Solving the Plan Coordination Problem

Master's Thesis

Joris Scharpff

Delft, August 19, 2010

Solving the Plan Coordination Problem

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Joris Scharpff
1263145
`jscharpff@hotmail.com`

Thesis committee

TU Delft
Delft University of Technology

Algorithmics group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Abstract

In this thesis we focus on implicit coordination for multi-agent planning problems. In such problems, agents are not able or willing to cooperate with each other and hence we need to perform *pre-planning* coordination in order to ensure that merging all their plans always results in a feasible joint plan. More specifically, we are interested in finding a minimal cardinality set of constraints such that when add this set to the multi-agent planning problem, no infeasible joint plan can be constructed, whatever local plan each agent develops. Finding such a minimal cardinality set is known as the PLAN COORDINATION PROBLEM (PC) which has been proven to be $\sum_2^p$-complete [44].

Previous work has focussed on approximation and special cases for PC, however some scenarios require or allow for exact solutions. Also, smaller instances might be solvable in reasonable time.

This thesis discusses several exact solving methods and combines them into one exact algorithm that is able to solve instances with task sizes up to 50 *planarcs* within the hour.

Table of Contents

Introduction **1**

I The Plan Coordination Problem 5

1 The Plan Coordination Problem **7**
- 1.1 Multi-agent planning . . . 7
- 1.2 Plan Coordination Framework . . . 9
- 1.3 Problem Definition . . . 16
- 1.4 Coordination Complexity . . . 19

2 Previous Work **25**
- 2.1 Depth Partitioning . . . 26
- 2.2 Intra-free Coordination . . . 29

II Solving the Plan Coordination Problem 33

3 Enumeration **36**
- 3.1 Enumeration algorithm . . . 36
- 3.2 Complexity of Enumeration . . . 43
- 3.3 Enumeration Optimisations . . . 45

4 Dynamic Programming **49**
- 4.1 DP and Plan Coordination . . . 50
- 4.2 DP and Coordination Verification . . . 56
- 4.3 Improving Dynamic Programming . . . 62

5 Kernelisation **65**
- 5.1 In-Out Pairs . . . 66
- 5.2 Task removal . . . 67
- 5.3 Kernel based algorithm . . . 68
- 5.4 Fixed constraints . . . 69

6 QBF Encoding **71**
- 6.1 Phase I: Designing the model . . . 72

6.2 Phase II: Formulating the QBF 75
6.3 Phase III: Transforming to CNF 76

7 Experiments **80**
7.1 Experimental Goals 80
7.2 Experimental Set-up 85
7.3 Results 87

III Discussion **101**

8 Conclusions **103**

9 Future Work **105**

10 Summary **106**

Appendices **110**

A Background Knowledge **110**
A.1 Approximation Classes 110
A.2 Ordered Sets 111
A.3 Polynomial Hierarchy 112
A.4 Student's t-test 113

List of Algorithms **114**

List of Figures **115**

Bibliography **116**

Introduction

Many of today's real life problems consist of complex interactions between different parties, of which each is responsible for only a part of the total set of tasks involved. In addition, these task sets are usually interdependent. Consider for instance the construction of a house in which we have a mason that is solely responsible for construction of all the brick walls. Although the mason can independently lay his bricks, he is dependent on the supply of these bricks by a supplier. This supplier should have delivered enough bricks before the mason starts his work, otherwise delays occur which incur certain costs. But these are not the only two tasks in the construction of a house and this is certainly not the only dependency. Planning all the tasks involved, while still respecting all dependencies, is a very complex problem [43] and is virtually impossible to solve by hand, therefore several algorithms have been developed to tackle this problem centralised [28, 30, 33].

The planning problem as described above is already very hard, but the real challenge arises when we allow the agents to come up with their own schedules and try to combine them into a feasible plan. A *feasible plan* in this context is a plan that never results in deadlock when executed. Based on the level of collaboration between the various agents, we can identify three main strategies for plan coordination mechanisms [40]. When agents are willing to collaborate intensively we let the agents come up with a joint plan simply by communicating about the planning choices made and updating their plans accordingly as in [14]. In other cases, agents are not able to or do not want to communicate about all their planning choices but are willing to modify their plans if required. In such a scenario we let each agent come up with its own plan, resolve all conflicts globally and let agents update their own plans to reflect the changes made globally. An example of this is given in [12].

Both of the approaches mentioned above are based on the assumption that agents are collaborating or at least willing to adapt their individual plans. However in several situations this is not the case, because either the agent is purely self-interested or simply because there are no possibilities to collaborate, as in emergency situations for example [23]. Although this makes planning very hard [44], such situations do occur in practice and hence we need a method to deal with this type of planning problem as well. Because agents do not cooperate we need to prevent infeasible plans from being constructed at all, whatever plan each agent comes up with for its own task set. Thus we still allow each agent to make its planning individually, however we constrain its choices such that any planning that would result in a possible conflict globally can never be produced. We can achieve this by *plan decoupling*.

This research focuses on the latter type of planning problem, in which agents are either unable or not willing to cooperate in constructing a global plan. In this thesis we refer to such agents as *autonomous agents*.

Autonomous agent planning requires a pre-emptive plan decoupling approach, disallowing any infeasible plans to be constructed. In the literature this is referred to as *pre-planning coordination* or *implicit coordination*. Plan decoupling comes down to finding a *coordination set* that prevents planning conflicts and checking whether the instance is indeed *plan coordinated*, i.e. no infeasible plan can be constructed. It turns out that just verifying the plan coordination property is already co-NP-complete and finding a minimal coordination set is $\sum_2^p$-complete [44] (see appendix A.3 for a more detailed explanation on the *polynomial hierarchy*).

The above results seem to offer not much hope of solving plan coordination efficiently, unless P = NP, and so previous work [47, 38] has only focussed on identifying 'easier' special cases or approximation algorithms to come up with a good enough solution in a short amount of time. Nevertheless some practical situations require or allow for exact solutions, no matter what time it takes. Moreover, as modern day computers increasingly gain in computational power, more complex problems such as plan coordination become more interesting to experiment with.

The main purpose of this thesis is to study several exact algorithms for the plan coordination problem and compare them. Through this study we hope to gain insight into the very complex plan coordination problem, which hopefully leads towards an algorithm that is able to solve moderately sized instances (10 to 50 planarcs) within reasonable time. In addition, this research also provides insight into a more difficult class of problems, higher in polynomial hierarchy. These insights might also be beneficial in solving other, perhaps similar, $\sum_2^p$-complete problem.

This thesis is divided in three parts. The first part is meant as an introduction to the plan coordination problem (chapter 1) and it summarises previous work (chapter 2).

The main part of this thesis consists of the exact algorithms we propose. We have studied several algorithmic approaches to tackle the plan coordination problem. In chapter 3 we begin with enumeration, primarily meant as a basis for further research. In chapter 4 we introduce a dynamic programming approach to solve the coordination verification sub problem. We study the possibilities of kernelisation in chapter 5. As a last solving method, we also provide an encoding of the plan coordination problem as a QUANTIFIED BOOLEAN FORMULA PROBLEM in section 6, allowing us to benefit from thoroughly researched and highly optimised solvers. All experiments have been summarised in chapter 7.

The last part is the discussion part in which we look back on the thesis. We discuss the experimental results and draw our conclusions in chapter 8. In chapter 9 we identify interesting leads for future research, resulting from our work. Finally, chapter 10 summarises the entire thesis.

Contribution

The main contribution of this thesis is an exact algorithm for the PLAN COORDINATION PROBLEM that combines all three techniques discussed in this thesis into one algorithm. This algorithm uses enumeration with the enumeration strategy optimisation from chapter 3 to find possible coordination sets. Then a novel dynamic programming approach which relies on the *summary constraints* introduced in chapter 4 is used to verify whether the coordination set indeed allows for autonomous planning. In addition, we can improve on the overall performance of our solver by applying the kernelisation proposed in chapter 5. The result of this is a solver that is able to solve almost all moderately sized instances within the hour.

Part I

The Plan Coordination Problem

Chapter 1

The Plan Coordination Problem

In the introduction we have informally introduced the plan coordination problem in the context of autonomous multi-agent planning. In this problem we want to find a set of additional restrictions for the agents such that they are able to plan their own activities autonomously, i.e. without having to cooperate with other agents. This chapter formalises the problem and using a framework proposed by several authors from the literature [39, 45]. This framework captures all the notions relevant to the (exact) solving of plan coordination problems and provides us a tool to reason about the theoretical aspects of the problem. However, before we can go into the details of coordinating planning problems, we will first need some knowledge about the multi-agent planning problem itself. Although we do not study the actual planning problem in this thesis, it is important to study the problem we want to coordinate and understand the need of coordination.

In section 1.1 we discuss the planning problem that we deal with in this study. Next we introduce a framework to capture the plan coordination problem in section 1.2. Using this framework, we work towards a formal definition of the plan coordination problem in section 1.3. Finally, in section 1.4 we comment on the computational complexity involved in the solving of plan coordination problems.

1.1 Multi-agent planning

Planning, and more specifically classical planning, is one of the best known and well studied PSPACE-complete problems in computer science. The problem is conceptually very simple but computationally very challenging [10]. Basically the problem is about reaching some specified goal state, from some given start state using a limited number of different operations. Many different formalisms of this simple informal definition exist, however the differences are mostly notational. We have adopted the definition from [43] given in definition 1.1.

Definition 1.1: CLASSICAL PLANNING PROBLEM (Kleinberg and Tardos, [43])

Given: *A planning instance consisting of a set of initial constraints C_0, a set of desired goal conditions $\hat{C}$ and a non-empty set of allowed operators O.*

Problem: *Does there exist a sequence of operators $\hat{O} = \{o_1, o_2, \ldots, o_k\} \subseteq O$ such that when we start from initial conditions C_0 and sequentially perform operations $\hat{O}$ we satisfy goal conditions $\hat{C}$?*

When looking at this definition for planning problems, one can intuitively see that finding such a sequence $\hat{O}$ is very complex. This is because of the exponential number of possible sequences that can be generated in the search for the correct sequence for which no intelligent selection procedure can be devised. Hence in the worst case we have to explore the entire exponential size search space to find such a sequence. Classical planning as defined in definition 1.1 is therefore considered intractable.

In this thesis we will study a different, less complex type of planning problem known as task-based planning. In task based planning we are given some complex task, comprised of several elementary or atomic sub tasks which must all be performed in order to complete the complex task. Task based planning as we study here only requires us to come up with a valid ordering of all these sub tasks, which can be found within polynomial time using for instance the topological ordering algorithm from [24].

Although we are not interested in the complexity of the the planning problem each agent faces individually, we do require the task assignment to be known in advance. This is why we study task-based planning as the underlying problem in this research. In addition, the concepts of task-based planning can be naturally extended to a framework for the plan coordination problem as we will do in section 1.2. However, the plan coordination problem itself is not dependent on the underlying planning problem and can be adapted to work with any type of planning problem, assuming that the task assignment is known in advance.

In definition 1.2 we introduce the task-based planning problem more formally.

Definition 1.2: TASK-BASED PLANNING PROBLEM

Given: *A complex task $\mathcal{T}$ consisting of the set $T = \{t_1, t_2, \ldots, t_i\}$ of elementary sub tasks that need to be performed in order to complete the complex task, a set of precedence constraints $\prec$ and a set of equality constraints $\equiv$.*

Problem: *Find a strict partial ordering of the set T such that all constraints in $\prec \cup \equiv$ are satisfied.*

Task based planning can be easily extended to multi-agent scenarios by specifying some task assignment of all the sub tasks in T over the set of participating agents $\mathcal{A}$. We can solve such a multi-agent planning problem by making a topological ordering of all tasks, using for instance the algorithm from [24]. Nevertheless, as the introduction has already noted, finding such an ordering of all tasks requires one central authority to solve the planning problem. In some situations this is not desired or simply impossible and we would like to allow all agents to make their own plans. In other words: we want each agent to solve their sub part of the planning individually. Then we can obtain the joint plan by merging all the plans the agents have come up with. The problem with this, however, is that we do not always end up with a feasible plan this way. Agents are not independent and these dependencies between agents must be considered in solving the planning problem. To allow for independent planning we need some mechanism that coordinates all the agents.

Plan coordination is such a coordination mechanism. Basically it tries to *decouple* all inter-agent dependencies by making them explicit in each agent's own planning problem. It does so by finding additional constraints that, when adhered to, guarantee the validity of the joint plan. This way agents are able to plan autonomously without having to cooperate with other agents. This type of coordination is known as *implicit* or *pre-planning* coordination.

In this thesis we will from now on only focus on the coordination of task-based planning problems and no longer on the planning problem itself. Planning is performed by each agent individually, using their own preferred planning tool, after imposing the coordination set found by solving the plan coordination problem.

1.2 Plan Coordination Framework

The previous section has introduced us to the task-based planning problem, in which we try to find a valid ordering of all elementary sub tasks that adheres to the given set of constraints. This type of planning problem is easily solvable when we agree on having one central authority that finds such an ordering. However when faced with multiple agents that do not wish to or are not able to cooperate, we need a *coordination mechanism* to make sure we always obtain a feasible joint plan. Solving the plan coordination problem is such a mechanism to decouple agents in way that they no longer have to cooperate. In the plan coordination problem we identify possible causes for infeasibility and 'coordinate' these by imposing additional constraints on the planning problem. This set of additional constraints then prevents any possible infeasible joint plan from being constructed at all, i.e. we can only obtain valid joint plans whatever the planning choices of each agent. However to make any such claims, we need a formalism that allows us to prove such properties.

Several authors [39, 40, 45] have proposed frameworks for the plan coordination problem. These frameworks are based on the formalism of the underlying task-based planning problem and incorporate new concepts relevant to plan coordination. Note again that although we formalise the plan coordination problem using task-based planning, we can adapt plan coordination to work with any planning problem, assuming the task assignment is known in advance. In this remainder of this section we will deliberate on a combined framework for plan coordination by providing a definition for all of the concepts involved.

Complex task

The complex task is the main part of the multi-agent planning problem[1] we wish to coordinate. As we have said before, a complex task consists of a set of elementary sub tasks, of which every sub task needs to be performed in order to complete the complex task. These elementary tasks (definition 1.3) represent exactly one unit of work and must be performed by exactly one agent. Which agent is responsible for what set of sub tasks is defined by the task assignment. In this thesis we assume that the task assignment is provided in advance, although this does not hold for planning problems in general. Indeed finding such a task distribution can be a very challenging problem in itself [27].

Definition 1.3: Elementary Task
An elementary task *is a simple task that:*

(i). must be executed exactly once
(ii). must be executed by exactly one agent

These elementary tasks are the work units that make up the complex task and they all need to be completed in order to complete it. This seems to be a very trivial problem: we can simply pick a random sub task, perform it and mark it complete until we have no more uncompleted tasks. Indeed this is true if the sub tasks are *independent*, but in practice we seldom encounter a planning problem instance without task dependencies. In most planning instances certain relations between sub tasks exist in the form of qualitative temporal constraints such as `before`, `overlaps`, `during`, `meets`, `starts`, `finishes` and `equals` [2]. These constraints capture most of the dependencies we encounter in planning problems and have the convenient property that they can all be written using only *precedence* and *equality* constraints [13]. This allows for a compact and comprehensive notation of the task dependencies in the form of these two types of constraints.

In complex tasks we express the relations between elementary sub tasks hence as combinations of precedence and equality constraints. Both constraints are defined in definition 1.4.

[1]We sometimes omit the term multi-agent from now on, however keep in mind that the plan coordination is only applicable to multi-agent planning problems.

Definition 1.4: Precedence and Equality
Let t_i and t_j be two distinct elementary sub tasks of some complex task, then we denote the precedence *relation 't_i must be completed before t_j begins' by*

$$t_i \prec t_j$$

Furthermore we denote the equality *relation 't_i and t_j must both begin and end at the same time' by*

$$t_i \equiv t_j$$

Together the set of elementary sub tasks and the sets of precedence and equality constraints make up the complex task. We have captured this in definition 1.5. Note that we have also included the notion of triviality in this definition. Of course if we would have a complex task consisting of only one task or without any task dependencies, we do not have to coordinate anything and hence the associated plan coordination problem is trivial to solve. This is also true if all the sub tasks of the complex task can be completed by a single agent. In this thesis we will omit the notion of triviality from now on and assume we are dealing with non-trivial complex tasks.

Definition 1.5: Complex Task
A non-trivial complex task *$\mathcal{T}$ is a task consisting of at least two sub tasks, which are interdependent and cannot be completed by one single agent, a set of precedence constraints and a set of equality constraints. We denote a complex task as the tuple $\mathcal{T} = \langle T, \prec, \equiv \rangle$ in which $T = \{t_1, t_2, \ldots, t_i\}$ denotes the set of elementary subtasks, $\prec$ the set of precedence constraints and $\equiv$ the set of equality constraints.*

Both the precedence and equality constraint sets are binary relations containing only sub tasks of $\mathcal{T}$, i.e. $\prec, \equiv \subseteq (T \times T)$. In the precedence constraint set we represent precedence relations in the form $t_i \prec t_j$ as ordered pairs (t_i, t_j). For the equality constraint set we also use a pair (t_i, t_j) to denote the constraint, however the equality relation is of course commutative as $t_i \equiv t_j \Leftrightarrow t_j \equiv t_i$. In one complex task we must have that both sets are disjoint, i.e. no pair (t_i, t_j) occurs in both sets otherwise we can never satisfy the set of constraints.

Because both the precedence and the equality sets have been specified as a part of the instance description, we often refer to these sets as the sets of *original constraints*.

Plans

In task-based planning we are given some complex task, as defined above, and we are to find some feasible plan for it. Such a feasible plan is simply a strict partial ordering[2] of all elementary sub tasks of the complex task. This strict partial ordering is enforced by additional constraints, captured in an *abstract plan*. More specifically, these additional constraints are included in the 'refinement' sets $\prec^*$ and $\equiv^*$. These sets are known as 'refinement' sets because they, by adding constraints, refine the original constraint sets to impose a strict partial ordering. Hence we must have $\prec \subseteq \prec^*$ and $\equiv \subseteq \equiv^*$ and all elementary sub tasks of the complex tasks must be included in at least one constraint. Definition 1.6 formally defines the notion of an abstract plan.

Definition 1.6: Abstract Plan
An abstract plan *P for some complex task $\mathcal{T}$ imposes a strict partial ordering on the elementary sub tasks of $\mathcal{T}$. We represent P by a tuple $\langle T, \prec^*, \equiv^* \rangle$ in which T is the set of elementary sub tasks and $\prec^*$ and $\equiv^*$ denote the precedence and equality refinement sets respectively. These sets are said to 'refine' the original constraint sets, because $\prec \subseteq \prec^*$ and $\equiv \subseteq \equiv^*$.*

The abstract plan, sometimes known as *extension* or *refinement* [44], is an extension of the original problem that partially orders the set of sub tasks. We call this plan abstract because it consists only of qualitative temporal constraints, which only capture sub task dependencies and no actual time values. We are solely interested in how the sub tasks relate to each other, not at what time they are actually performed. The latter is captured in a concrete plan or schedule, however in this thesis we are not interested in such quantitative information.

Agents

Up to now we have considered all concepts of the task-based planning problem mostly from a single agent perspective. In this thesis, however, we are studying the plan coordination problem which is solely applicable to multi-agent planning problems. If we would have only one agent, there would not be much need for coordination. Therefore we extend the previous concepts in the context of multi-agent planning.

In multi-agent planning problems we are also dealing with a complex task, consisting of a set of elementary sub tasks and two constraints sets, for which we want to find an abstract plan. In addition we have the set of agents and some task assignment function $f : T \rightarrow \mathcal{A}$ that associates each task with an agent. In this thesis we assume that such a task assignment is already known in advance, for this is the case in various real life problems. In many cases we already know which agent is responsible for which set of tasks, but we simply want to know in what order to execute the tasks. Nonetheless there do exist problems in which the task assignment is not known in advance; for instance when agents are equally capable of performing some sub task, we could decide

[2]See appendix A.2 for a recapitulation on set ordering theory.

to assign tasks during the planning in order to optimise the resulting plan.

Based on the task assignment we have been given, we can partition the task set T of the complex task such that for each agent we have a set of sub tasks for which it is responsible. Let $\mathcal{A}$ denote the set of agents involved in the problem and f some given task distribution function, then for each agent $A_i \in \mathcal{A}$ we have the task set $T_i = \{t_i \mid f(t_i) = i\}$. For the task assignment to be correct we must have that $\bigcup_{i=1}^{n} T_i = T$ and $\bigcap_{i=1}^{n} T_i = \emptyset$. Note that because of definition 1.5 we must have that $T_i \subset T$ for all T_i, otherwise the complex task is trivial.

Using the task partition defined above we can also partition the constraint sets $\prec$ and $\equiv$. Moreover, we distinguish between *intra-agent* and *inter-agent* constraints. Intra-agent constraints are constraints that consist of two different tasks belonging to the same agent, while the inter-agent constraints denote dependencies between two task belonging to two different agents. Note that we can have only one inter-agent set and as much intra-agent sets as there are agents for each type of constraint.

Definition 1.7: Intra-agent and Inter-agent Constraints

The intra-agent *constraint sets $\prec_i$ and $\equiv_i$ for agent A_i in a complex task $\mathcal{T} = \langle T, \prec, \equiv \rangle$ are given by*

$$\begin{aligned}\prec_i &= \{(t_a, t_b) \in \prec \mid t_a, t_b \in T_i\}\\ \equiv_i &= \{(t_a, t_b) \in \equiv \mid t_a, t_b \in T_i\}\end{aligned}$$

The inter-agent *sets $\prec_{inter}$ and $\equiv_{inter}$ are defined by*

$$\begin{aligned}\prec_{inter} &= \{(t_a, t_b) \in \prec \mid t_a \in T_i, t_b \in T_j, T_i \neq T_j\}\\ \equiv_{inter} &= \{(t_a, t_b) \in \equiv \mid t_a \in T_i, t_b \in T_j, T_i \neq T_j\}\end{aligned}$$

We summarise the above into a representation for the agents involved in the planning problem in definition 1.8. Note that if we would take the union of all agent tuples and the inter-agent constraint sets, we would obtain the original complex task tuple, i.e. $\langle \bigcup_{i=1}^{n} T_i, (\bigcup_{i=1}^{n} \prec_i) \cup \prec_{inter}, (\bigcup_{i=1}^{n} \equiv_i) \cup \equiv_{inter} \rangle = \langle T, \prec, \equiv \rangle = \mathcal{T}$.

Definition 1.8: Agent

An agent *A_i is an autonomous actor responsible for at least one sub task of some complex task $\mathcal{T} = \langle T, \prec, \equiv \rangle$ and is represented by the tuple $\langle T_i, \prec_i, \equiv_i \rangle$. In this tuple the set T_i denotes the set of sub tasks from T assigned to this agent by some task distribution function f. The sets $\prec_i$ and $\equiv_i$ denote the intra-agent constraint sets for this agent (definition 1.7).*

Planning arcs

The notion of planning arcs, or planarcs for short, is simply a convenient term for planning choices that can still be made. When solving their local planning problem, agents can add only arcs between their own tasks. The planarcs represent the degree of freedom each agent still has and also directly relate to the complexity of any instance of the plan coordination problem. The more planarcs any instance has, the more difficult it becomes to coordinate.[3]

In defining the set of planarcs, we only consider the set of planning choices that can actually lead to an inter-agent cycle. The rest of the planning choices are not relevant in solving the plan coordination problem and hence are omitted as planarcs. The only arcs that can ever contribute to an inter-agent cycle have to contain two tasks that are part of two different inter-agent constraints [44]. Other planning choices might still lead to a inter-agent cycle through a longer path, but they are subsumed by just those arcs, see for example figure 1.1. [4]

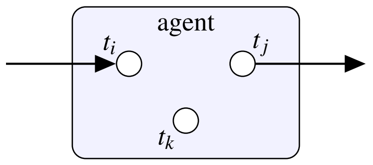


Figure 1.1: We can create an inter-agent cycle by either planning $t_i \prec t_j$ or $t_i \prec t_k$ and $t_k \prec t_j$. Note however that when we add coordination constraint $t_j \prec t_i$ both cycles are automatically coordinated.

In figure 1.1 we see that we can create a cycle in two different ways, assuming that there is some inter-agent path that connects t_j to t_i. Either by adding $t_i \prec t_j$ or by adding both $t_i \prec t_k$ and $t_k \prec t_j$ we would obtain an inter-agent cycle in our precedence graph. We can coordinate this agent by adding the constraint $t_j \prec t_i$ and both cycles are no longer possible. The first cycle is obvious: we cannot have both $t_i \prec t_j$ and $t_j \prec t_i$ in our constraint set. In addition, the second and longer cycle can also never be created because adding it would result in an *intra-agent cycle*, also known as a *local cycle*. Any solution for the agent's local planning problem containing a cycle implies that no partial order exists and hence is an invalid solution by the definition of the problem. Because of all the above, we always have to consider as planarcs only those planning choices that consist of two tasks that are both part of some possibly different inter-agent constraint.

We give the definition of planarcs below in definition 1.9. Note that although we use the term arc, which indicates a certain direction, we are store only pairs of tasks as planarcs. What actual constraint we might add to the coordination set, e.g. $t_i \prec t_j, t_j \prec t_i$ or $t_i \equiv t_j$, is not imposed by the planarc. The name planarc is somewhat misleading, however it has been introduced as such in the literature [44] and hence we will adopt this convention.

[3]This is confirmed in section 7.3, although the number of agents is also an important factor.
[4]See the next sub section for an explanation of this *precedence graph*.

Definition 1.9: Planning arcs
Given an instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ of the planning problem, we define the set of planning arcs *or* planarcs $\hat{E}$ *to be the set of possible planning choices we have to consider to coordinate the instance. This set is given by:*

$$\hat{E} = \{\{t_i, t_j\} \mid \{t_i, t_j\} \notin \prec \cup \equiv,\ f(t_i) = f(t_j),\ \exists t_k, t_l : \{t_i, t_k\}, \{t_j, t_l\} \in \prec_{inter} \cup \equiv_{inter}\}$$

In words, definition 1.9 captures the set of pairs of tasks within the same agent which do not occur together in some precedence or equality constraints, however both tasks are part of an inter-agent constraint.

Precedence Graph

We can visualise an instance of the multi-agent planning using a precedence graph. Such a graph consists of all the agents, their tasks and all dependencies between the tasks. See figure 1.2 for an example dependency graph.

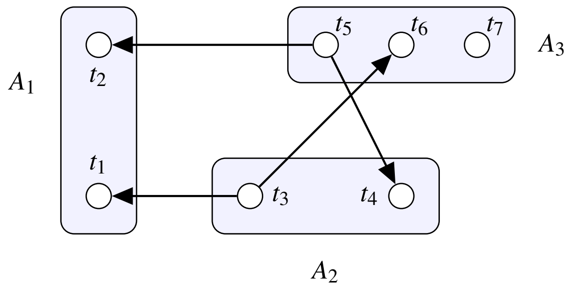


Figure 1.2: An example dependency graph with three agents. The tasks are depicted by white circles. Their relations, i.e. precedence constraints, are depicted by the black arrows. An arrow from t_i to t_j means that t_j depends on t_i or $t_i \prec t_j$.

In this figure we see three agents, illustrated by the rounded rectangles. The tasks assigned to the agents are contained within them and are depicted as white circles. For example we can see that $f(t_1) = A_1$ and $f(t_6) = A_3$. All precedence constraints, both intra- and inter-agent, are displayed as 'depends on' arrows. If some task t_i precedes another task t_j, we say that t_j depends on t_i, hence the direction of the arrow.

For each task we define its in- and out-degree as in definition 1.10.

Definition 1.10: Task In- and Out-degree
The in-degree *of any task t is defined as the total number of different tasks $t' \neq t \in T$ for which $\exists (t' \prec t) \in \prec$. Analogous we define the* out-degree *as the total number of different tasks $t' \neq t \in T$ for which $\exists (t \prec t') \in \prec$.*

Instance Tightness

In the sub sections above we have discussed several planning problem concepts using *tightly coupled* instances introduced below. However other types of instances do also exist. In [48] Zlot and Stentz introduce three types of coordination instances.

Loosely coupled
The first type is the *loosely coupled* instance, in which no *inter-agent* constraints are allowed. The absence of inter-agent constraints immediately makes all the local planning problems independent, because agents have no interdependencies and hence no constraints can be violated when merging all abstract plans. Therefore there is no need to coordinate the instance and there is no plan coordination problem for this type of instance.

Moderately coupled
The second type of planning problem, called *moderately coupled*, does have inter-agent constraints, but allows only for precedence constraints to be used.

Tightly coupled
The third type of planning problem is known as *tightly coupled* and allows for both precedence as well as equality constraints to be used. Although this type of instance may seem more difficult to coordinate in terms of computational complexity than their moderately coupled relatives, Steenhuisen et al. prove that the coordination problem for tightly coupled instances is not substantially more difficult than the one for moderately coupled instances in [38].

In this thesis we will discuss all theory using a framework that is able to represent tightly coupled instances, because a framework for this type of instances is also contains the other two. In our research, however, we will focus on moderately coupled instances because they are conceptually easier and have almost the same coordination complexity. Note that although Ter Mors et al. report that coordination using *simultaneity* constraints is Π_3^p-complete [44], they have a different definition of simultaneity. They define a simultaneity constraint $t_i \bowtie t_j$ to enforce that either $t_i \prec t_j$ or $t_j \prec t_i$ must hold. This is a different type of constraint than the equality constraint $\equiv$ introduced in Steenhuisen et al. [38].

1.3 Problem Definition

In the previous section we have introduced the concepts involved in multi-agent task-based planning. The framework we have discussed up till now is perfectly suited for solving the planning problem in a centralised fashion, i.e one central authority is able to solve this problem using our framework. However, as we have mentioned before, letting one central authority come up with a plan is not always desired or possible. But if we would allow each agent to develop its plan individually, one can imagine that the joint plan will not always be feasible. We need a coordination mechanism to ensure the feasibility of the joint plan. In this thesis we study plan coordination, a pre-planning mechanism that coordinates the multi-agent planning problem by adding additional constraints. These additional constraints, known as the *coordination set*, make sure the joint plan is always feasible, whatever planning choices each agent makes. Still, we have not defined what makes a plan feasible. Definition 1.11 defines the notion of a feasible plan in the context of a joint plan.

Definition 1.11: Feasible and Infeasible Joint Plan
We define the joint plan $\mathcal{P}$ *to be the union of all the abstract plans* $P_1, P_2, \ldots, P_n$ *developed by the agents and the inter-agent constraints, or*

$$\mathcal{P} = \langle \bigcup_{i=1}^{n} T_i, (\bigcup_{i=1}^{n} \prec_i^*) \cup \prec_{inter}, (\bigcup_{i=1}^{n} \equiv_i^*) \cup \equiv_{inter} \rangle$$

We say that a (joint) plan is feasible *if and only if defines a strict partial ordering on its tasks, i.e. the union of the constraint sets is acyclic. A joint plan is said to be* infeasible *if the union of its constraints is contains a cycle.*

From definition 1.11 we see that in order to obtain a feasible joint plan, we need to make sure that the combination of all local plans[5] results in a strict partial ordering of all tasks. The plan coordination problem provides this guarantee by disallowing any infeasible joint plans to be constructed at all. When no such infeasible plan can ever be constructed for a given planning instance, we say that the instance is *plan coordinated* (definition 1.12).

Definition 1.12: Plan Coordinated
We say that an instance of the planning problem is plan coordinated *if and only if no infeasible joint plan (definition 1.11) can be constructed. In other words, whatever abstract plan each agent develops for its own set of tasks, the joint plan again must be strict partially ordered, i.e. an abstract plan.*

The purpose of the plan coordination problem is to, by adding coordination constraints, obtain this plan coordinated property for the given instance. If the plan coordinated property holds, we can safely allow for autonomous planning because no infeasible joint plan can ever be constructed. Therefore we would like to be able to verify whether for any given planning problem instance this property holds and, if not, modify the instance such that it becomes plan coordinated, without changing the semantics of the instance. First we discuss the problem of verifying the plan coordinated property. This problem is known as the COORDINATION VERIFICATION PROBLEM, defined in definition 1.13.

[5]We use the term local plan to denote the plan developed by one individual agent. After plan coordination an agent only has to consider its own set of tasks and constraints in its planning problem, hence the term local.

Definition 1.13: COORDINATION VERIFICATION PROBLEM (CVP)

Given: *A multi-agent planning instance, represented by a complex task $\mathcal{T} = \langle T, \prec, \equiv \rangle$, a set of agents $\mathcal{A}$ and some task distribution function f.*

Problem: *Does the construction of a joint plan $\mathcal{P} = \langle \bigcup_{i=1}^{n} T_i, (\bigcup_{i=1}^{n} \prec_i^*) \cup \prec_{inter}, (\bigcup_{i=1}^{n} \equiv_i^*) \cup \equiv_{inter} \rangle$ always result in a feasible plan, whatever abstract plans the agents come up with? In other words, is the given instance plan coordinated?*

The COORDINATION VERIFICATION PROBLEM or CVP verifies whether any given planning problem instance is plan coordinated. It simply answers with yes or no, based on the constraints in the instance. The actual finding of a constraint set which enforces the plan coordinated property holds is known as the PLAN COORDINATION PROBLEM, which has CVP as a sub problem.

The set of constraints that enforces the plan coordinated property is known as the *coordination set* and is defined in definition 1.14. Note that we denote the multi-agent planning problem instance with the tuple $\langle \mathcal{T}, \mathcal{A}, f \rangle$, consisting of the complex task $\mathcal{T}$, the agent set $\mathcal{A}$ and some task assignment function f.

Definition 1.14: Coordination Set

A coordination set $\Delta = \langle \Delta^{\prec}, \Delta^{\equiv} \rangle$ *for a given instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ of the multi-agent planning problem is a set of constraints that, when added to the planning instance, enforces the plan coordinated property. Or in other words, the resulting instance*

$$\langle T, \prec \cup \Delta^{\prec}, \equiv \cup \Delta^{\equiv} \rangle$$

is plan coordinated.

In the PLAN COORDINATION PROBLEM we are interested in finding a coordination set Δ for the complex task $\mathcal{T} = \langle T, \prec, \equiv \rangle$. Moreover, we are interested in finding the coordination set Δ of minimal cardinality. In our research we define the cardinality for any coordination set as $|\Delta| = |\Delta^{\prec}| + |\Delta^{\equiv}|$. In definition 1.15, we define the problem of finding such a minimal coordination set.

Definition 1.15: PLAN COORDINATION PROBLEM (PC)

Given: *A multi-agent planning instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ and some positive integer K.*

Problem: *Does there exist a coordination set $\Delta = \langle \Delta^{\prec}, \Delta^{\equiv} \rangle$ with $|\Delta| \leq K$ for which the resulting instance $\langle T, \prec \cup \Delta^{\prec}, \equiv \cup \Delta^{\equiv} \rangle$ is plan coordinated?*

The whole point of the PLAN COORDINATION PROBLEM is to find constraints that effectively decouple the planning problem instance into multiple independent planning problems, one for each agent. This allows them to make their own plans individually, without having to communicate with other agents. In addition, we are interested in the coordination set with the smallest cardinality. Although this does not one-to-one translate to the largest degree of planning freedom for all agents, we want to impose a minimum number of additional restrictions on the local planning problems.

1.4 Coordination Complexity

In the introduction we have already mentioned that the PLAN COORDINATION PROBLEM is a $\sum_2^p$-complete problem[6] and therefore highly intractable. In this section we will demonstrate the complexity of both the COORDINATION VERIFICATION PROBLEM and the PLAN COORDINATION PROBLEM by providing a reduction from two known difficult problems, namely the PATH WITH FORBIDDEN PAIRS PROBLEM and the NO PATH WITH FORBIDDEN EXCLUSIVE ARC SETS PROBLEM. Both reductions have been obtained from [44]. Note that Ter Mors et al. only discuss moderately coupled instances (see section 1.2). The reductions are extended to tightly coupled instances in [38]. We will discuss both reductions for moderately coupled instances in the remainder of this section. For a formal proof of both we refer the reader to [45] and [38].

Coordination Verification

The COORDINATION VERIFICATION PROBLEM is the basis for plan coordination, because it verifies whether we have found a correct coordination set that plan coordinates a given planning instance. In [45] Valk proves that for moderately coupled instances the CVP is a co-NP-complete problem. This means that even simply checking whether a given instance is plan coordinated is already very hard. It is not hard to image that this is indeed the case: we have to check that for every possible combination of planning choices the agents make, no infeasible plan can be constructed. If one has some experience with algorithmics, this intuitively sounds like a NP-complete problem. More specifically, because we can easily verify a no certificate for this problem, using a simple cycle detection algorithm, we are dealing with a co-NP-complete problem.

[6]For a refresher on the polynomial hierarchy, see appendix A.3.

The formal proof for the co-NP-completeness of the COORDINATION VERIFICATION PROBLEM can be given by a reduction from the PATH WITH FORBIDDEN PAIRS PROBLEM to the complement of CVP: the COORDINATION FAILURE DETECTION PROBLEM. In definition 1.16 we first introduce the PATH WITH FORBIDDEN PAIRS PROBLEM used by Ter Mors et al. in [44].

Definition 1.16: PATH WITH FORBIDDEN PAIRS PROBLEM (PWFP) (Ter Mors et al., [44])

Given:	*A tuple $\langle G_0, C, s, t\rangle$ in which $G_0 = \langle V, E_0\rangle$ is a directed graph, $C = \{c_1, c_2, \dots c_n\}$ is a set of forbidden pairs of arcs in E_0 and s and t are two distinct vertices from V.*
Problem:	*Does there exist a path from s to t using at most one arc from every $c_j \in C$?*

In words the PWFP problem asks whether there exists a path from s to t such that for each two arcs from one forbidden pair $c_j \in C$ they never occur both in the same path. This problem is a known NP-complete problem, which has been proven by Gabow et al. in [20]. Now to prove that CVP is co-NP-complete, we reduce PWFP to its complement. This problem is known as the COORDINATION FAILURE DETECTION PROBLEM, stated in definition 1.17. The CFD asks whether an infeasible plan can be created given a planning instance.

Definition 1.17: COORDINATION FAILURE DETECTION PROBLEM (CFD)

Given:	*A planning instance $\langle \mathcal{T}, \mathcal{A}, f\rangle$.*
Problem:	*Does there exist any combination of abstract plans $P_1, P_2, \dots, P_n$ developed by the agents $A_1, A_2, \dots A_n \in \mathcal{A}$ respectively such that the joint plan $\mathcal{P}$ is infeasible?*

For the reduction of PWFP to CFD we need some special construction to model forbidden pairs of the first problem as planning choices in the latter. We do this using *path blocking gadgets*, illustrated in figure 1.3.

These path blocking gadgets are constructed for every forbidden pair of PWFP. In the PWFP problem, we can choose to include either zero or one arc from each forbidden pair $c_j = \{(x,y),(u,v)\}$ but not both. The path blocking gadget enforces the same choice in the multi-agent planning problem. When solving the planning problem, we can choose to include (x^j, y^j) or (u^j, v^j) or neither of them. Including both of them would result in an intra-agent cycle (also known as a local cycle) $x^j \prec y^j \prec u^j \prec v^j \prec x^j$ and therefore is not a legal solution to the planning problem.

In his thesis, Valk [45] discusses the complete reduction from a PWFP instance $\langle G_0, C, s, t\rangle$ to a CFD instance $\langle T, G = \langle V, E\rangle\rangle$. Note that Valk uses a slightly different notation for the CFD problem, however we can easily obtain the graph based notation Valk has used from any planning instance $\langle \mathcal{T}, \mathcal{A}, f\rangle$. We model each agent $A_i \in \mathcal{A}$

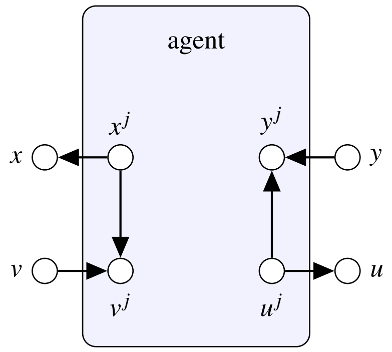


Figure 1.3: A path blocking gadget that models the forbidden pair $\{(x,y),(u,v)\}$ as a planning choice. We can plan either (x^j, y^j) or (u^j, v^j) but not both.

as some shape containing its assigned tasks T_i as a vertices and for each precedence relation $(t_i, t_j) \in \prec$ we add a directed edge from t_i to t_j.

Now that we have overcome the notational differences, we can go into the actual reduction, given below:

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

(ii). (Arcs) E is the smallest set of constraints 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 directed arcs:

$$(x, x^j), (y^j, y), (u, u^j), (v^j, v), (y^j, u^j), (v^j, x^j)$$

which together make up a path blocking gadget.

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

The correctness of this reduction is proven in [45] and will not discussed here, we will only summarise the result of this reduction. Theorem 1.18 is the result of the reduction given before.

Theorem 1.18: PWFP $\leq_P$ CFD
The PATH WITH FORBIDDEN PAIRS PROBLEM *is polynomial time reducible to the* COORDINATION FAILURE DETECTION PROBLEM.

Proof. Valk, [45]. □

Now it remains to prove that CVP is intractable because its complement, CFD, is also intractable. We have done this is theorem 1.19.

Theorem 1.19: co-NP-completeness of CVP
The COORDINATION VERIFICATION PROBLEM *is a co-NP-complete problem.*

Proof. From theorem 1.18 we have that PWFP $\leq_P$ CFD and therefore CFD is at least as hard as PWFP. In [20] PWFP has been proven to be NP-complete and hence so is CFD. We know that CFD is the complement of CVP, or CFD $= \overline{\text{CVP}}$. By definition of the complexity classes we have that for all $\Pi \in$ NP-complete it must hold that $\overline{\Pi} \in$ co-NP-complete, hence CVP is co-NP-complete. □

Plan Coordination

Theorem 1.19 proves that CVP is indeed a co-NP-complete problem and hence intractable. From section 1.3 we know that CVP is a sub problem of PC and therefore we have not much hope of PC being efficiently solvable. We will see that indeed CVP is a very complex problem, much harder than the CVP. Intuitively this seems correct: we can non-deterministically guess some coordination set and then verify it using an algorithm for the co-NP-complete CVP.

In his thesis, Valk [45] proves that the PLAN COORDINATION PROBLEM indeed has a $\sum_2^p$-complete complexity[7] and hence is *highly* intractable. The proof again has only been given for moderately coupled instances and is extended in [38].

The complexity proof for PC requires a reduction from the $\sum_2^p$-complete problem NO PATH WITH FORBIDDEN EXCLUSIVE ARC SETS PROBLEM, defined in definition 1.20,

[7]The polynomial hierarchy theory can be found in appendix A.3.

Definition 1.20: No Path With Forbidden Exclusive Arc Sets problem ($\exists\forall\neg$PWFP) (Ter Mors et al., [44])

Given: *A $\exists\forall\neg$PWFP instance $\langle G_0, C, s, t\rangle$ with $G_0 = \langle V, E_0\rangle$ a directed acyclic graph and some partitioning $\{C_1, C_2\}$ of the forbidden pair set C such that $C_1 \cup C_2 = C$ and $C_1 \cap C_2 = \emptyset$.*

Problem: *Does there exist an exclusive choice set X_1 from C_1, i.e. a set X_1 that contains exactly one arc from each forbidden pair in C_1 such that for every exclusive choice set X_2 from C_2 there does not exist a path from s to t in the arc set $E_0' = (E_0 \setminus C) \cup X_1 \cup X_2$? In this definition $E_0 \setminus C$ denotes the set of arcs from E_0 that do not occur in any forbidden pair.*

The definition of the No Path With Forbidden Exclusive Arc Sets problem seems rather complicated, but can be stated simply in words. We want to find a set of arcs X_1 consisting of one arc of each pair from C_1 such that no set X_2 can be found that still allows a path from s to t to be constructed.

Again we have to model this exclusive arc choice as some planning choice construction in order to reduce $\exists\forall\neg$PWFP to PC. In this reduction we use *forced choice gadget* to model this exclusive choice. The forced choice gadget is depicted in figure 1.4.

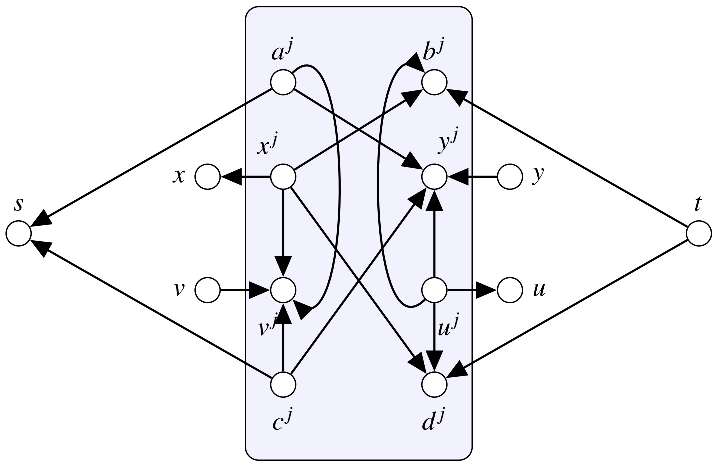


Figure 1.4: A forced choice gadget for the forbidden pair $c_j = \{(x,y),(u,v)\}$ from the set C_1. This gadget enforces that either one or two from $\{(a^j,b^j),(c^j,d^j)\}$ or exactly one arc from $\{(x^j,y^j),(u^j,v^j)\}$ can be chosen when solving the planning problem. Any other choice would introduce an intra-agent cycle which implies that no strict partial ordering exists for the agent's tasks.

The complete reduction from any $\exists\forall\neg$PWFP instance $\langle G_0, C, s, t\rangle$ and some partitioning $\{C_1, C_2\}$ to a PC instance $\langle T, G = \langle V, E\rangle\rangle$ can also be found in [45]. Note again the notational differences, which can be overcome using the method discussed for the CVP reduction.

The reduction can be done as follows:

(i). (Tasks) For every $v_i \in V$, $T_i = \{v_i\}$. For every forbidden pair $\{(x,y),(u,v)\}$ of C, we add nodes x^j, y^j, u^j and v^j to T_{n+j}. Moreover, for every forbidden pair $\{(x,y),(u,v)\} \in C_1$ we add four additional nodes to T_{n+j}: a^j, b^j, c^j and d^j. Finally, T_{n+m+1} (with $m = |C|$) contains the node s_0 and the $K+1$ nodes $t_0, \ldots, t_K$, with $K = |C_1|$.

(ii). (Arcs) The set of arcs E contains the following:

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

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

$$(x,x^j),(y^j,y),(u,u^j),(v^j,v),(y^j,u^j),(v^j,x^j)$$

(c) For every forbidden pair $\{(x,y),(u,v)\} \in C_1$, E contains the 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)

The result of this reduction is a correct mapping from any ∃∀¬PWFP instance to an PC instance. Theorem 1.21 summarises this result, of which the proof can be found in [45].

Theorem 1.21: ∃∀¬PWFP $\leq_P$ PC

The NO PATH WITH FORBIDDEN EXCLUSIVE ARC SETS PROBLEM *is polynomial time reducible to the* PLAN COORDINATION PROBLEM.

Proof. Valk, [45] □

From this theorem we regretfully have to conclude that the PLAN COORDINATION PROBLEM is a $\sum_2^p$-complete problem and therefore highly intractable, unless P = NP. Solving instances of this type of complexity scales very badly, much worse than the known intractable NP-complete problems. Because of this, all previous work in the field of PC solving has focussed on approximation methods. In the next chapter we will discuss two of these approximation methods, Depth Partitioning and Intra-free coordination. We will see that using these algorithms we are able to solve the problem in polynomial time, although the quality (i.e. coordination set size) of the solution might be far from optimal.

Chapter 2

Previous Work

In the literature there have been several authors who studied the problem of autonomous multi-agent planning in different ways. We can identify three main strategies to coordinate multi-agent planning from the research, based on the extent that agents are willing (or able) to collaborate.

In situations where agents are cooperative we can coordinate the joint plan by communicating each planning choice that is made. Then agents can either modify their plans accordingly or respond that the proposed planning action is accepted or infeasible. Such an approach has been taken in for instance [14].

A second strategy is conflict based coordination, used in for example [12]. In this method we allow agents to make their own plans and we try to merge all these plans into one joint plan. This joint plan will then most likely contain one or more infeasible parts and such conflicts need to be resolved. At this point we need some communication and negotiation procedure that informs agents about conflicts so that they can come up with an updated solution to deal with the specified conflict. This approach requires less collaboration, however agents must be willing to modify their plans to make the joint plan feasible. Moreover, conflict driven coordination might require an extensive amount of iterations before all planning conflicts are resolved.

The third approach, which we study in this thesis, is a non-collaborative approach. We want to decouple the multi-agent planning problem such that agents are able to autonomously plan their own part of the joint plan, without having to communicate about their planning choices. This coordination approach is known as *pre-planning* coordination and has been studied by various authors in, amongst others, [8, 40, 44, 45].

Although much research has been performed into the PLAN COORDINATION PROBLEM, there has been no attempt to solve it *exact*. In [38], Steenhuisen et al. propose an approximation algorithm, known as the Depth Partitioning algorithm, that uses the precedence and equality constraint depths to coordinate the PC. Other research, such as the study by Yadati et al. [47], considers a special case only for which the PC is ‘only’ NP-complete and its CVP can be solved in polynomial time.

With PC being an Σ_2^p-complete problem, approximation seems the more promising approach in solving PC. Nevertheless, because of the complexity of the problem, we do not have much hope for efficient approximation. Indeed Ter Mors et al. have proven that PC is an APX-hard problem and therefore it is very unlikely that a constant ratio approximation algorithm exists, unless $P = NP$.[1] Still, both these approaches aim to solve the PLAN COORDINATION PROBLEM and have been the basis of this thesis research, hence we discuss both methods in more detail in this chapter. Section 2.1 explains the Depth Partitioning algorithm by Steenhuisen et al. In section 2.2 we cover the Intra-Free coordination approach by Yadati et al.

2.1 Depth Partitioning

Depth Partitioning is a very intuitive approach to coordinating PC instances and is based on the prerequisite levels of each task. We have seen in chapter 1 that a coordination set must ensure that the joint plan is always feasible or, in other words, planning the tasks will never result in an inter-agent cycle. Depth Partitioning ensures this by making precedence levels, known as task depths, explicit in the form of coordination constraints. In [39] Steenhuisen et al. discuss Depth Partitioning for moderately coupled instances and they extend their algorithm for tightly coupled instances in [38].

The Depth Partitioning algorithm makes sure that plans stay acyclic by enforcing a set based order of tasks. Because of the definition of PC, we must have an acyclic constraint set and hence there must exist at least one task that has no other task preceding it. Now if we make sure in our planning that all such tasks are executed first, the set of tasks preceded by one task will become prerequisite free. Then, if we make sure that these tasks are the next to be executed, the set of tasks with two preceding tasks becomes prerequisite free. The Depth Partitioning algorithm continues this procedure until finally the last tasks that do not precede any other task are coordinated.

Basically the Depth Partition method makes sure that tasks from different depth levels are planned sequentially. The notion of depth is formalised in definition 2.1. Note that after the Depth Partitioning algorithm has been performed agents are still required to plan tasks within each task depth set. Adding constraints between two different depth sets would have no result when we add some constraint $t_i \prec t_j$ with $depth(t_i) < depth(t_j)$, because this is already implied by the Depth Partitioning set. On the other hand, adding a constraint $t_i \prec t_j$ with $depth(t_i) > depth(t_j)$ would result in an intra-agent cycle and hence is not allowed.

Definition 2.1: Task Depth

The task depth *of a task t is defined as the greater of the number of tasks that precede task t and the depth of equal tasks. It is defined by the recursive formula*

$$depth(t) = \begin{cases} 0 & \textit{if no task precedes } t \\ \max\{1 + \max\{depth(t_i)\}, & \\ \qquad \max\{depth(t_j)\}\} & \forall (t_i, t) \in \prec,\ \forall (t_j, t) \in \equiv \end{cases}$$

[1] Approximation classes and approximation ratio are summarised in appendix A.1.

Using the definition we can partition all tasks into depth sets T^d in which d denotes the depth. Computing task depth is easily performed by a depth first search of the precedence graph. After having found all task depths we want to partition these tasks per agent. Hence we make partition sets T_i^d which contain all those tasks of depth d that are assigned to agent i. Note that such a set could be empty because there might not exist a task of such depth for the specified agent. For each agent $A_i \in \mathcal{A}$ we order the partitions as $\{T_i^0, T_i^1, \ldots, T_i^d\}$ and remove the empty sets from this ordering. Then for each neighbouring pair in this ordering we add coordination constraint making this partitions explicit. Also, we enforce planning all equality constraints in the same depth partition by adding equality constraints to the coordination set. This procedure is summarised in algorithm 2.1. For a proof on the correctness of this algorithm we refer the reader to [38].

Algorithm 2.1 Depth Partitioning algorithm

Require: A CVP instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$.
Ensure: The algorithm returns a coordination set for the instance.

```
depth-partitioning ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$ ):
    1. Create agent depth partitions $T_i^d = \{t \mid t \in T_i, depth(t) = d\}$.
    2. For each agent $A_i \in \mathcal{A}$ create ordering $\hat{T}_i = \{T_i^0, T_i^1, \ldots, T_i^d\}$ and remove empty sets.
    3. For each pair $(t_i, t_j) \in \hat{T}_i^j \times \hat{T}_i^{j+1}$ and $(t_i, t_j) \notin \prec$, with $j = 1, 2, \ldots, |\hat{T} - 1|$ being the index of the set $\hat{T}_i^j$ in $\hat{T}_i$, add constraint $t_i \prec t_j$ to $\Delta^{\prec}$.
    4. For each inter-agent constraint $(t_i, t_j) \in \equiv$ we add $t_i \equiv t'$ to $\Delta^{\equiv}$ for all $t'$ with $f(t') = f(t_i)$ and $depth(t_i) = depth(t')$. For $t_j$ we add constraints in a similar fashion.
    5. Return $\Delta^{\prec} \cup \Delta^{\equiv}$
```

The algorithm given in algorithm 2.1 is a polynomial approximation algorithm for the PLAN COORDINATION PROBLEM, however we have no guarantees on the solution quality. Indeed we can show by worst case analysis that on a specific type of instance the Depth Partitioning algorithm produces a coordination set of size $|\hat{E}|$ while the minimal coordination set consists of only one constraint. This instance class is shown in figure 2.1.

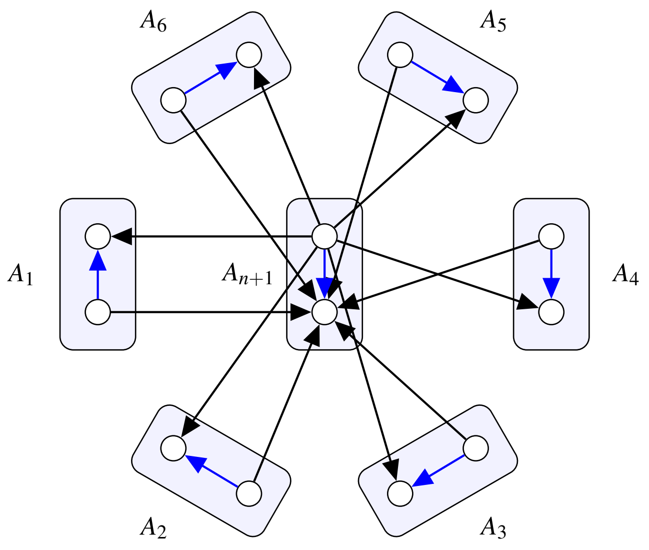


Figure 2.1: A class of PC instances on which the Depth Partitioning algorithm performs very poor compared to the optimal solution. Only the constraint in agent A_{n+1} is required to coordinate this instance, however because of the depths of the other tasks the Depth Partitioning algorithm adds all constraints depicted in blue.

2.2 Intra-free Coordination

Intra-free coordination, as discussed by Yadati et al. in [47], solves a special case of the PLAN COORDINATION PROBLEM identified by Ter Mors et al. in [44]. In intra-free coordination we only consider instances that do not contain any intra-agent constraints. Such instances are much easier to verify, because we do not have to worry about local cycles. We only have to make a dependency graph of the agents involved in the problem. Verifying that the instance is plan coordinated then comes down to checking whether the dependency graph contains a cycle or not. One can imagine that checking whether a graph contains a cycle is computationally much easier than verifying that no cycle can ever exist. Indeed, we can perform cycle detection in polynomial time using for instance the topological sort algorithm from [24]. Moreover, because the COORDINATION VERIFICATION PROBLEM can be solved in polynomial time for this class of instances, the complexity of coordinating intra-free instances is reduced to NP-complete.

Intra-free coordination instances are often encountered in logistic scenarios, in for example supply-chain management [11]. In these type of planning problems agents such as corporations, employees or other resources are only interested in the flow of goods and information *between* the agents. They do not impose restrictions on their own activities.

Still, although having a practical application, the problem of coordinating intra-free instances remains NP-complete and therefore intractable. Verifying the plan coordinated property for any given instance only requires a cycle detection on the agent dependency graph; for coordination, however, we need to identify which agents cause such a cycle and coordinate them. Hence we want to find the minimum set of nodes in the agent dependency graph that, when removed, makes the graph acyclic. This problem is known as the DIRECTED FEEDBACK VERTEX SET, see definition 2.2, and is a well known NP-complete problem [26].

Definition 2.2: DIRECTED FEEDBACK VERTEX SET (DFVS) (Karp, [26])

Given: *A directed graph $G = (V,E)$ and some positive integer K.*

Problem: *Does there exist a set $F \subseteq V$ of size K such that F contains exactly one node from every unique directed cycle in G?*

In [47], Yadati et al. propose an approximation algorithm that tackles intra-free instances by reducing the instance to an DIRECTED FEEDBACK VERTEX SET instance and solve them using a well known approximation algorithm by Even et al. [19]. Having found the minimum set of agents that need te be 'removed' from the agent dependency graph to make it acyclic, we coordinate these agents such that they cannot be part of any inter-agent cycle. To see how this algorithm approximates the PC, we first introduce intra-free instances in definition 2.3. Note that Yadati et al. define intra-free instances only for the moderately coupled case is discussed in [47].

Definition 2.3: (Strict) Intra-Free Instances (Moderately Couples)
Given an instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ of the PLAN COORDINATION PROBLEM*, we say that it is* intra-free *if and only if it does not contain any intra-agent constraint.*

In addition, we say that an intra-free instance is strict *if we can for each agent separate its tasks in two disjoint sets $T_i = In(T_i) \cup Out(T_i)$ such that $In(T_i)$ contains all nodes with zero out degree and $Out(T_i)$ contains all nodes with zero in degree (see definition 1.10).*

From the definition of strict intra-free instances we can see how to coordinate such a problem. If an agent A_i is part of a cycle, e.g. it is part of the set that the DFVS algorithm returned, then we can make sure no inter-agent cycle can be constructed by letting all tasks in $Out(T_i)$ precede all tasks in $In(T_i)$. Observe that we must have strict intra-free instances to coordinate the instance as such. Nonetheless, in [47] a reduction from intra-free instances to strict intra-free instances is also introduced. In this reduction each violating task, i.e. with both in degree and out degree more than zero, is split up into two tasks. One of these tasks has in degree 0 while the other has out degree 0. Continuously applying this splitting of tasks will eventually result in a strict intra-free instance. For more details see [47].

Coordinating strict intra-free instances requires construction of an agent dependency graph. This graph captures the precedence relations between various agents imposed by inter-agent constraints. The agent dependency graph can be looked upon as an abstraction or meta graph of the precedence graph. Formally the agent dependency graph for moderately coupled instances is given by definition 2.4.

Definition 2.4: Agent Dependency Graph (ADG)
Given an instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ of the PLAN COORDINATION PROBLEM*, we construct the* agent dependency graph *as follows:*

1. *For each $A_i \in \mathcal{A}$ we construct vertex A_i.*
2. *If there exists any $t_i \prec t_j$ with $t_i \in A_i$, $t_j \in A_j$ and $A_i \neq A_2$ we add a constraint $A_1 \prec A_2$.*

As we have mentioned before, we can transform this ADG to an instance of the DIRECTED FEEDBACK VERTEX SET to determine what agents should be coordinated. Then we coordinate each of these agents by making sure all sources precede all sinks within the agent. Hence the coordination set for the PC instance consist of all possible source-sink pairs for each agent returned by the DFVS algorithm. For a proof on the correctness of this coordination set we refer the reader to [47], however it is fairly easy to see that the found coordination set is indeed a valid solution.

Using all observations we have made above we introduce the intra-free approximation algorithm by Yadati et al. in algorithm 2.2. Note that the algorithm makes use of an DFVS approximation sub routine to compute the set of agents that need to be coordinated.

Algorithm 2.2 Intra-Free Approximation algorithm

Require: A CVP instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$.
Ensure: The algorithm returns a coordination set for the instance.

```
intrafree ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$ ):

  1. Transform to strict intra-free if required

  2. Construct the ADG for this instance

  3. $F = \text{DFVS}(ADG)$ for each $A_i \in F$ :

       for each $t_{in} \in In(T_i)$ :
            for each $t_{out} \in Out(T_i)$ :
                 $\Delta = \Delta \cup (t_{out} \prec t_{in})$

  4. Return $\Delta$
```

Part II

Solving the Plan Coordination Problem

The previous chapters have introduced the PLAN COORDINATION PROBLEM using a framework that captures all aspects of it. Also we have seen in section 1.4 that PC is highly intractable and previous work has therefore mainly focussed on approximating the solution (section 2.1) or a special case of the problem (section 2.2). However in practice there exist some situations in which we are interested in exact solutions for the PC. Although PC may be a $\sum_2^p$-complete problem, some instances are still solvable within reasonable time. For example smaller instances might lend themselves perfectly for exact solving.

In addition, in solving other $\sum_2^p$-complete problems exactly, such as the solving of the QUANTIFIED BOOLEAN FORMULA PROBLEM (QBF), some very promising results have been made [4, 31]. Hence there is some hope in solving PC exactly, especially if we manage to exploit the problem structure to a great extent. This way we may be able to quickly identify a large part of the solution set, only having to actually solve a small part of the original problem. This technique is known as *kernelisation* and is discussed in chapter 5.

Some instances, on the other hand, might not be reduced to smaller kernels, but there might simply be the need or desire for exact solving. In some situations we are not really interested in how much time it takes to come up with a good solution, as long as one is found within 'reasonable time'. In such cases, exact solving might be worthwhile.

This part discusses the exact solving methods we have applied on the PLAN COORDINATION PROBLEM. All theory and experiments have been performed on moderately coupled instances (see section 1.2) because moderately coupled instances are conceptually easier and their coordination problem is at least as hard at the one for tightly coupled instances [38]. As we have discussed in section 1.2, the equality constraints $\equiv$ introduced by Steenhuisen et al. are easier to deal with than the simultaneity constraints introduced by Ter Mors et al. in [44]. If we would consider the latter type of constraints, the complexity of PC would increases another step in the polynomial hierarchy to Π_3^p.

In our experiments, we started with a simple exact algorithm based on enumeration to get a feel of the problem and its difficulties. This enumeration technique and simple optimisations for it are discussed in Chapter 3. Next, in chapter 4, we focus on on Dynamic Programming, a more advanced technique for exact solving. In chapter 5 we investigate possible kernelisation of the problem so as to reduce the size of the exponential search space we have to explore. The last exact solving method we study in chapter 6 is encoding of the PC as a *Quantified Boolean Formula*, which can be solved by current state of the art solvers such as sKiZZo [6] or Qube++ [22]. This QBF encoding allows us to compare the competitiveness of our algorithms against already existing solver for a $\sum_2^p$-complete problem.

All the experiments and their results are included in chapter 7.

Chapter 3

Enumeration

Enumeration is one of the simplest techniques in exact solving. Basically, in enumeration we simply generate all possibilities and then verify whether the current possibility is the best so far. If so, the solution will be stored as the new best and the algorithm continues. Finally, after all configurations have been generated and verified, the algorithm concludes that the last stored best solution is indeed the optimal solution and it is returned as such.

Because of its conceptual simplicity and ease of implementation, we have decided to start with enumeration as an exact solving technique. The advantage of this is that we can quickly develop a correct algorithm for the PC. From there on, we can identify the difficulties in solving PC instances and try and improve the algorithm to deal with those more efficiently. On the downside, the enumeration strategy (unless optimised) in effect searches the entire search space of the problem. Nonetheless it provides a good basis for further research. Indeed, as we will see later on, many ideas have originated from our simple enumeration algorithm.

This chapter discusses the enumeration algorithm and its optimisations we have developed for the PLAN COORDINATION PROBLEM and its sub problem COORDINATION VERIFICATION PROBLEM. We start by introducing the basic enumeration algorithm in section 3.1 and we will go into the complexity of this algorithm in terms of the number of configurations we have to inspect in section 3.2. In section 3.3 we will outline several optimisations we have found and the impact of those on the run time of the algorithm.

3.1 Enumeration algorithm

The first implementation of this thesis research is a simple enumeration algorithm for both the PLAN COORDINATION PROBLEM and the COORDINATION VERIFICATION PROBLEM. In subsequent chapters we will introduce more advanced techniques to solve both problems, however we have started with a simple recursive enumeration procedure for both problems. In this section we will discuss both the PC and the CVP algorithm and also the complexity involved in solving these problems by enumeration.

PC algorithm

As we have mentioned before, the enumeration algorithm is conceptually very easy to understand and also simple to implement. The algorithm only has to be able to generate all possible configurations and then verify each of them. The basic algorithm we have implemented for PC is given in pseudo code in algorithm 3.1.

Algorithm 3.1 Enumeration algorithm for the PLAN COORDINATION PROBLEM.

Require: A PC instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$.
Ensure: The algorithm returns the minimal coordination set Δ_{min}.

```
pcpenum ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$ ):

  1. $\Delta_{min} = \emptyset$
  2. Generate the set of planarcs $\hat{E}$ from $\langle \mathcal{T}, \mathcal{A}, f \rangle$
  3. $enumerate(\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E}, \Delta)$
  4. Return $\Delta_{min}$

enumerate ( $\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E}, \Delta$ ) ):

  1. if $\hat{E} = \emptyset$:
       if CVP($\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E}, \Delta$) returns COORDINATED and $|\Delta| < |\Delta_{min}|$:
           $\Delta_{min} = \Delta$
       else
           Return
  2. Let $e = (t_i, t_j)$ be the first planarc from $\hat{E}$.
  3. $enumerate(\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta)$
  4. $enumerate(\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta \cup (t_i \prec t_i))$
  5. $enumerate(\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta \cup (t_j \prec t_i))$
```

In this algorithm we define $|\Delta_{min}| = \infty$ for $\Delta_{min} = \emptyset$.

The algorithm starts by setting the best found solution to $\emptyset$, indicating that no solution has been found yet. Then we use some sub routine that generates the set of planarcs from the problem instance, using definition 1.9. With all the required information we start the enumeration process. Note that we use the set of planarcs to enumerate over by considering three options for each planarc (t_i, t_j): not in the coordination set, in the set as $t_i \prec t_j$ or in the set as $t_j \prec t_i$.

This recursive enumeration routine first checks whether there are still arcs to add to the coordination set. If this is not the case, the current coordination set is verified using some sub routine that solves the CVP. We will discuss the CVP sub routine in more detail later on in this section. If the verification returns true, then the instance is plan coordinated and we store the coordination set if it is the smallest seen so far. Note that we define $|\emptyset| = \infty$ so that the solution set is correctly updated when the first solution is found.

When the planarcs set $\hat{E}$ is not yet empty, we do not perform verification but recurse on all three possibilities for the current planarc $e = (t_i, t_j)$. Either we chose to include the constraint $t_i \prec t_j$, the constraint $t_j \prec t_i$ or no constraint at all for the current planarc e. The enumeration algorithm indeed recurses on all three possible choices for the current planarc.

From algorithm 3.1 we quickly see that we are dealing with a recursive algorithm that has a fan-out of 3. Thus we exactly generate $3^{|\hat{E}|}$ configurations and we have to solve an equal number of CVP problems. This is of course not very scalable, as the CVP is already NP-complete, however keep in mind that this algorithm is a very naive implementation. A lot of optimisation can be done to reduce the size of the search space that is actually explored, which we will discuss in section 3.3.

CVP algorithm

The enumeration algorithm for the COORDINATION VERIFICATION PROBLEM is very similar to the one for PC in structure. However, it also requires actual verification of the coordination set. While the PC algorithm simply makes a call to the CVP and stores the result, the CVP algorithm has to incorporate a verification algorithm to check the coordination set generated by the PC algorithm.

Remember from section 1.3, more specifically definition 1.11 and definition 1.12, that in order for any instance to be plan coordinated, the construction of an infeasible plan is impossible. The CVP algorithm uses the converse of this definition to verify this property. When an infeasible joint plan is encountered during the enumeration process of possible joint plans, the algorithm returns NOT COORDINATED. Finally when it has checked all configurations of joint plans and no infeasible plan has been encountered, it is able to conclude that this instance is plan coordinated and it returns COORDINATED.

The detection of infeasible joint plans is based on definition 1.6 and definition 1.11. For a plan to be feasible, it must be an abstract plan. Or in other words, it must define some strict partial ordering on the set of tasks specified by the instance. By the definition of strict partial ordering, such a set must be acyclic. Again we use the converse of this to detect infeasibility of a plan: if during the enumeration some joint plan is generated that contains a cycle, the plan is not partially ordered and hence not an abstract plan. Therefore we can immediately conclude that there exists a joint plan that is infeasible and hence the CVP algorithm can safely return NOT COORDINATED.

The above may suggest that a simple polynomial cycle detection algorithm might suffice in solving the CVP. We could make a graph of the instance (like we did in section 1.4), add for each planarc (t_i,t_j) both constraints $t_i \prec t_j$ and $t_j \prec t_i$ and perform a simple Depth First Search (DFS) cycle detection which disallows going backwards on the same edge (otherwise we would find a cycle $t_i \prec t_j \prec t_i$ for each planarc we added). Using a counter that is increased every time we traverse an inter-agent constraint, we can distinguish intra-agent from inter-agent cycles. We need to distinguish these because intra-agent cycles can, by definition of the planning problem, never be created when solving the planning problem and should therefore not cause the plan to be considered infeasible. Inter-agent cycles should be the only cycle to indicate infeasibility.

The problem with the simple cycle detection is that it does not account for inter-agent cycles that contain intra-agent cycles as well. When the cycle detection algorithm suggested above encounters any inter-agent cycle, it returns NOT COORDINATED. Sometimes this results in an instance being falsely declared as NOT COORDINATED, as for example in figure 3.1.

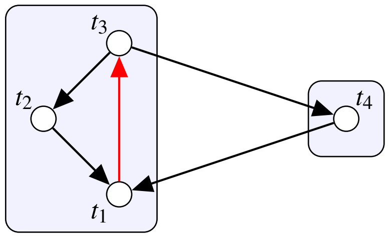


Figure 3.1: This CVP instance contains a inter-agent cycle that consists of an intra-agent cycle as well. The simple cycle detection algorithm would for this reason falsely declare this instance NOT COORDINATED. Nonetheless, this inter-agent cycle can never be constructed, because the intra-agent cycle is forbidden as a planning choice in the planning problem and can never be added.

In figure 3.1, the red constraint arc indicates the possible abstract plan of the leftmost agent. Indeed we now have a inter-agent cycle, namely $t_1 \prec t_3 \prec t_4 \prec t_1$. We also have an intra-agent cycle $t_1 \prec t_3 \prec t_2 \prec t_1$. Although the instance does have an inter-agent cycle, this scenario can never occur because it contains an intra-agent cycle forbidden by the planning problem itself. The cycle detection algorithm is not able to tackle this problem adequately and regretfully we are forced to implement an exponential algorithm. This exponential enumeration algorithm for CVP is given in algorithm 3.2. Note that we use $\hat{E} \setminus \Delta$ to denote the set of planarcs that is not already part of the coordination set. This notation is not entirely exact, because we can have for each planarc $(t_i,t_j) \in \hat{E}$ that either (t_i,t_j) or (t_j,t_i) is in the coordination set. In this algorithm the constraint (t_i,t_j) is removed, regardless which of the two possible constraints is in the coordination set.

Algorithm 3.2 is a very simple recursive enumeration algorithm with a mechanism to quickly propagate the finding of a cycle upwards. As with CVP, all possible configurations are generated and tested for the existence of cycles. The moment we find a cycle we know that the instance is not plan coordinated and the algorithm hence propagates NOT COORDINATED. Thus when a cycle is found, the algorithm will not test any more configurations. Concluding that an instance is coordinated, on the other hand, requires all configurations to be tested. Running the CVP algorithm on a plan coordinated instance therefore requires exactly $3^{|\hat{E}\setminus\Delta|}$ configurations to be verified for a given coordination set Δ.

Algorithm 3.2 Enumeration algorithm for the COORDINATION VERIFICATION PROBLEM.

Require: A CVP instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$, the set of planarcs $\hat{E}$ and the current coordination set Δ to verify.

Ensure: The algorithm returns COORDINATED if and only if the instance is plan coordinated. It will return NOT COORDINATED otherwise.

```
CVP ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$, $\hat{E}$, $\Delta$ ):
  1. if $\hat{E} \setminus \Delta = \emptyset$:
       if $containscycle(\langle \mathcal{T}, \mathcal{A}, f \rangle, \Delta)$:
           Return NOT COORDINATED
  2. Let $e = (t_i, t_j)$ be the first planarc from $\hat{E} \setminus \Delta$.
  3. if CVP($\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta$) = NOT COORDINATED or
     CVP($\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta \cup (t_i \prec t_j)$) = NOT COORDINATED or
     CVP($\langle \mathcal{T}, \mathcal{A}, f \rangle, \hat{E} \setminus e, \Delta \cup (t_j \prec t_i)$) = NOT COORDINATED:
       Return NOT COORDINATED
  4. Return COORDINATED
```

In algorithm 3.3 the cycle existence detection algorithm is outlined.

Algorithm 3.3 Cycle existence detection algorithm

```
Require: A CVP instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$ and the set of additional arcs $E$.
Ensure: The algorithm returns true if and only if the set $E$ contains an inter-agent
    cycle and no intra-agent cycle.

containscycle ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$, $E$ ):
    1. Create an array $tag$ of length $|T|$ and initialise values to NOT VISITED.
    2. for each $t \in T$ with $tag[t] = $ NOT VISITED:
        if $intracycle(\langle \mathcal{T}, \mathcal{A}, f \rangle, E, tag, t)$:
            Return $false$
    3. Reset values of $tag$ to NOT VISITED.
    4. for each $t \in T$ with $tag[t] = $ NOT VISITED:
        if $intercycle(\langle \mathcal{T}, \mathcal{A}, f \rangle, E, tag, t)$:
            Return $true$
    5. Return $false$

intracycle ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$, $E$, $tag$, $t$ ):
    1. $tag[t] = $ VISITED
    2. for each $t' \in T$ with $f(t') = f(t)$ and $(t' \prec t) \in E$:
        if $tag[t'] = $ VISITED:
            Return $true$
        else
            if $tag[t'] \neq$ INTRA DEAD END and $intracycle(\langle \mathcal{T}, \mathcal{A}, f \rangle, E, tag, t')$:
                Return $true$
    3. $tag[t] = $ INTRA DEAD END
    4. Return $false$
```

```
intercycle ( $\langle \mathcal{T}, \mathcal{A}, f \rangle$, $E$, $tag$, $e$ ):
  1. $tag[t] = $ Visited
  2. for each $t' \in T$ with $(t' \prec t) \in E$ and $tag[t'] \neq$ INTER DEAD END:
       if $tag[t'] = $ VISITED:
           Return $true$
       else
           if $f(t') \neq f(t)$:
               if $intercycle(\langle \mathcal{T}, \mathcal{A}, f \rangle, E, tag, t')$:
                   Return $true$
  3. $tag[t] = $ INTER DEAD END
  4. Return $false$
```

The algorithm for cycle existence detection first checks for local cycles and if none are found, it looks for inter-agent cycles. If the first is found, we conclude that this is an invalid configuration and return $false$. If the latter is found we know the instance is not plan coordinated and we return $true$. Only after we have checked all configurations and no infeasible plan is encountered we say that the instance indeed is coordinated and we return $false$.

In this algorithm we use an array to tag all tasks during the detection. We need to keep track of all the nodes we have already visited in order to detect possible cycles. In the intra-agent cycle detection it suffices to check whether or not a node is already visited *during this recursion*. We start from some task t and try to visit as many other tasks within the same agent as we can reach. If we visit some task twice, we have found an intra-agent cycle. If no task is traversed twice during the search, we trace back our recursion to mark all visited tasks as dead ends for future searches starting from other tasks. Note that this is correct, for if some task t marked a dead end would have been part of some intra-agent cycle we would have detected this cycle, no matter from which other task t' we reached t.

The inter-agent cycle detection uses a similar strategy to mark parts of the precedence graph (see again section 1.2) that are no longer of interest. Note that the cycle detection does not try to identify what type of cycle we are dealing with, i.e. intra-agent or inter-agent. Because the intra-agent cycle detection has already been performed, we know that no intra-agent cycle exists and hence any cycle we would now detect must be an inter-agent cycle.

3.2 Complexity of Enumeration

The enumeration algorithms for both PC and CVP we have discussed in the previous section are naive implementations that do not exploit problem specific characteristics. When we combine both algorithms we simply generate all possible solutions and verify them by enumerating all possible outcomes. This on the one hand makes enumeration algorithms easy to develop but, on the other hand, not suited for practical use.

We have seen that the enumeration algorithm for PC generates exactly $3^{|\hat{E}|}$ coordination sets, which have to be verified by the CVP algorithm. The CVP algorithm then has to verify all possible configurations of remaining planarcs for each coordination set. Although we also have a fan-out of 3 branches for every planarc in the CVP algorithm, we do not have to verify $3^{|\hat{E}|}$ possibilities in CVP every time. This is because some of these planarcs have been included in the coordination set by the PC algorithm en hence are no longer available as free planarcs to enumerate in the CVP. We need a more accurate analysis to determine the actual total number of instances that are generated and verified in the combined algorithm.

First we need to determine the number of generated coordination sets per coordination set size. Let $n = |\hat{E}|$ and $k = |\Delta|$, then we want to know how many unique configurations of k arcs we can make out of a total of n. For each planarc we can choose to either exclude the arc, include it as $t_i \prec t_j$ or include it as $t_j \prec t_i$. We can model this using the sum over the product of two binomial coefficients that denote our choices for the remaining planarcs.

For each set of size k we can choose 0 planarcs to be included as $t_i \prec t_j$ arcs and k included as $t_j \prec t_i$ arcs, then 1 of the first and $k-1$ of the latter, and so on. If we have included m arcs of the first type, we must choose $k-m$ arcs of the latter type over the $n-m$ remaining possible positions. Equation 3.1 can be used to compute $\Phi(k)$, the number of configurations possible for each set size k.

$$\begin{aligned}\Phi(k) &= \binom{n}{0}\binom{n}{k} + \binom{n}{1}\binom{n}{k-1} + \ldots + \binom{n}{k-1}\binom{n}{1} + \binom{n}{k}\binom{n}{0} \\ &= \sum_{m=0}^{k} \binom{n}{m}\binom{n-m}{k-m} \qquad (3.1)\end{aligned}$$

For each coordination set of size k, we must solve $\Phi(k)$ different CVP instances in which the remaining set of planarcs of size $n-k$ is enumerated. We can construct exactly 3^{n-k} different configurations of these $n-k$ planarcs per coordination set size. This is summarised in equation 3.2.

$$\begin{aligned}\#conf &= \Phi(0)\cdot 3^n + \Phi(1)\cdot 3^{n-1} + \ldots + \Phi(n-1)\cdot 3^{n-(n-1)} + \Phi(n)\cdot 3^{n-n} \\ &= \sum_{k=0}^{n} \Phi(k)\cdot 3^{n-k} \\ &= 5^n \qquad (3.2)\end{aligned}$$

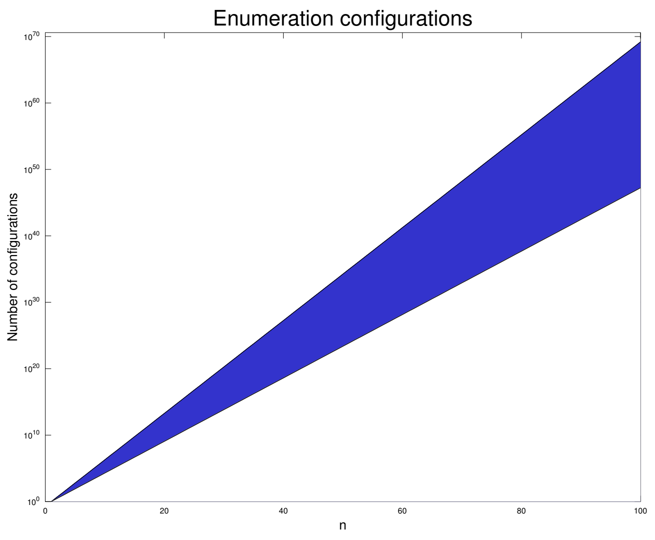


Figure 3.2: The area between the two lines contains the possible number of configurations for each set size k.

The total number of instances that is verified by the combined algorithm is equal to $\sum_{k=0}^{n} \Phi(k) \cdot 3^{n-k}$ which we verified numerically to be equal to 5^n. Keep in mind though that the number of generated and verified coordination sets does not correspond exactly to the time taken to solve the plan coordination. The hardest instances for the CVP to verify are the ones with the smallest coordination set size: because of the large degree of freedom in planarcs, we have to analyse a large number of configurations in order to find a possible cycle. The easiest CVP instances, on the other hand, are the ones with a large coordination set. However, as the coordination sets are binomially distributed over the size, we do not have a lot of such very difficult and easy CVP instances. The bulk of the coordination sets generated have set sizes around $\frac{1}{2}|\hat{E}|$.

Note that 5^n is the worst case number of configurations in the enumeration algorithm. The CVP algorithm uses an upwards propagation method that, when a cycle has been found, concludes this instance can not be plan coordinated and no further CVP configurations for this coordination set are generated. For example consider an instance with 20 planarcs for which we test a coordination set of size 1. It could be the case that, although we can generate $3^{n-k} = 3^{19} = 1,162,261,467$ possible configurations, the first one already contains a cycle. This would save us from checking $1,162,261,467 - 1$ more instances for the current coordination set.

In the best case we would always immediately find a cycle, which requires checking $\sum_{k=0}^{n} \Phi(k) = 3^n$ configurations. This is exactly the number of nodes in the PC search tree, because for each planarc three different possibilities are enumerated. Figure 3.2 shows the possible number of configurations for set sizes k up to 100.

3.3 Enumeration Optimisations

We have seen that simple enumeration of $n = |\hat{E}|$ planarcs requires at least 3^n and at most 5^n configurations. This means that the enumeration algorithm scales very badly in terms of input size: even if we would always have the best case, we still need to solve 3^n instances. This is because we always enumerate all possible coordination sets in the PC enumeration part. In this section we introduce some improvements over the naive enumeration algorithm that, in various cases, might greatly increase the performance.

Enumeration strategy

One of the major factors in the enumeration complexity is of course the number of generated coordination sets by the PC enumeration part. Because of its naive implementation, it always generates exactly 3^n configurations which all have to be verified by the CVP algorithm. This is because the simple enumeration algorithm can only conclude that some coordination set is optimal once it has verified that no better set exists. To verify this, it has to check *all* possible coordination sets.

We have made two simple observations about coordination set optimality that allows the algorithm to discard large parts of the search space. Remember that we are interested in the minimal coordination set for the PLAN COORDINATION PROBLEM in terms of set cardinality, i.e. we are looking for the smallest set Δ that still coordinates the PC instance. We can use this to greatly reduce the search space of the enumeration algorithm. As soon as we find some coordination set Δ that does coordinate the instance, we can immediately discard all coordination sets with a size larger than $|\Delta|$. These sets can never be optimal because we have already verified the existence of a smaller set. Moreover we can also discard all other coordination sets with an equal size, because we have already found one (currently) optimal set. Other sets of equal cardinality are also optimal an because we are not interested in what set is returned we do not have to search for other sets.

In addition to the first observation, we also know that when no coordination set of some given size exists, there also does not exist any coordination set of a smaller size. This statement is easy to prove although not trivial for our research, see theorem 3.1.

Theorem 3.1: Coordination Set Existence

If for any given PC *instance there does not exist any coordination set of size $k \leq |\hat{E}|$, then no coordination set with size $m < k$ exists.*

Proof. We prove this by contra positive. Lets assume there does exist some set Δ_m of size $m < k$ that coordinates the given PC instance and no coordination set of size k exists.

To the set Δ_m we can always add either $t_i \prec t_j$ or $t_j \prec t_i$ from one of the remaining planarcs $e \in (\hat{E} \setminus \Delta_m)$ and obtain a new coordination set Δ_{m+1} of size $m+1$. This must be true because, if no cycle could have been created while considering $\Delta_m \cup (\hat{E} \setminus e)$, either the set $\Delta_m \cup (t_i \prec t_j) \cup (\hat{E} \setminus \Delta_m \cup e)$ or $\Delta_m \cup (t_j \prec t_i) \cup (\hat{E} \setminus \Delta_m \cup e)$ or both must again be acyclic.

So we can always extend our original set Δ_m of size m to Δ_{m+1} of size $m+1$, until $m = |\hat{E}|$. This way we can always obtain a coordination set of size $|\hat{E}|$ however no coordination set of size $m < k \leq |\hat{E}|$ can exists; a contradiction! □

Based on the two observations above we have implemented three enumeration strategies that exploit them in a different way: binary, decreasing size and increasing size. Both the latter strategies are iterative improvement techniques, as we will see below.

Binary

The binary enumeration strategy reduces the remaining search space in each coordination set size iteration by using both observations. In this strategy we first initialise a lower bound lb to 0 and an upper bound ub to $|\hat{E}|$. Then in each iteration we try to find a coordination set of size $k = \lceil \frac{ub-lb-1}{2} \rceil + lb$. If one such a set exist, we know that the optimal solution can no longer be in the search space half with coordination set sizes equal to or larger than k, therefore we update our upper bound to k. If no such set exists we use theorem 3.1 to know that also no smaller set can exist and we update our lower bound to k. Continuing this way, we can reduce the search space each time we find a coordination set.

Although we have named the strategy binary search, it is not true that it indeed halves the search space in each time as binary search commonly does. In the PC enumeration algorithm, we cannot halve the search space in each step because of the binomial distribution of coordination set configurations. Only the first iteration exactly halves the search space, after that it depends on the set sizes tried and found in the enumeration. If the optimal solution has a set size around $\frac{1}{2}|\hat{E}|$, we still have a large PC search space when using the binary strategy. On the other hand, if the optimal coordination set size is close to 0 or $|\hat{E}|$, the PC search space is greatly reduced. Note that, although the PC search is greatly reduced when the optimal solution size is close to 0, the CVP enumeration algorithm's complexity increases because more possibilities arise for the introduction of cycles.

Decreasing size

The second strategy, named decreasing size enumeration, tries coordination sets of decreasing set sizes. We begin by trying to find a coordination set of size $k = |\hat{E}|$. If we cannot find such a set, we must conclude that no coordination set can exist for this instance and we quit. If we do find a set of size k, we start the next iteration in which we try to find a coordination set of size $k-1$. This process is continued until either $k = 0$ and we know that the instance was already coordinated by itself, or for some size m we can no longer find a coordination set. Because of theorem 3.1 we now know that no coordination set exists with a size equal to or smaller than m and hence we conclude that the last coordination set we found, of size $m+1$, must be optimal.

Increasing size

The third strategy is increasing size enumeration in which we keep increasing the size of possible coordination sets. We start with a coordination set of size $k = 0$ and if the instance is now plan coordinated, we return that no coordination set is required for this instance. Then in the next iteration, we try all coordination sets of size $k+1$. If no generated set coordinates the instance, we increase the set size again. This is continued until either $k = |\hat{E}| + 1$ and we know that no coordination set exists, or we find a coordination set of some size m. The first set that coordinates the instance, automatically is one of the optimal sets and is therefore immediately returned.

These three strategies all browse through the search space in different ways and are hence very effective on some instances, but do not perform well on others. In theory, the decreasing size strategy performs well if the optimal solution has a set size larger than $\frac{1}{2}|\hat{E}|$, otherwise it would have to search through the large number of coordination sets centred around the size $\frac{1}{2}|\hat{E}|$. In general one would expect that the increasing size strategy is most effective when the solution is *very close* to 0. Remember that there are not many different coordination sets with small sizes, but the verification problem is very complex when the coordination set is small. As coordination set size k decreases, the number of distinct CVP configurations 3^{n-k} increases. The binary strategy is likely to be the most effective when the optimal solution size is close to half the number of planarcs.

All these observations on the performance of all the strategies are however also influenced by the solution density. The binary search algorithm is most likely to profit most from an instance with a high solution density, because then it can quickly prove the existence of sets of various sizes. However, when there are not many sets that coordinate the instance in comparison to the number of sets one can generate, the binary strategy can be seen as finding a needle in a haystack: it needs to inspect a very large number of configurations without result.

Solution Minimisation

The idea of solution minimisation is loosely based on the proof of theorem 3.1. We know that the optimal coordination set is subsumed by all other sets that contain the same arcs. Solution minimisation tries to obtain the optimal set from the currently found solution set by removing arcs from it. This way we might be able to find better solutions of smaller set sizes quickly and eliminate large parts of the search tree very fast.

Solution minimisation can be applied every time we find a coordination set for the PC instance. We recurse on the sub sets of the coordination set and try to improve the solution quality in terms of coordination set size. If, by removing some arc from the coordination set, a sub set looses the coordinated property, its recursion branch is abandoned and another arc is recursed on. When the sub set still coordinates the instance, we store this as the best solution so far and we recurse further.

Solution minimisation indeed does commonly eliminate large parts of the search space, however if we would try all possible sub sets of any found solution, it is in itself again an enumeration algorithm. We need to restrict the number of sub sets that the minimisation technique considers. To realise this, we have implemented a minimisation *depth* parameter. The minimisation algorithm will try to find some better solution that has a size at least *depth* less than the coordination set Δ given at the start of the algorithm. Once a branch has resulted in a coordination set of size $|\Delta| - depth$ the algorithm will continue the current branch but will not try another branch. This way we will in the worst case check at most $|\Delta| \times 2^{depth-1}$ sub sets. This worst case only occurs when each branch of the original set has sub sets of size $|\Delta| - depth + 1$ that still coordinate the instance, but none of those sets has a sub set of size $|\Delta| - depth$ that coordinates the instance.

Chapter 4

Dynamic Programming

Dynamic programming, or DP, is a technique that tries to solve a large problem by solving easier sub problems and combining the results. The technique as we use it in algorithmics was introduced by Bellman in [5] and has proven its worth on various computational problems, of which computation of the Fibonacci series [46] and the Knapsack problem [34] are two famous examples.

Dynamic programming is very effective when it is easier to solve multiple sub problems than solving the entire problem at once. Commonly it is implemented as a recursive function that combines the results obtained from computing its sub problems. It seeks to profit from the fact that it is easier to compute m sub problems of size $c^{n/m}$ than solving the entire problem of size c^n.

In addition, dynamic programming can also be used very effectively when there is a large overlap in sub problems and we can efficiently remember the solutions to these sub problems. Overlap in this context means that some sub problems are encountered more than once in the recursion tree. When we are able to store and retrieve the solutions for these sub problems, we only have to compute their solution once. For each equivalent sub problem we simply retrieve the previously computed solution. This *memoization* can prevent a lot of redundant computations. Computation of the Fibonacci series is a prime example of memoization, see [46].

Memoization for NP-complete problems usually has a worst case space complexity that is exponential in the size of the input. Nevertheless, with space being a resource often in plenty, such a space-for-time trade-off is acceptable in most cases. For instance there is an algorithm for the Knapsack problem that uses $O(2^n)$ space with memoization, but does allow for pseudo-polynomial solving [46].

In this chapter we will discuss using dynamic programming in solving the PLAN COORDINATION PROBLEM. As we will see in section 4.1, DP can not be used to solve PC. Nonetheless, section 4.2 discusses a DP approach for CVP that does indeed contribute to exact solving. Moreover we also comment on a possible optimisation that should improve performance of the DP algorithm in section 4.3.

4.1 DP and Plan Coordination

In the introduction of this chapter we have mentioned that dynamic programming is effective when we can recursively split up the problem into smaller, more easily solvable sub problems, solve them and combine the results to form the solution for the larger problem. Also, DP can be very effective when the same sub problems are encountered often in the search and we can reuse the solution computed the first time for all subsequent times. However, as we will see, the PLAN COORDINATION PROBLEM does not allow for dynamic programming solutions more efficient than simple enumeration. Whether we base our recursion on agents or planarcs, we can not guarantee optimality when combining sub problems.

Agent sub problems

The dynamic programming approach requires some method to split up problems into smaller sub problems. The first approach we have taken is based on splitting the set of agents in each iteration, until only one-agent sub problems remain. In order to split up the agent set, we have to find for both halves some way to remove the other half while still preserving its information. To this end we introduce *summary constraints*, given in definition 4.1.

Definition 4.1: Summary Constraint

A summary constraint $t_{out} \precsim t_{in}$ *denotes a possible inter-agent cycle between tasks* t_{in} *and* t_{out}*. A possible cycle is a path P of arcs from* t_{out} *to* t_{in}*, such that* $\forall (p \in P) : p \in (\prec \cup \hat{E})$ *and at least two arcs are inter-agent arcs, i.e.* $|P \cap \prec_{inter}| \geq 2$.

A summary constraint $t_i \precsim t_j$ simply states that an inter-agent path from t_i to t_j through the other half of agents *is possible* and therefore if we are able to create a path from t_j to t_i in this half, the instance is not plan coordinated. Continuously splitting up the problem replacing the other half by summary constraints eventually results in one-agent problems in which we know about all possible inter-agent cycles. Simply put, summary constraints allow the agent to remain aware of its 'context' within the entire problem, although this context is represented in a much more compact way. In each next recursion we do not have to go over the other half again, the summary constraints provide the same information about possible cycles.

In order to split the set of agents $\mathcal{A}$ into two sets $\mathcal{A}_1$ and $\mathcal{A}_2$, we need to check for each task $t_i \in \mathcal{A}_1$ that precedes some other task $t_j \in \mathcal{A}_2$ which tasks $T' \in \mathcal{A}_1$ it can reach with a path through the set $\mathcal{A}_2$. For each of these tasks $t' \in T'$ we then add a summary constraint $t_i \precsim t'$, then we remove the inter-agent constraint $t_i \prec t_j$ and all inter-agent constraints $\exists t_k : t_k \prec t'$. See figure 4.1 for an example of this.

In the figure we have the agent set $\mathcal{A}$ which we want to split in two sets: $\mathcal{A}_1$ and $\mathcal{A}_2$. To this end we have added two summary constraints: $t_1 \precsim t_7$ and $t_2 \precsim t_7$. The first summarises the direct path $t_1 \prec t_3 \prec t_5 \prec t_7$, whilst the second captures a *possible* path that can only exist when the arc $t_4 \prec t_5$ is planned. The dashed inter-agent arcs will be removed after adding the summary constraints. Note that both paths will never lead to a cycle, because task t_7 is not connected to t_1 or t_2. In the next DP recursion, these summary constraints will be treated as new inter-agent constraints between the half containing only A_1 and the half containing only A_2 and because no task in A_1 can be reached with a path through A_2 both will be removed. If a path were possible, these summary constraints would have been replaced as before.

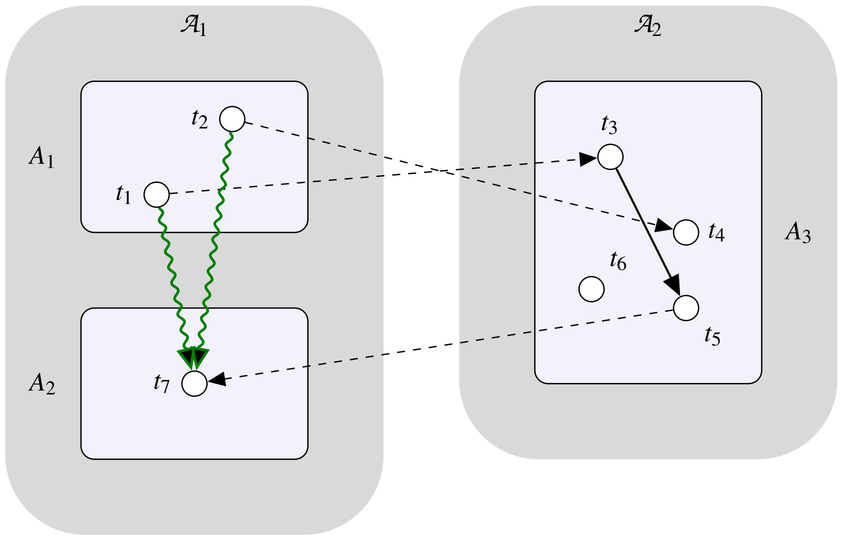


Figure 4.1: An example of summary constraints. The dashed inter-agents constraints depict the constraints that are removed after adding the summary constraints, illustrated by the green lines.

In figure 4.1 we see that each possible inter-agent cycle can be replaced by a summary constraint. A summary constraint between two tasks t_{out} and t_{in} tells the agent that a path starting at t_{out} and ending at t_{in} might be created and hence we can introduce a cycle by connecting t_{out} to t_{in}. To prevent this possible cycle from being constructed, we can decide to add the constraint $t_{out} \prec t_{in}$. This way the cycle can never be constructed and we seem to have a working strategy to coordinate.

Using summary constraints we can indeed implement dynamic programming as a strategy to find coordination sets, however not minimal: consider as an example the instance given in figure 4.2. Using summaries to solve this instance would coordinate the instance, however the result will be a coordination set of size 2 if we halve the agent set in each step of the recurrence, whilst a size of 1 is minimal.

One can see why by going through the recurrence step by step. In the root of our recurrence we have all three agents in the set $\mathcal{A}$. Then we halve this set into two new sub problems, one containing A_1 and A_2, the other only containing A_3. Again we recurse the first one level, because our first set still consists of multiple agents. Now we have three sub problems and its corresponding recurrence tree is given in figure 4.3.

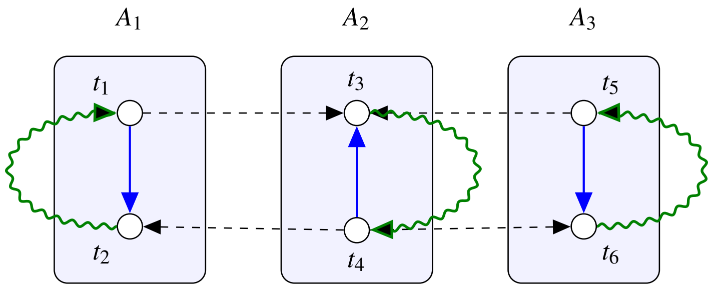


Figure 4.2: The dynamic programming algorithm using summaries would find coordination set $\Delta = \{(t_1,t_2),(t_4,t_3)\}$ or $\Delta = \{(t_4,t_3),(t_5,t_6)\}$ while only the constraint $t_4 \prec t_3$ is sufficient to coordinate this instance because all possible cycles include the arc $t_3 \prec t_4$.

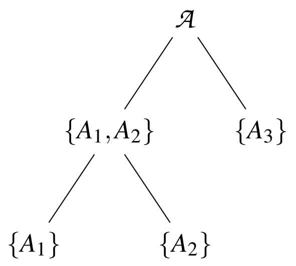


Figure 4.3: The recurrence tree corresponding to the sub problems created in agent based dynamic programming. Each leave represents an actual sub problem of just one agent. All those sub problem solutions have to be combined in each sub tree's root.

Given the sub problems for A_1, A_2 and A_3, we can solve each of them independently, resulting in three coordination sets, each of size 1 (depicted as blue arrows in figure 4.2. Now we want to combine these results and obtain the total coordination set. Hence we recurse back until we have merged all the solutions up to the root. First consider merging the solutions for A_1 and A_2. At this point we can select a coordination set from either sub problem, as they both coordinate the possible cycles present in the joint problem. In the next step we want to merge the joint solution of A_1 and A_2 with the coordination set for A_3, however this depends on what choice we have made for the first merge. We can obtain the optimal coordination set only when we have chosen the coordination set $t_4 \prec t_3$ (see figure 4.2) and not the other. Nonetheless, at the point of merger, both were equally good and so the algorithm has had no incentive to actually choose that one. It might just as well have chosen the other. This is why we cannot guarantee optimality using agent based recursion.

Planarc sub problems

Instead of using the agent set as the base for dynamic programming, we can also create sub problems by considering planarcs. There are several ways to recurse on planarcs, however for dynamic programming to be of any use we need to be able to easily solve sub problems and merge them optimally. In order to guarantee this optimality the sub problems need to be independent so that every root in the search tree is able to find an optimal solution for its own sub tree. Then, by merging all sub trees upwards, we get an optimal solution at the root of the search tree.

We have considered two such dynamic programming approaches that guarantee optimality for the PLAN COORDINATION PROBLEM. Both base their recursive function on the planarcs of the problem, either bottom-up or top-down. The bottom-up algorithm starts with an empty coordination set and tries to find the minimal coordination set by considering all possibilities for each planarc. Its cost function OPT is given in equation 4.1. In this function PVP is used to denote the result of the corresponding CVP instance that has to be solved, i.e. PVP means the instance is plan coordinated while $\neg PVP$ denotes its opposite. The set $\hat{E}$ is the set of all available planarcs, which is used as the iterator in this recursion. The set Δ_{min} is the solution set that is constructed during the recursion and is empty at the start.

$$OPT(\Delta_{min}, e \in \hat{E}) = \begin{cases} \infty, & \text{if } \hat{E} = \emptyset \wedge \neg PVP \\ 0, & \text{if } \hat{E} = \emptyset \wedge PVP \\ \min\{OPT(\Delta_{min}, \hat{E} \setminus e), & \\ 1 + OPT(\Delta_{min} \cup (t_i, t_j), \hat{E} \setminus e), & \\ 1 + OPT(\Delta_{min} \cup (t_j, t_i), \hat{E} \setminus e)\}, & \text{otherwise} \end{cases} \tag{4.1}$$

We can see from the cost function for bottom-up recursion that in fact we are doing the same as in the enumeration algorithm. Indeed this DP approach for PC requires verifying exactly $O(3^n)$ configurations: for each planarc we explore all three possibilities. Still, this algorithm might outperform the enumeration algorithm if we have a lot of overlapping sub problems and we can find some memoization function that enables us to exploits this. However if we again look at the cost function of our recursion we can see that there is no overlap: we generate exactly 3^n unique sub problems. Bottom-up recursion of our planarcs therefore performs as bad as simple enumeration.

The other type of recursion based on planarcs is a top-down approach that starts with a coordination set of size $|\hat{E}|$ and in each recursion tries removing one planarc of the problem. This recursion can be implemented using the function defined in equation 4.2.

$$OPT(\Delta_{min}, c \in \Delta) = \begin{cases} \infty, & \text{if } \Delta = \emptyset \land \neg PVP \\ |\Delta_{min}|, & \text{if } \Delta = \emptyset \land PVP \\ \min\{OPT(\Delta_{min} \setminus c, \Delta \setminus c), & \\ 1 + OPT(\Delta_{min}, \Delta \setminus c)\}, & \text{otherwise} \end{cases} \tag{4.2}$$

Again we denote the solution set as Δ_{min}, however this time it is initialised differently. We would like to start this recursion with the complete coordination set given by $\Delta^* = \{\{t_i, t_j\} \in \hat{E}\}$, however this poses some implementation difficulties. The complete set includes all possible constraints that can be part of the coordination set and therefore has to include an arc for both directions of each planarc of $\hat{E}$. Thus for each $(t_i, t_j) \in \hat{E}$ the set Δ contains both $t_i \prec t_j$ and $t_j \prec t_i$ and the set is therefore cyclic. To overcome this problem we do not generate the complete coordination set as a whole, however we generate all 2^n (see below) possible coordination sets of size $|\hat{E}|$ that are acyclic and evaluate all of them using our recursive cost function. Each evaluation starts with both Δ_{min} and Δ equal to this initial configuration set. We can find the optimal coordination set for the original PC instance by performing a min operation on all evaluations.

Each evaluation starts with two sets including all possible coordination constraints and for each constraint in Δ it checks whether it is better to keep the constraint in the coordination set or remove it from the set. Basically we try to remove as many constraints as possible while preserving the plan coordinated property of the instance. If we do violate it, i.e. CVP returns NOT COORDINATED then we incur a penalty of infinite cost and discontinue this branch. The penalty cost makes sure that the branch is not considered as a possible solution when evaluating the min function. Note that again Δ_{min} keeps track of the solution and we now use Δ to iterate over the constraints of the coordination set we started with.

Using this approach we again have independent sub problems and therefore we can solve the PLAN COORDINATION PROBLEM exact. However, in terms of computational complexity, this technique performs even worse than enumeration. To see this we require a number of observations on the number of configurations we have to verify. First we evaluate this function for every possible configuration of constraints from $|\hat{E}|$ such that the set of constraints is still acyclic. Hence, for each planarc $(t_i, t_j) \in \hat{E}$ we can choose to either include $t_i \prec t_j$ or $t_j \prec t_i$ in our constraint set. Let $n = |\hat{E}|$ then we can make exactly 2^n unique acyclic sets and we have to evaluate our recursive function for all of them.

The recursive function branches on two possibilities for each constraint of the set Δ. Either the constraint is kept in the minimal coordination set Δ_{min}, increasing the set size of the found solution by 1, or the constraint is not included in the minimal set. Hence we have to evaluate $2^{|\Delta|}$ possible configurations and because $|\Delta| = |\hat{E}|$, we must again check 2^n configurations. In total we need to verify $2^n \times 2^n = 4^n$ configurations in order to solve PC using top-down recursion as such and is therefore outperformed by the $O(3^n)$ configurations of the simple enumeration algorithm.

There is one improvement we can make, however, that reduces the number of configurations we have to solve by the CVP algorithm. For the top-down approach we can apply memoization in order to exploit the overlap in the sub problems. To illustrate this overlap, consider two coordination sets $\Delta_1 = \{(t_i,t_j),(t_k,t_l)\}$ and $\Delta_2 = \{(t_i,t_j),(t_l,t_k)\}$, both belonging to different sub problems. During evaluation of the *OPT* function on these coordination sets, they will both try the coordination set $\Delta_{min} = \{(t_i,t_j)\}$ in order to compute the optimum solution. If we are able to store the result of running the CVP algorithm on this coordination set, we do not have to solve the (NP-complete) CVP again on the same coordination set the second time. We can simply reuse the answer we have computed earlier.

Memoization would indeed reduce the number of CVP instances we need to solve, however keep in mind that in our evaluation function we need to check all *unique* configurations at least once. Memoization would reduce the cost of checking each configuration more than once to some polynomial (usually linear) time complexity introduced by looking up the value computed before, but we still need to check all 3^n unique configurations. Therefore memoization would reduce the complexity in terms of CVP instances we have to solve from $O(4^n)$ to $O(3^n)$ but this is again no improvement over the simple enumeration algorithm.

For the PLAN COORDINATION PROBLEM we have not found any DP approach that could perform better than simple enumeration. The agent based approach we have seen in the beginning of this section is not able to guarantee optimality of the found coordination set and is therefore unusable for exact solving. Recursing on planarcs does not have this problem, but both the bottom-up as well as the top-down approach do not offer any better complexities than simple enumeration. Hence dynamic programming is most likely not the right algorithmic technique in solving the PC, although we have not studied all possible ways to implement DP. There might exist a recursive function for the PC for which a dynamic programming algorithm exists that does decrease solving complexity, however based on our research we are sceptic towards application of dynamic programming on PC solving.

4.2 DP and Coordination Verification

The dynamic programming approach does not seem to provide better results than the enumeration algorithm for solving the PLAN COORDINATION PROBLEM exactly. Recursion on the set of agents yields easier sub problems, however the solutions resulting after merging sub problem solutions are not guaranteed to be optimal. Recursion on planarcs does not have this problem but now we have to solve approximately as many CVP instances as with enumeration discussed in chapter 3. Hence DP is most likely not suitable to solve the PC exactly. The COORDINATION VERIFICATION PROBLEM, on the other hand, does allow for a dynamic programming approach that performs better than its enumerative counterpart.

In solving CVP, we are not interested in optimising some specified criteria, we want to know whether *any* cycle can be introduced when merging all individual agent plans. In addition it does not matter exactly which planarcs are involved in any possible cycle. Intuitively this seems to make CVP a lot easier to divide into smaller sub problems than the PC.

In our research we have implemented a dynamic programming algorithm that recurses on the set of agents to construct sub problems. In each recursion we halve the set of agents until only one agent remains. We halve this set using summary constraints (see definition 4.1) to capture inter-set agent dependencies. This recursion is continued until we have sub problems with only one agent. These sub problems are solved independently, resulting in either $true$ if no cycle can be created in this sub problem, or $false$ otherwise. We combine the results in each sub tree root by a simple AND function. If the result in the root node of the search tree is $true$, then we know that the instance is plan coordinated. The DP algorithm for CVP is given in algorithm 4.1.

Algorithm 4.1 Dynamic Programming algorithm for the COORDINATION VERIFICATION PROBLEM.

Require: A CVP instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$, the set of planarcs $\hat{E}$ and the current coordination set Δ to verify.

Ensure: The algorithm returns COORDINATED if and only if the instance is plan coordinated. It will return NOT COORDINATED otherwise.

```
CVP ( $\langle \mathcal{T},\mathcal{A},f \rangle, \hat{E}, \Delta$ ):
    1. $\prec_{inter} = \{c = (t_i,t_j) \mid c \in \prec, f(t_i) \neq f(t_j)\}$
    2. $\prec' = (\prec \cup \Delta) \setminus \prec_{inter}$
    3. $E = \{(t_i,t_j) \in \hat{E} \mid (t_i,t_j) \notin \Delta \wedge (t_j,t_i) \notin \Delta\}$
    4. Return $pvpdp(\mathcal{A}, f, \prec', E, \prec_{inter})$

pvpdp ( $\hat{A}, f, \prec', E, \prec_{inter}$ ):
    1. if $|\hat{A}| > 1$ :
        $A_1 =$ first half of $\hat{A}$
        $A_2 =$ last half of $\hat{A}$
        $\langle E_1, \prec'_1, \precsim_1 \rangle = splitagent(A_1, f, \prec', E, \prec_{inter}$
        $\langle E_2, \prec'_2, \precsim_2 \rangle = splitagent(A_2, f, \prec', E, \prec_{inter}$
        $\prec_{inter} = \prec_{inter} \setminus \{(t_i,t_j) \mid (t_i,t_j) \in \prec_{inter}, f(t_i) \neq f(t_j)\} \cup \precsim_1 \cup \precsim_2$
        if $pvpdp(A_1, f, \prec'_1, E_1, \prec_{inter}) =$ NOT COORDINATED :
            Return NOT COORDINATED
        else
            if $pvpdp(A_2, f, \prec'_2, E_2, \prec_{inter}) =$ NOT COORDINATED :
                Return NOT COORDINATED
        Return COORDINATED
    2. else
        if $E \cup \prec_{inter} \cup \prec'$ contains an inter-agent cycle :
            Return NOT COORDINATED
        else
            Return COORDINATED
```

```
splitagent ( $A, f, \prec', E, \prec_{inter}$ ):
    1. $E_A = \{(t_i, t_j) \in E \mid f(t_i) = A\}$
    2. $\prec'_A = \{(t_i, t_j) \in \prec' \mid f(t_i) = A\}$
    3. $\precsim_A = makesummary(A, \prec', E, \prec_{inter})$
    4. Return $\langle E_A, \prec'_A, \precsim_A \rangle$
```

As we can see in algorithm 4.1 we start by initialising sets required for computations further along the algorithm. First the set of inter-agent constraints is retrieved from the original problem constraint set. In the second step we construct the set of all imposed intra-agent constraints, either by the original constraints or by the coordination set. From this set we remove all inter-agent constraints that were part of the original constraint set $\prec$. Finally we also obtain the set of planarcs that are still available after application of this coordination set. Having done all the necessary initialisations we can now start the dynamic programming recursion.

In this recursion, we halve the agent set and split up the remaining planarc set accordingly in the sub routine *splitagent*. In this sub routine we find for both agent sets the set of imposed constraints, remaining planarcs and summary constraints; we will discuss the latter in more detail in the next sub section. Using the summary constraints we found, we replace all inter-agent constraints between the two sets, i.e. all constraints such that either $t_i \in A_1, t_j \in A_2$ or $t_i \in A_2, t_j \in A_1$, by their summary counterparts. See figure 4.1 in section 4.1 again to recall how these summary constraints are obtained. For each pair of outgoing and incoming inter-agent constraints (definition 4.2) we have exactly one summary constraint. Note that although we can make summary constraints for all pairs of tasks, it suffices to capture possible cycles only involving *planarcs* that are connected to both an incoming and an outgoing constraint.

Definition 4.2: Incoming and Outgoing Inter-Agent Constraints and Tasks

Given two disjunct sets of agents A_1 and A_2. We say that an inter-agent constraint $t_i \prec t_j$ is incoming *for set A_1 iff $t_i \in A_2$ and $t_j \in A_1$. Analogue we have that an inter-agent constraint is said to be* outgoing *for set A_1 iff $t_i \in A_1$ and $t_j \in A_2$. Moreover note that any incoming arc for A_1 is an outgoing arc for A_2 and vice versa.*

We say that a task t is an incoming task *there exists some incoming constraint $t_i \prec t$. Also, a task is said to be an* outgoing task *if there exists some outgoing constraint $t \prec t_i$.*

The last step of algorithm 4.1 requires a cycle detection algorithm, that checks the union of the planarc, imposed constraint and summary constraint sets, all specific to this agent. For this we have used a slight modification of the simple enumeration algorithm of section 3.1. This algorithm does still have an exponential run time complexity, however we have greatly reduced the number of planarcs in this sub problem and hence enumeration finds solutions a lot faster. Of course we will have to solve more problems this way, but each one is a lot easier than solving the entire problem at once. Indeed, as we will see in chapter 7, the total time taken to solve all sub problems is far less than trying to solve the entire CVP instance at once.

Making Summaries

For dynamic programming to be applicable on a problem, we need to be able to recursively decouple problems into smaller sub problems that are easier to solve. As we have seen before, the DP algorithm for CVP does just this and its decoupling is based on summary constraints (see definition 4.1). The summary constraints $t_{out} \precsim t_{in}$ indicates the possibility of a cycle being created if we would plan $t_{in} \prec t_{out}$. Keep in mind that a summary constraint only indicates the *possibility* of a cycle though these tasks. It does not have to be the case that planning $t_{in} \prec t_{out}$ does always result in a cycle: maybe some other planarc on the path is planned such that the cycle can no longer be created.

Still, in the COORDINATION VERIFICATION PROBLEM it suffices to know whether a cycle is possible at all. If this is the case, the PC instance we have been given is not coordinated by our current coordination set because of definition 1.12. Hence if we find a summary constraint $t_{out} \precsim t_{in}$ and we are able to add the constraint $t_{in} \prec t_{out}$ *without introducing a local cycle* then we can create a cycle and the instance is not plan coordinated. As we have discussed in section 3.1, local cycles are disallowed in the planning problem and therefore adding planarcs that introduce such a cycle makes the CVP instance invalid (see figure 3.1 for an illustration of this).

These local cycles are also the most difficult part of the summary creation algorithm, shown in algorithm 4.2. We have to take into account that we can not simply add a planarc in order to detect a possible cycle. Before trying the arc as a part of our path we must first assure that it does not introduce a local cycle.

Algorithm 4.2 Summary creation algorithm for the CVP DP algorithm.

Require: The current agent set $\hat{A}$, the set of constraints $\prec'$ of both agent sets, the set of remaining planarcs E of both agent sets and the set of remaining inter-agent constraints $\prec_{inter}$

Ensure: The algorithm returns a set of summary arcs that contains exactly one summary arc for each pair of incoming and outgoing constraints of agent A.

```
makesummary ( $\hat{A}, \prec', E, \prec_{inter}$ ):
  1. $\precsim_{\hat{A}} = \emptyset$
  2. for each outgoing constraint $(t_i, t_j) \in \prec_{inter}$:
       $T' = \emptyset$
       $\prec_{inter} = \prec_{inter} \setminus (t_i, t_j)$
       $markincoming(\hat{A}, \prec', E, \prec_{inter}, T', t_j)$
       for each $t_{in} \in T'$:
           if $(t_i \precsim t_{in}) \notin \precsim_{\hat{A}}$ :
               $\precsim_{\hat{A}} = \precsim_{\hat{A}} \cup (t_i \precsim t_{in})$
  3. Return $\precsim_{\hat{A}}$
```

```
markincoming ( $\hat{A}, \prec', E, \prec_{inter}, T', t$ ):
  1. mark $t$ visited
  2. for each $(t_i, t_j) \in \prec_{inter}$ with $t_i = t$ :
       if $t_j \in \hat{A}$ :
           $T' = T' \cup t_j$
           $\prec_{inter} = \prec_{inter} \setminus (t_i, t_j)$
       else
           if $t_j$ has not been marked visited :
               $markincoming(\hat{A}, \prec', E, \prec_{inter}, T', t_j)$
  3. for each $(t_i, t_j) \in \prec'$ with $t_i = t$ :
       $markincoming(\hat{A}, \prec', E, \prec_{inter}, T', t_j)$
  4. for each $(t_i, t_j) \in E$ with $(t_i, t_j) \notin \prec'$ and $(t_j, t_i) \notin \prec'$ :
       $t_1 = t$
       if $t_i = t$ :
           $t_2 = t_j$
       else
           $t_2 = t_i$
       if $t_2$ has not been marked visited :
           $\prec' = \prec' \cup (t_1 \prec t_2)$
           if $\prec'$ is acyclic :
               $markincoming(\hat{A}, \prec', E, \prec_{inter}, T', t_2)$
           $\prec' = \prec' \setminus (t_1 \prec t_2)$
```

The summary creation algorithm seems very complex, however it is not much more advanced than a simple depth first search algorithm for directed graphs. Basically the sub routine *makesummary* identifies all outgoing constraints of this agent set and for each of them *markincoming* marks what incoming arcs can be reached by some path through the other set. For each outgoing constraint in $\prec_{inter}$ we find the set of tasks T' in the same agent set that can be reached through inter-agent constraints. Note that we immediately remove the outgoing constraint as it will be replaced by one or more summary constraints after the mark round.

The marking part consists of a recursive depth first search of all reachable tasks by traversing the currently imposed constraints. For each task in the precedence graph we try to find a path in three different ways. First, we check the inter-agent constraint sets to see whether we can reach tasks in other agents by some inter-agent constraint. If a task in another agent indeed is preceded by our current task we either have found an incoming constraint or just an inter-agent constraint in the other halve of $\hat{A}$. In the case we have found an incoming constraint we store the task that is preceded, i.e. the task in the set $\hat{A}$, in the set of reachable tasks T' and remove the incoming constraint from the set $\prec_{inter}$. If the constraint we found is no incoming constraint, we simply continue depth first search from the preceded task.

The second type of arcs on the path of a possible cycle are the constraint arcs imposed by the set $\prec'$. This set contains the original problem constraints and the coordination constraints. These are fixed at the point of running the CVP algorithm (as are the inter-agent constraints) and we traverse them in the same way as inter-agent constraints. Hence if we are currently at task t_i then we recurse our depth first search on each other task t_j for which a pair $(t_i,t_j) \in \prec'$ exists.

Finally we also need to traverse all *possible* constraints in order to make correct summary arcs that indicate possible cycles. The set E contains all these possible pairs of tasks that can still be added as a constraint arc to the current path. Hence for each pair in E we check whether either of the tasks is the current task and we try to add a precedence constraint in which our current task precedes the other. Of course we will only add such a constraint $t_i \prec t_j$ if it is not already on our path and its converse $t_j \prec t_i$ is not either. When we are allowed to add the constraint, we do so and continue the marking from the preceded task.

As we have mentioned before, the depth first search algorithm for marking has to account for local cycles during the search for a possible inter-agent cycle. In algorithm 4.2 we can see in step 4 of the marking routine that we check the set $\prec' \cap (t_i.t_j)$ for cycles before actually continuing our marking process. This small part of the algorithm is very important for the correctness of the summary constraints. It guarantees that the path we have found that may indeed lead to a possible inter-agent cycle does not also introduce a local cycle. The latter type of cycle is forbidden by definition of the planning problem and hence agents can never introduce such a cycle. We have discussed the need for local cycle detection in section 3.1, however figure 4.4 is a simple example that shows how summary creation could be wrong if it would not account for local cycles as well.

4.3 Improving Dynamic Programming

We will see in chapter 7 that the basic DP CVP algorithm we proposed above already outperforms the enumeration algorithm from chapter 3, however we can do even better. In this section we will discuss one improvements from which our CVP implementation will most likely benefit very much.

Data Structure

In our study we have implemented the Dynamic Programming algorithm using a square matrix M to represent constraints between tasks in the instance. Each constraint (t_i,t_j) of our instance is stored at entry $M_{i,j}$ in our constraint matrix. For enumeration it suffices to simply flag entry $M_{i,j}$ when we have a precedence constraint $t_i \prec t_j$, however we need a slight modification to cope with summary constraints. We modify the matrix

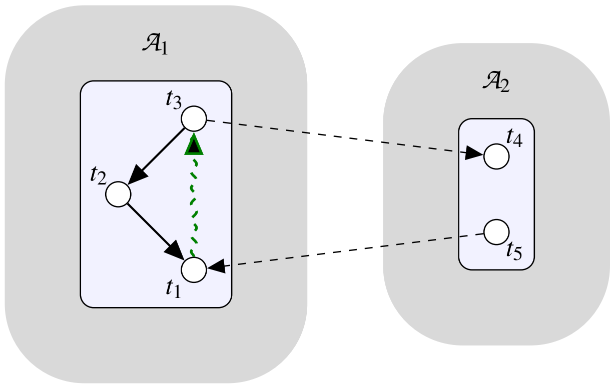


Figure 4.4: An example in which finding summary constraints requires local cycle detection. We want to find the summary arcs for the set $\mathcal{A}_2$, hence we trace the constraint $t_5 \prec t_1$ into the set $\mathcal{A}_1$. If we would trace the arc $t_1 \prec t_3$, depicted in green, we would eventually find that $t_5 \precsim t_4$. However this path can never be constructed, because one of its arcs ($t_1 \prec t_3$) causes a local cycle. Therefore we do not include the summary constraint $t_5 \precsim t_4$. Without local cycle detection, the DP algorithm might incorrectly conclude that a cycle can be created by adding $t_4 \prec t_5$.

to work with bit flags, telling what type or types of constraints are imposed between each pair of tasks.

In terms of computational complexity, the constraint matrix is an efficient data structure for immediate storage and retrieval of constraints, if the task numbers are known. I.e. checking whether task t_i precedes t_j requires merely a matrix retrieval of element $M_{i,j}$ and maybe a bitwise comparison, depending on whether the type of constraint matters. Both these operations, as well as storing at given task numbers, can be done in $O(1)$. However, it has a major drawback: if we want to know what tasks precede some given task t, we need to iterate through an entire column of the matrix. Of course, when the matrix gets bigger, such an operation becomes increasingly costly. Regretfully this operation is very frequently used in both the creation of summaries and the cycle existence check.

Another issue is that it is very complex to divide such a constraint matrix in the DP separation step. It does not allow for easy dividing because of the task numbers we use as the index to this array. Halving constraint matrices results in gaps in the index numbers and hence inefficient iteration. We can compensate this by some mapping function, that maps each halves task numbers to their corresponding matrix entries but this makes it very complex altogether. Nevertheless if we would not split the constraint matrices then we would have to go through the *entire* constraint matrix every time, even if we are solving a one agent sub problem. To exploit the power of Dynamic Programming even more we must use a different data structure and more efficient way of accessing than a constraint matrix.

We propose to replace our constraint matrix by an agent based data structure in which we also make task relations explicit in each task. We store for each agent the tasks it controls, the intra-agent constraints and the planarcs it contains. For each task we store the agent it belongs to, which other tasks it precedes and in which arcs it occurs. Also we have one container for the CVP instance that contains all the agents, the inter-agent constraints and the current coordination set.

Each CVP run starts by adding the coordination set constraints to the precedence lists of the correct tasks. Hence for each constraint $t_i \prec t_j \in \Delta$ we add t_j to the precedes list of t_i. This will enables us to quickly traverse the precedence graph of the instance.

Having added all precedence relations, we begin the recursive division of the problem. In each step we again halve the agents sets by replacing intra-agent constraints with their summary counterparts. If we look back at the summary algorithm in algorithm 4.2, we see that the mark algorithm is a depth first scan of the precedence graph. Using a constraint matrix to perform this would require checking $O(|T|)$ neighbours *each time* we have travelled to a new task. In the worst case we would have to scan $|T|-1$ other tasks in order to find that we do or do not need to add a summary constraint for some pair of tasks. Hence we would search $O(|T|^2)$ tasks to make a summary for just one pair of tasks.

Storing the precedence relations in the tasks enables us to immediately identify all neighbours of the task and we only have to iterate through $\deg(t)$ nodes. Although this can be equal to $|T|$, this very rarely occurs. More commonly the degree of a task is very small compared to the total number of tasks. The same holds for the enumeration algorithm that is used to solve the remaining NP-complete problem of the one agent.

Chapter 5

Kernelisation

In the previous chapters we have seen exact approaches to solve the PLAN COORDINATION PROBLEM. All algorithms we have proposed solve the problem with a run time that is exponential in the number of planarcs. The naive enumeration algorithm for example has a run time complexity of $O(5^n)$ in the worst case in which n denotes the number of possible planarcs. With some improvements we can obtain a best case complexity of $O(3^n)$ (see section 3.3), however this is still an exponential complexity. Assuming that $P \neq NP$, as the vast majority of computer scientists believe to be true, we have no hope of finding an efficient algorithm that solves the PC in polynomial or even sub-exponential time and other ways of reducing complexity have to be studied.

One such an approach is kernelisation, in which we try to find the smallest possible subset of the input that needs to be solved exactly, known as the *kernel*. The key idea is that some part of the input can be solved in polynomial time using a set of predetermined rules and that the remaining part has to be solved using an exponential algorithm. If we can find such a polynomial function, we can reduce the complexity of solving the problem to $O(f(k) \cdot |n|^{O(1)})$, with f being an exponential function and k the size of the kernel, smaller than n. Problems that allow for such an algorithm are known as *Fixed-Parameter Tractable* (FPT) problems (see [18]). One of the most famous examples of a FPT problem that allows for kernelisation is the MINIMUM VERTEX COVER PROBLEM. By finding structures known as crowns we can reduce the size of the part that has to be solved by an exponential algorithm [1].

During our research we have identified several structures in PC instances that we can exploit to reduce the number and possible configurations of planarcs we have to consider. In section 5.1 we prove that we can reduce the number of arcs we have to inspect in solving the CVP. Section 5.2 discusses removal of tasks that are irrelevant for the coordination problem. Then in section 5.3 we introduce an algorithm that uses both these techniques. We will see in chapter 7 that this kernelisation algorithm indeed outperforms our previous algorithms.

In the process of finding kernelisation techniques we also investigated fixed constraints, which indeed reduce solving time significantly but do not always result in an optimal coordination set. Section 5.4 discusses these fixed constraints and illustrates when this technique fails to return an optimal solution.

5.1 In-Out Pairs

In section 1.2 we have introduced the notion of planarcs (see definition 1.9) and there we reasoned that we only need to consider task pairs for which both tasks belong to some inter-agent constraint. This is because all other possible inter-agent cycle paths are dominated by the planarcs containing two tasks also belonging to inter-agent constraints. By dominated we mean that both the creation and prevention of these inter-agent cycle paths can be realised using only such planarcs. See figure 1.1 again for an illustration of this.

We can reduce the set of planarcs we have to consider even more, however only for the COORDINATION VERIFICATION PROBLEM. For any given CVP instance it is sufficient to consider only *in-out pairs*, which we define in definition 5.1.

Definition 5.1: In-Out Pairs
A pair of tasks (t_i, t_j) is called an In-Out pair *if $f(t_i) = f(t_j)$ and there exists at least one inter-agent constraint $t_i \prec t'$ and one inter-agent constraint $t'' \prec t_j$ with $f(t_i) \neq f(t')$ and $f(t'') \neq f(t_j)$.*

We say that we can connect an in-out pair if we can add the constraint $t_{in} \prec t_{out}$ without causing a local cycle.

The main idea behind this kernelisation is that if we cannot connect the task t_{in} to t_{out} directly then we can also not connect t_{in} to t_{out} through some other path. The proof for this is given in theorem 5.2.

Theorem 5.2: Inter-agent cycles and In-Out Pairs
Assuming that $\{t_{in}, t_{out}\}$ is not already in $\prec_i$, if for some agent $A_i = \langle T_i, \prec_i, \equiv_i \rangle$ there exists an inter-agent path P that exits the agent at t_{out} and enters the agent at t_{in} then this path can only be cyclic if we can add $t_{in} \prec t_{out}$ to $\prec_i$ without making the set $\prec_i$ acyclic.

Proof. To see that we can have an inter-agent cycle if we can add $t_{in} \prec t_{out}$ to $\prec_i$ we study both the case in which we (i) can and (ii) cannot add the arc. We need to prove that there exists a path from t_{in} to t_{out} if and only if we are allowed to add $t_{in} \prec t_{out}$ to $\prec_i$.

When we want to add the constraint $t_{in} \prec t_{out}$ we must have that $t_{out} \prec t_{in} \notin \prec_i$, otherwise adding the constraint is not allowed by the definition of the PLAN COORDINATION PROBLEM. To be precise, the case in which $t_{out} \prec t_{in}$ is already in $\prec_i$ is subsumed by the case in which adding $t_{in} \prec t_{out}$ would cause a local cycle (ii).

(i). For the reasons stated above we have $t_{out} \prec t_{in} \notin \prec_i$ and hence we are able to add the constraint. We can now have that adding the constraint either results in a cyclic or an acyclic set. Note that the set $\prec_i$ is acyclic, again by definition.

The trivial case is when adding the constraint $t_{in} \prec t_{out}$ results in an acyclic set $\prec_i$, because now we already have a path from t_{in} to t_{out} and the path $P \cup (t_{in} \prec t_{out})$ is indeed cyclic.

(ii). In the case that adding $t_{in} \prec t_{out}$ would cause $\prec_i$ to become cyclic, there must exist some path P' starting in t_{out} and ending in t_{in} such that $P' \cup (t_{in} \prec t_{out})$ is cyclic. And because such a path P' must exist, we cannot add any path P'' from t_{in} to t_{out} to $\prec_i$. Adding such a path would always cause a cycle $P' \cup P''$ within the set $\prec_i$ and hence we can never add such a path to the inter-agent path P and therefore P can only be acyclic.

□

Using theorem 5.2 we can reduce the complexity of our CVP algorithm. Proving that no cycle can be created comes down to testing for each in-out pair (t_{in}, t_{out}) whether we can connect t_{in} to t_{out} without creating a local cycle. In combination with summaries, this simple observation can increase performance greatly: not only do we have to consider fewer arcs, for each arc we now only have to consider one option. We only have to test whether we can add $t_{in} \prec t_{out}$ and no longer the other way around. Moreover we can already test for cycles during the creation of summary constraints. The moment we have found a summary we can immediately test if we can connect the in-out pair. Applying this notion to the Dynamic Programming algorithm proposed in section 4.2 would reduce the complexity of CVP to almost polynomial; only when we encounter local cycles during summary creation we would have to use a backtracking approach to find all possible summaries (see section 5.3).

5.2 Task removal

In PC instances we usually encounter a sub set of tasks that are either not part of any planarc or their associated planarcs can never lead to any cycle at all. Removing these tasks and their planarcs again might reduce the number of possibilities we have to consider.

We can identify two different tasks that can be removed immediately from any PC instance without harming the finding of an optimal solution. First, because of the definition of planarcs (see definition 1.9), we can remove all tasks that are not part of any inter-agent constraint. Such tasks are never part of any planarc and hence will never be included in any coordination constraints. This is because any coordination path through such a task is dominated by a coordination path staring and ending in two other tasks that are part of an inter-agent constraint. If such a task t has both in degree and out degree greater than zero, we can replace all pairs of in constraints $t_i \prec t$ and out constraints $t \prec t_j$ by a new constraint $t_i \prec t_j$.

Iterative removal of such tasks will eventually result in a PC instance with just those tasks that are part of some inter-agent constraint. Therefore in the resulting instance we must have either a constraint or a planarc for each pair of tasks within the same agent.

Another type of task can be removed from the PC instance with a little more effort. One can imagine that not all parts of the PC instance have a cyclic dependence that must be coordinated. There might exists agents, for instance, which only contain incoming inter-agent agent constraints but no outgoing constraints. Of course we can also have the other way around. In such a case, this agent can never contribute to any inter-agent cycle and we can even remove the entire agent from the instance, including all inter-agent constraints originating from or ending at that agent. Removing an agent results in a decrease of the number of planarcs in the instance.

Identifying such irrelevant parts of the instance can be done using the agent dependency graph (see definition 2.4) as introduced in [47]. We can run a depth first search on the ADG to find agents that cannot be part of any agent cycle and remove them from the instance. Note that we also need to remove the inter-agent constraints that have one of their tasks in the agent. If removing this inter-agent constraint would cause the task in another agent to no longer be part of any inter-agent constraint, we can remove that task as well, as we have discussed previously in this section.

5.3 Kernel based algorithm

We have combined the techniques discussed in this chapter with the dynamic programming algorithm for the COORDINATION VERIFICATION PROBLEM introduced in section 4.2 into a new kernel based algorithm. This algorithm first runs a pre-processing phase in which all irrelevant tasks are removed as we discussed in section 5.2. In the solve phase we have used the PC enumeration and CVP dynamic programming approaches.

The PC enumeration algorithm is based on algorithm 3.1, although we have also implemented the enumeration strategy improvement discussed in section 3.3. This algorithm enumerates all possible configurations of the coordination set, which are then verified by a slightly modified dynamic programming algorithm. The overall idea of the new DP algorithm is similar to algorithm 4.1. It captures possible cycles by making summaries and then verifies whether we can complete the cycle or not. However, we have made some improvements that allow for faster verification of coordination sets.

First of all, using theorem 5.2, we can reduce the number of arcs we have to consider when verifying the coordination set. Inter-agent cycles through any in-out pair can exist if and only if we can connect that in-out pair. Hence we only have to check these in-out pairs, not all the planarcs in $\hat{E}$.

In addition we do not have to verify all possible configurations of these in-out pairs. Again by theorem 5.2 it suffices to check if we can connect those in-out pairs (t_{in}, t_{out}) that also have a summary cycle $t_{out} \precsim t_{in}$. This can be done using a very simple polynomial depth first search that checks whether $t_{in} \prec t_{out}$ causes a local cycle. If this indeed causes a local cycle, we know that the inter-agent cycle is not possible. If we are able to connect t_{in} to t_{out} then we report a cycle possibility and the CVP returns NOT COORDINATED.

During the creation of summary arcs we can already detect possible cycles if both the tasks of the summary constraint belong to the same agent. In this case we have found an in-out pair and we can immediately verify if we can introduce a cycle, i.e.

if we can connect this in-out pair. When we can indeed connect the pair, we know that a cycle is possible and we can return NOT COORDINATED instantly, without having to summarise the entire instance. Note that immediate cycle detection is always performed when halving a set means that only one agent remains in that set. This is because any summary we find for a set consisting of only one agent must have two tasks belonging to the same agent.

Instead of the dynamic programming summary creation algorithm we could also implement a pair-wise cycle existence detection algorithm. This algorithm would check for each in-out pair (t_{in}, t_{out}) whether we can reach t_{in} from t_{out} (i.e. $t_{out} \precsim t_{in}$) and if we can connect the pair. This approach would also work, although this means that for each in-out pair we will have to analyse the entire dependency graph. Moreover, as we have seen in section 4.2, we need to consider all inter-agent paths without local cycles, of which an exponential amount might exist in the dependency graph. Using the pair-wise method we would have to backtrack a lot of possibilities for each in-out pair and hence we prefer the dynamic programming approach which halves the size of this 'hard' part in each iteration.

5.4 Fixed constraints

In our study into kernelisation we also investigated typical substructures of the PC that can be coordinated instantly. Using a pre-processing routine we find such *fixed constraints* in polynomial time, thus reducing the size of the 'hard' part to be solved exactly. Regretfully we have not been able to identify fixed constraints that can be added to the instance while preserving solution optimality.

Nonetheless we have found a substructure in PC instances that can be exploited to reduce solving time significantly while producing near optimal solutions. In some cases it even produces the optimal solution, although much faster than solving the instance exactly. Exploiting this substructure might be useful in situations where we want to produce a good, but not necessarily minimal, solution within a (more) reasonable amount of time.

Our thought was that when we have an in-out pair (t_{in}, t_{out}) for which there already exists an inter-agent path from t_{out} to t_{in} consisting only of original constraints, we can immediately add $t_{out} \prec t_{in}$ to the coordination set. This assumption seems valid, however in some scenarios we can find an even smaller coordination set that implicitly enforces $t_{out} \prec t_{in}$. Consider for instance figure 5.1, a generated PC instance (see section 7.2) on which the fixed constraints approach fails.

In figure 5.1 we see the coordination set that is proposed by the algorithm in blue dashed arrows. The size of this coordination set is 5, while an optimal set of size 4 exists (see figure 5.2). The fixed constraints method fails to make use of local cycles in such a way that a smaller coordination set implicitly also coordinates the fixed constraint arc. To account for this, we would have to generate all other coordination sets and see whether they (1) prevent the arc from being added and (2) are of a smaller cardinality. There are exponentially many of such sets to verify before being able to decide on optimality and hence this pre-processing technique itself poses a NP-complete problem. Therefore it is not of any practical use as a pre-processing step.

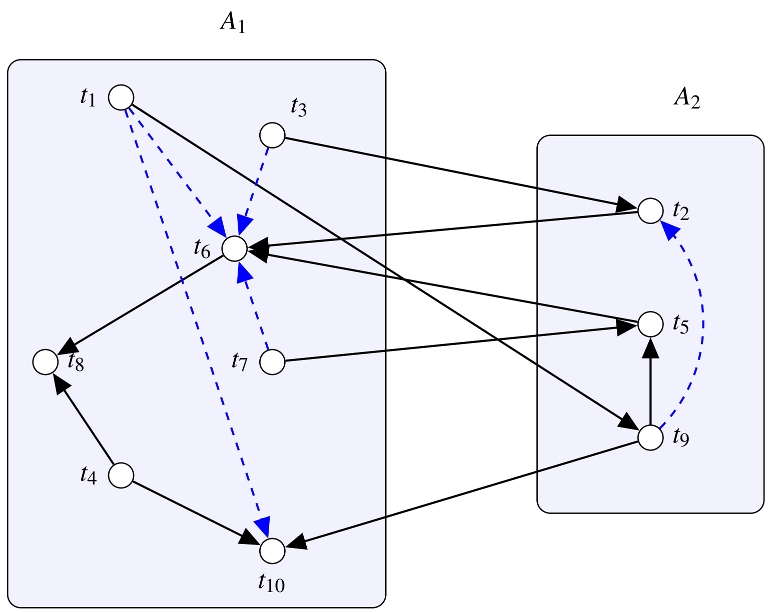


Figure 5.1: The coordination set found by applying the fixed constraint method. The blue dashed arrows depict the proposed coordination set of size 5.

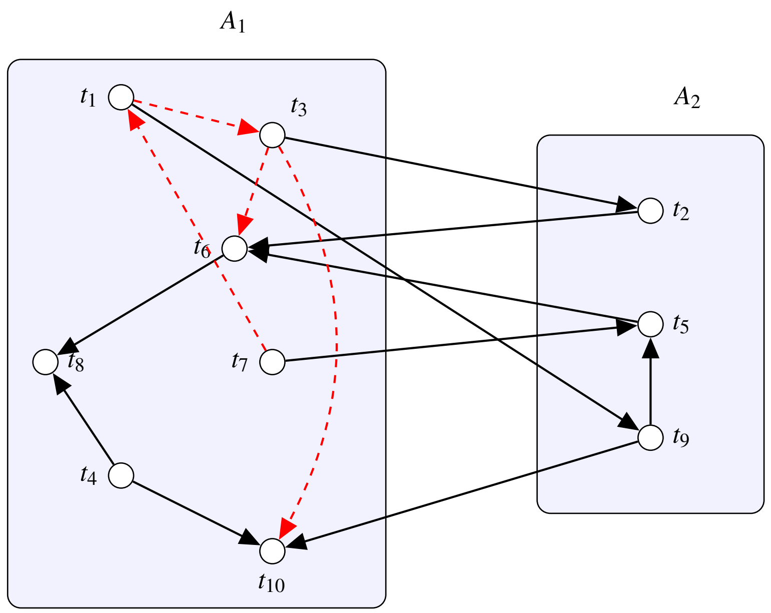


Figure 5.2: The optimal coordination found by enumeration is depicted by red dashed arrows. The size of this set is only 4.

Chapter 6

QBF Encoding

So far there have been no studies yet into solving the PLAN COORDINATION PROBLEM exactly; only approximation algorithms have been proposed for the problem. In order to have some form of comparison to other techniques, we intend to formulate the PC as a Quantified Boolean Formula (QBF) and solve it using a state-of-the-art solver. This provides us with some indication about the performance of our algorithms.

The QUANTIFIED BOOLEAN FORMULA PROBLEM (QBF) is a well known $\sum_2^p$-complete problem[1] for which many solvers have already been developed; see [31] for a nice evaluation of current day solvers. QBF is an extension of the classical satisfiability problem in which we are able to quantify the propositional symbols or variables of the problem. In classical satisfiability problems we want to know whether there exists an assignment of its variables such that all clauses of the instance are satisfied, i.e. evaluate to true. Clauses in these problems are usually disjunctions of variables and the formula is commonly denoted as a conjunction of all clauses, known as the Conjunctive Normal Form, or CNF.

QBF formulation allows for a larger degree of freedom in expressing computational problems. By applying combined quantifiers on the variables of a satisfiability problem we can model more difficult problems quite naturally. Finding a planning of a fixed length k, for example, can be modelled directly as a QBF formula [35]. Also QBF is commonly studied in the field of adversarial game theory in which we want to find a dominating strategy for the player. An example of this can be found in [3].

Encoding and solving the minimal PC as a QBF requires an iterative approach. Although we can model finding an assignment of coordination constraints such that no cycle can be created as a QBF, we can impose no restrictions on its size. The QBF solver would simply return the first assignment it finds that does not allow for any inter-agent cycle. In order to solve the PC minimally, we need to encode it such that we model the finding of a coordination set of size K in QBF. Then we iterate until we have found a smallest set, i.e. no set with size $K-1$ coordinates the instance. For this we can apply similar strategies to those discussed in section 3.3.

[1]The actual complexity of QBF is determined by the number of quantifiers used in the formula and can be arbitrarily hard. Most research, however, has focussed on the class of $\sum_2^p$-complete formulas.

Analogue to [3], we develop our QBF encoding in different phases. First we design the variables and axioms associated to the PLAN COORDINATION PROBLEM section 6.1. Then in section 6.2 we formulate our PC encoding as a QBF and in section 6.3 we transform the instance into CNF, as is the proposed standard in [32]. In our encoding we exploit the kernelisation techniques discussed in chapter 5 to minimise the size of the encoding.

6.1 Phase I: Designing the model

Encoding a problem as a QBF requires choosing variables that represent possible choices in the problem and incorporating rules to state legal options and goal conditions. The goal of the QBF solver then is to find an assignment of our variables such that no rule is violated and the goal condition is achieved. Therefore we need to model our rule such that they immediately make the QBF instance unsatisfiable when violated. Also the QBF instance should be satisfiable if and only if the goal condition is met, however this modelling will be discussed in the second phase.

In this phase we will introduce the variables to model the PC and the encodings of the rules of the PC. As we have discussed before, we cannot impose any restrictions on the coordination set size using QBF encoding. Hence the encoding we propose is dependent on some integer k that denotes the size of the coordination set to be found. Recall that we are given a PC instance $\langle \mathcal{T}, \mathcal{A}, f \rangle$.

Variables

As basic variables in our encoding we have the planning options for each planarc and coordination set inclusion variables. Hence for every $e_x = (t_i, t_j) \in \hat{E}$ we include four variables $e_{i,j}$ and $e_{j,i}$. If we choose to plan arc $t_i \prec t_j$ we have $e_{i,j} = true$, the case for $e_{j,i}$ being analogous. If we add $e_{i,j}$ to the coordination set, we set $\delta_{i,j} = true$, again the case for $\delta_{j,i}$ is similar. Note that we need rules to prevent illegal combinations of planarcs such as both $e_{i,j} = true$ and $e_{j,i} = true$ or $e_{i,j} = false$ and $\delta_{i,j} = true$ at the same time.

Our variable sets are given by

$$E = \{e_{i,j}, e_{j,i} \mid e = (t_i, t_j) \in \hat{E}\}$$

and

$$\Delta = \{\delta_{i,j}, \delta_{j,i} \mid e = (t_i, t_j) \in \hat{E}\}$$

Planarc exclusion

For each combination of planarc variables $e_{i,j}$ and $e_{j,i}$ we add a mutual exclusion clause $(\neg e_{i,j} \vee \neg e_{j,i})$ so that we cannot have both constraints at the same time. There will be $|\hat{E}|$ of such clauses. We represent all the mutual exclusion clauses by the set ME, defined as

$$ME = \bigwedge_{(t_i,t_j)\in\hat{E}} (\neg e_{i,j} \vee \neg e_{j,i})$$

Coordination set

If we want to add a constraint $e_{i,j}$ to the coordination set we set its corresponding coordination set variable $\delta_{i,j}$ to true. However, this can only be done if $e_{i,j}$ is added in the first place. We need to prevent $\delta_{i,j}$ from being added to the coordination set when its associated arc is not. Therefore we introduce the set of coordination set clauses as

$$CS = \bigwedge_{e_{i,j},\delta_{i,j}} (\delta_{i,j} \rightarrow e_{i,j})$$

Note that we do not have to add another mutual exclusion constraint for these coordination set variables, because this already enforced by the mutual exclusion constraints for their associated arcs.

Intra-agent Cycles

There are two possible causes for intra-agent cycles to occur, both the choice of planarcs and the choice of coordination set constraints may lead to local cycle. Planarcs might cause a local cycle, however this should not result in an unsatisfiable formula. In practice such cycles can never be created by the agents and we only need to detect them to make sure that inter-agent cycles do not also cause local cycles. Inter-agent cycles that also introduce local agent cycles have been created by illegally adding planarcs (i.e. those that cause a local cycle) and can therefore never exist in practice.

The latter type of local cycle, introduced by the choice of coordination constraints, should cause the formula to become unsatisfiable however. The purpose of a coordination set is to enforce that the multi-agent planning problem will always remain acyclic. Choosing a coordination set such that it introduces a cycle itself should therefore be forbidden. We do this by copying the intra-agent cycle clauses for the planarcs and replace all arc choice variables $e_{i,j}$ by their associated coordination set variable $\delta_{i,j}$.

The detection of intra-agent cycles requires an exhaustive enumeration of all possible cycles within the agent. From [42] we know that this enumeration is exponential in the number of cycles and hence we will obtain an exponential amount of clauses. We can perform cycle enumeration using the algorithm from [29], slightly adapted to deal with planarcs.

Using this adapted algorithm we find all possible cycles and for each cycle in the form $C = \{e_x, e_y, \ldots, e_z\}$ we create two clauses that are falsified if the cycle exists because of the current assignment. The first clause detects intra-agent cycles for all planning choices while the second clause only involves the coordination set variables. Hence for each cycle C we have clauses

$$C_{\hat{E}} = \neg(e_x \wedge e_y \wedge \ldots \wedge e_z)$$

and

$$C_{\Delta} = \neg(\delta_x \wedge \delta_y \wedge \ldots \wedge \delta_z)$$

The union of all these cycle detection clauses evaluates to true if and only if all clauses are satisfied and hence when no intra-agent cycles occur. We take the union of both constraint types separately because the first type of cycles is allowed in general and is only used to mark illegal inter-agent cycle whilst the second constitutes an illegal choice for the coordination set. These unions are given by

$$Intra_{\hat{E}} = \bigwedge C_{\hat{E}}$$

and

$$Intra_{\Delta} = \bigwedge C_{\Delta}$$

Inter-agent Cycles

For inter agent-cycles we use almost the same approach as with intra-agent cycles, however because of the observation in section 5.1 we do not have to consider all possible cycles in the precedence graph. Using this observation we can conclude that inter-agent cycles only consist of inter-agent constraints and connected in-out pairs. Therefore we only have to enumerate all cycles that can exist consisting only such constraints. This prevents us from enumerating through all intra-agent paths; instead we only have to consider in-out pairs. Again we can use a modified version of the cycle enumeration algorithm from [29] to find all such cycles. Note that verifying whether we can connect an in-out pair is already covered by the intra-agent cycle detection clauses and hence we do not have to encode extra rules for this.

In this set of inter-agent clauses there might be again some overlap. Consider for instance two inter-agent cycles that enter and leave some agent A_i in the points t_{in} and t_{out}, respectively. The first cycle can be created by directly connecting t_{in} to t_{out}, while the second cycle exits the agent through some other task, enters it again through yet another task t' and can then be created when we connect t' to t_{out}. This latter cycle is subsumed by the first and hence we can remove the larger of these two from our set of clauses. This notion is important also for the transformation to CNF we dicuss in section 6.3.

We use the set $Inter$ to denote all inter-agent cycle detection clauses.

Goal condition

The goal condition is the only part of the QBF encoding that depends on the parameter k. To reduce the number of clauses we require to represent the goal condition, we introduce an auxiliary variable $\hat{\delta}_{i,j}$ for each planarc $e = (t_i, t_j)$. This variable is true if the planarc is in the coordination set, i.e. $(\delta_{i,j} \vee \delta_{j,i}) \rightarrow \hat{\delta}_{i,j}$.

Using these auxiliary variables we can formulate the goal as finding a permutation of these variables such that k variables have the value $true$ and the rest $false$. We can generate exactly $\binom{|\hat{E}|}{k}$ of such clauses. An example for $k = 3$ and $|\hat{E}| = 5$ is given below:

$(\hat{\delta}_{1,2} \vee \hat{\delta}_{2,3} \vee \hat{\delta}_{3,5} \vee \neg\hat{\delta}_{4,5} \vee \neg\hat{\delta}_{5,8}) \wedge (\hat{\delta}_{1,2} \vee \hat{\delta}_{2,3} \vee \neg\hat{\delta}_{3,5} \vee \hat{\delta}_{4,5} \vee \neg\hat{\delta}_{5,8}) \wedge (\hat{\delta}_{1,2} \vee \hat{\delta}_{2,3} \vee \neg\hat{\delta}_{3,5} \vee \neg\hat{\delta}_{45} \vee \hat{\delta}_{,58}) \wedge \ldots \wedge (\neg\hat{\delta}_{1,2} \vee \neg\hat{\delta}_{2,3} \vee \hat{\delta}_{3,5} \vee \hat{\delta}_{4,5} \vee \hat{\delta}_{5,8})$

We denote the set of auxiliary variables by $\hat{\Delta}$ and the goal condition clauses for constraint set size k by the set G^k.

6.2 Phase II: Formulating the QBF

Now that we have designed our QBF encoding in terms of variables and clauses, we must formulate the problem using existential qualifiers. As we have discussed previously in this chapter, we want the QBF encoding to be satisfiable if and only if we have a coordination set of size k. Recall from chapter 1 that when we add the coordination set to any PC instance, the resulting constraint set must remain acyclic. Also it must prevent the creation of cycles for all valid choices of possible planning choices of each agent.

In terms of Quantified Boolean Formula we are looking for an assignment of each of the coordination set variables $\delta_{i,j}$ such that for all assignments of planarcs $e_{i,j}$ the resulting formula is satisfiable. Moreover, to enforce a coordination set of specific size, this assignment can only be valid if it in addition also satisfies the goal condition clauses. The QBF encoding that corresponds to this is given in equation 6.1.

$$\exists \Delta \forall E : CS \wedge Intra_{\Delta} \wedge (Intra \wedge ME \rightarrow Inter) \wedge G^k \tag{6.1}$$

This formula asks whether there does exist an assignment of variables in Δ such that for all variables in E it holds that (1) each arc associated with the coordination set constraint is included in the instance, (2) the coordination set does not introduce a local cycle itself, (3) if no intra-agent cycles occur and mutual exclusion is respected then no inter-agent cycle should exist and (4) only k constraints have been used. We discuss the clauses below in more detail.

(1) CS

The clauses in CS make sure that whenever we select an arc $i \prec j$ as a coordination constraint, by setting $\delta_{i,j}$, the associated arc variable $e_{i,j}$ is also set. If this is not implied then we might select coordination constraints without actually enforcing them on the instance.

(2) $Intra_{\Delta}$

In order to produce a valid coordination set we must make sure that the set itself does not impose any local cycles when added to the instance. In the QBF encoding we allow local cycles, because they can only result from making illegal planning choices and hence cannot exist in practice. Agents will not create such cycles when solving their local planning problem. Coordination sets, however, are imposed by solving the PC and may introduce local cycles if chosen badly. A set that makes the instance cyclic after adding it to the original constraints is not a valid coordination set and hence should be forbidden. The clauses in $Intra_{\Delta}$ prevents the coordination set from introducing local cycles.

(3) $Intra \wedge ME \rightarrow Inter$

This part of the formula 'tests' the correctness of the coordination set we are trying as our current assignment. When valid planning choices have been made then no inter-agent should possibly exits, otherwise our coordination set does not coordinate the instance. The validity of planning choices is tested by $Intra$ and ME. The first is satisfied if and only no inter-agent cycle occurs, while the later is satisfied if and only if no mutual exclusion is violated. Note that we are not interested whether an inter agent-cycle exists or not when invalid planning choices have been made, hence only the single implication.

(4) G^k

The goal clauses are rather trivial, however we must make sure that we choose a coordination set with exactly k coordination constraints. Otherwise we may find a coordination set of random size and we can never work towards an optimal (minimal) set size.

6.3 Phase III: Transforming to CNF

In order to solve our QBF encoding of the PLAN COORDINATION PROBLEM by any of the state-of-the-art solvers currently around, it remains to transform our formula into Quantified CNF, using the QDIMACS format. This format has been proposed in [32] and has been adhered by almost all state-of-the-art QBF solvers. Hence it seems worthwhile to transform our encoding into that format.

Variables of the our encoding need no transformation; we only have to transform our rules into a conjunction of clauses which consist only of disjunctions of variables. This, however, is not straightforward for our problem: encoding the implication $Intra \wedge ME \rightarrow Inter$ naively would require an enormous amount of clauses. This is because transforming an implication into CNF naively results in a Cartesian product of the sets on both sides of the implication. To prevent this, we introduce auxiliary *indicator variables* as done in [3].

To each clause in all of the aforementioned sets we add an indicator variable that must evaluate[2] to $true$ if and only if that clause is satisfied. Then we combine the

[2]Although we use the term 'evaluate' here, it is not true than a QBF solver indeed sets the indicator variable the moment a monitored clause is falsified. Only because there is no other satisfying assignment for

indicator variables for each set into one additional *set indicator variable* that evaluates to $true$ if and only if no indicator variable for that set is assigned $true$. The implication of the three sets can then be replaced by a three variable implication only containing the set indicator variables, which requires only 3 clauses. First we will demonstrate how we can monitor clauses and sets then after that we will discuss the cost of monitoring clauses.

By monitoring a clause, we want to assign an indicator variable to it such that it evaluates to $true$ if and only if that clause is satisfied. Hence for any random clause $C_i = (x_1 \vee x_2 \vee \ldots \vee x_m)$ we want to assign a indicator variable c_i such that:

$$(x_1 \vee x_2 \vee \ldots \vee x_m) \iff c_i \tag{6.2}$$

We obtain the clauses to add by rewriting both implications of equation 6.2 into boolean logic:

$$\begin{aligned}
(x_1 \vee x_2 \vee \ldots \vee x_m) \rightarrow c_i &= \neg(x_1 \vee x_2 \vee \ldots \vee x_m) \vee c_i \\
&= (\neg x_1 \wedge \neg x_2 \wedge \ldots \wedge \neg x_m) \vee c_i \text{ (DeMorgan)} \\
&= (\neg x_1 \vee c) \wedge (\neg x_2 \vee c_i) \wedge \ldots \wedge (\neg x_m \vee x_c) \qquad (6.3)
\end{aligned}$$

$$\begin{aligned}
(x_1 \vee x_2 \vee \ldots \vee x_m) \leftarrow c_i &= c_i \rightarrow (x_1 \vee x_2 \vee \ldots \vee x_m) \\
&= \neg c_i \vee (x_1 \vee x_2 \vee \ldots \vee x_m) \\
&= (\neg c_i \vee x_1 \vee x_2 \vee \ldots \vee x_m) \qquad (6.4)
\end{aligned}$$

We do not make any assumptions on the sign of each of the literals in the original clause C, i.e. each variable x_i can either denote x_i or $\neg x_i$. Note that it does not matter whether clause C is a disjunction or conjunction of literals, in the latter case we would obtain the same clauses although the sign of each of the variables $x_1, x_2, \ldots, x_m$ is inverted. This is because

We can combine all indicator variables into a set indicator variable in a similar fashion. We generate one clause that should result in the set indicator variable assigned to $true$ if and only if at least indicator value is true, resulting in clauses alike the results of equation 6.3 and equation 6.4. Of course, we can also let it evaluate to $false$ by inverting the set indicator literal or we can force all indicator values need to be true by inverting the indicator variable signs. For the purpose of demonstration we assume that we want the set indicator variable to evaluate to $true$ if and only if all indicator variables evaluate to $true$. This is the case with our mutual exclusion set indicator for instance.[3] This results in the following clauses for the set indicator over n indicator variables:

that clause, the QBF solver will eventually conclude that the formula can only be satisfied when setting the indicator variable.

[3]And it also demonstrates that we obtain a similar set of clauses when adding an indicator variable to a conjunctive clause.

$$\begin{aligned}
(c_1 \wedge c_2 \wedge \ldots \wedge c_n) \rightarrow c* &= \neg(c_1 \wedge c_2 \wedge \ldots \wedge c_n) \vee c* \\
&= (\neg c_1 \vee \neg c_2 \vee \neg \ldots \vee \neg c_n) \vee c* \text{ (DeMorgan)} \\
&= (\neg c_1 \vee \neg c_2 \vee \neg \ldots \vee \neg c_n \vee c*) \qquad (6.5)
\end{aligned}$$

$$\begin{aligned}
(c_1 \wedge c_2 \wedge \ldots \wedge c_n) \leftarrow c* &= c* \rightarrow (c_1 \wedge c_2 \wedge \ldots \wedge c_n) \\
&= \neg c* \vee (c_1 \wedge c_2 \wedge \ldots \wedge c_n) \\
&= (\neg c* \vee c_1) \wedge (\neg c* \vee c_2) \wedge \ldots \wedge (\neg c* \vee c_n) \qquad (6.6)
\end{aligned}$$

Although using indicator variable and set indicator variable we can easier encode our implication, we still need to encode these additional variables and clauses for them given by equation 6.3, equation 6.4, equation 6.5 and equation 6.6. We have seen that in order to monitor any clause of size m we require m additional clauses in order to correctly set or unset the indicator variable. Also, in order to combine the result of all n monitored clauses, we need another $n+1$ clauses. To compare with the Cartesian product we need to analyse the resulting sets more carefully. We analyse set sizes in terms of planarcs n, number of unique intra-agent cycles C_{intra} and number of unique inter-agent cycles C_{inter}.

Mutual exclusion

Encoding the mutual exclusion constraints requires exactly n clauses without monitoring. By adding indicator variables to these clause we would require 2 additional clauses per planarc. The set indicator variable requires another $n+1$ clauses, hence we obtain a total of $3n+n+1 = 4n+1$ clauses for mutual exclusion.

Cycle clauses

To encode all C_{intra} intra-agent cycles into QBF we need to add exactly C_{intra} clauses, however of various length. Let l_1 denote the average cycle length in arcs of these cycles, then we require $l_1 \times C_{intra}$ additional indicator clauses. Combining the indicator variables into a set indicator variable requires $C_{intra}+1$ more clauses. This results in a total of $(l_1+2) \times C_{intra}+1$ clauses.

For the inter-agent cycles the number of clauses is computed similar to the intra-agent clauses, however with a different average cycle length l_2. Therefore we obtain a total of $(l_2+2) \times C_{inter}+1$ clauses.

Set indicator implication

If we use set indicator variables, we can enforce our implication $Intra \wedge ME \rightarrow Inter$ with just three variables: $Intra*, ME*$ and $Inter*$. This results in just one clause $(\neg Intra* \vee \neg ME \vee Inter)$.

Combining the total number of clauses required for this encoding we obtain $4n+(l_1+2) \times C_{intra}+(l_2+2) \times C_{inter}+4$. Bounding the average cycle length to the maximum cycle length of n, we get $4n+(n+2) \times C_{intra}+(n+2) \times C_{inter}+4$.

Cartesian Product

Generating the Cartesian product of $Intra \wedge ME \rightarrow Inter$ requires a substantial amount of clauses. Recall that we generate n clauses for mutual exclusion C_{intra} inter-agent cycle clauses and C_{inter} agent cycle clauses. Although for the real number of generated clauses we also need to know the lengths of the clauses in the Cartesian product, in order to show the exponential blow up it suffices to assume all clauses to be of length 1.

The number of clauses we generate with the Cartesian product is, assuming a clause length of 1, equal to $(clauses(Intra) + clauses(ME)) \times clauses(Inter)$. We have analysed the required number of clauses for each set before, resulting in a total of $(n + C_{intra}) \times C_{inter}$. As the number of cycles in a directed graph can be exponential in terms of its tasks [7], the number of resulting clauses rapidly grows. This growth is exponentially larger than that of the monitoring approach.

Indeed using the indicator variables we end up with a much smaller encoding, already when assuming that the clause lengths in the Cartesian product all equal 1. Larger clause lengths result in an even worse growth rate of Cartesian product clauses. Therefore we encode our planning problem using indicator variables.

Conditional Variables in QBF

During experiments with our encoding we have seen that already for very small PC encodings, the QBF solver we used had great difficulty in determining whether the formula is satisfiable. In [3], Anóstegui et al. identify a pitfall of many modern QBF solvers. Almost all current day solvers use a top-down approach to tackle QBF formulas, which does not work very well with indicator variable. To benefit more from indicator variables, they propose to make the QBF solver aware of such conditional variables. This could allow a solver to immediately backtrack when a conditional variable is falsified, to prevent searching large irrelevant areas of the search space. As the bulk of our encoding consists of such conditional variables, we expect to see enormous performance improvements when using a solver that accounts for them equal to the result obtained in [3].

Chapter 7

Experiments

In this thesis we have proposed several approaches to solve the PLAN COORDINATION PROBLEM. Our research has focussed mainly on exact solving of the problem, however also previously developed approximation methods by others have been discussed. In order to get insight into the performance of the various algorithm, we will experimentally verify and compare them. In these experiments we will focus both on the time required to solve instances of the PC, as well as the obtained coordination set size.

As we have stated before, there has not yet been any attempt to solve the PC exact. As a consequence, our experiments will be mostly of an exploratory nature. Nevertheless we also study the influence of various characteristics of PC instances to gain more knowledge about the complexity and scaling of the problem. We have developed several hypotheses concerning the PC that we wish to verify experimentally.

This chapter starts with a discussion of all PC instance characteristics to understand all variables that are involved in our experiments. Using these variables we have formulated several hypotheses that we wish to study in our experiments. Both are discussed in section 7.1 Then, before we go into the experiments themselves, we describe our test set and experimental set-up in detail in section 7.2. Section 7.3 discusses the results we have obtained from our experiments.

7.1 Experimental Goals

In order to do perform useful experiments not only of exploratory nature, we have devised a set of test goals in advance to allow for concise testing. This section discusses those experimental goals that, from our perspective, are the most relevant in the context of this thesis. Each sub section presents an important aspect to verify experimentally and introduces hypotheses underlying our tests. The validity of each of these hypotheses will be discussed using our experimental results in section 7.3.

Previous work

We have mentioned a few times that previous work has only focussed on approximation or special cases (see chapter 2), however these methods have not yet been tested in practice. In [47] only a theoretical comparison of the coordination set sizes is done. This analysis has never been verified experimentally.

In this thesis we compare the coordination sets produced by both methods and try to verify the claim made in Yadati et al. in [47] about the produced coordination set sizes. Moreover, we also relate the set sizes of approximation with the set size of the exact solution. For these tests we have the following two hypotheses:

Hypothesis H 1: Depth Partitioning versus Intra-free Coordination

Based on the theoretical analysis from [47] we expect the Intra-free coordination method to produce smaller coordination sets on average than the Depth Partitioning method for the same instances. Hence want to verify the null hypothesis $H_0 : \mu_{IF} < \mu_{DP}$*. In addition, it is likely that the depth partitioning algorithm is more effective on instances with smaller instance depths (see definition 2.1).*

Hypothesis H 2: Approximation versus optimal

Both the Depth Partitioning and Intra-free Coordination methods produce will most likely produce much larger coordination sets compared to the exact solvers. Moreover we expect the ratio $|\Delta_{apx}|/|\Delta_{opt}|$ *to become increasingly large when the number of planarcs increases. This is because the number of coordination possibilities greatly increases for each additional planarc, while requiring only a small amount of additional coordination constraints in most cases.*

Solver comparison

We have developed three different solvers to tackle the PC exact, based on three different algorithmic techniques. Our first solver is the **pcpenum** solver and is based on simple enumeration as introduced in chapter 3. This solver uses enumeration to generate configurations for both the PC and the CVP problems.

The **pvpdyna** solver from chapter 4 aims to tackle the CVP sub problem more efficiently by applying the dynamic programming technique to split up the problem in smaller, more easily solvable sub problems. It still relies on the enumeration procedure to generate PC configurations.

Finally the **pvpkern** solver extends our **pvpdyna** solver with the kernelisation discussed in chapter 5. This solver treats planarcs slightly different to reduce the number of options that do actually have to be considered in solving the instance. By focussing only on the so called In-Out pairs, we can eliminate several possibilities that will never lead to any coordination constraint in advance. Also, it uses pre-processing to remove tasks and sometimes even agents that can never be part of the optimal solution anyway.

As each subsequent solver has been developed from the previous one by extending it with more enhanced search space elimination methods, we expect to see that our **pvpkern** solver will outperform the **pvpdyna** solver, which on its turn outperforms the **pcpenum** solver. This is formulated in hypothesis H 3. Although verifying this hypothesis seems unnecessary, we are still interested to see how they compare.

Hypothesis H 3: Solver comparison

*We expect the **pvpkern** solver to perform best of our three solvers. Also, the **pvpdyna** solver is expected to perform better than the **pcpenum** solver.*

Complexity causes

One can immediately see that the complexity of such an instance does not depend on just one factor. Several instance characteristics affect the run time of solving a PC instance. One of the main goals in our research, and hence our experiments, is to study these characteristics and their influence on the solving hardness. Ideally, we would like to find a good estimator for the 'difficulty' of any given PC instance, so that we can a priori determine what instances are hard and what instances are easy to solve. In addition, we are also interested in finding a heuristic for the enumeration strategy (chapter 3) to use. This allows us to apply the, in most cases, right strategy on each instance.

In order to obtain any such an estimator or heuristic, we need to study all the individual factors in detail. In PC instances we can immediately identify a few of such contributors. Recall from chapter 1 that a PC instance can be represented by a complex task $\mathcal{T} = \langle T, \prec, \equiv \rangle$, a set of agents $\mathcal{A}$ and a task distribution function f. In our experiments we have focussed on the agent set size and the number of planarcs, because the task set and constraint set sizes themselves will not affect performance much. Increasing the task or constraint set size while keeping both the agent set size and planarc set size the same will only incur a relatively very small additional cost because of the way our solvers are implemented.

In the results section (section 7.3) we will see that the **pvpkern** solver outperforms our other solvers on all instances. Therefore we have studied the complexity contribution of each variable using only the **pvpkern** solver. Note that the variables are most likely to affect each type of solver differently, hence the hypotheses and conclusions we present only apply to our **pvpkern** solver.

In hypothesis H 4 and hypothesis H 5 we formulated our expectations concerning the influence of the agent set size and number of planarcs.

Hypothesis H 4: Agent set size

The size of the agent set will most likely have a great impact on the run time required to solve an instance. Increasing the number of agents while keeping all other variables fixed will allow the solver to benefit more from the dynamic programming approach used to solve the CVP*. Smaller sub problems can be created that can be solved more easily.*

Hypothesis H 5: Planarcs

Together with the agent set size, we expect the number of planarcs to be one of the most important factors in the run time complexity. Each additional planarc causes a multiplication of the number of planning possibilities by at least a factor 2*. Hence the run time complexity is expected to be exponentially dependent on the planarc set size.*

In addition to these characteristics, the set size of the optimal solution is also most likely to affect the run time required to solve PC instances. However, the extend to which the optimal size will affect the run time is greatly dependent on the enumeration strategy we use (see next sub section). For example, a relative small optimal size compared to the number of planarcs will most likely cause the increasing size strategy to perform more effective, while the binary and decreasing size strategies will probably have a harder time solving such an instance. Because this property is correlated to the strategy we use, we study this using hypothesis H 6 introduced in the next sub section.

Optimisations

For each different technique we have proposed optimisations to increase its effectiveness. In our experiments we want to investigate how much each optimisation affects the run time required to solve PC instances. In addition we would like to gain some insight into the type of instances for which each optimisation is best applicable.

In chapter 3 we have introduced the strategy optimisation for the PC enumeration procedure. In theorem 3.1 we have already proven that we can indeed greatly reduce the size of the remaining search space using this optimisation, however we do not know how effective this reduction is in practise. Also, as we have stated in section 3.3, the choice of strategy depends greatly on the size of the optimal coordination set Δ compared to the number of planarcs $|\hat{E}|$.

We can be certain that using the enumeration strategy always outperforms simple enumeration. Only in the worst case, the enumeration strategy requires checking all possible configurations, which the simple enumeration always has to. Hence we do not have to verify that the optimisation is useful. Our experiments therefore focus on the type of strategy to apply based on the instance properties, see hypothesis H 6.

Hypothesis H 6: Enumeration strategy

As discussed in section 3.3 the effectiveness of the enumeration strategy is expected to depend on the ratio between the optimal coordination set size and the number of planarcs $r = |\Delta|/|\hat{E}|$. When r is small, the increasing size strategy is expected to perform best. When r is close to .5, the binary strategy is to be preferred. For values of r from .75, the decreasing size strategy will most likely do best. Although this seems promising, we think it is unlikely there exists a heuristic based on predetermined instance characteristics that is able to indicate the best strategy. This is because instances with the same properties can have varying optimal set sizes.

In addition to the enumeration strategy we also study the minimisation introduced in section 3.3. Solution minimisation quickly tries to improve any currently best optimal solution by iteratively removing planarcs constraints from it, while ensuring that the coordinated property is preserved by the coordination set. To prevent the solution minimisation from exhaustively searching all possible sub sets of the current best solution we included a minimisation depth level. This level corresponds to the minimum improvement the solution minimisation has to achieve before abandoning the search.

Our experiments concerning the depth parameter will be focussed on finding a ‘good’ value, if such a value exists. We state this in hypothesis H 7.

Hypothesis H 7: Solution minimisation

The solution minimisation parameter depth controls the minimum required improvement of the currently best solution before abandoning the improvement search. For large values of the parameter, the minimisation optimisation will more resemble an exhaustive enumeration of all possible sub sets of the currently best solution and hence such a value for the depth parameter is not desired. We expect that for a value of 2 *or* 3 *the solution minimisation performs best, as we expect this to be a good trade-off between the improvement the solution and the time required to find such an improvement.*

Chapter 5 about kernelisation introduces a task removal pre-processing optimisation, exploiting the fact that some tasks and their corresponding planarcs can never introduce inter-agent cycles. Such tasks and their associated planarcs can therefore be removed in advance in polynomial time, before starting the exponential PC algorithm. As we will see in section 7.3, our solvers benefit much from a reduction in the number of planarcs.

In our experiments we want to gain knowledge about the actual performance of this pre-processing approach in terms of the instance characteristics. In hypothesis H 8 we have summarised our expectations concerning the pre-processing of PC instances.

Hypothesis H 8: Pre-processing

We expect the pre-processing introduced in section 5.2 to be dependent on both the number of planarcs and the number of agent of an instance. When the number of planarcs becomes larger, more tasks will be part of interesting constraints and hence we expect the pre-processing effectiveness to decrease for larger numbers of planarcs. Moreover, when the number of agent increases and the number of planarcs is kept constant, the pre-processing will most likely perform better. This is because with more agents, the probability of an agent not participating in any possible inter-agent cycle increases and hence it can be removed from the instance.

QBF Encoding

In chapter 6 we introduce a QBF encoding for the PC that enables us to solve the problem using highly optimised state-of-the-art solves from one of the most actively studied fields of computer science. This way we can obtain a benchmark to compare our exact solvers to. Moreover, we want to verify that for the PC a specialised solver is able to do better than a generalised solver using an encoding of the PC, stated in hypothesis H 9.

Hypothesis H 9: QBF encoding

The QBF *encoding proposed in chapter 6 is expected to be outperformed by our specialised* ***pvpkern*** *solver. This assumption is based on the fact that our specialised solver is designed to exploit the problem structure, whilst the generalised solved* QBF *simply performs an exhaustive search through the entire search space of encoding variables.*

7.2 Experimental Set-up

For our experiments we have both developed an instance generator, discussed in detail below, and a script to enable automatic testing. The test script simply iterates through all generated instances and stores the results in an output file, such that we can extract results from it later.

We have also included a time limit in our experiments. When solving any instance takes more than this time limit, we abandon the process and report that it has been exceeded. This has been done to prevent waiting for ours or even several days before obtaining a coordination set for some instances and allow us to test a lot of different instances. Also, when solving an instance takes a lot of time we can question its applicability in practice, although this depends on the situation in which the algorithm is used.

Instance generator

In order to obtain a large test set we have implemented a simple generator that is able to produce PC instances randomly, based on the supplied parameters. To generate an instance we have to specify the number of agents, tasks, constraints and planarcs we would like to have. In addition, we specify the number of instances to create with these properties.

The generator simply creates the number of tasks specified and distributes them randomly over all agents. After that, it will randomly create constraints and add them to the instance, however a little care has to be taken when adding constraints. Any constraint that would cause the constraint set to become cyclic is rejected and a new constraint is generated for it. To prevent the generator from looping infinitely, we impose a maximum number of rejections before giving up. If the solver does give up, we simply try generating an instance again, albeit with a different seed for its random function. Hence we use a Monte Carlo method to generate our instances.

When the generator has succeeded in creating an instance, we verify that is has the required number of planarcs. If this is not the case, again the generator starts over with a different seed. This way we are sure that the resulting instance has all the requested characteristics. The generation is terminated when either the required number of instances is generated or the maximum seed value is reached. In the latter case, less instances are generated than we would have liked, however this occurs only when the number of planarcs is relatively high in comparison to the number of tasks and constraints. When such is the case, the number of instances becomes rare and it is hard to generate such an instance.

Using our generator we have built a substantial test set with instances of different characteristics, as we will discuss in the next sub section.

Test set

Our test set used with our own algorithms consists of 13.187 different instances, all with varying combinations of characteristics. The number of agents varies between 2 and 6, for the task set size we have chosen the values 8, 10 and 12. The constraint set sizes are chosen between 4 and 20. Then for each combination of these values, we have generated 10 instances for each choice of planarc size between 1 and 40. Hence the set we have generated represents more or less a Cartesian product of all values for the different variables and hence allows us to study the influence of each variable individually.

For the comparison of Depth Partitioning and Intra-free coordination we require a special class of instances. We have slightly adapted our generator to also be able to generate such intra-free instances. This special test set contains instances with much larger agent, task, constraint and planarc sizes because both Depth Partitioning and Intra-free Coordination can polynomially solve the problem. We have run both algorithms over a very large set of instances varying from 2 to 10 agents, 10 to 250 tasks and 10 to 250 constraints. The intra-free set contains a total of 4.527 instances.

7.3 Results

In this section we present the results of running our experiments on the test set described in section 7.2. All experiments have been performed on a Intel© i7 Quad-core$^{\text{TM}}$ processor with a clock rate of 2.66 GHz and a machine with 6 GB of RAM. The time limit in our experiments has been chosen 15 minutes (900 seconds) in order to test as many instances as possible, solvable within 'reasonable' time.

Previous work

Before discussing our own algorithms we first go into the Depth Partitioning [38] and Intra-free coordination [47] algorithms proposed in previous work. Remember that, although both methods are able to approximate a coordination set for the PC, intra-free coordination is only applicable to *intra-free instances*. We have used the intra-free instance set, as discussed in section 7.2.

For the DFVS part of the intra-free coordination algorithm, we have used the $O(mn)$ algorithm from [16], in which m and n denote the number of arcs and agents in the *Agent Dependency Graph* respectively. Note that better DFVS approximation algorithms exist, such as the $O(\log|F^*|\log\log|F^*|)$ algorithm by Even et al. [19], however they rely on very complex implementations. For our purposes, the simple $O(mn)$ algorithm suffices.

In figure 7.1 we have made a scatter plot of the coordination set size produced by the Depth Partitioning versus the Intra-free coordination set size. The points of this graph are determined by the pair $(|\Delta_{DP}|, |\Delta_{IF}|)$ for each distinct instance.

Looking at figure 7.1 it seems that the Depth Partitioning algorithm generally produces larger coordination sets, because the most points are below the line $|\Delta_{DP}| = |\Delta_{IF}|$. Using the *Student's t-test* [15] we test the likelihood of our null hypothesis $\mu_{IF} < \mu_{DP}$, as introduced in hypothesis H 1, to be true.[1] Using a statistical tool we computed that the hypothesis $\mu_{IF} < \mu_{DP}$ is true with a probability of 99.997% and hence we can safely accept it.

As for hypothesis H 2, we have plotted the coordination size found by depth partitioning and the optimal coordination set for all 13.187 instances in our regular test set. This is depicted in figure 7.2.

In the figure we have made a plot of both the Depth Partitioning and optimal coordination set sizes, for various numbers of planarcs. Also, to test hypothesis H 2, we have computed the linear regressions of both data sets, illustrated by the lines in the figure. From the figure it is not very clear to see that the linear regression for the Depth Partitioning data set has a larger slope value. Using a mathematical tool we found that the linear regression of the DP data set is given by $y = 16.812 + 0.262x$ and for the optimal set by $y = 0.213 + 0.211x$.

This evidence is not very convincing that indeed the ratio $|\Delta_{apx}|/|\Delta_{OPT}|$ becomes larger when the number of planarcs increases, for the slopes of both linear regressions do not differ much. We can conclude that indeed the Depth Partitioning method produces much larger coordination sets, however our expectations concerning the ratio of this set have not been verified.

[1]The Student's t-test is included as an appendix in appendix A.4.

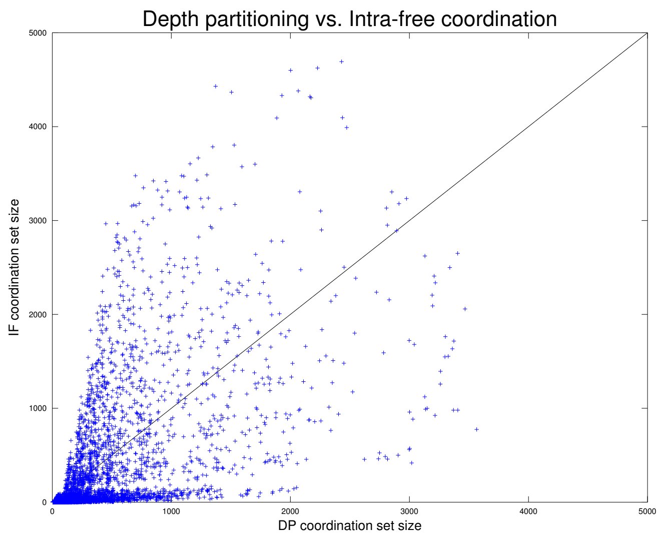


Figure 7.1: The coordination set sizes of the sets produced by the Depth Partitioning and the Intra-free Coordination algorithms depicted as a scatter plot.

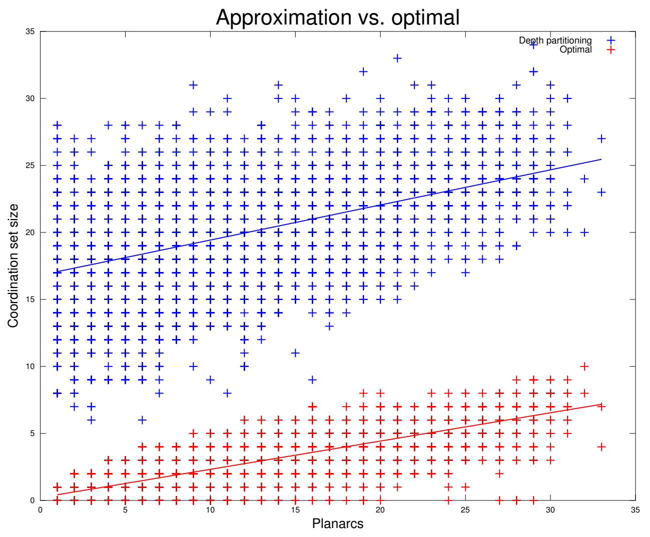
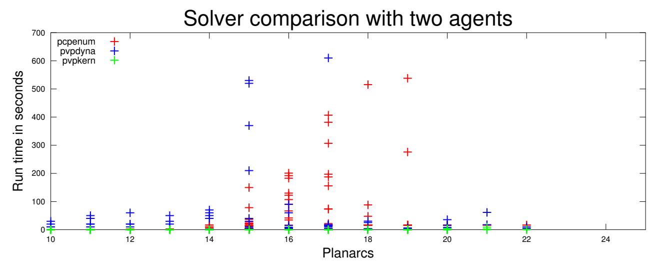


Figure 7.2: Coordination set sizes produced by Depth Partioning versus the optimal set size. The lines illustrate the linear approximations for both data sets.



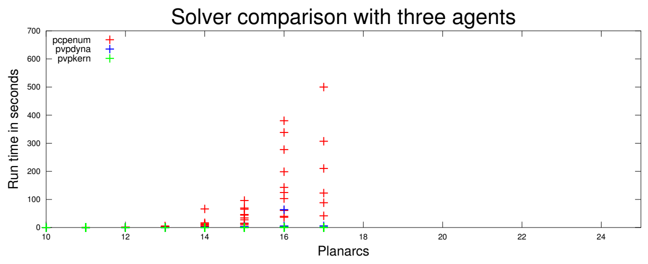


Figure 7.3: The different run times required for each solver against the number of planarcs. The top figure is for instances with 2 agents, the bottom figure shows run times for three agent instances.

Solver comparison

In this thesis we have studied three different algorithmic approaches to tackle the PC. We have developed three solvers using these techniques: the **pcpenum** solver uses enumeration to solve both the PC and the CVP, the **pvpdyna** solver is an enhancement of **pcpenum** using dynamic programming to deal with the CVP sub problem and finally our **pvpkern** solver is extended to also exploit kernelisation. Because of this sequential development of solvers in our work, it seems trivial to verify hypothesis H 3 experimentally. It is almost certain that each subsequent solver will outperform its predecessor.

For the reason stated above we focus more on the limits of each solver and the characteristics that makes a solver more effective than another. In figure 7.3 we have two figures illustrating the time each solver needed to solve an instance with the given number of planarcs. For each number of planarcs the solvers all solved the same ten different instances. The top figure illustrates the run times for instances with two agents and the run times for three agent instances are displayed in the bottom figure.

From the figure we see that indeed the **pvpkern** solver dominates the other two: nearly all instances are solved within the second, while the other solvers require more time. According to our expectations in hypothesis H 3, the **pvpdyna** solver should also outperform the **pcpenum** solver. This does not seem to be the case, at least not with the two agent instances. Especially for smaller planarcs sizes, the **pcpenum** solver performs better.

We believe this to be caused by the additional overhead required by the **pvpdyna** solver to split up the CVP problem. While the **pcpenum** solver tries to find solutions in a brute-force manner right away, the **pvpdyna** solver first splits up the two agents and then performs the brute-force enumeration approach on each individual agent. With only two agents, the **pvpdyna** solver can not benefit much by solving smaller sub problems.

Indeed, looking at the bottom figure, we see that when dealing with three agents, the **pvpdyna** solvers already performs much better than the **pcpenum** solver. With larger agent set sizes, solving the entire problem at once becomes more complex and the **pcpenum** solver has a harder time solving such instances. The **pvpdyna** solver, on the other hand, has to solve three smaller and easier sub problems and hence performs much better now. For instances with four or five agents, this pattern continues and the **pvpdyna** solver performs increasingly better compared to **pcpenum**.

It is safe to say that hypothesis H 3 is true for the case $|\mathcal{A}| > 2$. When $|\mathcal{A}| = 2$, the **pcpenum** solver is able to outperform **pvpdyna**.

Complexity causes

Now that we have established that the **pvpkern** solver is by far the most effective in solving PC instances, we use this solver to try and determine what makes the PC so complex. In section 7.1 we have identified the two most significant characteristics that might affect the run time required to solve PC instances: the number of agents and the number of planarcs. As stated in hypothesis H 4, we expect the run time to decrease when the number of agents increases while keeping the number of planarcs constant. This is because the solver can more effectively use the dynamic programming routine for CVP. In addition, hypothesis H 5 states that the run time is expected to be exponential in the number of planarcs.

We have have a three dimensional plot of agent set size and planarcs versus the required time to solve an PC instance using the **pvpkern** solver. In order to compare run times fairly, we have averaged the run times of all three different strategies for each instance. This plot is shown in figure 7.4.

Looking at figure 7.4 it seems that both hypothesis H 4 and hypothesis H 5 seem to be true. We can see that indeed the run times seem to decrease when the number of agent increases. The run time increases when the number of planarcs increases, however each line has a peak at the end of it after which the run time seems to decrease. This effect is due to the fact that unsolved instances have not been included and the **pvpkern** solver has only been able to solve one or two instances with a high number of planarcs within the given time limit. Most likely these one or two instances have been more or less ‘lucky’ hits by the solver: it was able to conclude quickly on the existence of a coordination set for each size it tried. The cases for which the solver has not been so lucky took more than 15 minutes and hence it abandoned the search.

Note that due to the assignment of tasks and constraints over the agents, the number of instances with large planarc sets decrease as the number of agents increases. When the number of tasks per agent decreases, the set of possible planarcs becomes smaller

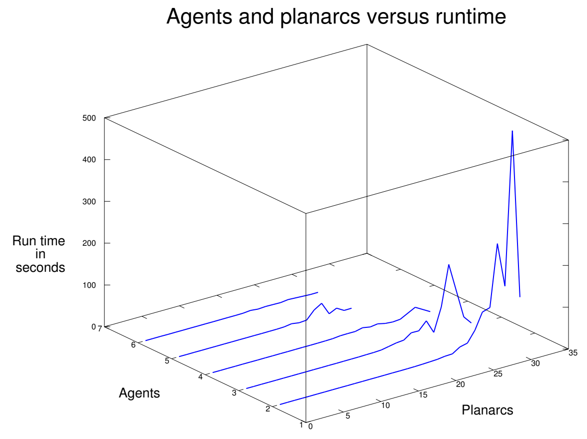


Figure 7.4: The three dimensional plot depicts the run times of the **pvpkern** solver for various agent set sizes and planarcs.

because of definition 1.9 and hence our generator is not able to produce sets with large amounts of planarcs for these instances.

Optimisations

In our thesis we have proposed several optimisations for our solvers, which we have experimented with to study the actual benefit from each optimisation. In this sub section we study all three optimisations: enumeration strategy, minimisation and preprocessing.

Enumeration strategy

The first optimisation we proposed is the enumeration strategy, see section 3.3. The enumeration strategy uses theorem 3.1 to quickly prune large parts of the search space. We introduced three different strategies: binary enumeration, increasing size enumeration and decreasing size enumeration. Our expectation is that the choice of strategy is greatly dependent on the instance we are dealing with, as we state in hypothesis H 6.

More specifically, let r denote the ratio between the optimal coordination set size and the number of planarcs, i.e. $r = |\Delta|/|\hat{E}|$. For the reasons stated in section 3.3 we believe that the increasing size strategy is to be most efficient for small r, the binary strategy for r around .5 and the decreasing size for larger r.

In figure 7.5, figure 7.6 and figure 7.7 we have plotted the number of agents and planarc versus the required run time for all three strategies with $r < .5$, $.25 < r <$

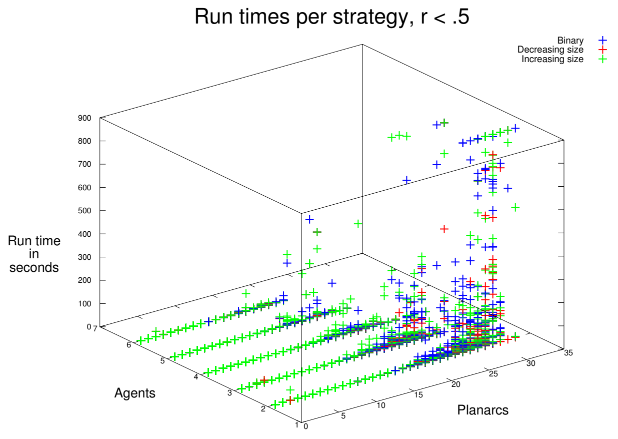


Figure 7.5: Agents and planarcs versus the run time required to solve the instances for each strategy. In this figure we have used instances with $r < .5$.

.75 and $r > .5$ respectively.

The figures indeed seem to provide some evidence for hypothesis H 6, however the decreasing size strategy performs much better in practice than we would have expected. We see that it indeed performs best on instances with $r > .5$, but in addition it seems to be also effective on smaller r, although it is not the strategy to be preferred. We believe this is because although the decreasing size strategy starts its search on the ‘wrong side’ of the search space, it is able to draw conclusions about coordination set existence quickly and hence it rapidly decreases towards the optimal set size. While one would expect the binary search strategy to perform better for the case $r < .5$, binary search also tries coordination sets smaller than the optimal coordination set and has to inspect all possible configurations of these sizes to conclude that no such set exists. This could cause the binary enumeration strategy to require more time to solve such instances.

In figure 7.6 we see that indeed the binary search strategy is able to solve instances faster overall, although again the decreasing size strategy performs well. Here the increasing size strategy is clearly outperformed by both others. For larger values of r, illustrated in figure 7.7, we see that indeed the decreasing size strategy is to be preferred.

The evidence provided by our experiments are not convincing enough to accept hypothesis H 6 to be true. The binary search strategy performs not as well as we would have expected from our theoretical analysis and the decreasing size strategy performs much better than we assumed.

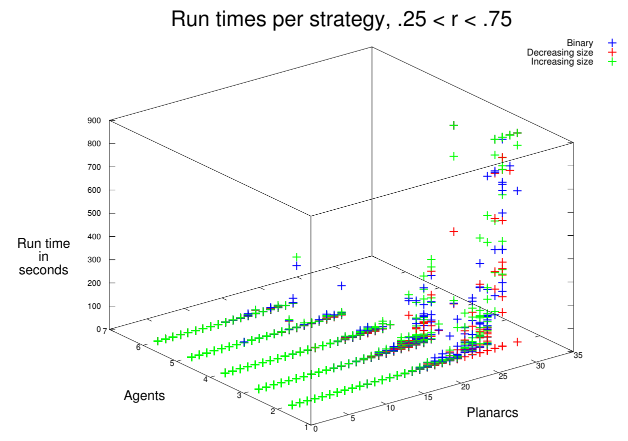


Figure 7.6: Agents and planarcs versus the run time required to solve the instances for each strategy. In this figure we have used instances with $r < .5$.

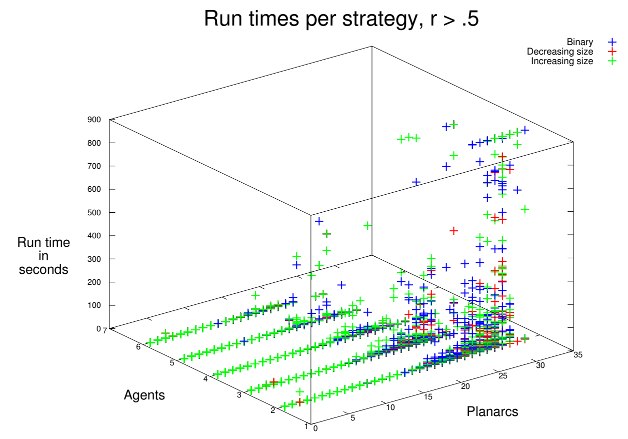


Figure 7.7: Agents and planarcs versus the run time required to solve the instances for each strategy. In this figure we have used instances with $r < .5$.

Solution minimisation

Solution minimisation, as proposed in section 3.3, tries to make quick improvements on any coordination set found during the PC solve process. It tries to reduce the size of the coordination set by removing constraints from the set and testing whether they are still plan coordination. This way we can rapidly establish existence of coordination sets of smaller sizes and we hope to reduce the number of exhaustive searches performed by the PC enumeration algorithm. For instance, if the solution minimisation establishes within a few tries that the current optimal set of size K has a sub set of size $K-1$ it would save checking a potentially very large number of possible configurations in the PC algorithm.

In order to prevent the solution minimisation technique from searching the entire search space of sub sets, we have included the depth parameter that controls the minimum improvement to be made. When we have found an improvement of the specified depth, the minimisation procedure abandons its further searches. This way we can make a trade-off between improvement quality and the time required to find such an improvement. Moreover, for some coordination sets no improvement can be made without violating the coordinated property. The depth parameter prevents the sub routine from inspecting all possible sub sets of such a coordination set.

In our experiments we have used various settings for the depth parameter, varying between 1 and 10. In figure 7.8 and figure 7.9 only the run times for the depth values 1, 2 and 3 rare illustrated. This is for the simple reason that for the case $depth > 3$ the run time rapidly increases and is always outperformed by running the solver without solution minimisation. We have depicted the run times per planarc and agent for both the binary and decreasing size enumeration strategies. Note that solution minimisation is not applicable to increasing size enumeration, for the first coordination set that is encountered in increasing size enumeration is also the optimal one.

In hypothesis H 7 we assumed that a set size of about 2 or 3 is to be preferred. A depth of 1 will not make substantial progress, while depth values larger require to much time to be of any use. Indeed this claim seems to be supported by both figures, although we should account for the strategy we use.

In figure 7.8 we have used the binary enumeration strategy to solve the instances. Here we see that minimisation does indeed prove to be effective. Although it cannot be observed from the figure clearly, we established using a data analysis tool that the binary strategy performs best on average with a minimisation depth of 3.

The decreasing size enumeration, displayed in figure 7.9, does not perform well with a minimisation depth of 3. For this strategy, the value 1 provides the best overall results, which we did not expect in advance. The most likely reason for this is that the decreasing size enumeration strategy is able to quickly find coordination sets, especially when the size it searches for is large. This enables it to conclude on the existence of such a coordination set fast, while the solution

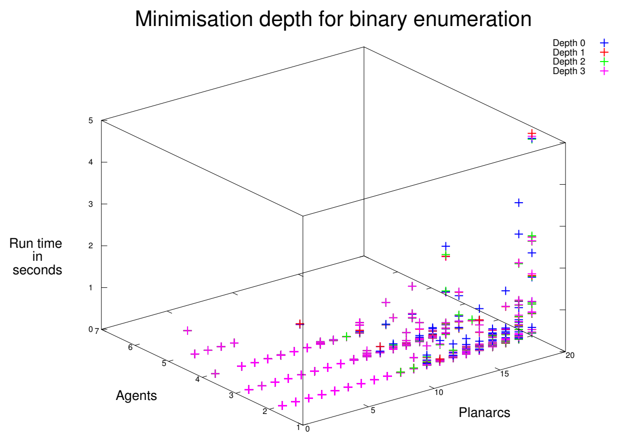


Figure 7.8: Run times for various planarc and agent set sizes while using the binary enumeration strategy.

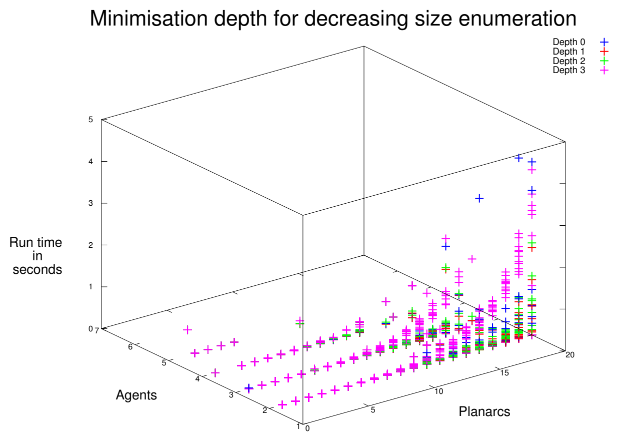


Figure 7.9: Run times for various planarc and agent set sizes while using the decreasing size enumeration strategy.

minimisation routine searches a much larger space before reaching a conclusion about a possible improvement when using larger values for the depth parameter.

The results show that the solution minimisation technique is effective in practice, although the choice for the value of depth depends on the strategy to use. Therefore our expectations in hypothesis H 7 are partially true. For the binary enumeration strategy a depth of 3 is preferable, while decreasing size performs better with a depth value of 1.

Pre-processing

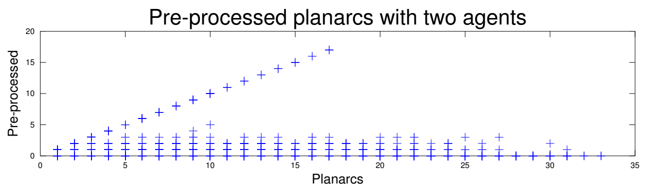
In section 5.2 we propose task removal as a pre-processing approach in solving the PC. This optimisation exploits the fact that some tasks can never be part of any coordination set constraint, because no inter-agent cycle can ever be constructed through that task. Such tasks can be removed from the instance and hence we have less tasks to consider in solving.

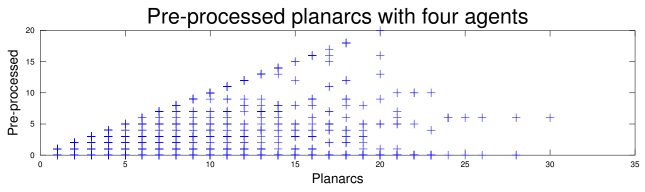
Removing tasks itself is not very interesting; as we have mentioned in section 7.1 this will not affect solver performance much. Nonetheless, removal of tasks can sometimes result in removal of planarcs, which is very interesting. Recall that the PC complexity is exponential in the number of planarcs and therefore reducing this set size is very beneficial.

We have illustrated the number of planarcs pre-processed for various agent set sizes in figure 7.10. Note in each figure the 'line' $|\hat{E}| = |preprocessed|$. Points on this line are caused by instances that have been pre-processed entirely. Such instances have no cycle in their agent dependency graph (ADG, see definition 2.4) and hence no inter-agent cycle can ever exist at all. For these instances the optimal coordination set is the empty set.

Our expectation in hypothesis H 8 concerning the number of planarcs does not seem to be verified by the results presented in figure 7.10. Although the number of pre-processed planarcs seems to decrease, it is not substantially proven by these results that there indeed exists such a correlation.

The size of the agent set, on the other hand, does seem to influence the number of planarcs that can be pre-processed. Indeed, for larger agent set sizes, more planarcs can be pre-processed, as we expected in hypothesis H 8. The most likely reason for this is that in instances with the same number of planarcs and larger agent set sizes, the chance of an agent not being part of any cycle in the ADG increases. Looking at the middle figure, we can see that a lot of instances allow a large number of planarcs to be pre-processed. In the bottom figure, however, most points are located on the x axis. This is because the number of agents is relatively large compared to the number of tasks and constraints, hence the probability of an agent not involved in any inter-agent cycle decreases.





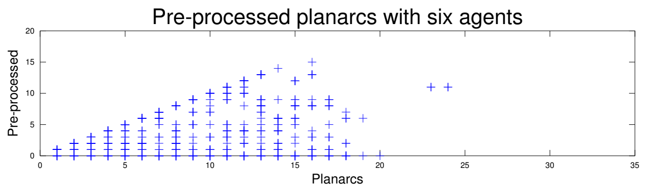


Figure 7.10: Pre-processed number of planarcs for different agent set sizes.

QBF encoding

We have implemented a small program that transforms PC instances into QBF encodings, using the transformation described in chapter 6. Basically it enumerates on coordination set sizes and for each set *k* it transforms the PC instance into a QBF formula that is satisfiable if and only if for some choices ($\exists$) of *k* coordination set variables and all choices ($\forall$) for the arcs no cycle can occur. We have used two QBF solvers to test satisfiability: QuBE7.0 and sKizzo. However, it quickly seemed that sKizzo was not able to cope with even small instances of our problem and hence we abandoned that solver.

Regretfully even with QuBE7.0 we have not been able to obtain any good quality results to which we can compare our own algorithms. Very small instances have been solved using QuBE7.0, however it already took 15 minutes to solve encodings of instances with only 10 tasks. Even our **pcpenum** solver was able to resolve them faster, mostly within a single minute. Hence with our approach we have to conclude hypothesis H 9 indeed holds: our specialised solver is able to tackle PC instances more effectively.

In chapter 6 we propose the incorporation of conditional variables in QBF solvers as discussed in [3], with which QBF solvers might be able to tackle PC alike instance a great deal faster. However in our thesis we do not further investigate this. Conditional variable might be a good subject for future research, as a lot of other QBF encoding might profit from such a modification.

Part III

Discussion

Chapter 8

Conclusions

Our research has focussed on the exact solving of the PLAN COORDINATION PROBLEM, a $\sum_2^p$-complete problem that is has to be solved when implicit coordination of multi-agent problems is possible. In chapter 3 to chapter 6 we have discussed several techniques to solve the PC exactly and we have compared them in chapter 7. There we also discussed the results we obtained from our experiments. In this chapter we continue this discussion by drawing conclusions on our work.

The combined solver works reasonably well

As we can see from the results, our combined solver is able to produce exact solutions to moderate size instances (10 to 50 planarcs) typically within the hour. Because the PC is a $\sum_2^p$-complete problem, we know that run times scale very badly in terms of the input; planarcs in our case. As its computational complexity is given by the product of two exponential problems, solving instances of such size can be seen as an achievement on its own.

Solving the PC exact is forbiddingly complex

Although we knew in advance that the PLAN COORDINATION PROBLEM was such a highly intractable problem, we did not expect it to scale this badly. Solving instances with more than 50 planarcs quickly requires the solver several days to come up with an exact answer. We therefore consider studying exact PC solving only valid from a theoretical point of view; it is very unlikely that such a complex problem also has practical applications, besides solving very small instances.

The complexity of PC instances depends most on the agent and planarc set sizes

From our experiments on hypothesis H 4 and hypothesis H 5 we have seen that the complexity of a PC instance mainly depends on the agent and planarc set sizes. For our solvers, the complexity increases exponentially for larger planarc sets while on the other hand the complexity decreases when the number of agents increases. The latter is caused by the dynamic programming approach of our CVP routine. The more agents we have (while keeping the number of planarcs constant), the smaller the sub problems become and therefore we can solve them much easier.

Approximation yields restrictive coordination sets

The theory in chapter 2 already mentions that the coordination set sizes return by approximation can be arbitrarily bad and this is confirmed by the results in chapter 7. Although approximation might seem the best way to go for the PC in terms of run time, the produced coordination is always much larger than the optimal coordination set size. Although we expected the approximated coordination set to become increasingly worse as the the number of planarcs increases (hypothesis H 2) we have not been able to find evidence to back up this claim.

Specialised solving outperforms generalised solving

In the last decade it has become increasingly popular to encode problems as a satisfiability problem and solve these encodings with highly optimised SAT solvers. Two of the many examples are temporal reasoning [36], and planing [17]. Because SAT solvers have been researched exhaustively, they incorporate some of the most innovative and effective strategies in NP-complete problem solving. Hence it might be worthwhile to encode a problem as a satisfiability problem to benefit from this.

As we have seen in section 7.3 this does not (yet) apply to our $\sum_2^p$-complete solver. Our specialised solver obtains far better results than the QBF encoding we introduced. Note however that with the modifications from [3] this might no longer be the case, however that might be subject of further study.

Chapter 9

Future Work

During our research we have encountered several leads for possible future studies, originating from the work we have performed. Some of them have been left out because there is simply not enough time to them include all in our study, others might be causes for additional detailed research. Below we have made a list of these subjects.

Experimental verification in practice

Aside from some manually constructed test instances, all of our experiments have been performed on randomly generated instances using a custom generator (see section 7.2). This generator did allow us to run the algorithms on instances of various numbers of agents, tasks, constraints and planarcs, however these generated instances are purely artificial. We are very interested to see how our solver performs on instances originating from practice, however this requires some method to transform multi-agent planning problems into the PC instances required in our solver.

Conditional QBF solving

From the experiments in chapter 7 we can see that verifying satisfiability of the QBF encoding we propose poses a difficult task for the QuBE7.0 solver [21] we used in our research. However we feel that the modification Anóstegui et al. discuss in [3] might greatly affect the QBF solvers run time.

In our encoding we used *indicator variables* that monitor the satisfiability of clauses. Combining all indicator variables into one *set indicator variable* can drastically reduce the number of clauses required in an encoding. Its drawback, however, is that the QBF solver is easily lead astray by such variables and spends a lot of time searching through the uninteresting part of the search space with indicator variables. If we would model these indicator variables as *conditional variables* as proposed in [3], we can prevent this from occurring. Considering that for n planarcs our encoding contains only $4n$ normal variables and an exponential number of indicator constraints, it is most likely that QBF profits enormously from such a modification. It would be worthwhile to compare such an adapted QBF solver against our combined solver.

Chapter 10

Summary

In this thesis we have studied solving the PLAN COORDINATION PROBLEM (PC) exactly using enumeration, dynamic programming, kernelisation and a QBF encoding of the problem. Our research has lead to a PC solver that combines the three first mentioned techniques to tackle moderately size instances within reasonable time. Moderate instances have tasks sizes between 10 and 50 planarcs, most of which our solver is able to tackle within at most one hour. For an intrinsically complex problem as PC, being a $\sum_2^p$-complete problem, we consider this a fairly good result. Indeed, compared to solving a QBF encoding of the problem, our solver performs several orders of magnitude better.

Below we will give a quick recapitulation on our study, chapter by chapter.

Chapter 1 introduces the PLAN COORDINATION PROBLEM and proposes a framework that we use to capture the problem formally. We introduce several important concepts that are assumed as preliminaries in the remainder of the thesis.

Chapter 2 provides us with a short overview of related work on the PC. Several authors [44, 39, 47, 45, 9] have studied the problem and acknowledged its intractability. Therefore only approximation algorithms have been proposed, of which we discuss the Depth Partitioning algorithm by Steenhuisen et al. [38] and Intra-free Coordination by Yadati et al. [47].

After these introductory chapters, we discuss the first exact solving method in chapter 3. In this chapter we study simple enumeration as a starting point for our further research. As we see later in the thesis, enumeration is indeed not very effective at solving PC instances. Nevertheless by using a search strategy optimisation, we can make enumeration more competitive. Indeed, our combined solver still relies on the optimised enumeration algorithm for the PC proposed in this chapter.

In chapter 4 we explain our second algorithmic approach, dynamic programming, to solve the PC exact. Although the dynamic programming method cannot be used to improve the PC part of the problem, the COORDINATION VERIFICATION PROBLEM sub part benefits greatly from this technique. In our approach we halve the set of agents until only one-agent problems remain, which can be solved easily. To this end we introduce summary constraints that, for each time we halve the agent set, capture the creation possible inter-agent cycles through the other half. This enables us to split up the agent set and preserve the context of the agent within the entire problem.

From the research and experiments with the two techniques mentioned above, we have been able to identify several kernelisation possibilities. In chapter 5 we summarise these and discuss how they can be combined with previous techniques to obtain a very promising solver for PC. We also describe the identification of fixed constraints, which eventually lead to near optimal solutions.

As we are the first to study exact solving of the PC, there are no means of comparing our results. Chapter 6 therefore introduces an encoding of the PC as a QUANTIFIED BOOLEAN FORMULA PROBLEM. This way we can use any QBF solver, such as for example Qube++ [22] or sKizzo [6], to solve PC instances and we can compare our work against solvers originating from the vast literature of satisfiability solving. A test which our solver has passed with flying colours.

Chapter 7 discusses our experiments and the results we have obtained. From these experiments we see that our combined solver is able to tackle moderately sized instances within reasonable time. Regretfully the encoding we proposed for QBF solving proved to give both the QuBE7.0 [21] and sKizzo [6] a hard time and we were not able to reasonably compare this to our own solver.

In chapter 8 we draw conclusions from our work also based on the experiments from the chapter before. Our conclusions are that although we have developed a PC solver that performs reasonably well, solving the PC exact is forbiddingly complex and thus time consuming. On the other hand, approximation results in very restrictive coordination sets for the problem. In addition, our specialised algorithm for PC performs better than a generated encoding for QBF, which for now votes against generalised problem solving using QBF as is done often for SAT.

Finally chapter 9 lists some open questions for future research that originated from our work.

Appendices

Appendix A

Background Knowledge

This appendix provides the reader with a refresher of some important concepts encountered in solving the plan cooridnation problem. These sections are not meant as a comprehensive summary of the entire theory; only the subjects required for the understanding of this thesis are included. For further reading on the discussed topics, we refer to the cited articles.

A.1 Approximation Classes

As with time and space complexities, we can also identify several distinct classes of approximability. These *approximation classes* define levels of 'quality' of approximating a problem. The class NPO is the class of polynomially approximable problems, i.e. for all problems in NPO there exists an algorithm that approximates the problem in polynomial time. For any problem in NPO we do not necesarily have any guarantees on the quality of the approximated solution. These guaratees are provided by the sub classes of NPO in terms of the *approximation ratio*, defined in definition A.1

Definition A.1: Approximation Ratio (Kann [25])

The approximation ratio *for any given problem* Π *provides a bound on the relative error of any solution* δ *for* Π *in respect to the optimal solution* OPT_Π *for that problem. This ratio r is defined as:*

$$r = \begin{cases} OPT_\Pi/\delta, & \textit{if } \Pi \textit{ is a minimisation problem} \\ \delta/OPT_\Pi, & \textit{if } \Pi \textit{ is a maximisation problem} \end{cases}$$

We say that an r-approximation algorithm for a given minimisation problem approximates the problem with an error of at most r times the optimal solution. This error is at most $1/r$ *when dealing with a maximisation problem.*

Based on this approximation ratio we can identify several sub classes of NPO.

The most important classes are APX and its sub classes PTAS and FPTAS. The class APX is the class of problems that allow for constant factor approximation algorithms, but not necesarily an integer constant. Problems in this class might as well have an approximation ratio that is relative to the size of the input.

The problems in PTAS (Polynomial-Time Approximation Scheme) provide even better guarantees. A minimisation problem in PTAS can be approximated with a ratio of $1+\varepsilon$ ($1-\varepsilon$ for maximisation problems) for any $\varepsilon > 0$, nevertheless the exponent of its polynomial complexity profile depends on ε and might be intractable for small ε.

The easiest known approximation algorithms with a guaranteed error bound are in FPTAS (Fully Polynomial-Time Approximation Scheme). The approximation ratio for such minimisation problems is also given by $1+\varepsilon$ (again $1-\varepsilon$ for maximisation) for any $\varepsilon > 0$, however now the time complexity only depends on $1/\varepsilon$. This latter class hence provides the same freedom in choosing approximation ratio but its time complexity does not increase as quickly as PTAS when selecting smaller values for ε.

The classes is NPO are related in the following way:

$$FPTAS \subseteq PTAS \subseteq APX \subseteq NPO$$

These relations are strict if $P \neq NP$, however this is still an open problem. In chapter 2 we mention that the minimal PLAN COORDINATION PROBLEM is an APX-hard problem. This means that the problem is a member of the APX class but not of any of its sub classes and hence there exists no integer constant approximation algorithm for the minimal PC unless P = NP. As a consequence we can only find approximation algorithms that bound the relative error in terms of the input size n.

As an example of this lets assume that we have an $O(\log n)$-approximation algorithm for the PC. The larger n becomes, the more inacurate the results of our approximation algorithm will be. For an instance of size 8, the algorithm will find a solution that is at most $\log 8 = 3$ times as large as the optimum. But when the size goes up to 16 we already have that the algorithm might produce a solution that is $\log 16 = 4$ times as large as the optimum. This is why we prefer approximation algorithms with a integer constant, however as Ter Mors et al. prove in [44] none such an algorithm exist for the PLAN COORDINATION PROBLEM unless P = NP.

A.2 Ordered Sets

In the context of this thesis only two concepts of set theory are of importance: partially and totally ordered sets. A partially ordered set is a set that is ordered by some binary operator. The term partially is used to indicate that not all pairs of elements need to be related by this binary operator. When this is the case, we speak of a totally ordered set. Moreover, we are only interested in the *strict* sets for which the *irreflexive* property holds. The formal definition of both sets are given in definition A.2.

Definition A.2: Strict Partially and Totally Ordered Set (Schröder [37])

A strict partially ordered set *(or* strict poset*) is an ordered pair* $(P, \circ)$ *of a set P and a binary relation* $\circ$ *contained in* $P \times P$*, called the order on P, such that:*

(i). *The relation* $\circ$ *is irreflexive, that is for every* $p \in P$ *we have* $\neg(p \circ p)$.

(ii). *The relation* $\circ$ *is antisymmetric, that is for every* $p, q \in P$ *we have* $(p \circ q) \implies \neg(p \circ q)$.

(iii). *The relation* $\circ$ *is transitive, that is for every* $p, q, r \in P$ *we have* $(p \circ q) \wedge (q \circ r) \implies (p \circ r)$.

The set P is called strict totally ordered *if in addition to the above we have that for all* $p, q \in P$ *either* $p \circ q$ *or* $q \circ p$ *holds.*

A plan for the task based planning problem is such a partially ordered set, ordered by the binary relation $\prec$.

A.3 Polynomial Hierarchy

The polynomial-time hierarchy, or polynomial hierarchy in short, is a recursive generalisation of the complexity classes P, NP and co-NP in terms of oracle machines. For a very detailed discussion on this subject we refer the reader to the article by Stockmeyer [41]. This appendix only discusses the basics required for comprehension of the polynomial hierarchy and its consequences for problem complexity. The polynomial hierarchy is defined in definition A.3.

Definition A.3: Polynomial Hierarchy (Stockmeyer [41])
We define $\Delta_0^P = \sum_0^P = \prod_0^P = P$*, where P is the set of problems solvable in polynomial time. Then the* polynomial hierarchy *for any natural number* $i \geq 0$ *is defined by:*

(i). $\Delta_{i+1}^P = P^{\Sigma_i^P}$

(ii). $\sum_{i+1}^P = NP^{\Sigma_i^P}$

(iii). $\prod_{i+1}^P = co\text{-}NP^{\Sigma_i^P}$

in which A^B *denotes the set of problems in complexity class A augmented by an oracle of class B.*

In this report we only encounter the class $\sum_2^p$-complete. According to the polynomial hierarchy $\sum_2^p$ equals $\mathrm{NP}^{\mathrm{NP}}$. Thus $\sum_2^p$-complete problems are non-deterministic solvable with the use of an non-deterministic oracle. This is the case when we have an NP-complete algorithm that uses another NP-complete algorithm as a sub routine in *each iteration*. In terms of problem solving this means that such problems are highly intractable even for very small instances. In NP-complete problems we can sometimes still solve exactly when the problem size is small, however solving $\sum_2^p$-complete problems exactly is prohibiting due to its complexity.

A.4 Student's t-test

In order to make any claims about differences between two distinct variables we need to prove that such a claim is indeed statistically valid to make. In this thesis we use the Student's t-test [15] to support or reject null hypotheses concerning the relation between two independent variables.

Using the Student's t-test, we can determine whether some correlation between two variables is caused by a mere coincidence or is likely to exist. In the former case, we must reject our null hypothesis because ther is not enough statistical evidence for it. In the latter case we can say that a correlation exists *with high probability*.

To perform a t-test on two different data sets A and B we first compute the means and sum of squared deviations using the formulas below. In these formulas we have data set $X = x_1, x_2, \ldots, x_n$ of size n_x, μ_X represents the mean and SS_X the sum of squared deviation of X.

$$\begin{aligned} \mu_X &= \frac{\sum x_i}{n_x} \\ SS_X &= \sum (x_i - \mu_X)^2 \end{aligned}$$

We can perform a t-test over the combination of data sets A and B by estimating the combined mean and standard deviation using the formulas:

$$\begin{aligned} \hat{\mu} &= \mu_A - \mu_B \\ s_p^2 &= \frac{SS_A + SS_B}{(n_A - 1) + (n_B - 1)} \\ \hat{\sigma} &= \sqrt{\frac{s_p^2}{n_A} + \frac{s_p^2}{n_B}} \end{aligned}$$

Using these combined mean and standard deviation we can compute the t ratio and the sampling size $\hat{n}$:

$$\begin{aligned} t &= \frac{\hat{\mu}}{\hat{\sigma}} \\ \hat{n} &= (n_A - 1) + (n_B - 1) \end{aligned}$$

Having computed the values of t and $\hat{n}$ we can use a statistical tool to compute either the left tail probability ($\mu_A < \mu_B$), right tail probability ($\mu_A > \mu_B$) or non-directional probability ($\mu_A = \mu_B$) for various significance levels α, depending on our null hypothesis. We can accept a null hypothesis if t is smaller than the corresponding probability, i.e. left tail, right tail or non-directional, we find at significance level α. In our thesis we have used a significance level $\alpha = 0.5$.

List of Algorithms

Depth Partitioning algorithm 27
Intra-Free Approximation algorithm 31
Enumeration algorithm for the PLAN COORDINATION PROBLEM. 37
Enumeration algorithm for the COORDINATION VERIFICATION PROBLEM. . . . 40
Cycle existence detection algorithm 41
Dynamic Programming algorithm for the COORDINATION VERIFICATION PROBLEM. 57
Summary creation algorithm for the CVP DP algorithm. 60

List of Figures

1.1 Planarcs 14
1.2 Example dependency graph 15
1.3 Path blocking gadget 21
1.4 Forced choice gadget 23

2.1 Depth Partitioning worst case 28

3.1 CVP instance falsely declared Not Coordinated. 39
3.2 Possible configurations in enumeration 44

4.1 Summary constraints 51
4.2 Example of non-optimal DP solution. 52
4.3 Recurrence tree for agent based DP 52
4.4 Local cycle during summary creation 63

5.1 Coordination of fixed constraints is not optimal 70
5.2 Optimal set for the same instance 70

7.1 Depth partitioning versus Intra-free coordination 88
7.2 Approximation coordination set size versus optimal. 88
7.3 Solver run time comparison 89
7.4 Agents and planarc versus run time 91
7.5 Run times per strategy, $r < .5$ 92
7.6 Run times per strategy, $.25 < r < .75$ 93
7.7 Run times per strategy, $r > .5$ 93
7.8 Minimisation depth for binary enumeration 95
7.9 Minimisation depth for decreasing size enumeration 95
7.10 Pre-processed number of planarcs 97

Bibliography

[1] F.N. Abu-Khzam, R.L. Collins, M.R. Fellows, M.A. Langston, W.H. Suters, and C.T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. Citeseer.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[3] C. Ansótegui, C.P. Gomes, and B. Selman. The Achilles' heel of QBF. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 20, page 275. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[4] B. Becker and T. Schubert. SAT, SMT, and QBF Solving in a Multi-Core Environment. 2009.

[5] R. Bellman. The theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–719, 1952.

[6] M. Benedetti. sKizzo: a suite to evaluate and certify QBFs. *Automated Deduction–CADE-20*, pages 369–376.

[7] JC Bermond and C. Thomassen. Cycles in digraphs-a survey. *Journal of Graph Theory*, 5(1):1–43, 2006.

[8] Pieter Buzing, Adriaan ter Mors, Jeroen Valk, and Cees Witteveen. Task coordination for non-cooperative planning agents. In C. Ghidini, P. Giorgini, and W. van der Hoek, editors, *Proceedings of the 2nd European Workshop on Multi-Agent Systems (EUMAS 2004)*, pages 87–98, dec 2004.

[9] Pieter Buzing, Adriaan ter Mors, Jeroen Valk, and Cees Witteveen. Coordinating self-interested planning agents. *Autonomous Agents and Multi-Agent Systems*, 12:199–218(20), March 2006.

[10] T. Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, volume 1, pages 274–279. Citeseer, 1991.

[11] M.C. Cooper, D.M. Lambert, and J.D. Pagh. Supply chain management: more than a new name for logistics. *The International Journal of Logistics Management*, 8(1):1–14, 1997.

[12] J.S. Cox and E.H. Durfee. Efficient mechanisms for multiagent plan merging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1342–1343. IEEE Computer Society Washington, DC, USA, 2004.

[13] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[14] K.S. Decker and V.R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multi-Agent Systems*, volume 73, page 80, 1995.

[15] M. Dekking, C. Kraaikamp, and HP Lopuhaa. *A modern introduction to probability and statistics: understanding why and how*. Springer Verlag, 2005.

[16] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems in directed graphs* 1. *Information Processing Letters*, 86(3):129–136, 2003.

[17] M. Do, B. Srivastava, and S. Kambhampati. Investigating the effect of relevance and reachability constraints on SAT encodings of planning. In *Proc. 5th International Conference on AI Planning and Scheduling*, page 982, 2000.

[18] R.G. Downey and M.R. Fellows. *Parameterized complexity*. Springer New York, 1999.

[19] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.

[20] HN Gabow, SN Maheshwari, and LJ Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, pages 227–231, 1976.

[21] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. *Automated Reasoning*, pages 364–369.

[22] E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube++: An efficient qbf solver. In *Formal Methods in Computer-Aided Design*, pages 201–213. Springer.

[23] JR Harrald. Supporting agility and discipline when preparing for and responding to extreme events. *Proc. of the 2nd ISCRAM*, 2005.

[24] AB Kahn. Topological sorting of large networks. 1962.

[25] V. Kann. *On the approximability of NP-complete optimization problems*. Citeseer.

[26] R.M. Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, 43:85–103, 1972.

[27] S. Kraus and T. Plotkin. Algorithms of distributed task allocation for cooperative agents. *Theoretical Computer Science*, 242(1-2):1–27, 2000.

[28] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30(1):119–169, 2000.

[29] G. Loizou and P. Thanisch. Enumerating the cycles of a digraph: A new preprocessing strategy. *Information Sciences*, 27(3):163–182, 1982.

[30] D. McAllester and D. Rosenblitt. Systematic nonlinear planning, 1991.

[31] M. Narizzano, C. Peschiera, L. Pulina, and A. Tacchella. Evaluating and certifying QBFs: A comparison of state-of-the-art tools. *AI Communications*, 22(4):191–210, 2009.

[32] M. Narizzano and A. Tacchella. QDIMACS prenex CNF standard ver. 1.1, 2005. *Available on-line from http://www.qbflib.org/qdimacs.html*.

[33] D. Nau, Y. Cao, A. Lotem, and H. Muftoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[34] G.L. Nemhauser and L.A. Wolsey. *Integer and combinatorial optimization*. Wiley New York, 1999.

[35] C. Otwell, A. Remshagen, and K. Truemper. An effective QBF solver for planning problems. *Proc. MSV/AMCS*, pages 311–316.

[36] D. Pham, J. Thornton, and A. Sattar. Towards an efficient SAT encoding for temporal reasoning. *Principles and Practice of Constraint Programming-CP 2006*, pages 421–436, 2006.

[37] B.S.W. Schröder. *Ordered sets: an introduction*. Birkhauser, 2002.

[38] J. Renze Steenhuisen and Cees Witteveen. Plan decoupling of agents with qualitatively constrained tasks. *Multiagent and Grid Systems*, 5(4), dec 2009.

[39] J. Renze Steenhuisen, Cees Witteveen, Adriaan ter Mors, and Jeroen Valk. *Framework and Complexity Results for Coordinating Non-cooperative Planning Agents*, pages 98–109. Springer Berlin / Heidelberg, 2006.

[40] J. Renze Steenhuisen, Cees Witteveen, and Yingqian Zhang. Plan-coordination mechanisms and the price of autonomy. *Computational Logic in Multi-Agent Systems*, 5056/2008:1–21, 2008.

[41] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1 – 22, 1976.

[42] O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding Separation Formulas with SAT.

[43] E. Tardos and J. Kleinberg. *Algorithm Design*. Reading (MA): Addison-Wesley, 2006.

[44] Adriaan ter Mors, Cees Witteveen, and Jeroen Valk. Complexity of coordinating autonomous planning agents. Technical Report 2004-002, Delft University of Technology, 2004.

[45] J.M. Valk. *Coordination among Autonomous Planners*. PhD thesis, Delft University of Technology, 2005.

[46] D.B. Wagner. Dynamic programming. *The Mathematica Journal*, 5(4):42–51, 1995.

[47] Chetan Yadati, Cees Witteveen, and Yingqian Zhang. Coordinating agents: An analysis of coordination in supply-chain management like tasks. In *The 2nd International Conference on Agents and Artificial Intelligence (ICAART)*, 2010. To appear.

[48] Robert Zlot and Anthony Stentz. Market-based multirobot coordination for complex tasks. *The International Journal of Robotics Research*, 25(1):73–101, January 2006.