segment_numbers[i] == EMPTY)
      unfinishedtask = 1;
  }

  if(unfinishedtask == 0)
    done = 1;
}

inline double HeuristicAgent::costSegmentSize()
//Policy 2
{
  int tasksinsegment, i;
  double cost;

  tasksinsegment = 0;
  for(i=0; i < number_tasks; i++)
  {
    if(segment_numbers[i] == segment_number)
      tasksinsegment++;
  }

  cost = 1.0 - (double) tasksinsegment / (double) number_tasks;
  return cost;
}
```

```
inline double HeuristicAgent::costPrecsSatisfied()
//Policy 3
{
  int i,j, count;
  double cost;
  //if there aren't any global precs, there is no need to delay
  //scheduling
  if(total_num_global_precs == 0)
    return 0.0;

  count = 0;
  for(i=0; i < number_tasks; i++)
  {
    if(segment_number == segment_numbers[i])
    {
      count += num_global_precs[i];
    }
  }

  cost = (double) count / (double) total_num_global_precs;
  return 1.0 - cost;
}

inline double HeuristicAgent::splitCost(int mode)
//determines the cost of making a split with the current segment
{
  switch(mode)
  {
    case 0:
    {
      return costSegmentSize();
    } break;
    case 1:
    {
      return costPrecsSatisfied();
    } break;
    default:
    {
      return costSegmentSize();
    } break;
  }
}

inline void HeuristicAgent::buildSegment()
//assigns the current segment number
//to all tasks that can be scheduled now
{
  int taskid, row, col, precs_left, i;
  //look in global_store to see if some
  //global precedences have been scheduled
  for(row=0; row < number_tasks; row++)
  {
    //if the task at global_precs[row][col] is zero, then there
    //are no more precs from global_precs[row][col] unscheduled
    int size = global_precs[row].getSize();
    for(col = 0; col < size; col++)
```

```
    {
      taskid = global_precs[row][col];
      if(taskid != EMPTY)
      {
        //if the task id at global_precs[row][col] has
        //been scheduled, remove it from the global_precs matrix
        if((*global_store).isSet(taskid))
          global_precs[row].removeNumber(taskid);
      }
    }
  }

  //now identify all tasks that have neither local
  //nor global precedences left unscheduled
  row = 0;
  while(row < number_tasks)
  {
    //task at row hasn't been scheduled
    if(segment_numbers[row] == EMPTY)
    {
      //any global_precs left?
      int global_precs_left = global_precs[row].getElementCount();
      if(global_precs_left == 0)
      {
        //any local precs left?
        precs_left = 0;
        for(col = 0; precs_left == 0 && col < number_tasks; col++)
        {
          if(local_precs[row][col] == 1)
            precs_left = 1;

          //oddly, else seems to be necessary, otherwise
          //some other else statements aren't executed..
          else
            ;
        }
        if(precs_left == 0) //no local precedence found either
        {
          //update local_precs
          for(int k = 0; k < number_tasks; k++)
          {
            local_precs[k][row] = EMPTY;
          }

          //add task to segment
          segment_numbers[row] = segment_number;
          tasks_added_to_segment = 1;

          //restart the outer while loop, because now tasks
          //that couldn't be scheduled at first may be scheduled
          row = 0;
        }
        else
          row++;
      }
      else
```

```
        row++;
    }
    else
      row++;
  }
}

inline void HeuristicAgent::initialize(int num_tasks, IntArray *task_matrix, BitVector *store)
{
  int i, j, index, k, taskid, lengte;
  number_tasks = num_tasks;
  global_store = store;

  tasks_added_to_segment = 0;
  //the first task are scheduled in segment number one
  segment_number = 1;

  //init taskids
  for(i=0; i < number_tasks; i++)
  {
    taskids[i] = task_matrix[i][0];
  }

  //local precs is a square matrix
  local_precs = new IntArray[number_tasks];
  global_precs = new IntArray[number_tasks];

  //for use in heuristics functions
  num_global_precs = new int[number_tasks];
  total_num_global_precs = 0;
  for(i=0; i < number_tasks; i++)
  {
    num_global_precs[i] = 0;
  }

  //now, fill both global_precs and local_precs
  //outer for: rows in the matrix
  for(i=0; i < number_tasks; i++)
  {
    //precedences are in the 1 to the nth
    //column in task_matrix
    lengte = task_matrix[i].getElementCount();
    for(j = 1; j < lengte; j++)
    {
      //find out if task_matrix[i][j] is a local
      //or a global constraint
      index = -1;
      taskid = task_matrix[i][j];
      for(k = 0; k < number_tasks; k++)
      {
        if(taskid == task_matrix[k][0])
        {
          //task matrix[i][0] is dependent on
          //task matrix[k][0]
          index = k;
          //leave this for-loop
```

```
          break;
        }
      }

      if(index == -1) //global precedence constraint
      {
        global_precs[i].addInt(taskid);
        num_global_precs[i]++;
        total_num_global_precs++;
      }
      else //local precedence constraint
      {
        //i depends on k
        local_precs[i][k] = 1;
      }
    }
  }

}
```

Appendix C

Complexity of Quantified PWFP

In this appendix, we will show that the quantified path with forbidden pairs problem — the $\exists\forall\neg$PWFP problem defined in Chapter 3, Definition 3.4.3 — is Σ_2^p-complete. This proof consists of a reduction from QSAT_2, a quantified version of 3-SAT that is known to be Σ_2^p-complete. This reduction is an adaption of a reduction from 3-SAT to PWFP, presented in [24]. The difference between Szeider's reduction and ours is that we deal with directed graphs.

There are a number of preliminary matters we have to mention before we can present the reduction:

- In Chapter 3, we defined the negated version of quantified PWFP, i.e., asking for a solution such that *no* $s-t$ path exists. To reduce QSAT_2 to quantified PWFP, it is more convenient to use the 'positive' version, i.e., asking whether there *does* exist an $s-t$ path.
- We will rename the set of forbidden pairs in the PWFP instance, to avoid name-clashes with the elements of a QSAT_2 instance. We will represent a quantified PWFP instance as: $(G = (V, E), F = \{F_1, F_2\}, s, t)$, where F is the set of forbidden pairs (instead of C).
- Note that $\exists\forall$PWFP is in Σ_2^p: nondeterministically, guess an exclusive choice X_1 for F_1, and verify whether there exists, regardless of the exclusive choice X_2 for F_2, an $s-t$ path in the graph $G' = (V, E')$, with $E' = (E \setminus F) \cup X_1 \cup X_2$. This verification can be done using an NP-oracle.
- The QSAT_2 problem can be defined as follows: Let Y be a set of boolean variables, $\{Y_1, Y_2\}$ a partition of Y, and $\phi = \{C_1, \ldots, C_n\}$ a collection of clauses, such that each C_i consists of three literals. Does there exist a truth assignment for Y_1, such that for all truth assignments for Y_2, ϕ is satisfiable?

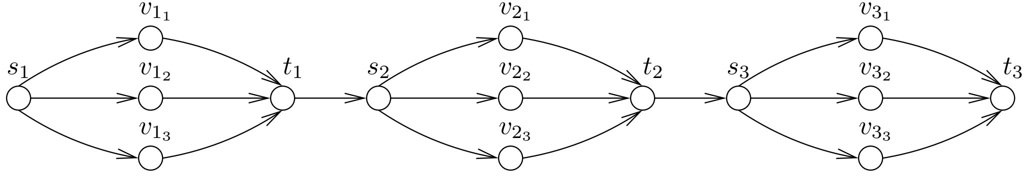


Figure C.1: *The graph G associated with a three-clause $QSAT_2$ instance.*

In transforming QSAT_2 to $\exists\forall$PWFP, we associate a sub-graph G_i with every clause C_i, and then chain these sub-graphs together. The transformation is illustrated in Figure C.1. Specifically, the transformation from QSAT_2 to $\exists\forall$PWFP is specified as follows:

1. For each C_i, we construct the sub-graph $G_i = (V_i, E_i)$, where $V_i = \{s_i, v_{i_1}, v_{i_2}, v_{i_3}, t_i\}$, and $E_i = \bigcup_{j=1}^{3}\{(s_i, v_{i_j}), (v_{i_j}, t_i)\}$.

2. To connect the sub-graphs G_i, we add the arcs $\bigcup_{i=1}^{n-1}(s_i, t_{i+1})$.

3. We choose $s = s_1$, and $t = t_n$

4. A pair of arcs $f = \{(s_i, v_{i_j}), (s'_i, v_{i'_{j'}})\}$ is in F_1, if $x_{i_j} = \neg x_{i'_{j'}}$, and $x_{i_j} \in Y_1$; f is in F_2 if $x_{i_j} = \neg x_{i'_{j'}}$, and $x_{i_j} \in Y_2$.

There is a very intuitive correspondence between a solution for $\exists\forall$PWFP and a solution for QSAT_2: In QSAT_2, at least one literal must be true per clause; in $\exists\forall$PWFP, we must choose one vertex v_{i_j} to reach t_i from s_i in the sub-graph G_i.

Bibliography

[1] Fahiem Bacchus. Aips'00 planning competition (artificial planning and scheduling 2000). *"AI" Magazine*, 2001.

[2] Blai Bonet and Hector Geffner. Heuristic search planner ver. 2.0. *AI Magazine*, pages 77–80, 2001.

[3] Micheal D. Bordo, Alan M. Taylor, and Jeffrey G. Williamson. *Globalization in Historical Perspective*. University of Chicago Press, 2002.

[4] Jeffrey M. Bradshaw. An introduction to software agents. MIT Press, 1997.

[5] K. Decker and V. R. Lesser. Quantitative modeling of complex computational task environmnents. In *Proceedings of the 12th international workshop on distributed artificial intelligence*, pages 67–82, Hidden Valley, Pennsylvania, 1993.

[6] Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence (DAI-94)*, pages 65–84, 1994.

[7] Edmund H. Durfee. Scaling up agent coordination strategies. *Computer*, 2001.

[8] Eithan Ephrati and Jeffrey S. Rosenschein. Multi-agent planning as the process of merging distributed sub-plans. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (DAI-93)*, pages 115–129, May 1993.

[9] Guy Even, Joseph Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2), 1998.

[10] R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.

[11] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—the planning domain definition language, 1998.

[12] Nicholas R. Jennings, Katia Sycara, and Michael Woolridge. A roadmap of agent research and development. Kluwer Academic Publishers, 1998.

[13] H. Moravec. *Mind Children: The Future of Robot and Human Intelligence*. Harvard Univerisity Press, 1992.

[14] Bernard Moret. *The Theory of Computation*. Addison Wesley Longman, 1998.

[15] Hyacinth S. Nwana. Software agents: an overview. In *Knowledge Engineering Review*, 1996.

[16] Doowon Paik, Sudhakar Reddy, and Sartaj Sahni. Vertex splitting in dags and applications to partial scan designs and lossy circuits.

[17] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of encounter: Designing Conventions for Automated Negotiation among Computers*. MIT Press, 1994.

[18] Mohanbir Sawhney. Decouple and conquer. *CIO Magazine*, april 2003.

[19] J. R. Searle. Minds, brains and programs. *Behavioral and Brain Sciences*, 3, 1980.

[20] J. R. Searle. *The Rediscovery of the Mind*. MIT Press, 1992.

[21] P.D. Seymour. Packing directed circuits fractionally. *Combinatorics*, 15, 1995.

[22] Yoav Shoham and Moshe Tennenholtz. On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1–2):231–252, 1995.

[23] L. Steels. Cooperation between distributed agents through self organization. In Y. Demazeau and J.P. Müller, editors, *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, pages 175–196, Amsterdam, The Netherlands, 1990. Elsevier Science Publishers B.V.

[24] Stefan Szeider. Finding paths in graphs avoiding forbidden transitions. *Discrete Applied Mathematics*, 126:261–273, 2003.

[25] Adriaan ter Mors. Coordination mechanisms for autonomous agents: A survey of the multi-agent coordination literature, 2004.

[26] A.M. Turing. Computing machinery and intelligence. *Mind*, 59, 1950.

[27] Jeroen Valk. *Coordination in Multi-Agent Systems*. PhD thesis, Delft University of Technology, 2004.

[28] Jeroen Valk and Cees Witteveen. Multi-agent coordination in planning. In *Seventh Pacific Rim International Conference on Artificial Intelligence*, pages 335–344, Tokyo, Japan, august 2002. Springer.

[29] J.M. Valk, A. Bos, J. Rogier, J.F.M. Tonino, and C. Witteveen. An approximation algorithm for a distributed planning problem. In *Proceedings of the ISCS Conference on Intelligent Systems and Applications (ISA-00)*, pages 419–424. ICSC Academic Press, december 2000.

[30] Frank von Martial. *Coordinating Plans of Autonomous Agents*, volume 610 of *Lecture Notes on Artificial Intelligence*. Springer Verlag, Berlin, 1992.