UNIVERSITÉ JOSEPH FOURIER - GRENOBLE 1
SCIENCES. TECHNOLOGIE. MÉDECINE

THÈSE

pour le grade de **Docteur**
de l'université Joseph Fourier
Discipline : **Informatique**

préparée au laboratoire VERIMAG - Institut IMAG

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement le 2 Octobre 2006 par

Abdelkarim Aziz KERBAA
Ingénieur d'état en Recherche Opérationnelle
DEA Informatique
DEA Recherche Opérationnelle

TITRE

Stratégies d'Ordonnancement Conditionnelles Utilisant des Automates Temporisés

JURY

Anatoli IOUDITSKI,	Président
Eric RUTTEN,	Rapporteur
Eugène ASARIN,	Rapporteur
Oded MALER,	Directeur de thèse
Marius Dorel BOZGA,	Co-directeur de thèse
Yasmina ABDEDDAÏM,	Examinateur

Conditional Scheduling Strategies Using Timed Automata

Abdelkarim Aziz KERBAA

October 2, 2006

Remerciements

Je voudrais exprimer ma gratitude à Oded Maler, Directeur de Recherche au CNRS et chercheur au laboratoire Verimag, pour m'avoir donné l'opportunité de faire cette thèse. Je le remercie en particulier pour sa disponibilité et pour les longues discussions qui ont permis de jeter les bases de modélisation de ce travail. Ce fut la période où il fallait se montrer très engagé par rapport aux Automates Temporisés. Je le remercie d'avoir toujours été impliqué dans mon sujet et de la liberté et l'autonomie qu'il a fini par m'accorder pour mener mes recherches dans des directions qui me plaisaient particulièrement et qui m'ont permis d'aboutir aux types de résultats que je voulais au départ pour cette thèse. Tout ceci, m'a permis donc de porter un double regard sur les problèmes d'ordonnancement, un qui tient plus à l'informatique et basé sur les automates, puis l'autre qui trouve aussi son originalité dans certains aspects de la Recherche Opérationnelle, ma discipline d'origine, et c'est heureux ... Enfin, voilà Oded, merci pour tout.

Je me dois de remercier particulièrement Marius Bozga, Chercheur à Verimag pour avoir accepté de co-diriger cette thèse. Il a toujours montré sa disponibilité et fait preuve de beaucoup de patience lors des discussions souvent longues concernant mes recherches. Par ses conseils avisés et son grand savoir faire en matière d'implémentation, et les cours qu'il n'a pas hésité à me donner au début, j'ai appris à mieux concevoir mes programmes, ce qui m'a permis de bien mettre en oeuvre mes résultats de recherches. Je le remercie d'avoir été à l'écoute des idées que je pouvais proposer, d'avoir su encourager les orientations que j'ai pu donner à mes recherches et pour ses critiques constructives appréciables qui ont permis de faire aboutir cette thèse. Ce fut pour moi un privilège et un grand plaisir d'avoir travaillé dans une ambiance toujours joviale avec un esprit aussi instructif. Merci Marius, merci beaucoup.

Je tiens à exprimer ma reconnaissance à Anatoli Iouditski, Professeur à l'Université Joseph Fourier, pour m'avoir fait l'honneur de présider le jury de cette thèse.

Je remercie Eric Rutten, Chercheur à l'INRIA, pour avoir accepté d'examiner

ce travail.

Eugène Asarin, Professeur à l'Université Paris 7 et ancien membre de l'équipe, trouvera ici l'expression de mes remerciements.

Je n'aurai garde d'oublier tout le personnel administratif de Verimag, de l'école doctorale et de la bibliothèque de l'IMAG.

Je remercie toutes les personnes sympathiques parmi mes camarades thésards et autres, pour toutes les pauses café passées à parler d'informatique, d'avenir, du temps qu'il va faire demain, des choses de la vie, de tout et n'importe quoi, ainsi que tous ceux parmi les chercheurs qui m'ont témoigné leur sympathie au cours de mon séjour scientifique parmi eux.

Plus personnellement, je remercie mes amis de Grenoble et les personnes que j'ai pu rencontrer, je ne peux tous les citer, qui m'ont témoigné leur affection et leur sympathie durant ce séjour, spécialement ceux que j'ai connu à la cité universitaire du village olympique et à la résidence Ouest, pour les heureux moments qu'on a pu partager et toutes les sorties qui m'ont permis d'oublier le stress de cette pénible épreuve.

Finalement, que Dieu bénisse mes parents. Ils m'ont toujours encouragé et aidé tout au long de mon cursus. Parce qu'ils ont su me donner toutes les chances pour réussir, qu'ils trouvent dans l'achèvement de ce travail l'aboutissement de leurs efforts et sacrifices ainsi que l'expression de mon éternelle gratitude. Ce travail a été fait en pensant à eux et je n'ai fait que suivre leur exemple, il leur est donc naturellement dédié ainsi qu'à tous les membres de ma famille.

Table des matières

Introduction

Le problème d'ordonnancement optimal peut être défini comme étant la recherche d'une allocation des ressources sur l'axe de temps pour exécuter un ensemble de tâches afin de minimiser certains critères sous certaines contraintes. C'est un problème d'optimisation combinatoire dont l'essence est l'ensemble des tâches et des ressources. Les problèmes d'ordonnancement viennent de la Recherche Opérationnelle et sont formulés dans différents contextes, les plus représentatifs étant les systèmes informatiques, la productique et l'ordonnancement de projets [Bak74, Jr.76, Pin95, BEP+96, Bru97].

Les contraintes de précédences constituent une partie importante de tout problème d'ordonnancement et surgissent dans différents domaines. Elle peuvent être utilisées en productique pour exprimer un ordre technologique comme par exemple dans le problème d'ordonnancement d'ateliers de type job-shop [Fis73], ou peuvent surgir sous forme de dépendance entre tâches dans le contexte de l'ordonnancement de projets ou en programmation parallèle [Gra66, Man67, ACD74, KI99b, TKK00]. Les contraintes de précédences ont deux représentations classiques utilisant les graphes orientés sans circuit (DAG pour *Directed Acyclic Graph*). La première est le graphe *tâche-sur-arc*, où les arcs représentent les tâches et les noeuds représentent des évènements temporels dans un ordonnancement. La deuxième est le graphe *tâche-sur-noeud* où les noeuds correspondent aux tâches et les arcs capturent les précédences entre ces dernières. La première est principalement utilisée dans le contexte d'ordonnancement de projets [TP78, Weg99], alors que la dernière est largement utilisée en productique ainsi qu'en ordonnancement de programmes informatiques [Jr.76, Pin95, BEP+96, Bru97]. Aussi, en programmation parallèle, le DAG est un modèle abstrait employé pour représenter le diagramme de flots de données et est habituellement extrait automatiquement du programme

pendant la phase de compilation.

Les méthodes exactes pour résoudre le problème d'ordonnancement DAG sont essentiellement basées sur la programmation dynamique [RCG72] et la programmation mathématique. Cependant, ce problème étant NP difficile au sens fort [GJ79], il demeure impossible (à moins que P=NP) de le résoudre de manière polynomiale, ni même de lui trouver un schéma d'approximation pleinement polynomial. C'est pourquoi un bon nombre de recherches ont été menées dans l'objectif de trouver des solution proches de l'optimum dans un temps relativement raisonnable [Gra66, Man67, ACD74, TKK00]. La plupart de ces méthodes sont basées sur les algorithmes de listes de priorités.

Un cadre de travail pour exprimer et résoudre les problèmes d'ordonnancement en utilisant les automates temporisés a été développé au cours des dernières années [Abd02], [AAM06], [AM02]. L'automate temporisé [AD94] est l'outil naturel pour modéliser l'évolution de l'*état* du problème d'ordonnancement comme résultat d'actions discrètes (commencement et terminaison d'une tâche) et du passage de temps. Des outils tels que Kronos [Yov97], IF [BFG+99, BGM02] et Uppaal [LPY97] exploitent le fait que le problème d'atteignabilité pour les automates temporisés est décidable et est ainsi justiciable à l'algorithmique de vérification (model-checking). Bien que la motivation initiale pour les automates temporisés fût la vérification des propriétés qualitatives, ce travail donnera une évidence additionnelle quant à l'applicabilité de tels modèles à certains problèmes d'ordonnancement de programmes, qui suggèrent parfois une meilleure qualité de solutions que celle basées sur les listes de priorités.

En pratique, il existe des situations habituelles qui présentent de l'incertitude par rapport à l'ensemble des tâches devant être exécutées. Par exemple, on peut avoir le cas où les tâches doivent être ordonnancées seulement sous certaines conditions bien spécifiques. Cette situation est typique des programmes contenant des instructions *if-then-else*. Malheureusement, le modèle basique du graphe de tâches ne capture pas de tels comportements conditionnels. L'incertitude conditionnelle dans l'ordonnancement des programmes a retenu plus d'attention ces dernières années ; les lecteurs intéressés peuvent trouver des résultats préliminaires dans [KW00].

Dans cette thèse nous considérons le problème de l'ordonnancement efficace des programmes conditionnels sur une architecture de processeurs homogènes et parallèles. Nous introduisons une nouvelle représentation appelée Graphe de Précédences Conditionnel (CPG pour *Conditional Precedence Graph*), qui est une extension du modèle DAG pour représenter le problème d'ordonnancement conditionnel. Cette représentation couvre deux types de contraintes : contraintes de précédences et contraintes d'activation. L'approche traditionnelle de la programmation par contraintes a été employée pour résoudre un problème similaire dans [KW02]. Dans ce travail, nous étendons le cadre de travail des automates temporisés pour modéliser et résoudre le problème d'ordonnancement conditionnel [BKM04]. L'espace d'états est représenté par un automate de jeux temporisé défini comme un produit d'interaction d'automates pour chaque tâche et dans lequel on cherche des stratégies pire cas optimales. Plusieurs techniques de recherche sont proposées et évaluées. En outre, pour améliorer l'efficacité d'une telle approche, nous présentons de nouveaux résultats théoriques concernant les propriétés de domination entre différents types de stratégies, et qui seront utilisés pour renforcer le modèle initial dans le but de restreindre l'espace d'état de recherche. Nous investiguerons l'efficacité d'une telle méthodologie d'un point de vue expérimental sur un ensemble de benchmarks, puis nous apporterons un nouveau résultat d'approximabilité afin de montrer la qualité des solutions obtenues d'un point de vue théorique.

Organisation de la thèse

La thèse comporte deux parties. Dans la première on étudie l'ordonnancement sur graphe de tâches *déterministe*. Cette première partie est organisée comme suit :

Chapitre 2 : On introduit le problème d'ordonnancement sur graphe de tâches déterministe. Différents types d'ordonnancement sont décrits, formalisés et classifiés selon un critère de domination. Enfin, la qualité théorique des ordonnancements de listes est donnée.

Chapitre 3 : Ce chapitre décrit notre approche globale. Celle-ci consiste à transformer le problème en produit d'automates tel que les traces d'exécution

dans l'automate global correspondent aux ordonnancements réalisables.

Chapitre 4 : Ce chapitre donne un rappel sur les techniques exactes de recherche de plus court chemins dans un graphe acyclique positivement pondéré.

Chapitre 5 : Ce chapitre présente plusieurs heuristiques basées sur une recherche guidée qui permet de trouver de bons chemins en un temps relativement raisonnable. La fonction d'estimation utilisée est décrite ainsi que ses propriétés. Enfin, les résultats expérimentaux sur des exemples de benchmarks sont présentés à la fin du chapitre.

Dans la seconde partie du document, du *non-déterminisme* est introduit dans le problème. Le type d'incertitude qu'on étudie provient du fait que l'ensemble des tâches devant être exécutées dépend essentiellement du résultat d'autres tâches qui ne devient connu qu'après leurs terminaison. C'est une situation typique des programmes contenant des instructions *if-then-else*. Cette seconde partie est organisée comme suit :

Chapitre 6 : Ce chapitre introduit le problème aux travers d'un exemple de programme. Après avoir donné la façon d'ordonnancer (clairvoyant vs. non-clairvoyant), on donne le modèle CPG qui sert à spécifier le programme avec une sémantique opérationnelle bien définie.

Chapitre 7 : Ce chapitre présente le modèle d'automates temporisés augmentés par des automates booléens, qui sera ensuite réduit en se focalisant sur les ordonnancements non-lazy et immédiats sur un graphe de jeux orienté et pondéré. Des propriétés de domination entre les différents types de stratégies sont montrées ainsi qu'une preuve formelle montrant que la déviation des stratégies *sans attente* de l'optimum est au maximum 2. Enfin, on présente l'adaptation de la décomposition en chaînes pour un codage efficace de l'automate produit.

Chapitre 8 : Ce chapitre décrit les algorithmes exacts et approchés de recherche dans les graphes de jeux. Il présente les façons les plus prometteuses de recherche en avant sans avoir à construire tout le graphe. La recherche est guidée par une

fonction d'estimation qui est expliquée en détail ainsi que ses propriétés. Enfin, les résultats expérimentaux et théoriques sont présentés afin d'évaluer la qualité des ordonnancements obtenus par notre approche.

Chapitre 1

Introduction

The problem of optimal scheduling can be defined as searching for an allocation of ressources over the time axis to execute a set of tasks in order to minimize some criterion under some specific constraints. It is a combinatorial optimization problem for which the essence are tasks and ressources. The scheduling problems come from Operational Research, and are formulated in several contexts, the most representative being computer systems, manufacturing and project scheduling [Bak74, Jr.76, Pin95, BEP+96, Bru97].

Precedence constraints are an important part of any scheduling problem and arise in different areas. They can be used in manufacturing to express a technological order like in the job-shop scheduling problem [Fis73], or may rise up in the form of tasks dependencies in the context of project scheduling or parallel processing of computer programs [Gra66, Man67, ACD74, KI99b, KI99a, TKK00]. Precedence constraints have two classical representations using directed acyclic graphs (DAG). The first one is the task-on-arc graph, where arcs stand for tasks and nodes represent time events in a schedule, and the second is the task-on-node graphs where nodes correspond to tasks and arcs capture precedence between them. The former is principally used in the context of projects scheduling [TP78, Weg99], while the latter is widely used in manufacturing and computer scheduling problems [Jr.76, Pin95, BEP+96, Bru97]. In parallel processing, the DAG is an abstract model used to represent the dataflow chart of the program and is usually extracted during the compilation process.

Exact methods for solving DAG scheduling problem are essentially based on

dynamic programming [RCG72] and mathematical programming. However, since this problem is shown to be strongly NP hard [GJ79], it is impossible (unless P=NP) to solve it polynomially, or to get a fully polynomial time approximation scheme. This is why a large body of research was conducted toward getting near optimal solutions within a reasonable execution time [Gra66, Man67, ACD74, TKK00]. Most of these methods are based on list scheduling.

In the last couple of years a framework for expressing and solving scheduling problems using timed automata has been developed [Abd02], [AAM06], [AM02] and have been applied successfully to different case studies [NY00, BF01]. The timed automaton [AD94] is the natural tool for modeling the evolution of the *state* of the scheduling problem as a result of discrete actions (starting or ending a task) and of the passage of time. Tools such as Kronos [Yov97], IF [BFG$^+$99, BGM02], and Uppaal [LPY97] exploit the fact that the reachability problem for timed automata is decidable and hence can be subject to algorithmic verification (model-checking). Although the initial motivation for TA models was verification of qualitative properties, as for example in reactive systems [KY03, ACMR03, MR02, AGS02], this work will give additional evidence for the applicability of such models to certain problems of programs scheduling, which suggest, sometimes, a better quality of solutions than classical approaches based on priority lists.

In real world applications, there are usual situations that present uncertainty with respect to the set of tasks to be executed. For example, it may be the case that tasks have to be scheduled only under some specific conditions. This typical situation is in scheduling of programs containing *if-then-else* instructions. Unfortunately, the basic model of task graph do not capture such conditional behaviour. Conditional uncertanty in program scheduling has retained more attention in last few years ; readers interested can find some preliminary results in [KW00].

In this thesis we consider the problem of efficiently scheduling conditional programs on architecture of parallel homogeneous processors. We introduce a new representation named Conditional Precedence Graph, which is an extension of the DAG model, to represent conditional scheduling problem. In this representation, we have two types of constraints : precedence constraints and activation constraints. Traditional constraint programming approach has been used to solve similar problem in [KW02]. In this work, we extend the timed automata framework to model and solve conditional scheduling problem [BKM04]. The state

space is represented by a timed game automaton which is a product of interacting automata for each task, and in which we search for the optimal worst case strategies. Several search techniques are proposed and evaluated. In addition, to improve the efficiency of such an approach, we present new theoretical results concerning domination properties between different types of strategies, and will use them to strengthen the initial model in order to restrict the state space search. We investigate the efficiency of such a methodology from experimental side on a set of benchmarks, and come up with a new approximability result in order to demonstrate the quality of the obtained solutions from theoretical aspect.

Organization of the thesis

The first part of the thesis studies *deterministic* task graph scheduling. We introduce the problem in Chapter 2 and present some known results about different classes of solution and the the worst-case performance of (greedy) list scheduling. In Chapter 3, we present timed automata and show how they can be used to model the problemn and reduce its solution to finiding shortest paths in directed weighted graphs. In Chapter 4 we survey exact algorithms for finding shortest paths followed, in Chapter 5, by a discussion of several heuristics for guiding the search with and without guarantee of optimality. Finally, we discuss their theoretical quality and test their empirical performance on a set of benchmark examples.

The second part is dedicated to the problem of conditional scheduling where the set of tasks to be executed is not known completely in advance but becomes known progressively as other tasks terminate. In chapter 6 we give the problem statement and introduce the conditional precedence graph model. The modeling framework based on timed game automata is presented in Chapter 7 as well as classification of strategies and approximability results. Finally we present exact and heuristic algorithms for searching game graphs with and without optimality (Chapter 8) and test them on a set of examples.

Première partie

Deterministic Task Graph Scheduling

Chapitre 2

Task-Graph Scheduling

2.1 Problem Definition

In non technical terms, the problem of scheduling can be phrased as follows. There is some quantity of work to be performed. This quantity is distributed over several units of work that we call *tasks*, each having a *duration*, the time it takes to terminate it once started. Tasks are related to each other by *precedence constraints* that prevent some tasks from being performed before other tasks have terminated. The scheduling problem is to assign to each task an interval of time in which it is executed while respecting the precedence constraints and optimizing some performance measure, for example, the termination time of the last task (called "makespan" in the context of job-shop scheduling). The mathematical object that describes the tasks and their relationships is the *task graph*.

Definition 2.1 (Task Graph) *A task graph is a triple $G = (P, \prec, d)$ such that $P = \{p_1, \ldots, p_m\}$ is a set of m tasks, $\prec$ is a strict partial-order relation on P (precedence) and $d : P \to \mathbb{N}$ is a function which assigns a duration to each task.*

In a world of unlimited resources, only the precedence constraints restrict the times when tasks can be executed. Hence the optimal schedule can be obtained by a greedy approach : start every task as soon as it is *enabled*, that is, all its predecessors have terminated. Such problems are common in project management where the solution method is called PERT/CPM (critical path method). A critical path is a path in the task graph whose sum of task durations is maximal. For the

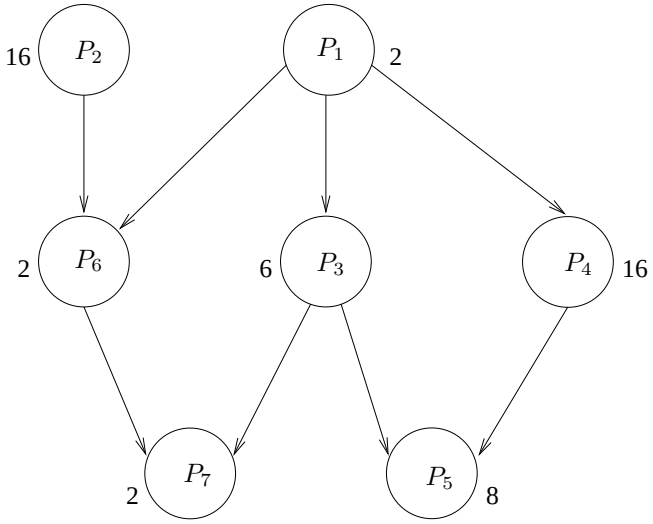


FIG. 2.1 – A task graph.

graph appearing in Figure 2.1 the critical path is $p_2 \cdot p_6 \cdot p_7$ whose length is 20 and hence no schedule can terminate all tasks before 20 time units. The computation of the longest path is polynomial in the size of the graph. The PERT method also assigns to each task an interval between the earliest time it can start and the latest time it can start executing without affecting global termination time. The earliest start time of a task is simply the length of the longest path leading to it. The latest start time of a task which is not on the critical path depends on how much it can be postponed without becoming critical, for example task p_3 may start anywhere in the interval $[2, 12]$ and still terminate not later than p_6. Needless to say, tasks on the critical path admit no margins and any postponement beyond their earliest start time will delay global termination.

The scheduling problem becomes much more difficult after the introduction of *resource constraints*. Resources are re-usable objects (machines, containers) allocated to tasks during their execution and cannot be used by more than one task at a time. When their number is limited it may happen that two or more tasks are ready for execution (in terms of precedence constraints) but only one of them can execute because of a conflict on a resource. In that case, the scheduler must decide to which task to give the resource and allow it to execute while letting the others wait. Such decisions make the scheduler a much more complicated object (unlike

the greedy scheduler when resources are unbounded) and renders the problem of finding the optimal schedule NP-hard.

There are many versions of scheduling problems and we will focus in this work on a commonly-used variant of the task-graph scheduling problem which is motivated by the execution of a computer programs on a parallel architecture consisting of a fixed number of *identical* machines with negligible communication costs. In this case the resource constraint is expressed by the number n of machines, and no more than n tasks can execute simultaneously. It should be noted that our framework applied to the task graph scheduling problem (with and without deadlines and release times) initially in [Ker02] and taken again in [Abd02, AKM03], applies as well to the case where there are different types of machines, like the job-shop scheduling problem [AAM06]. We also assume that tasks cannot be preempted once started but this assumption can be relaxed as well [AM02].

Before defining formally what feasible schedules are in this setting, let us note that the structure of the task graph imposes by itself a limit on the number of simultaneously executing tasks. The *width* of a partial order relation is the maximal number of uncomparable elements, and since the order here means precedence, the width of the task graph bounds the number of concurrent tasks. Hence beyond that number, additional machines cannot improve the optimum.

Let us denote by $\Pi(p)$ the set of immediate predecessors of p. Given a set of n parallel identical machines, we look for a schedule that minimizes the total execution time and respects both the precedence and resource constraints.

Definition 2.2 (Feasible and Optimal Schedules)
A feasible schedule for a task graph $G = (P, \prec, d)$ and n machines is determined by the function $st : P \to \mathbb{R}_+$ indicating the start time of each task. In a deterministic setting, the end time of a task is $en(p) = st(p) + d(p)$. A schedule is feasible if it satisfies :

1. *Precedence : for every $p, p' \in P$, $p \prec p' \Rightarrow en(p) \leq st(p')$.*
2. *Resources : every $t \in \mathbb{R}_+$ belong to at most n execution intervals of the form $[st(p), en(p)]$.*

The length of the schedule is $\max\{en(p) : p \in P\}$. *An optimal schedule is a schedule whose length is minimal.*

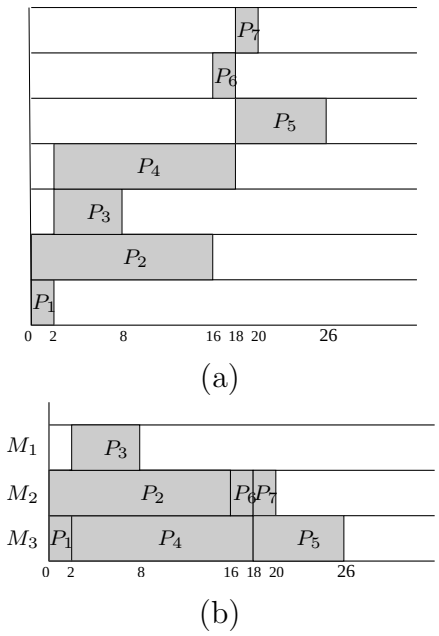


FIG. 2.2 – (a) An optimal schedule of the task graph of Figure 2.1 when the number of machine is unlimited. (b) An optimal schedule on 3 machines.

Note that the precedence constants constitute a *conjunction* of *difference constraints* of the form $st(p) - st(p') \geq d(p)$, a special class of a convex linear program solved easily by finding the earliest start time of each task. On the other, the resource constraints (2) reduce eventually to a combination of constraints that disallow two tasks to execute simultaneously : $[st(p), en(p)] \cap [st(p'), en(p')] \neq \emptyset$. Such conditions are *disjunctions* of the form

$$st(p) - st(p') \geq d(p) \ \vee \ st(p') - st(p) \geq d(p')$$

rendering the set of feasible solutions highly non-convex with exponentially many disconnected sets of feasible solutions. This is what makes the problem NP-hard.

The schedule of Figure 2.2-(a) is an optimal schedule for the graph of Figure 2.1 when the number of machines is unlimited and we have a distinct machine for each task. Since the width of the graph is 3, the same schedule can be obtained using 3 machines, see Figure 2.2-(b).

On the other hand, if we have only 2 machines the number of enabled tasks may exceed the number of available machines and the conflict should be resolved

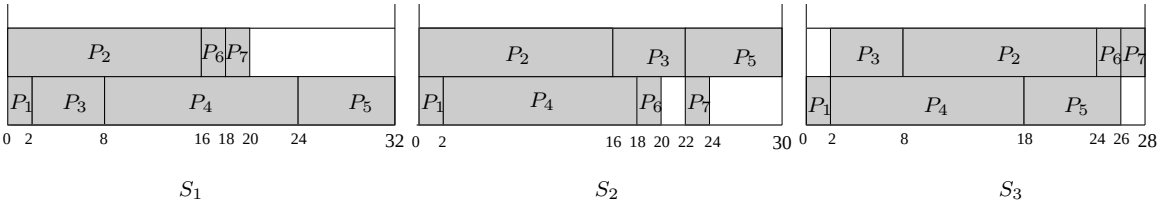


FIG. 2.3 – Three feasible schedules of the task graph of Figure 2.1 on 2 machines.

by the scheduler. We can see in schedules S_1 and S_2 of Figure 2.3 that at $t = 2$, p_2 is already occupying one machine where both p_3 and p_4 become enabled. Schedule S_1 gives the remaining machine to p_3 while S_2 gives it to p_4. Unlike the case of infinitely many machines, an optimal schedule may be obtained by choosing at some point not to execute an enabled task. For example, schedule S_3 achieves the optimum while not starting task p_2 immediately although it is enabled at $t = 0$.

2.2 Types of Schedules

In principle, the set of feasible schedules is an infinite and uncountable subset of $\mathbb{R}_+^m$, for any instance of the task graph scheduling problem, because an arbitrary amount of idle time can be inserted at any machine between adjacent pairs of tasks. There are certain interesting finite subsets of the set of feasible, some of which are guaranteed to contain the optimum and our algorithms will take advantage of this fact. To explain these classes of schedules we need the following auxilary definition of the well known *left shift*.

Definition 2.3 (Left Shift) *Let st be a feasible schedule. A feasible schedule st' is a left shift of st if there is a task p such that $st'(p) < st(p)$ and $st'(p') = st(p')$ for every $p' \neq p$. A left shift is local if for every $p' \neq p$, $en(p') \leq st(p)$ implies $en(p) \leq st'(p)$.*

Shifts and local shifts are best illustrated graphically using the Gantt chart that corresponds to the schedule (see Figure 2.4). A shift just moves the execution interval of p earlier (while maintaining precedence and resource constraints). A local shift restricts this leftward "movement" of the interval by disallowing its left endpoint to cross the right endpoint of another task. If st' is a left shift of st, its

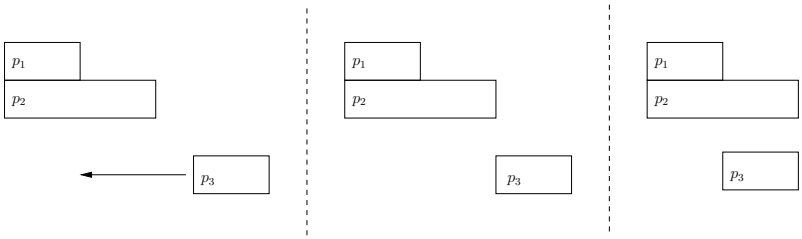


FIG. 2.4 – A schedule, a local shift of p_3 and a general shift of p_3.

length is, of course, lesser or equal to the length of st. Given a schedule st, we say that a task is *enabled* at time t if $en(p') \leq t$ for every $p' \prec p$. We are now in the position to define various classes of feasible schedules.

Definition 2.4 (Types of Schedules) *A feasible schedule st is :*

1. *Integer, if* $en(p) \in \mathbb{N}$ *for every* p *;*
2. *Immediate, if it admits no local shift ;*
3. *Non-lazy, if it admits no shift ;*
4. *Greedy, if there is no time* t *in which the number of active tasks is smaller than* n *and there is a task* p *enabled in* t *such that* $st(p) > t$*.*

All these classes form an inclusion hierarchy where the class of greedy schedules is the smallest. All classes except the greedy schedules are known to intersect the class of optimal schedules for *every* problem (see Figure 2.5). In the framework of timed automata, these property of immediate and non lazy schedules[1] has been exploited in [Abd02, AAM06] to restrict the analysis of timed automata to a finite (and relatively-small) number of runs.

Before describing our modeling framework we mention a known result concerning the deviation of a greedy schedule from the optimum. When resources are bounded, a greedy approach might miss the optimum. For example it might be the case that a non-critical task p is enabled while a critical one p' is not. Thus p will get the machine and still occupy it when p' becomes enabled. Postponing p'

[1] An exact reinvention of the well known *active schedules* (or left shifted schedules) used in Operational Research, initially in the context of the job-shop, to restrict the search space to a small set of schedules (cf. [Bak74] pp. 181).

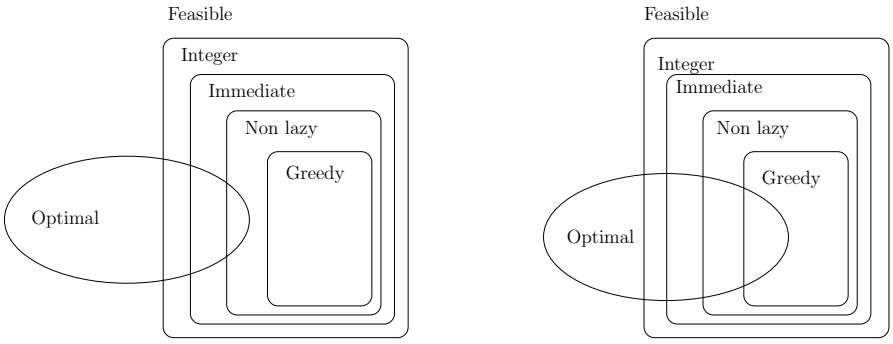


FIG. 2.5 – The inclusion relation between schedule classes for a given problem. For every scheduling problem there is a non-lazy optimal schedule. Some problems admit a Greedy solution (right) and some problems do not (left).

may delay termination, and this could be avoided by a non-greedy schedule that would let p wait and the machine idle, anticipating p', and executing p some time later. Since the Greedy solution is so easy to compute, it is worthwhile to compare its performance with an optimal algorithm. In the literature greedy algorithms are called *list scheduling* and they are based on the following principles : at any moment start as many tasks as you can. If the number of idle machines is smaller than the number of enabled tasks than choose tasks according to a fixed priority relation. Variants of list scheduling differ in the way the priority relation is defined but they all give an implicit priority to tasks that are already enabled. For example in the *Critical Path Scheduling* which is a variant of list scheduling, the priority list is sorted by the non increasing order of the *levels*, where the level of each task can be defined as the longest path from this task in the original task graph. Ties are broken by preferring tasks that have more immediate successors.

The following result gives an upper bound on the performance of a greedy schedule (employing any priority) compared to the optimal one. The following proof is an adaptation from [DRV00].

Theorem 2.1 (Graham [CG72]) *Let $G = (P, \prec, d)$ be a directed acyclic graph to be scheduled on m machines. Let C^{opt}_{max} the length of the optimal schedule, and C^{L}_{max} the length of a schedule given by a priority list L of G. We show that :*

$$C^{L}_{max} \leq (2 - 1/m)C^{opt}_{max}$$

Proof We first have to show the following lemma :

Lemma 2.1 *There exists a dependence path c in G such that :*

$$tm \leq (m-1) \times \sum_{p \in c} d(p)$$

where tm is the total idle time.

Proof Let p_{i_1} the latest finishing task in the schedule i.e.

$$st(p_{i_1}) + d(p_{i_1}) = C^L_{max}$$

Now consider t_1 the largest instant smaller than $st(p_{i_1})$ such that there exists an idle machine during the interval $[t_1, t_1 + 1[$ ($t_1 = 0$ if such a time step does not exist). Since $st(L)$ is a list schedule, no task is free at t_1, otherwise the idle machine would schedule it. In addition, it must be a task p_{i_2} which is a predecessor (not necessarily an immediate one) of p_{i_1} that is being executed at time t_1 ; otherwise p_{i_1} would have been started at time t_1 by the idle machine. By the definition of t_1, we know that all machines are occupied between the termination of p_{i_2} and the beginning of p_{i_1}.

We start the construction again from p_{i_2} so that we obtain a task p_{i_3} in a way that all machines are occupied between the termination of p_{i_3} and the beginning of p_{i_2}. Repeating the process, we end up with a set of l tasks p_{i_l}, $p_{i_{l-1}}$, ..., p_{i_1} that forms a path c in G such that all machines are occupied except perhaps during their execution. In other words, the idleness of some machines can only occur during the execution of these l tasks, during which at least one processor is active (the one that schedules the task). Hence :

$$tm \leq (m-1) \times \sum_{j=1}^{l} d(p_{i_l})$$

We have,

$$m \times C^L_{max} = tm + \sum_{p \in P} d(p)$$

which comes from a geometrical interpretation of the Gantt chart : its total rectangular surface is $m \times C^L_{max}$ can also be expressed by the sum of tasks durations

and the total idle time.
because every schedule is longer than the longest path, a-fortiori c we have

$$\sum_{p \in c} d(p) \leq C_{max}^{opt}$$

in addition,

$$\sum_{p \in P} d(p) \leq m \times C_{max}^{opt} (\text{we have equality if there is no idle time})$$

finally we obtain,

$$\begin{aligned} m \times C_{max}^{L} &= tm + \textstyle\sum_{p \in P} d(p) \\ &\leq (m-1) \times \textstyle\sum_{p \in c} d(p) + \sum_{p \in P} d(p) \\ &\leq (m-1) \times C_{max}^{opt} + m \times C_{max}^{opt} \end{aligned}$$

∎

Example 2.1 *Figure 2.7 shows possible schedule st for the task graph of figure 2.6 on 3 machines where no local left shift is possible. The schedule st′ is derived from st by a global left shift of task p_2. In st′, no other left shift is possible, and it is optimal. Note that for this example, there is no optimal greedy schedule.*

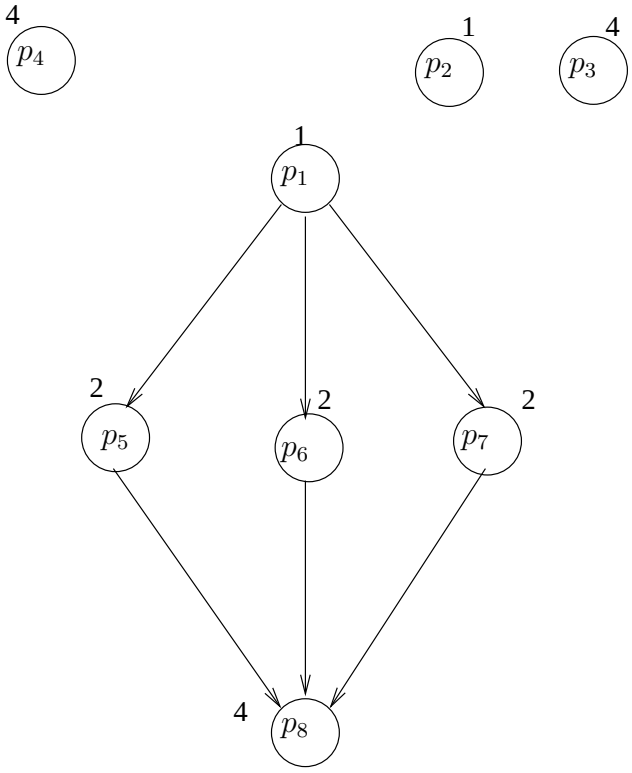


FIG. 2.6 – A task graph

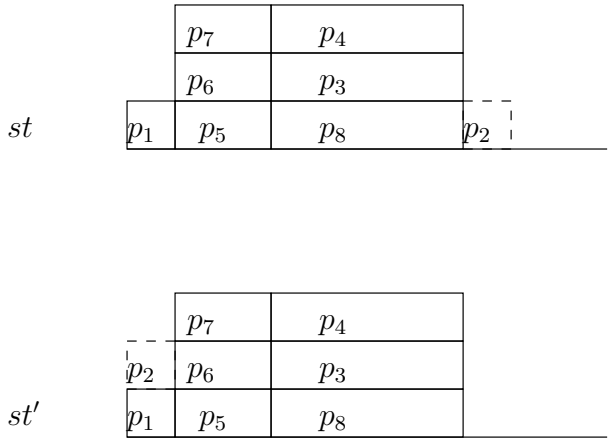


FIG. 2.7 – Possible schedules for the task graph of figure 2.6

Chapitre 3

Modeling Task Graph Scheduling with Timed Automata

Our approach to scheduling consists in transforming the problem into a product of timed automata such that the runs of the global automaton correspond to feasible schedules, and hence the optimal schedule is the shortest run in the automaton in terms of total elapsed time.

Timed automata [AD94] are automata augmented with continuous clock variables whose values grow uniformly at every state. Clocks can be reset to zero at certain transitions and tests on their values can be used as conditions for the enabledness of transitions. Hence they are ideal for describing concurrent time-dependent behaviors. Our definition below is an "open" version of timed automata which can refer to the states of other automata, ranging over Q', in their transition guards. The clocks constraints that we use are slightly less general than in the standard definition of timed automata. Instead of considering a general model using synchronizations, we choose this open version because it is sufficient to capture precedence constraints.

Definition 3.1 (Timed Automaton)
An open timed automaton *is* $\mathcal{A} = (Q, C, I, \Delta, s, f)$ *where*

- Q *is a finite set of states ;*
- C *is a finite set of clocks ;*
- I *is the staying condition (invariant), assigning to every* $q \in Q$ *a conjunction* I_q *of inequalities of the form* $c \leq u$*, for some clock* c *and integer* u *;*

- Δ *is a transition relation consisting of elements of the form* (q, ϕ, ρ, q') *where*
  - q *and* q' *are states ;*
  - $\phi = \phi_1 \wedge \phi_2$ *is the transition guard where* ϕ_1 *is a formula characterizing a subset of an external set of states* Q' *and* ϕ_2 *is a conjunction of constraints of the form* $(c \geq l)$ *for some clock* c *and some integer* l *;*
  - $\rho \subseteq C$ *is a set of clocks to be reset ;*
- s *and* f *are the initial and final states, respectively.*

A *clock valuation* is a function $\mathbf{v} : C \to \mathbb{R}_+ \cup \{0\}$, or equivalently a $|C|$-dimensional vector over $\mathbb{R}_+$. A configuration of the automaton is hence a pair $(q, \mathbf{v})$ consisting of a discrete state (also known as *location*) and a clock valuation. Every subset $\rho \subseteq C$ induces a reset function Reset_ρ defined for every clock valuation $\mathbf{v}$ and every clock variable $c \in C$ as

$$\text{Reset}_\rho \, \mathbf{v}(c) = \begin{cases} 0 & \text{if} \quad c \in \rho \\ \mathbf{v}(c) & \text{if} \quad c \notin \rho \end{cases}$$

That is, Reset_ρ resets to zero all the clocks in ρ and leaves the other clocks unchanged. We use $\mathbf{1}$ to denote the unit vector $(1, \ldots, 1)$ and $\mathbf{0}$ for the zero vector.

A *step* of the automaton is one of the following :

- A discrete step : $(q, \mathbf{v}) \xrightarrow{0} (q', \mathbf{v}')$, where there exists $\delta = (q, \phi, \rho, q') \in \Delta$, such that $\mathbf{v}$ satisfies ϕ and $\mathbf{v}' = \text{Reset}_\rho(\mathbf{v})$.
- A time step : $(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1})$, $t \in \mathbb{R}_+$ and $\mathbf{v} + t\mathbf{1}$ satisfies I_q.

A *run* of the automaton starting from a configuration $(q_0, \mathbf{v}_0)$ is a finite sequence of steps

$$\xi : \quad (q_0, \mathbf{v}_0) \xrightarrow{t_1} (q_1, \mathbf{v}_1) \xrightarrow{t_2} \cdots \xrightarrow{t_n} (q_n, \mathbf{v}_n).$$

The *logical length* of such a run is n and its *metric length* is $t_1 + t_2 + \cdots + t_n$. Note that discrete transitions take no time.

For every task p we build a 3-state automaton with one clock c and a set of states $Q = \{\overline{p}, p, \underline{p}\}$ where $\overline{p}$ is the *waiting* state before the task starts, p is the *active* state where the task executes and $\underline{p}$ is a *final* state indicating that the task has terminated. The transition from $\overline{p}$ to p resets the clock and can be taken only if all the automata corresponding to the tasks in $\Pi(p)$ are in their final states. The transition from p to $\underline{p}$ is taken when $c = d(p)$.

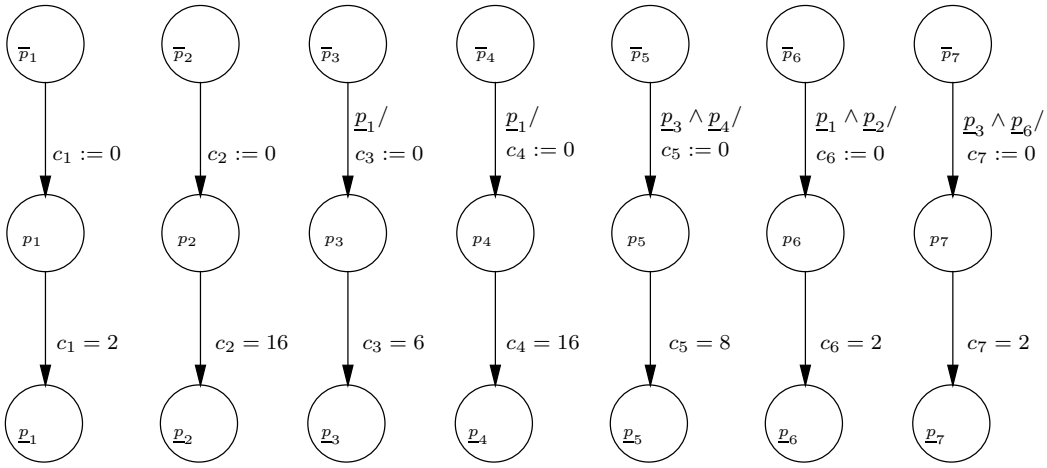


FIG. 3.1 – The automata for the task graph of Figure 2.1

Definition 3.2 (Timed Automaton for a Task) *For every task $p \in P$ its associated timed automaton is $\mathcal{A}_p = (Q, \{c\}, I, \Delta, s, f)$ with $Q = \{p, \overline{p}, \underline{p}\}$ where the initial state is $\overline{p}$ and the final state is $\underline{p}$. The staying conditions are* **true** *for $\overline{p}$ and $\underline{p}$ and $c \leq d(p)$ in p. The transition relation Δ consists of the two transitions :*

$$start: \qquad (\overline{p}, \bigwedge_{p' \in \Pi(p)} \underline{p}', \{c\}, p)$$

and

$$end: \qquad (p, c = d(p), \emptyset, \underline{p})$$

Note that the clock is active only in state p and its value in $\overline{p}$ is not important as it is reset to zero before being tested. The automata for the task graph of Figure 2.1 appear in Figure 3.1.

To obtain the timed automaton representing the whole scheduling problem we need to compose the automata for the individual tasks. The composition takes care of the precedence constraints by allowing an automaton to make a start transition only when the automata for its predecessors are in their respective final states. Mutual exclusion constraints are enforced by forbidding *conflicting* global states, that is, states of the form $q = (q^1, \ldots, q^n)$ where more than m automata are in their respective active states.

Definition 3.3 (Mutual Exclusion Composition) *Let $G = (P, \prec, d)$ be a task graph and let $\mathcal{A}^i = (Q^i, C^i, I^i, \Delta^i, s^i, f^i)$ be the automaton corresponding to each*

task p_i. Their mutual exclusion composition (assuming m machines) is the automaton $\mathcal{A} = (Q, C, I, \Delta, s, f)$ such that Q is the restriction of $Q^1 \times \ldots \times Q^n$ to non-conflicting states, $C = C^1 \cup \ldots \cup C^n$, $s = (s^1, \ldots, s^n)$, $f = (f^1, \ldots, f^n)$, the staying condition for a global state $q = (q^1, \ldots q^n)$ is $I_q = I_{q^1} \wedge \ldots \wedge I_{q^n}$ and the transition relation Δ contains all the tuples of the form

$$((q^1, \ldots, q^j, \ldots, q^n), \phi_2, \rho, (q^1, \ldots, r^j, \ldots, q^n))$$

such that the source and target states are non-conflicting, $(q^j, \phi_1 \wedge \phi_2, \rho, r^j) \in \Delta^j$ for some j and ϕ_1 is satisfied by $(q^1, \ldots, q^n)$.

In the automata derived from tasks, the formula ϕ_1 in the guard for the *start* transition specifies that the automata for the preceding tasks are in their respective final states. This way runs of the product automaton satisfy precedence constraints by construction. A run of $\mathcal{A}$ is *complete* if it starts at $(s, \mathbf{0})$ and the last step is a transition to f. From every complete run ξ one can derive in an obvious way a schedule where $st(p_i)$ is the time the $start_i$ transition is taken. The length of the schedule coincides with the metric length of ξ. Note that the interleaving semantics inserts some redundancy as there could be more than one run associated with a feasible schedule in which several tasks start at the same time.

The construction just described, although it gives an exact representation of the scheduling problem is very inefficient when implemented. Each global state is represented as a $2n$-tuple consisting of the state and clock of *every* task automaton. When we treat thousands of tasks, just scanning the states and generating successors becomes a very heavy procedure. We take advantage of the fact that the number of concurrently active tasks is bounded by the width of the graph, and create a more efficient translation with fewer automata. The idea is to use an automaton for a chain of linearly-ordered tasks. If $p \prec p'$ than after terminating p the automaton moves to state $\overline{p'}$. If p' has additional predecessors in another chain, they are mentioned in the guard of its corresponding *start* transition. This is formalized below.

Definition 3.4 (Chain) *A chain in a partially-ordered set $(P, \prec)$ is a subset P' of P such that for every $p, p' \in P'$ either $p \prec p'$ or $p' \prec p$.*

Definition 3.5 (Chain Cover) *A chain covering of a partially-ordered set $(P, \prec)$ is a set of chains $\mathcal{H} = \{H_1, \ldots H_k\}$ satisfying*

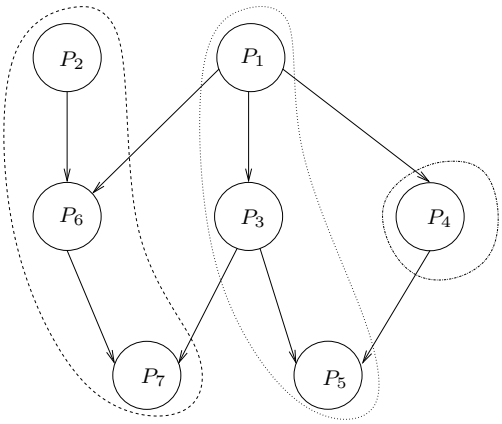


FIG. 3.2 – A chain covering of the task graph of Figure 2.1

1. *Each* H_i *is a linearly ordered subset of* P.
2. $H_i \cap H_j = \emptyset$ *pour* $i \neq j$
3. $\bigcup_{i \leq k} H_i = P$

An example of a chain cover for our task graph appears in Figure 3.2.

The *external predecessors* of a task $p \in H_i$ are the predecessors of p outside its chain, that is,

$$\Pi'(p) = \Pi(p) \cap (P - H_i).$$

Given a chain $H = p_1 \prec p_2 \prec \cdots \prec p_k$ its automaton consists of a pair of states $\{\overline{p_i}, p_i\}$ for every p_i and a final state f. The *start* transition from $\overline{p_i}$ to p_i is then enabled if for every $j \neq i$ and $p' \in \Pi(P) \cup H_j$, the automaton for the chain H_j is in a state beyond p'. We denote this condition by :

$$\bigwedge_{p' \in \Pi'(p)} > p'.$$

The resulting automata appear in Figure 3.3. Applying the mutual exclusion composition to the chain automata we obtain an automaton isomorphic to the product of the task automata with the same properties with respect to feasible schedules.

It is worth mentioning that chain covers are related to the width of a partial order via Dilworth's theorem [Dil50].

Theorem 3.1 (Dilworth) *The width of a partial order is equal to the minimal number of chains needed to cover it.*

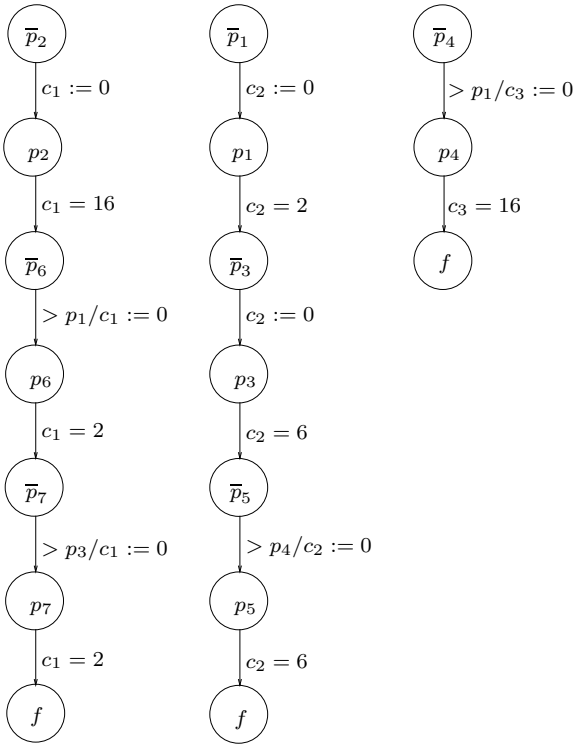


FIG. 3.3 – The automata for the chain cover of Figure 3.2

Although the computation of the width and its associated cover is polynomial, we do not compute it exactly but use a fast and simple algorithm to approximate it [Ker02]. An exact algorithm is obtained via a transformation into a max-flow min-cut problem, and the greedy algorithm used here and initially in [Ker02] constitutes a good approximation [FRS03]. The use of chain automata improves significantly the efficiency of the initial model in terms of size and analysis, particularily in practice, because in the most of real applications we have to deal with task graphs that have a very small width compared to the number of tasks.

Figure 3.4 depicts part of the automaton obtained by composing the automata of Figure 3.2 when there are 2 machines. This automaton has only 3 clocks (the number of chains in the cover). In the initial state, where tasks P_2, P_1 and P_4 are waiting, there are only two possible successors, to start P_2 (state $(p_2\ \overline{p}_1\ \overline{p}_4)$) or to start P_1 (state $(\overline{p}_2\ p_1\ \overline{p}_4)$). The transition to the state $(\overline{p}_2\ \overline{p}_1\ p_4)$ is disabled because task P_1 has not terminated. No start transition can be taken from $(p_2, p_3, \overline{p}_4)$

because all the machines are occupied in this state.

As mentioned in Section 2, previous results [Abd02, AAM06] show that it is sufficient to explore finite set of runs that correspond to non-lazy schedules in order to find the optimum. However, the detection of a useless waiting that makes a run lazy cannot always be done at the moment the *wait* transition is taken. Suppose we are in a global state q in which a task p_i is enabled, a processor is available and among the active tasks, task p_j has the least remaining execution time d. If the duration of p_i is less than d then choosing to wait will clearly lead to laziness because p_i could be executed without blocking. If, however the duration of p_i is larger than d, the laziness depends on the number of tasks that may become enabled after the termination of p_j, their durations, their successors and so on. For this reason we generate runs that correspond to the larger class of immediate schedules which are easier to generate without lookahead : at every global state q in which there is at least one free machine we generate a *start* successor for each enabled task and one *wait* transition of duration d. Thus the whole problem can be reduced to finding the shortest path in a weighted acyclic graph which corresponds to these runs.

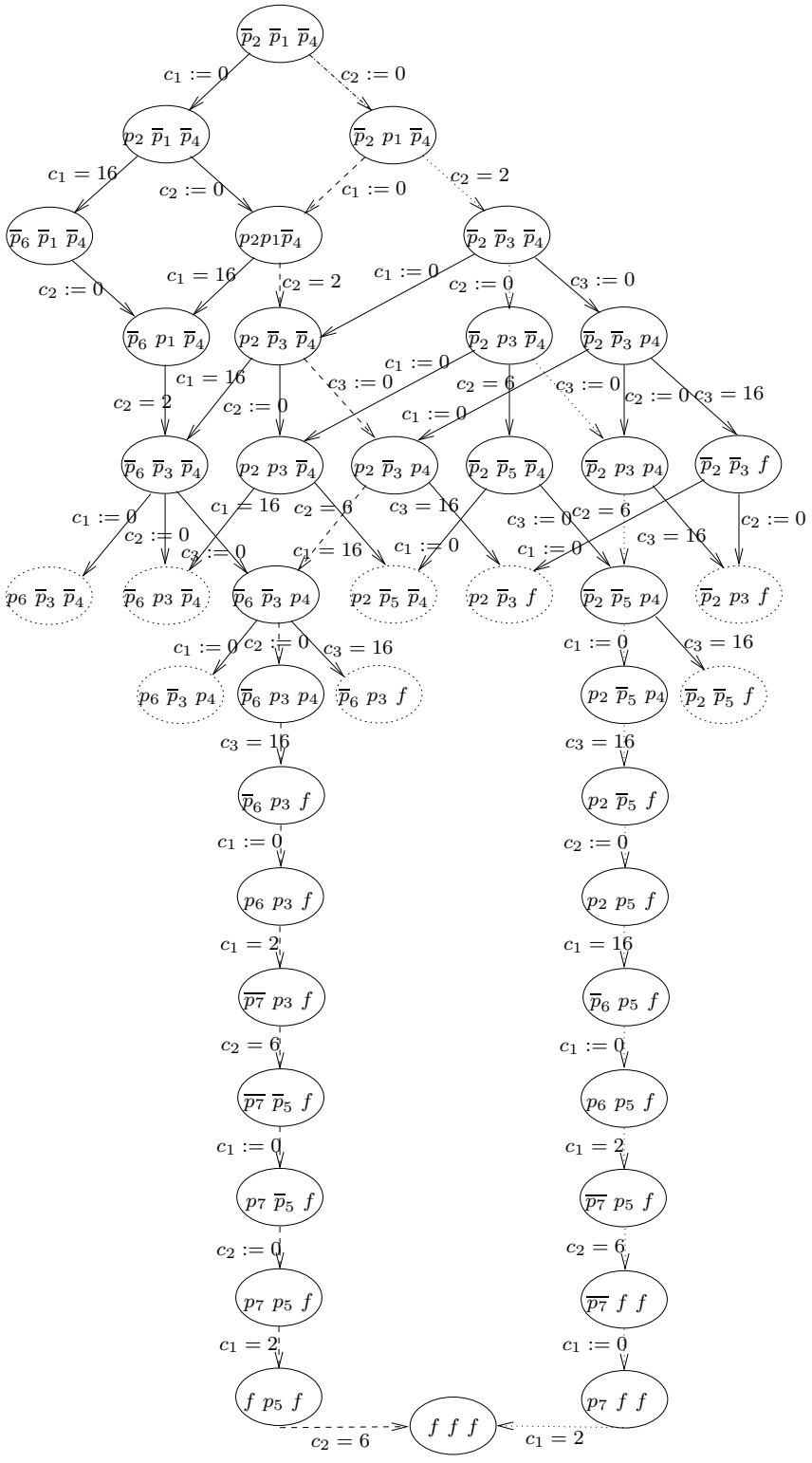


FIG. 3.4 – The timed automaton obtained by composing the automata of Figure 3.3 for the case of 2 machines. The two runs corresponding to the schedules S_1 and S_3 are indicated by the dashed and dotted lines, respectively.

Chapitre 4

Shortest Paths in Acyclic Graphs : Exact Algorithms

In this chapter we survey various approaches for finding shortest paths in acyclic graphs with non-negative edge weights. Although some of the concepts and results carry over to more complex graphs, we ourselves to the type of graphs that we use for modeling and solving our scheduling problems. Readers interested by more detailed formalism can consult [Nil71, Zha99, CLRS01]

4.1 Basic Definitions

Definition 4.1 (Directed Graph) *A directed graph is $G = (Q, \delta)$ where Q is a finite set and $\delta \subseteq Q \times Q$ is a set of directed edges (also called transitions)*

We denote the fact that $(q, q') \in \delta$ by $q \rightarrow q'$ and the fact that (q, q') belongs to the *transitive closure* of δ by $q \Rightarrow q'$. We would like sometime to be able to refer to a *specific path* from q to q', and since there are no multiple edges between any two nodes, we can unify such a path with a sequence of states of the form

$$\xi = q \cdot q_1 \cdots q_{k-1} \cdot q' \tag{4.1}$$

with $k \geq 1$, also written sometimes as

$$q \rightarrow q_1 \rightarrow \cdots \rightarrow q_{k-1} \rightarrow q'.$$

We use the notation $q \stackrel{\xi}{\Longrightarrow} q'$ to denote the fact that ξ is a path from q to q' which is defined recursively as

$$q \stackrel{q \cdot q'}{\Longrightarrow} q' \quad \text{if } q \to q'$$
$$q \stackrel{\xi \cdot q'}{\Longrightarrow} q' \quad \text{if } \exists q'' \text{ s.t. } q \stackrel{\xi}{\Longrightarrow} q'' \text{ and } q'' \to q'$$

We say that a *path* such as (4.1) is of *logical length* k. A graph is acyclic if $q \not\Rightarrow q$ for every q and in this case $\Rightarrow$ is a *strict partial-order* relation. Our graphs, which are constructed (more or less) as direct products of chains admit some extra properties. First, they have distinguished *initial* and *final* states, s and f which are, respectively, the minimal and maximal elements of $\Rightarrow$. Secondly, all paths between a pair of nodes have the same number of edges, which eliminates the possibility of δ including a pair (q, q') which is connected by a longer path.These graphs can be be partitioned into levels $Q_0, Q_1, \ldots, Q_k$ such that Q_i consists of all nodes reachable from s by paths of length i. We use the term *ranked graphs* for them. We define the auxiliary *successor* and *predecessor* functions, respectively, as

$$\sigma(q) = \{q' : q \to q'\} \quad \text{and} \quad \pi(q) = \{q' : q' \to q\},$$

and extend them to sets. Hence $Q_i = \sigma(Q_{i-1}) = \pi(Q_{i+1})$. We call k the *depth* of the graph. Paths in the graph admit the obvious *prefix* relation with $\xi \prec \xi \cdot \xi'$, visulaizable in a tree form. Figure 4.1 depicts such a graph and an enumeration of its paths.

Definition 4.2 (Weighted Directed Graph) *A weighted directed graph is* $G = (Q, \delta, w)$ *where* (Q, δ) *is a directed graph and* $w : \delta \to \mathbb{N}$ *is a cost/weight assignment to edges.*

By transitivity we can lift w to paths by letting

$$w(\xi \cdot q \cdot q') = w(\xi \cdot q) + w(q, q').$$

We sometimes write such weighted paths as

$$q \xrightarrow{d_1} q_1 \xrightarrow{d_2} \cdots q_{k-1} \xrightarrow{d_k} q'$$

Finally in our algorithm descriptions we will sometimes represent such paths as sequences of *valued nodes* of the form

$$(q, v_0) \cdot (q_1, v_1) \cdots (q_{k-1}, v_{k-1}) \cdot (q', v_k)$$

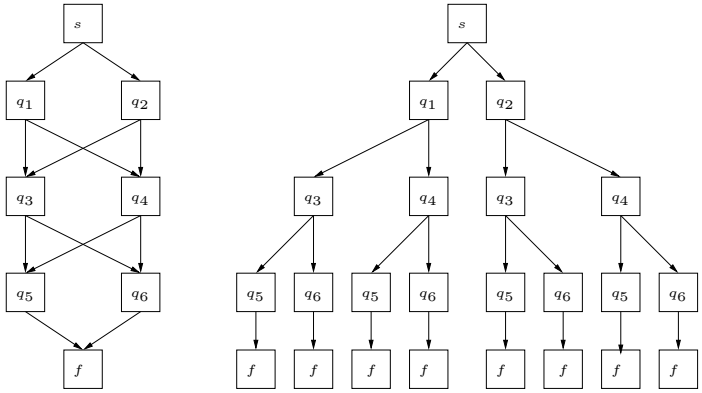


FIG. 4.1 – A depth-4 ranked directed acyclic graph and the structure of all its paths from s t of.

where $v_0 = 0$ and $v_i = v_{i-1} + d_i$, that is, $v_i = w(q \cdot q_1 \cdots q_i)$. A path whose first state is s and last state is f is called a *complete* path. For every pair of states q and q' we let $\Pi(q, q') = \{\xi : q \xrightarrow{\xi} q'\}$ denote the set of paths from q to q'.

Definition 4.3 (Potential Function) *A potential function associated with a weighted directed graph is a function $V : Q \times Q \to \mathbb{N} \cup \{\infty\}$ defined as*

$$V(q, q') = \min\{w(\xi) : \xi \in \Pi(q, q')\}.$$

In other words, the potential function assigns to every pair of states the shortest path between them. We adopt the convention that $V(q, q') = \infty$ when $\Pi(q, q') = \emptyset$. A path $\xi \in \Pi(q, q')$ such that $w(\xi) = V(q, q')$ is called *optimal*.

Definition 4.4 (Cost to Come and Go) *For a directed weighted graph, the functions $\overleftarrow{V}$ (cost-to-come) and $\vec{V}$ (cost-to-go) are defined as :*

$$\overleftarrow{V}(q) = V(s, q) \quad \textit{and} \quad \vec{V}(q) = V(q, f)$$

We want to compute the shortest path between s and f, that is,

$$V(s, f) = \overleftarrow{V}(f) = \vec{V}(s).$$

4.2 Path Enumeration Algorithms

As $\overleftarrow{V}(f)$ is the minimal cost over all complete paths we can just enumerate and compare them. There are two common ways to perform the enumeration, by exploring the graph either in the depth-first (DF) or the breadth-first (BF) order. We will comment on the relative algorithmic advantages of those after we describe them below. These algorithms are wasteful and impractical for graphs which are not trees and are presented here for semantic and didactic purposes without trying to "optimize" them.

The DF algorithm uses a *LIFO (last in first out) stack* S, a data structure that keeps elements ordered according to the temporal order in which they have been inserted, and provides for removal and extraction of the *last* element.The sequence of stack elements represents a path starting from s augmented with additional information for search purposes. Stack elements are thus of the form (q, v, L) where q is a node in the path, v is the cost of the path until q and L is the list of the unexplored successors of q, that is, successors that lead to paths that have not yet been generated. The operation $Path(S)$ extracts from the stack the corresponding path. The $Push$ operation inserts elements at the end of the stack and the Pop operation removes the last element. We assume a $Last$ operation that makes the last element accessible and manipulable without being removed.

Algorithm 4.1 (DF Path Enumeration)

$\overleftarrow{V}(f) := \infty$
$Push(S, (s, 0, \sigma(s))$
while S *is not empty* **do**
  $(q, v, L) := Last(S)$
  if $L = \emptyset$ **then**
    $Pop(S)$
    if $q = f \wedge v < \overleftarrow{V}(f)$ **then**
      $\overleftarrow{V}(f) := d$
      $OptPath := Path(S)$
  else
    $q' :=$ *some element in* L
    $L := L - \{q'\}$
    $L' := \sigma(q')$
    $v' := v + w(q, q')$
    $Push(S, (q', v', L'))$
end

The set of all enumerated paths is illustrated in Figure 4.2. The algorithm is under-specified with respect to the *order* in which the successors of a given node are explored. When this order coincides with the left-to-right order in the figure, partial paths are explored according to the order appearing in Figure 4.3-(a). The content of the stack while visiting the path $s \cdot q_1 \cdot q_4 \cdot q_5 \cdot f$ is :

$$(s, 0, \{q_2\}) \cdot (q_1, 3, \emptyset) \cdot (q_4, 5, \{q_6\}) \cdot (q_5, 10, \emptyset) \cdot (f, 13, \emptyset).$$

The breadth-first algorithm uses a *FIFO (first in first out) queue* F which, like the stack, maintains elements according to their insertion order but removes them in the same order, that is, from the other side of the queue. The elements of F are of the form (ξ, q, v) where ξ is a path leading to q, and $v = w(\xi)$. Insertion to the queue is done via the Enq operation while the Deq operation extracts and removes the first element.

Algorithm 4.2 (BF Path Enumeration)

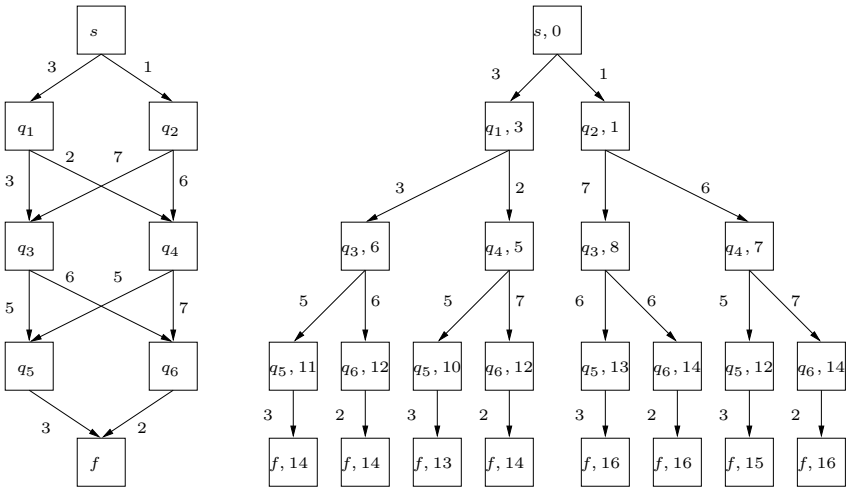


FIG. 4.2 – A weighted acyclic graph and the paths emanating from s. Each node represents a unique path whose length is written inside the node.

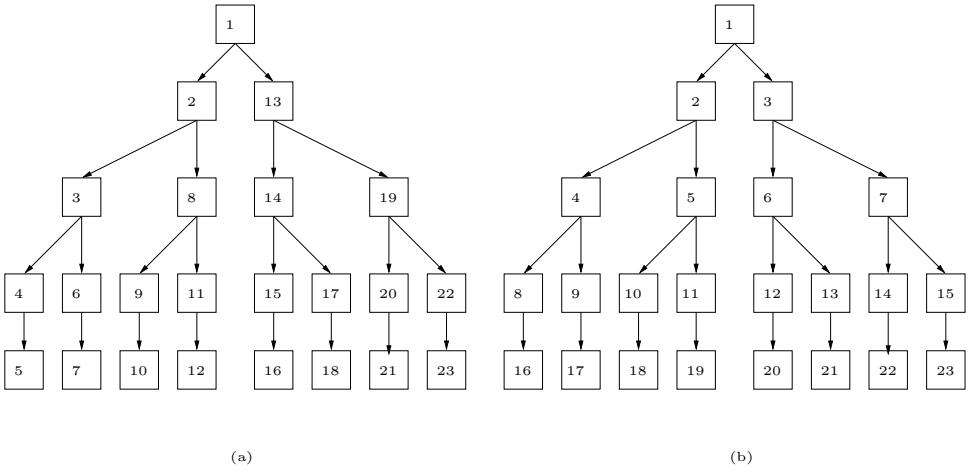


FIG. 4.3 – The order of path exploration according to DF path enumeration (a) and BF path enumeration (b).

$\overleftarrow{V}(f) := \infty$
$Enq(F, (s, s, 0))$
while $F \neq \emptyset$ **do**
  $(\xi, q, v) := Deq(F)$
  if $q = f$ **then**
    if $v < \overleftarrow{V}(f)$ **then**
      $\overleftarrow{V}(f) := v$
      $OptPath := \xi$
  else
    for *each* $q' \in \sigma(q)$ **do**
      $Enq(F, (\xi \cdot q', q', v + w(q, q')))$
    end
end

Like the DF algorithm, this one is not specific about the order in which successors of a given node are generated. Assuming a left-to-right order among successors, paths are generated according to the order depicted in Figure 4.3-(b). Note that for graphs like ours, if both algorithms use the same order among successors, they generate *complete* paths in the *same* order. Albeit the difference between DF and BF order, both algorithms are similar in the sense that they start with $(s, 0)$ and accumulate the cost as they *go down* the path.

Both algorithms admit "reversed" versions that enumerate all the paths leading to f (Figure 4.4) and compute the cost-to-go $\vec{V}(s)$. These versions are obtained by interchanging s by f and σ by π. This is equivalent to running Algorithms 4.1 and 4.2 on the reversed graph, the graph obtained by replacing every $(q, q') \in \delta$ by (q', q) and flipping s and f.

Enumeration is an exponential process for any decent problem and these algorithms are not practical. The DF algorithm has a relative advantage in terms of memory usage as it keeps at most one complete path in memory at every moment, while the BF algorithm may need to store an exponential number of partial paths before reaching one complete path. Hence for prohibitively large graphs the DF algorithm will, at least, always give *some* solution, optimal or not (provided, of course, that the graph is acyclic).

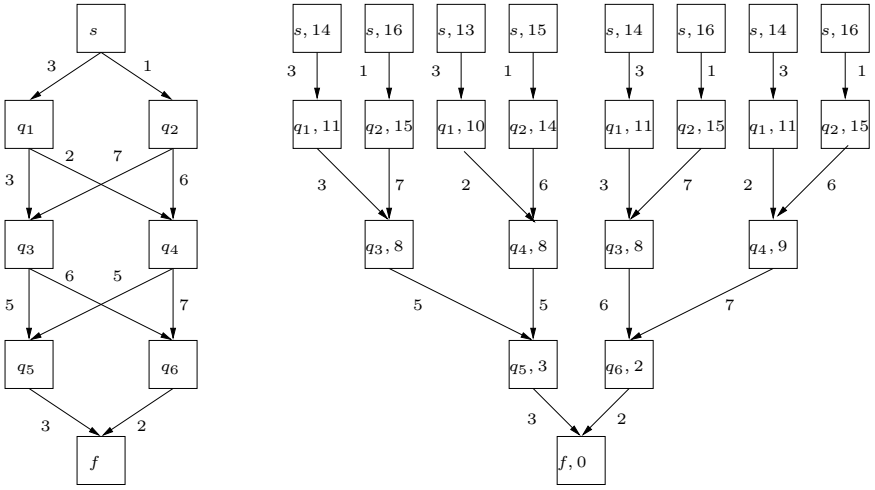


FIG. 4.4 – All the paths leading to f.

4.3 Non-Enumerative Algorithms

Algorithm 4.1 and 4.2 compute and compare the cost-to-come for the final state but not for intermediate states along the paths. The following principle [Bel57] allows one to use much more efficient algorithms.

Theorem 4.1 (Bellman Principle [Bel57]) *Let q'' be a node appearing in an optimal path from q to q'. Then*

$$V(q, q') = V(q, q'') + V(q'', q'). \tag{4.2}$$

Note that for a node q residing on an optimal *complete* path, (4.2) specializes into

$$\overleftarrow{V}(f) = \overleftarrow{V}(q) + \vec{V}(q) = \vec{V}(s),$$

as can be demonstrated if Figure 4.5.

The Bellman principle can be rephrased as defining $\vec{V}$ and $\overleftarrow{V}$ recursively :

$$\begin{aligned} &\vec{V}(f) = 0 \\ &\vec{V}(q) = \min\{w(q, q') + \vec{V}(q') : q' \in \sigma(q)\} \end{aligned} \tag{4.3}$$

and

$$\begin{aligned} &\overleftarrow{V}(s) = 0 \\ &\overleftarrow{V}(q) = \min\{\overleftarrow{V}(q') + w(q', q) : q' \in \pi(q)\} \end{aligned} \tag{4.4}$$

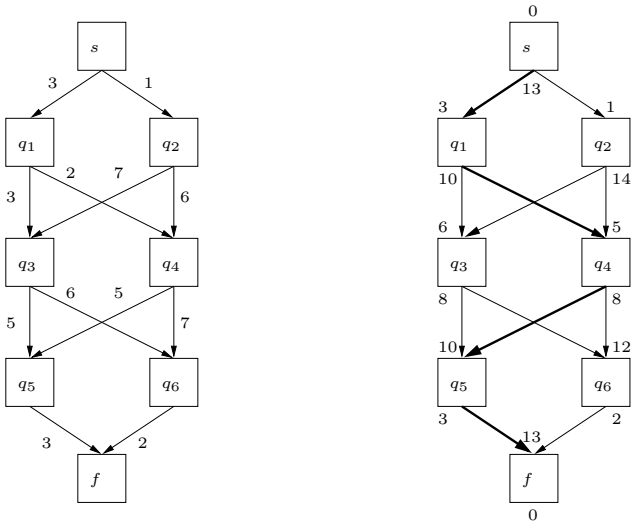


FIG. 4.5 – Cost-to-come (above each node) and cost-to-go (below each node) for the example. The optimal path is denoted by thicker arrows.

These can be exploited for finding shortest paths without enumerating all paths.We start with a BF algorithm which uses a FIFO queue F containing nodes that need to be explored and an array $\overleftarrow{V}$ that contains temporary and, eventually, the final values of the cost-to-come. In this sense the algorithm is an incremental realization of equation (4.3). Unlike the path enumeration algorithms, every node is inserted only once into F, at the price of having to scan F each time a new successor is generated.

Algorithm 4.3 (BF Shortest Path)

$\overleftarrow{V}(s) = 0$
$\overleftarrow{V}(q) = \infty$ *for all* $q \neq s$
$Enq(F, s)$
repeat
  $q := Deq(F)$
  for *every* $q' \in \sigma(q)$ **do**
    if $q' \notin F$ **then**
      $Enq(F, q)$
    $\overleftarrow{V}(q') := \min\{\overleftarrow{V}(q'), \overleftarrow{V}(q) + w(q, q')\}$
  end
until $F = \emptyset$

The behavior of the algorithm on our example, when successors are generated from left to right is illustrated in Figure 4.6. The correctness of the algorithm stems from the fact that a node q reaches the front of the queue after all its predecessors has been explored and the shortest path that leads to it has been determined. Thus, at this point $\overleftarrow{V}(q)$ is indeed its cost-to-come. Algorithm 4.3 can be viewed as a specialized version of Dijkstra's algorithm, adapted for ranked acyclic graphs, whose complexity is linear in the size of the graph. For cyclic graphs one should indeed use Dijkstra's algorithm which is super-linear. For acyclic graphs which are not ranked, one should first perform a topological sort and generate nodes in an order consistent with the topological order, that is, never generate a node before its predecessor. This comes for free for ranked graphs.

As in the path enumeration algorithms, the algorithm can be run backwards, starting from f and using π instead of σ. In this case it will compute the cost-to-go (See Figure 4.7) The backward version can be viewed as a specialization of the Bellman-Ford algorithm for ranked acyclic graphs.

4.4 Depth-First Shortest Path

While the BF algorithm can be seen as an improved and non-redundant version of the BF path enumeration, the DF algorithm described below differs from its enumerative version in the sense that is computes the *cost-to-go* rather then the

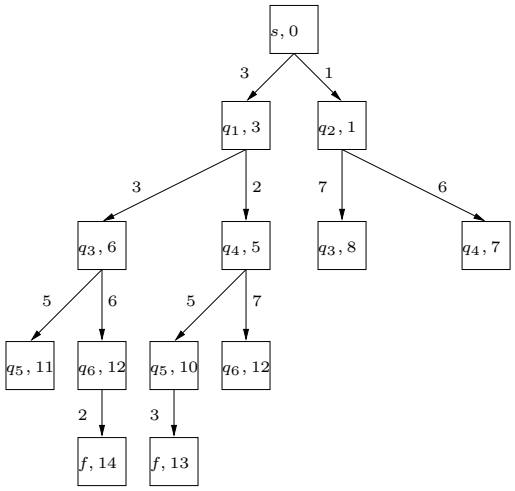


FIG. 4.6 – Computing cost-to-come by forward BF, interpreted as selective path enumeration. We show all valued nodes that have been computed, but at each level, when there are several copies for a node q, we show only the successors of the valued node (q, v) such that $v = \overleftarrow{V}(q)$

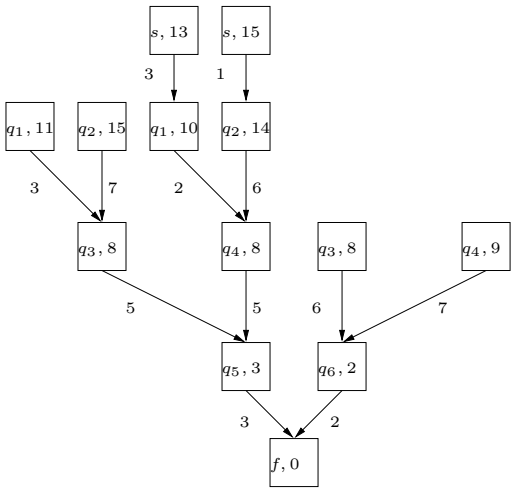


FIG. 4.7 – Computing cost-to-go by backward BF.

cost-to-come and can be seen as an algorithmic realization of equation (4.4). To explain why this is the only way to exploit Bellman's principle in a DF manner, let us look at the information obtained by a DF path enumeration algorithm after having explored the paths $s \cdot q_1 \cdot q_3 \cdot q_5 \cdot f$ and $s \cdot q_1 \cdot q_3 \cdot q_6 \cdot f$ (see Figure 4.1). At this point there is no node for which all the *incoming* paths have been exhausted but all the paths *outgoing* from q_3 have been explored.

Hence the DF algorithm goes down from s to f and when it "backtracks" from a node q it is an indication of having computed its cost-to-go, which is communicated to its predecessor as a possibility for computing the cost-to-go of the predecessor as in equation (4.4). The Bellman principle is exploited when q is later reached via another path and its already-computed cost is used instead of going down the graph again.

It is worth mentioning that the need to communicate the cost-to-go to one's predecessors corresponds to what is called in some programming circles *head recursion* where some computation needs to be done after backtracking, in contrast with the *tail recursion* inherent in the cost-to-come algorithm where accumulated values are sent down. This type of recursion requires more information to be kept on the LIFO stack which includes triples of the form $(q, Current, L)$ where q is a node, $Current$ is its successor being currently explored and L is a set of unexplored successors. In addition, the algorithm uses an array $\vec{V}$ to store temporary and final values of the cost-to-go as well as a bit-array $Finished$ to indicate whether the definitive cost-to-go of a node has already been computed.This prevents the value function from being recomputed when a node is revisited.

Algorithm 4.4 (DF Shortest Path)

$\vec{V}(f) := 0$
$\vec{V}(q) := \infty$ *for every* $q \neq f$
$Finished(q) := false$ *for every* q
$Push(S, (s, \perp, \sigma(s)))$
while S *is not empty* **do**
  $(q, Current, L) := Last(S)$
  if $Current = q'$ **then**
    $\vec{V}(q) = \min\{\vec{V}(q), w(q, q') + \vec{V}(q')\}$
  if $L = \emptyset$ **then**
    $Finished(q) := true$
    $Pop(S)$
  else
    $q' :=$ *some element in* L
    $L := L - \{q'\}$
    $Current := q'$
    if $\neg Finished(q')$ **then**
      $Push(S, (q', \perp, \sigma(q'))$
end

The behavior of the algorithm is illustrated in Figure 4.8 The DF algorithm is linear in the number of nodes. Amateurs of duality can develop a "height-first-search" version of this algorithm which computes cost-to-come by starting at f, going recursively to s and then propagating down.

This conclude the discussion on polynomial graph algorithms that work on graphs that are small enough the be given explicitly using some representation of δ or of its inverse (if we go backward). All this is very nice but not of much help for many real problems, including ours, where the graph is given implicitly as a product of many small graphs and the only viable solution is *not* to generate the whole graph beforehand but rather on-the-fly, and explore as few nodes as possible, without deviating too much from the optimum.

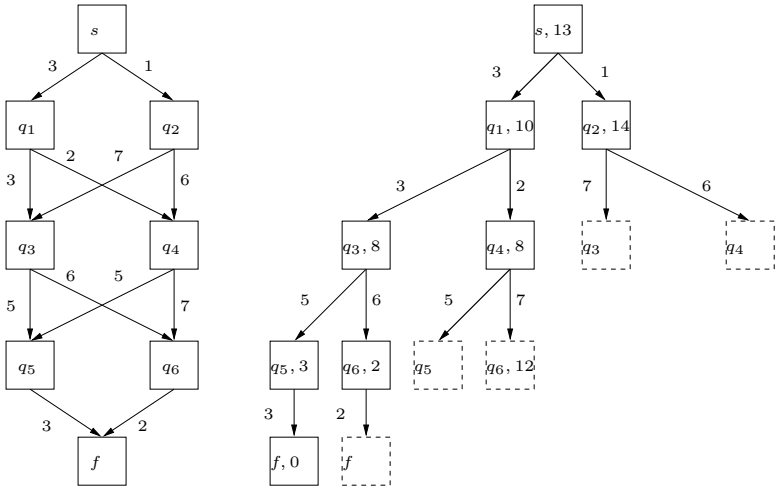


FIG. 4.8 – The behavior of the shortest path DF algorithm. Dashed nodes indicate visits to nodes after their cost-to-go has been computed. The cost is written inside nodes and (the explored part of) the optimal path is indicated by a thicker line

Chapitre 5

Shortest Paths in Acyclic Graphs : Heuristic Algorithms

In this chapter we present several heuristic versions of the generic algorithms of Chapter 4 in order to be able to find reasonable paths with bounded effort.

5.1 How to Direct the Search

Any search algorithm that does not run to completion, will be stopped after having explored only a subset of the set of paths and partial paths in the graph. It is our goal to direct the search toward interesting paths that are more likely to give the optimum. At the end of our time-bounded search we want nearly-optimal paths to be included in the explored subtree.

The most general situation where search guidance is manifested is the following. We have two (or more, not necessarily distinct) partial paths ξ and ξ' we have to choose which one to explore further, that is, continue with some $\xi \cdot q$ or $\xi' \cdot q'$. Let us use the cost-to-come notation $\overleftarrow{V}(\xi)$ to denote the length of a path. We want to define a measure of goodness for each of these paths according to which we make our decision. A natural measure would be something of the form

$$E(\xi \cdot q) = \overleftarrow{V}(\xi \cdot q) + V(q)$$

where $V(q)$ is some domain-specific function (see more details in Section 5.6) that approximates somehow the cost-to-go $\vec{V}(q)$, the length of the best path from q

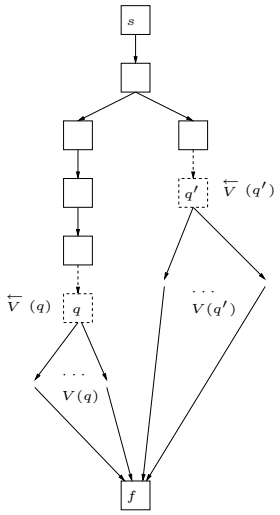


FIG. 5.1 – Choosing between $\xi \cdot q$ and $\xi' \cdot q'$.

to f (see Figure 5.1). In many cases $V(q) < \vec{V}(q)$ for every q : it is a lower-bound (optimistic estimation) of the cost-to-go. Note the $\overleftarrow{V}(q)$ is "a bird in hand", something that we know that can be achieved while $V(q)$ is a wishful thinking. Let us call algorithms that, this way or another, use E to guide the exploration as *best-first* algorithms and see the possible best-first variations of the BF and DF search strategies.

5.2 Best first BF

The standard BF algorithm explores paths in an order consistent with their logical length, that is, all paths consisting of k transitions will be explored before any path with $k+1$ transitions (see the left part of Figure 5.2). This is a major drawback of applying BF on large graphs because at some depth the number of partial paths explodes and we do not have any complete solution.

A best-first version of BF (used in [Ker02] and [AKM03]) maintains the queue ordered according to E. Let us consider first a degenerate version of E which only summarizes the past without "predicting" the future, $E(q) = \overleftarrow{V}(q)$. The influence of using this E as a selection criterion on the exploration order, amounts to deforming the path tree such that (vertical) distance between nodes corresponds

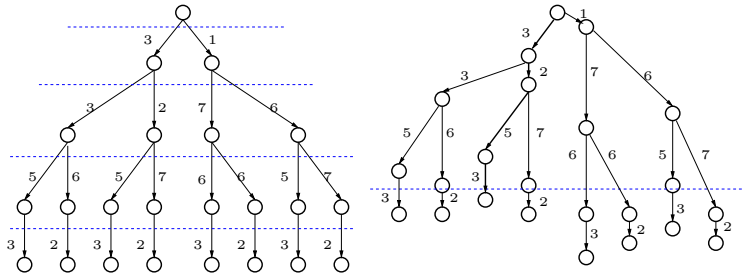


FIG. 5.2 – From breadth-first (left) to best-first (right) : paths are explored according to their metric, rather than logical, length.

to the length of the path between them (right of Figure 5.2). This way no path is explored before any shorter path. In particular, the optimal path (thick line) is not reached before all partial paths of smaller length) those above the horizontal dashed line) are explored. This gives a slight improvement compared to standard BF but not much. The reason is that any partial path which is explored deeper, becomes inferior to a less developed path. Hence the BF nature of the search, along with its main defficiency, is maintained.

Using $E(q) = \overleftarrow{V}(q) + V(q)$ we can distinguish between two partial paths $\xi \cdot q$ and $\xi' \cdot q'$ that have similar metric length but q is a more advanced state than q'. However, the BF nature of the algorithm still remains. For example, when ξ is much shorter than ξ', we are most likely to have $E(\xi \cdot q) < E(\xi' \cdot q')$ and we will have to postpone the exploration of ξ' and develop ξ. To direct the algorithm to go "deeper" we need to discount the optimistic nature of V by letting $E(q) = \overleftarrow{V}(q) + (1 + \alpha) \cdot V(q)$ where α is parameter representing the fact the $V(q)$ is an optimistic estimation. The choice of α may depend on the structure of the problem, for example on some ratio between the width of the task graph and the number of machines that characterize the plausibility of realizing the optimistic estimation. It may also depend on the length of the path so far.

5.3 Best-first DF

The unguided DF algorithm explores sons in an arbitrary order, for example leftmost-first. Hence when run on a large graph, it will explore some leftmost

subtree of the search tree. Replacing the arbitrary order by a best-first order (according to E), the explored paths will concentrate on the leftmost part of the re-ordered search tree. Not surprisingly the pros and cons of DF are the opposite of those of BF. It will produce many complete solutions but they will be all close to each other : they will all make the same decisions at early stages and will differ only in late decisions.

Another particularity of best-first DF is that it always compares *brothers*, paths of the form $\xi \cdot q$ and $\xi \cdot q'$. This property, combined with the particularity of graphs originating from scheduling problems may lead to a situation where $E(\xi \cdot q) = E(\xi \cdot q')$ for almost all q and q'. This is because *start* transitions do not affect $\overleftarrow{V}$ and V immediately, but this can be fixed either by collapsing several transitions or by using a more sophisticated E with some look ahead (see Section 5.6). As noted there, the solutions produced by best-first DF will be similar those produced by list scheduling in the sense that they will tend to prefer action over waiting. This can be overcome by a more intelligent V that will take the blocking effect of a *start* action into account.

5.4 Bounded-width Best-first BF

This is the variant used in our previous work [Ker02, AKM03]. It attempts to benefit from the advantages of BF (a diversity of paths) while avoiding explosion in the number of paths. At each level Q_i we select the best w paths, compute their successors, and select the best w paths among them and so on. The number w can be fixed or vary according to the level. Unlike DF, it is supposed to give some chance to paths that are not so promising in the first steps but can prove later to be good. However, upon closer inspection we see that the benefits of waiting are manifested only when we go to some depth. Consider a good path $\xi = s \cdot q_1 \cdots q_k \cdots f$ such that its prefixes up to $s \cdot q_1 \cdots q_k$ are inferior to their brothers and cousins in the corresponding levels. At early levels when the number of paths is small, the prefixes of ξ may be included in the best w paths, but as we go deeper, prefixes of ξ will have to compete with more and more cousins and are likely to be removed from the queue.

5.5 Best-first DF with Non-standard Backtracking (DFSBT)

If the previous section is an attempt to deform BF to become more depth like, non-standard backtracking makes DF more breadth like. The idea is illustrated in Figure 5.3. Suppose we go down along the leftmost (in terms of E) path. Instead of backtracking to the brother of the last node, we compare all the brothers of nodes along the path and backtrack to the best among them (note that they all are maintained in the stack). When the graph is a tree it is relatively-easy to see what goes on in terms of partial coverage of the path tree. Playing with the discount factor, one can make the system backtrack to arbitrary levels and finish the search with an interesting set of explored paths.

However when we deal with non-tree graphs a major difference with respect to classic DF is manifested : there when you backtrack from a node, you have computed its exact cost-to-go, and if you reach it through another path you can use the memorized value instead of going down again. This is not the case with non-standard backtracking.

Suppose that we backtrack to q_2' and then its best son is again q_3. Now we have several possibilities. We can take the value of q_3 (which is based only on one path emanating from q_3) and add it to the cost-to-come of q_2'. This is interesting only if the path to q_2' is shorter than the path to q_2. Alternatively we may use the opportunity to explore other continuations from q_2'. Which should we choose using E and the information gathered from the explored path ? Using E in the strict sense we will go down until the one before last branch and then take the brother of the already-explored leaf. How to combine already-accumulated knowledge is not a simple thing. One advantage of DF is that going down we discover something about conflicts and bottlenecks that we do not see immediately. Is there a way to summarize this knowledge in order to guide the search after backtracking ?

The algorithm can turn until no other neighbor path is better than the best path gotten. Solutions hence found are not necessarily optimal (because of the revisiting rule), but good.

Algorithm 5.1 is the main routine for $DFSBT$. It searches for an initial path by invoking the procedure *computePath* from the root q_0 of the graph. As long as the stop condition is false, it backtracks to a selected node, computes new partial

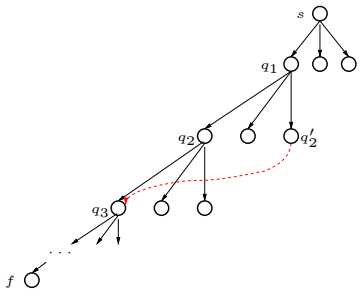


FIG. 5.3 – Non-standard backtracking : after exploring $s \cdot q_1 \cdot q_2 \cdot q_3 \cdots f$ we may backtrack to $s \cdot q_1 \cdot q_2'$.

path rooted at q_{choice} then improves the current path using *propagateUpward* if the new propagated cost is less than the current one. The algorithm is linear in the size of the graph and terminates with the best solution found.

Before giving the algorithm, let us introduce some notations :

q_0 : the initial state (the root of the graph),
q_{split} : the state where we want to update,
q_{choice} : the new choice that we want to make at q_{split}
$S(q)$: the choice made for q. $S(q) \in \sigma(q)$
$P(q)$: the parent node for q, if q is in the current path
$H(q)$: the cost of the current path starting at q
$T(q)$: the time to reach q in the current path

Algorithm 5.1 (Dfs with selective backtracking)

Algorithm *DFSBT*
begin
  $computePath(q_0, \perp, 0)$
  while *(¬stop condition)* **do**
    $q_{split} := \perp$,
    $q_{choice} := \perp$
    $findBackTrackNode(q_0)$
    $computePath(q_{choice}, q_{split}, T(q_{split}) + w(q_{split}, q_{choice}))$
    if $(H(q_{choice}) + w(q_{split}, q_{choice})) < H(q_{split}))$
      $S(q_{split}) := q_{choice}$
      $P(q_{choice}) := q_{split}$
      $propagateUpward(q_{split})$
    endif
  end while
end

The procedure *computePath* goes down recursively from a node q and constructs a complete path by selecting the best successor at every state using a selection rule $bestOf$. The choice made at q is memorized in $S(q)$. During the search process, the cost-to-come for each explored node is computed and stored in $T(q)$.

Several stop conditions have been implemented, and gives the algorithm an *anytime aspect* :

- no more improvement : the algorithm stops when all other backtracking nodes have greater estimation than the cost of the best strategy found.
- time bound : the algorithm stops when a time limit is released.
- memory bound : the algorithm stops when no more memory is available.

Procedure 5.1 (Computes a Path beginning from a node)

Algorithm *computePath*(q, p, t)
q *: the current state*
p *: the parent state of* q
t *: the cost to come at* q
begin
$P(q) := p$
$T(q) := t$
if q *is terminal*
  $H(q) := 0$
endif
if $S(q) \neq \perp$ *return*
else
  begin
    $q' := bestOf(\sigma(q))$
    $computePath(q', q, t + w(q, q'))$
    $H(q) := H(q') + w(q, q')$
    $S(q) := q'$
  end
endif
end

Note that each time a node q is encountered, the condition $S(q) \neq \perp$ is checked. This prevents from re-exploring an already chosen node ; because the selection rule is deterministic if leads necessarily to the same choice $S(q)$ and finally to the same cost $H(q)$.
The backtracking rule consists of choosing the most promising node likely to improve the current path among all the pending successors of each node in the current Path.

Procedure 5.2 (Selection of a backtracking edge)

Algorithm *FindBacktrackNode(q))*
begin
if q *is terminal*
  return
else
  begin
  select most promising $q' \in \sigma(q) \backslash \{S(q)\}$
  if $q_{choice} = \perp \vee q'$ *is most promising than* q_{choice}
    $q_{split} := q$
    $q_{choice} := q'$
  endif
  $FindBacktrackingNode(S(q))$
  end
end

The procedure is guided by the selection of the most promising node according to some estimation criterion. The estimation can be parameterized by the discount factor α as shown in section 5.2.
The procedure PropagateUpward updates the values of the current path, and stops at the initial state.

Procedure 5.3 (Backward propagation)

$PropagateUpward(q)$
begin
  $h := H(S(q)) + w(q, S(q))$
  $H(q) := h$
  if $P(q) \neq \perp$
    $PropagateUpward(P(q))$
end

5.6 Estimation function for DAG scheduling

All algorithms presented in the present chapter use an estimation function in the selection rule on the newly generated states. To be more specific, a state q in the global automaton is a generalized section in the task graph. From q, we have a finite set of possible actions, most of them are *start* transitions for enabled tasks, and one is a *wait* transition that terminate one or more tasks, leading hence to a new state. We can either use state-based or action-based measures. State based measures are more suited to breadth first search because there we have to compare successors of different states, while action-based measures are more suited for a depth first search where we compare successors of the same state.

Simple estimation functions $V(q)$ that give a lower bound on the time remaining until termination from a state, can be obtained by relaxing some constraints from the original problem. However, reader interested by more improved lower bounds is refered to [HC94, JR94, FB73]. The first, μ, relaxes resource constraints. It can be constructed by first associating with each task p the length $\mu(p)$ of the longest path in the task graph from p to some terminal task. Then, the estimation $\bar{\mu}$ of the value of the global state, is the maximum of μ over all tasks which are waiting or active in this state, i.e. the tasks that are in the section of the task graph.

The estimation for each task can be computed as follow :

$$\mu(p) = d(p) + \max_{p' : p \prec p'} \mu(p').$$

From μ we can define first estimation function $\bar{\mu}$ over the states and clock values of the $\mathcal{A}_p$ automaton by letting $\bar{\mu}(\overline{p}, c) = \mu(p)$, $\bar{\mu}(p, c) = \mu(p) - c$ and $\bar{\mu}(\underline{p}, c) = \mu(p) - d(p)$.

The second estimation, ν, relaxes the precedence constraints and gives the lower bound implied by the ratio between the amount of remaining work and the number of machines :

$$\nu(q) = \lceil \sum_{p \in P} (d(p) - c_q(p))/m \rceil,$$

where $c_q(p)$ is the amount of time elapsed from the beginning of p in q.

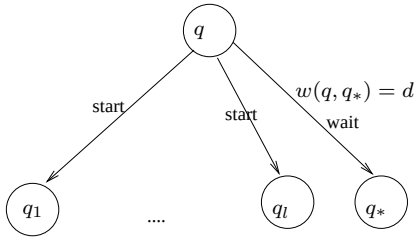


FIG. 5.4 – The *start* successors and *wait* successor of q

Observation 5.1 (Greediness) *Let us now investigate the use of these estimations in the DFS search. Let* $\sigma(q) = \{q_1, \cdots, q_l, q_*\}$ *be the successors of a global state* q *where* $q_1, \cdots, q_l$ *are the new generated states with* start *transitions, and* q_* *is the new state with the* wait *transition. The effect of* start *transition is not visible immediately and we will have* $\mu(q) = \mu(q_1) = \cdots = \mu(q_l)$ *as well as* $\nu(q) = \nu(q_1) = \cdots = \nu(q_l)$*. On the other hand, the effect of a* wait *transition of length* $w(q, q_*) = d$ *depends on whether tasks in the critical path are executing in* q *and on the number of active tasks* k *in* q*. In simple case we have* $\mu(q_*) = \mu(q) - d$ *(if no other path became critical) and* $\nu(q_*) = \nu(q) - \lfloor \frac{k}{m} \times d \rfloor$*. The cost-to-come is part of the estimation, this means that we have always* $E(q_*) \geq E(q_i) = E(q)$*. Equality is obtained when the time* d *added to the cost-to-come is removed from the estimation, when there is no idle critical path, and when* $k = m$ *i.e. when all machines are used in* q*. All these facts encourage greediness but do not suggest priorities among* q_i*'s.*

We can give the priority to the q_i which starts the critical p which has the maximum number of immediate successors in the task graph. This way, the solutions found by a depth first search are always better than those produced by critical path scheduling, and for which the *worst case performance is guaranteed.* In addition, solutions obtained by DFS heuristics employing this selection rule have great chances to produce solutions very close to the optimum at a very early stage of the search. Note that this does not necessarily mean that this heuristic generates only greedy schedules; playing with the parameter α of the estimation (see Section 5.2), one can let the algorithm backtrack to an edge which includes waiting. The question is then to know if it is better to include waiting sooner or later in the schedule. Answering such a question remains open for future work.

5.7 Experimental Results

We have implemented a tool that, given a task graph, it translates it into chains automata, and explores its underlying state space according to different strategies.

To appreciate the contribution of the chain decomposition based model, we have tested a depth first search with domination on particular task graphs which have a fixed chains structure. The method used is exhaustive. The results are summerized in table 5.1. As one can see, the time and memory consumption are drastically reduced.

It should be noted here that the problem becomes more difficult to solve (in terms of state space cardinality) when we have more machines because the state space with $m-1$ machines is included in that of m machines.

We have tested our heuristics on a set of benchmarks problems from [TKK00, TK02] having up to few thousands of tasks on an architecture Pentium 1.4 Ghz, 2 Gb of memory. The task graphs have different topologies and tasks's durations have different distributions. These benchmarks are issued from random generations as well as industrial applications such as robot control and compilation of programs. The results are shown in tables 5.2 and 5.3. As we can see, our chain decomposition algorithm, although not optimal, it gets a number of chains very close to the exact width. We also note that when the number of tasks is about thousands, the number of chains does not exceed few hundreds (see table 5.3).

The bounded width heuristic was tested with several values of w; the best results have been obtained by $w = 5$ and selected for presentation. The execution time was from few seconds for graphs with small number of chains to few minutes for graphs with great number of chains. As one can see, this heuristic gives good schedules for which the average deviation from the optimum is less than 0.21% (see table 5.2 column BFS-w). Let us mention the fact that a less quality schedules were obtained for $w \geq 5$.

The selective backtracking heuristic was implemented with several selection and backtracking rules. The results are depicted in table 5.2, column DFSBT. The time bound lets the algorithm turns until 7'30", although it can stops earlier for some instances or more for other, returning good solutions. Here again, the

Instance	T	C	m	*opt*	$\|Q\|$	$\|\Sigma\|$	t_{nc}	t_c	%	mem_{nc}	mem_c	%
3-10	31	3	1	60	4963	7262	0'00"	0'00"	0	0	0	0
			2	30	18403	37142	0'00"	0'00"	0	0	0	0
			3	20	28073	66152	0'00"	0'00"	0	0	0	0
3-15	46	3	1	90	15618	23042	0'00"	0'00"	0	0	0	0
			2	46	59403	120737	0'03"	0'01"	66.66	71 MB	42 MB	40.84
			3	30	92433	219827	0'06"	0'03"	50	142 MB	61 MB	57.04
3-20	61	3	1	120	35723	52922	0'03"	0'00"	100	68 MB	30 MB	55.88
			2	60	137603	280682	0'08"	0'01"	87.5	251 MB	95MB	62.15
			3	40	216343	516902	0'15"	0'04"	73.33	423 MB	106 MB	74.94
3-25	76	3	1	150	68278	101402	0'05"	0'02"	60	125 MB	50 MB	60
			2	76	265003	541727	0'18"	0'04"	77.77	600 MB	158 MB	73.66
			3	50	419303	1004627	0'35"	0'06"	82.85	1074 MB	274 MB	74.48
3-30	91	3	1	180	116283	172982	0'07"	0'01"	85.71	260 MB	74 MB	71.53
			2	90	453603	928622	0'38"	0'08"	78.94	1164 MB	247 MB	78.78
			3	60	720813	1730252	1'38"	0'16"	83.67	2152 MB	509 MB	76.34

TAB. 5.1 – The comparison table for the effect of chain decomposition on particular graphs. Column Instance designs the input example and colum C gives the number of chains. Columns T stands for the number of tasks. m, *opt*, $|Q|$ and Σ show respectively, the number of machines, the optimal solution, the number of generated states (w.r.t. domination) and the number of transitions in the state space reduced to non lazy runs. t_{nc}, t_c, mem_{nc} and mem_c stand for the time and memory consumption without and with chain decomposition. % stands for the percentage improvement

				BFS-w			DFSBT		
Instance	T	M	opt	best	%	time	best	%	time
*proto*151	80	16	**119**	**119**	0%	0'01"	**119**	0%	0'04"
*proto*054	158	12	**630**	**630**	0%	0'00"	**630**	0%	0'00"
*proto*001	473	4	**1178**	1182	0.3%	0'02"	**1178**	0%	0'04"
*proto*000	452	20	**537**	**537**	0%	0'01"	**537**	0%	0'00"
*proto*018	730	10	**700**	704	0.5%	0'03"	**700**	0%	7'30"
*proto*074	1007	12	891	894	0.3%	0'02"	947	6.28%	0'10"
*proto*027	1055	5	**2000**	2003	0.15%	2'22"	2001	0.05%	7'30"
*proto*021	1145	20	605	612	1.15%	0'02"	650	7.4%	3'05"
*proto*228	1187	8	**1570**	1574	0.25%	0'13"	**1570**	0%	7'30"
*proto*071	1193	20	629	634	0.79%	0'04"	656	4.29%	7'30"
*proto*026	1239	8	**1500**	1501	0.06%	1'44"	**1500**	0%	2'00"
*proto*025	1258	10	**1188**	1191	0.25%	0'13"	1189	0.084%	7'30"
*proto*271	1348	12	1163	1164	0.08%	0'06"	1189	2.235%	7'30"
*proto*028	1424	9	**1504**	1505	0.066%	0'38"	**1504**	0%	0'43"
*proto*237	1566	12	1340	1342	0.15%	0'10"	1357	1.268 %	7'30"
*proto*231	1694	16	-	1137	-	-	1183	-	3'35"
*proto*235	1782	16	-	1150	-	-	1165	-	7'30"
*proto*233	1980	19	1118	1121	0.268%	0'20"	1135	1.52%	7'30"
*proto*294	2014	17	1257	1261	0.31%	0'12"	1335	6.2%	7'30"
*proto*295	2168	18	1318	1322	0.3%	4'37"	1337	1.44%	7'30"
*proto*292	2333	3	**8009**	**8009**	0%	0'41"	8011	0.025%	7'30"
*proto*298	2399	10	2471	2473	0.08%	0'39"	2485	0.56%	7'30"

TAB. 5.2 – The results for the big examples. Column T, M represents respectively the number of tasks and the number of machines. *Opt* stands for the optimum. The rest of the table give respectively, the best solution found (best), the running time (time) and the deviation from the optimum (%) for respectively the bounded width heuristic BFS-w with $w = 5$ and the selective backtracking DFSBT. The time bound for DFSBT was set to 7'30"

heuristic gives schedules that are very close to the optimum, with average deviation less than 6.5%.

Due to its anytime aspect, the DFSBT heuristique was able to find more optimal schedules than the BFS-w.

Instance	T	C	width	%
*proto*151	80	16	13	23.07%
*proto*054	158	11	8	37.5%
*proto*001	473	125	106	17.92%
*proto*000	452	43	35	22.85%
*proto*018	730	175	137	27.73%
*proto*074	1007	66	52	26.92%
*proto*027	1055	788	783	0.63%
*proto*021	1145	88	74	18.91%
*proto*228	1187	293	239	22.59%
*proto*071	1193	124	95	30.52%
*proto*026	1239	674	641	5.14%
*proto*025	1258	282	230	22.60%
*proto*271	1348	127	97	30.92%
*proto*028	1424	455	392	16.07%
*proto*237	1566	152	117	29.91%
*proto*231	1694	101	79	27.84%
*proto*235	1782	218	167	30.53%
*proto*233	1980	207	159	30.18%
*proto*294	2014	141	108	30.55%
*proto*295	2168	965	892	8.18%
*proto*292	2333	318	243	30.86%
*proto*298	2399	303	229	32.31%

TAB. 5.3 – The greedy and exact width for the big examples. Column T, C and *width* represents respectively the number of tasks, the number of chains computed by our greedy algorithm and the exact width of the graph. % represents the deviation of C from the exact width

Deuxième partie

Scheduling under Uncertainty

Chapitre 6

Conditional Precedence Graphs

In the first part of the thesis, the model presented was *deterministic* in the sense that the set of tasks to be executed is known in advance to the scheduler. The only non-determinism in the specification of the problem comes from the scheduler's decisions and once they are chosen, the system exhibits a unique run/schedule with pre-determined start times for each task. In this second part, we extend the framework to a new problem of scheduling under uncertainty. The kind of uncertainty which we want to study comes from the fact that the set of tasks to be executed depends essentially on the "outcome" of other tasks which becomes known only after their termination. A typical case is the scheduling of programs with *if-then-else* instructions where evaluation of conditions and, hence, the choice of branches are not known in advance.

6.1 The problem

Consider the following program :

```
prog1 : input y
x0 := f0(y)
x1 := f1(y)
if x0 = 0
then
    x2 := f2(y)
    x3 := f3(y)
    x4 := f4(x2, x3)
```

Each f_i is a function (task) with a known computation time. All functions except f_4 depend only on some inputs available when the program is invoked and hence can be executed immediately. The functions have no side effects and their only dependencies are via argument passing. Using data-flow analysis we can deduce that f_4 cannot be executed before both f_2 and f_3 terminate. In addition, these three statements are executed *conditionally*, only if $x_0 = 0$ holds, a fact to be revealed only after the termination of f_1. This kind of situations can be captured by the *conditional precedence graph* (CPG). Figure 6.1 represents the CPG associated with the above program where each statement $x_i = f_i$ is represented as a task p_i with a given duration and the condition $x_0 = 0$ is modeled as a special Boolean task b with a zero duration. The precedence constraints between tasks are drawn as arrows and the conditional invocation of p_2, p_3 and p_4 is represented by arcs emanating from the boolean node b indicating under which value of b the tasks have to be executed (1 for b and 0 for $\neg b$).

Suppose we have to execute this program as quickly as possible on a two processors architecture, assuming negligible communications cost. At time $t = 0$ we have two tasks, p_0 and p_1, ready and we can start executing both. If after the termination of p_0 at $t = 3$, b evaluates to false, then all we need to do is to wait for the termination of p_1 at $t = 7$. If, however, b evaluates to true, p_2 and p_3 become enabled, but since p_1 still occupies one processor (we assume no preemption), we can only execute them sequentially, a fact that will delay the execution of p_4 resulting in termination at $t = 15$. The only reasonable alternative to this strategy is to postpone the execution of p_1 until $t = 3$ and then base our decision on the evaluation of b. If it turns out to be false, we start p_1 and terminate with a slight delay at $t = 10$. If, however, b is true, we have two free processors on which we

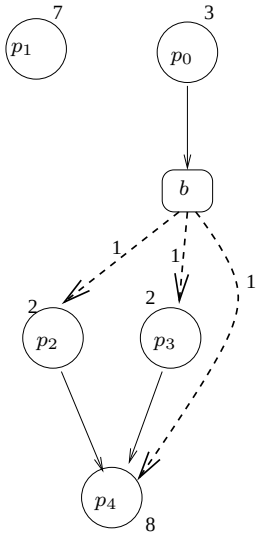


FIG. 6.1 – A conditional task graph representation of the program.

can execute p_2 and p_3 in parallel and only then execute p_1 and p_4 and terminate at $t = 13$. The schedules obtained by these two strategies (we call them s_1 and s_2, respectively) on the two cases (b and $\neg b$) are illustrated in Figure 6.2.

Which strategy do we prefer ? The answer depends on our evaluation criteria. If we want to be optimal with respect to the *worst case*, we will prefer strategy s_2 because $\max\{10, 13\} < \max\{7, 15\}$. If, however, we estimate the probability of b to be true by $\lambda \in [0, 1]$ and want to optimize with respect to the *average case*, we should compare the expected termination times of the two strategies, that is, $(1 - \lambda) \cdot 10 + \lambda \cdot 13$ and $(1 - \lambda) \cdot 7 + \lambda \cdot 15$, in order to choose. It is not hard to see that s_1 is preferable when $\lambda < 3/5$ and that s_2 is preferable otherwise.

6.2 Non Clairvoyant Scheduling

The only uncertainty in problem is due to the fact that not all tasks need to be executed at every invocation (instance) of the program. Whether or not some task should be executed is not known in advance but is revealed as the computation goes on. Hence the program admits a finite number of execution "scenarios", each

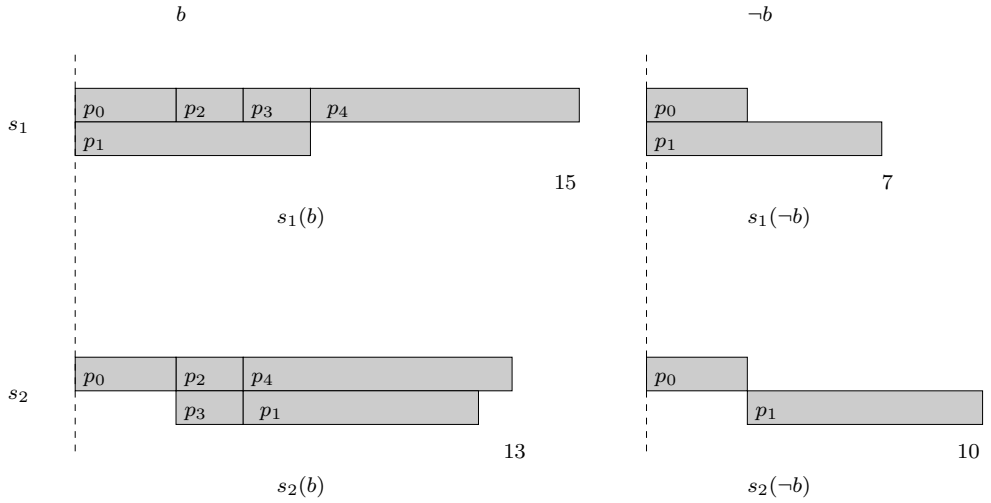


FIG. 6.2 – The results of applying strategies s_1 and s_2 to instances b and $\neg b$.

corresponding to a subset of the set of tasks. We call this scenarios *instances* of the scheduling problem. A naive approach to solve this problem would be to enumerate all instances and solve a (deterministic) scheduling problem for each of them, but this approach ignores the dynamic nature of the uncertainty and assumes a "clairvoyant" scheduler who sees into the future. The optimal schedule achieved by such a scheduler gives only a *lower-bound* on the worst-case computation time of the entire program. As an illustrations consider the CPG of Figure 6.3 : a clairvoyant scheduler can achieve an optimal solution of length 7 (Figure 6.4), while a non-clairvoyant scheduler cannot do better than 9 (Figure 6.5).

The fact that the uncertainty is *bounded* allows us to compute a dynamic scheduling strategy *off-line*, without the overhead associated with online re-scheduling. The scheduler can then be implemented as a simple add-on to the compiled code which is invoked when tasks terminate and decides, based on a pre-computed look-up table, which tasks to execute next (or to wait for the next event). In the next section we give a more detailed yet intuitive explanation of the abstract problem that we solve.

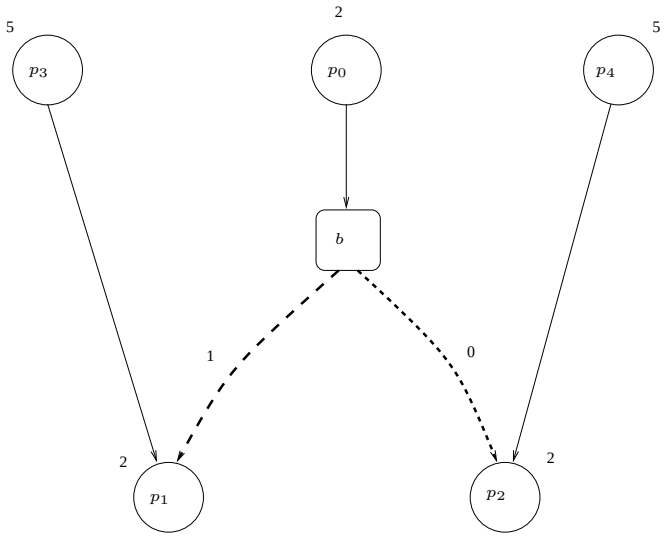
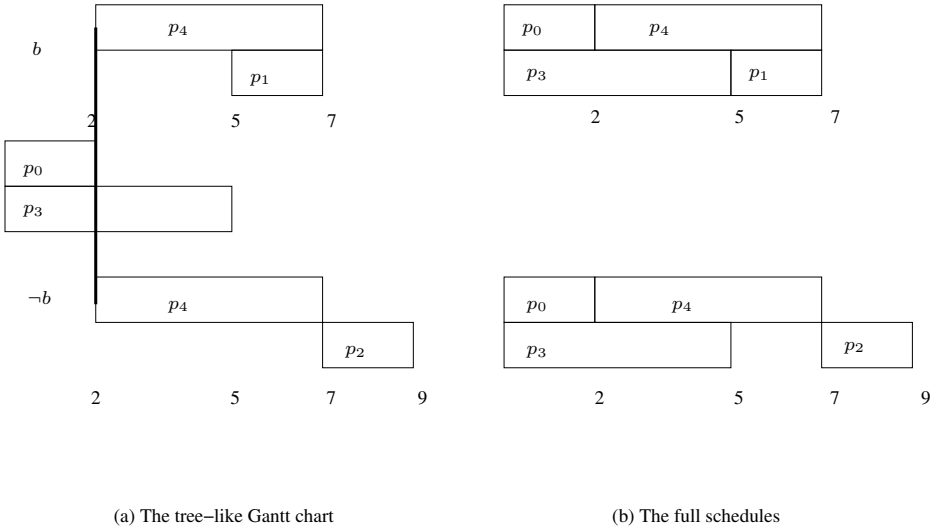


FIG. 6.3 – An example CPG.

(a) Optimal schedule for b

(b) Optimal schedule for instance $\neg b$

FIG. 6.4 – The optimal clairvoyant schedules for the CPG of figure 6.3.



(a) The tree−like Gantt chart (b) The full schedules

FIG. 6.5 – An optimal non clairvoyant strategy for the CPG of figure 6.3.

6.3 Conditional Precedence Graphs

In what follow we extend the task graph model (cf. previous chapters) to express conditional execution. This is done by introducing a special type of Boolean tasks with the following features :

1. They can be preceded by other tasks ;
2. They take zero time to execute ;
3. They terminate with a result which is either true or false ;
4. The execution of other tasks may depend on the results of the Boolean tasks ;

The second hypothesis can be relaxed if testing the condition takes non-negligible amount of time. Such tests can be decomposed into an ordinary task and a zero duration test.

To express the activation conditions of tasks we will use functions over a set B of Boolean variables that encode the results of the Boolean tasks. To simplify the presentation we restrict ourselves to the class $\mathcal{F}(B)$ of functions that can be written as a conjunction of positive and negative occurrences (literals) of distinct Boolean variables, e.g. $b_1 \wedge \neg b_2 \wedge b_3$. We denote by $V(f)$ the set of variables appearing in f. The partial order on Boolean functions is defined as $f \leq f'$ if

for every Boolean vector v, $f(v) \leq f'(v)$. Syntactically this means that the set of literals in f is a superset of those of f'. We say that f and f' are contradictory if $f \wedge f' =$ false, which is the case when at least one variable appears positively in one and negatively in the other. Note that the conjunctions can be evaluated to true only if *all* variable values are known, while their falsity can be sometimes deduced from partial information.

Definition 6.1 (Conditional Precedence Graph)
A conditional precedence graph (CPG) is $G = (P, B, \prec, A, d)$ *where* $P = \{p_1, \ldots, p_n\}$ *is a set of tasks,* $B = \{b_1 \ldots, b_m\}$ *is a set of Boolean tasks,* $\prec$ *is a strict partial order precedence relation on* $P \cup B$, $A : P \cup B \rightarrow \mathcal{F}(B)$ *is an activation function assigning a Boolean function over* B *to every task, and* $d : P \rightarrow \mathbb{N}$ *specifies task durations.*

We use notation A_p for $A(p)$ and say that task p is *less general* than p' if $A_p < A_{p'}$. We denote this fact by $p \sqsubset p'$. We say that a Boolean task b influences a task p if $b \in V(A_p)$ and denote it by $b \rightharpoonup p$.

Definition 6.2 (Consistent CPG) *A CPG is* consistent *iff the following two conditions holds*

- *No speculation : for every* $b \in B$, *and* $p \in P \cup B$ *: if* $b \rightharpoonup p$ *then* $b \prec p$. *No task can be executed before it is known whether it has to be executed.*
- *No contradiction : for every* $p, p' \in P$ *if* $p \prec p'$ *then* A_p *and* A'_p *are not contradictory.*

The first assumption can be relaxed if we want to move to the realm of *speculative execution*, used extensively in hardware. The idea is that if you have many processors you may save time by executing alternative conditional branches in parallel and then using only the outcome of the branches that really need to be executed. In that case we replace non-speculation with the weaker *non circularity* condition : if $p \preceq b$ then $b \not\prec p$. In other words, the termination of a task cannot be pre-requisite for determining whether it is to be executed.
We allow consistent CPGs to include precedence $p' \prec p$ when $p' \sqsubset p$, that is, p may depend on a task p' which might not be executed in all situations where p is. We interpret it as a *conditional dependency*, that is p' needs to wait for p only

when A_p evaluates to true. Nevertheless, we disallow precedence between tasks whose activation functions are contradictory.

Example 6.1 *Figure 6.6 represents the CPG of the following program :*

prog2 : inputs $(y_1, y_2, y_3, y_4, y_5)$
p_1 : $x_1 := y_1$
p_2 : $x_2 := y_2$
p_3 : $x_3 := f(x_2)$
b_1 : **if** $h(x_3)$
p_4 : $x_4 := g(x_2)$
else
p_5 : $x_5 := f_5(x_1)$
b_2 : **if** $h(x_5)$
p_6 : $x_6 := y_3$
p_7 : $x_7 := y_4$
else
p_8 : $x_8 := y_5$
p_9 : $x_9 := f_9(x_5)$
p_{10} : $x_{10} := f_{10}(x_1, x_3, x_4)$
p_{11} : $x_{11} := f_{11}(x_3, x_7)$

Ordinary arrows represent the tasks dependencies, and the set of dashed arc arrows entering each task represent the activation conditions. For example, although the activation condition of p_7 $Ap_7 = \neg b_1 \wedge b_2$ *is less general than* $Ap_{11} = true$*, the precedence* $p_7 \prec p_{11}$ *holds only if the branch is taken, but depends only on* p_3 *if not taken.*

6.4 Feasible schedules

The definition of feasible schedules for ordinary deterministic task-graph problems is simple. It is an assignment of start times to all tasks such that precedence constraints are satisfied and that the number of tasks active simultaneously

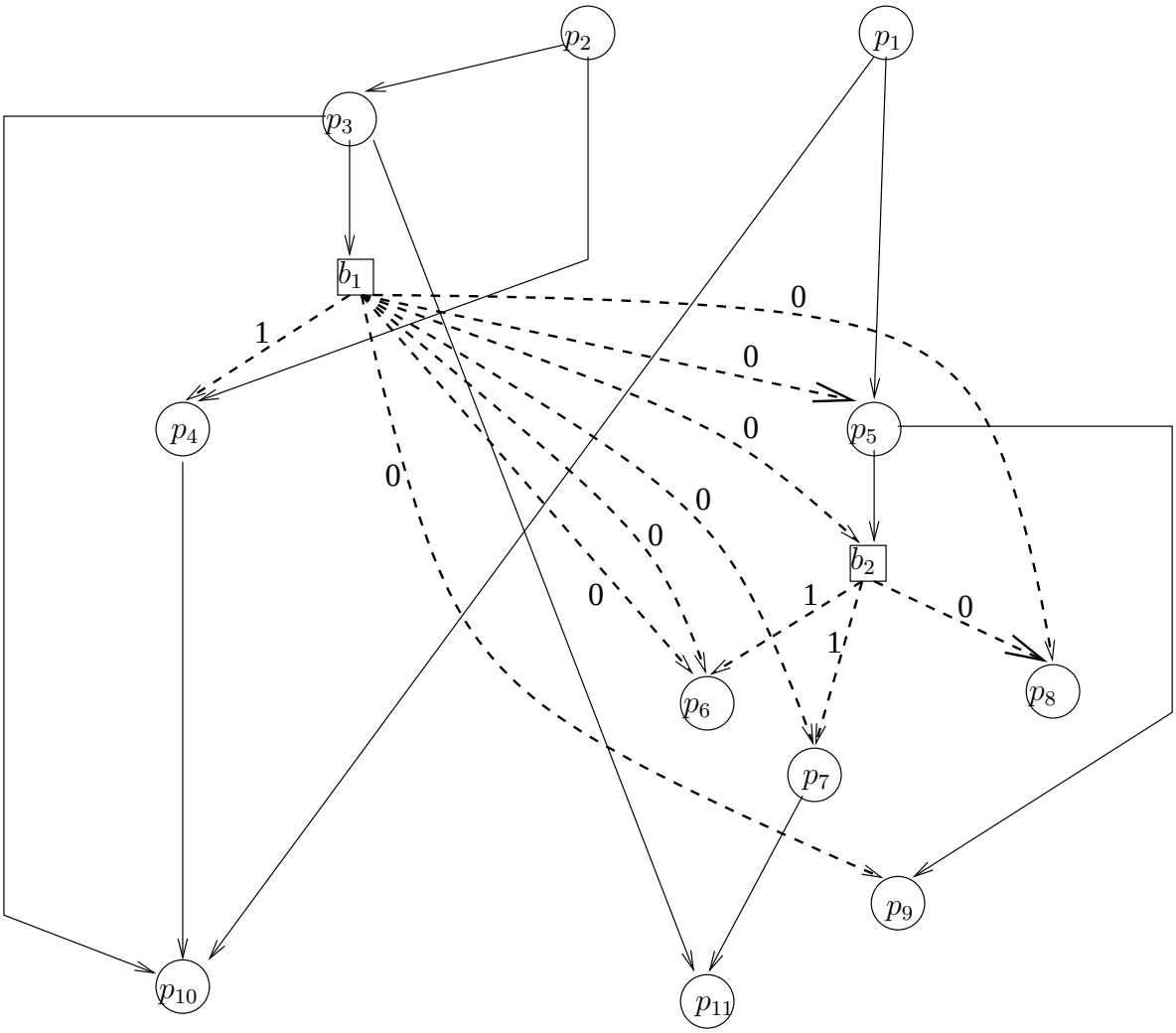


FIG. 6.6 – The CPG of the above program

at every moment is bounded by the number of machines. The adaptation of the definition to CPGs is more involved because different values of the B variables correspond to different sets of tasks to be executed.

An *instance* of the scheduling problem is an augmented Boolean vector $v : B \to \{0, 1, \star\}$ where $v(b) = \star$ ("don't care") indicates that $A_b(v)$ is false and b need not be executed. Such situations may occur when the program admits nested *if* statements. A partial instance is an instance which may be undefined for some variables whose values are not known yet. We say that v' *extends* v if it agrees with v on all variables defined in v. The set of tasks associated with an instance v is

$$P_v = \{p \in P \cup B : A_p(v) = true\}.$$

A *schedule* for an instance v is a function $st : P_v \to \mathbb{R}_+$ indicating the start times of tasks. From st we can derive for each task its termination time, $en(p) = st(p) + d(p)$ and its execution interval $I(p) = [st(p), en(p))$. The number of active tasks at time t is $\beta(t) = |\{p : t \in I(p)\}|$.

Definition 6.3 (Feasible Schedules) *A schedule st for an instance v is feasible on an architecture with k machines if*

1. *Precedence : for every $p \in P_v$, $st(p) \geq \max\{en(p') : p' \in P_v \wedge p' \prec p\}$. A task may start only after all its predecessors have terminated.*
2. *Resource constraints : for every $t \in \mathbb{R}_+$, $\beta(t) \leq k$. No more than k tasks may be active simultaneously.*

The length of a schedule is defined as the termination time of the last task, that is, $\max_{p \in P_v} en(p)$. We would like to obtain schedules that are optimal for all instances, but since instances reveal themselves progressively as more Boolean tasks are evaluated, we should restrict ourselves to *causal scheduling strategies* that can base their decisions only on information available at decision time.

At any given moment the state of a scheduling problem, of the type definable by a CPG, consists of the following information :

1. Which tasks have already been executed, and for the Boolean tasks also what their result was.
2. Which tasks are currently executing and for how long.

3. For which tasks it is known whether they should be executed.
4. Which tasks, among those that should be executed, are enabled for execution (all their predecessors have terminated and not all machines are busy).

   The state of the schedule determines which future evolution is possible. Passage of time increases the elapsed execution time of active tasks and allows them eventually to terminate. Termination of tasks may cause the evaluation of Boolean tasks and make some other tasks enabled. Starting a task, an action performed by the scheduler, moves a task from the waiting list to the active list and resets its timer. These actions belong to three categories :

   (a) *Deterministic actions* : these are actions that will always happen at certain states. They include termination of a task p exactly $d(p)$ time after its initiation, and the re-evaluation of a task activation function when some new Booleans terminate.

   (b) *Scheduler actions* : these are the decisions of whether or not to start an enabled task.

   (c) *Adversary actions* : the choices of the results of Boolean tasks on which we have no control. Note that they are, nevertheless, deterministic with respect to time and happen immediately after they become enabled.

   A scheduling strategy is thus a function that assigns to each state of the schedule one of the scheduler actions enabled at this state, including the special waiting "action" which means to do nothing and wait for the next event, while the active tasks keep on executing. In the next chapter we show how to extend the timed automaton modeling framework and use it to formalize the notion of a scheduling strategy, where the state of the schedule is represented by the state any clock values of the automaton.

Chapitre 7

Modeling Conditional Scheduling with Timed Automata

The model described in Chapter 3 was designed for (unconditional) task graphs. This model can easily be extended to express the behavior of Boolean tasks and their effects on the activation conditions for tasks. For the scheduling problem defined by CPGs, the resulting model is the composition of interacting automata consisting of timed automata for ordinary tasks and Boolean tasks. For the ordinary tasks we add an initial state expressing the fact that we don't know yet if the task have to be executed or not. Once the activation of a task evaluates to true, then the rest of the automaton is the same as the deterministic model

As in the deterministic case we first describe the basic model which is then reduced, by focusing on non-lazy and immediate schedules to a weighted directed graph. We also adapt the chain decomposition techniques to obtain a more efficient encoding of the automaton.

7.1 The basic model

The model for the CPGs scheduling problem is a product of interacting timed automata of two types : timed automata for ordinary tasks and automata for Boolean ones.

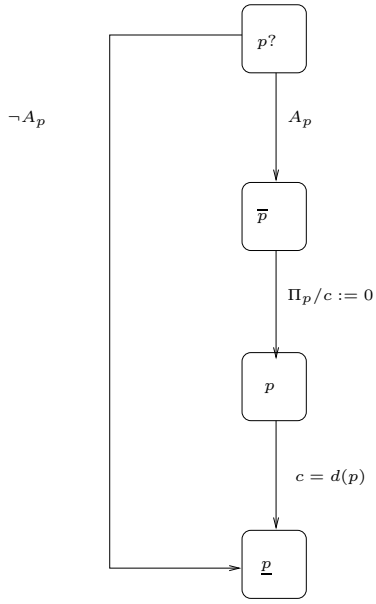


FIG. 7.1 – Modeling ordinary tasks with timed automata.

7.1.1 Modeling ordinary tasks

For each ordinary task p we construct a timed automaton $\mathcal{A}_p$ with four states and one clock c as depicted in Figure 7.1. State $p?$ is the initial state where it is not known yet whether p is to be executed. Once A_p evaluates to true, the automaton moves to a waiting state $\overline{p}$. We *can* leave this state and move to the active state p as soon as the condition Π_p holds, where Π_p is a conjunction of conditions indicating that for every $p' \prec p$, automaton $\mathcal{A}_{p'}$ is in its terminal state. Whether or not to take this transition when Π_p holds is a *decision of the scheduler* and when it is taken the clock is reset to zero. After spending $d(p)$ time in the active state the automaton moves to the terminal state $\underline{p}$. If A_p evaluates to false, the automaton goes from $p?$ directly to $\underline{p}$. In the rest of the section, we will use the notation $A_p =?$ to denote the fact that A_p is not yet evaluable. Formally we have,

Definition 7.1 (Timed automaton for ordinary task) *For every ordinary task $p \in P$ its associated timed automaton is $\mathcal{A}_p = (Q, \{c\}, I, \Delta, s, f)$ with $Q = \{p, p?, \overline{p}, \underline{p}\}$ where the initial state is $p?$ and the final state is $\underline{p}$. The staying conditions are* **true** *for $\overline{p}$ and $\underline{p}$, $c \leq d(p)$ in p and $A_p =?$ in $p?$. The transition relation*

Δ *consists of the four transitions :*

$$start : \quad (\overline{p}, \bigwedge_{p' \in \Pi(p)} \underline{p'}, \{c\}, p)$$

$$end : \quad (p, c = d(p), \emptyset, \underline{p})$$

$$activate : \quad (p?, A_p, \emptyset, \overline{p})$$

$$skip : \quad (p?, \neg A_p, \emptyset, \underline{p})$$

7.1.2 Modeling Boolean tasks

The automaton for each boolean task is shown in Figure 7.2 has four states : one initial state where the boolean task is not yet evaluated, and three terminal states, the state $b\star$ which indicates that the activation condition of b is false and hence it has not to be executed, and the states b and $\neg b$ where the automaton should move immediately to as soon as its activation condition evaluates to true and all its predecessors have terminated. The choice between these two transitions is the source of uncertainty in this scheduling problem. The evaluation of activation conditions of other tasks that mention b is done on the basis of the states of the corresponding $\mathcal{A}_b$ automata, where $b?$ is interpreted as "unknown" and $b\star$ as "don't care".

Definition 7.2 (Boolean automaton) *For each boolean task $b \in B$ its associated untimed automaton is $\mathcal{A}_b = (Q, \{\}, I, \Delta, s, f)$ with $Q = \{b?, b\star, b, \neg b\}$ where $b?$ is the initial state and the final states are b, $\neg b$ and $\star$. The staying conditions are* **true** *for the final states and $I_{b?} = A_b? \wedge \neg\Pi(b)$ for $b?$. The transition relation Δ consists on the following transitions :*

$$branch : \quad (b?, (\Pi_b \wedge A_b), \{b, \neg b\})$$

$$don't\ care : \quad (b?, \neg A_b, \star)$$

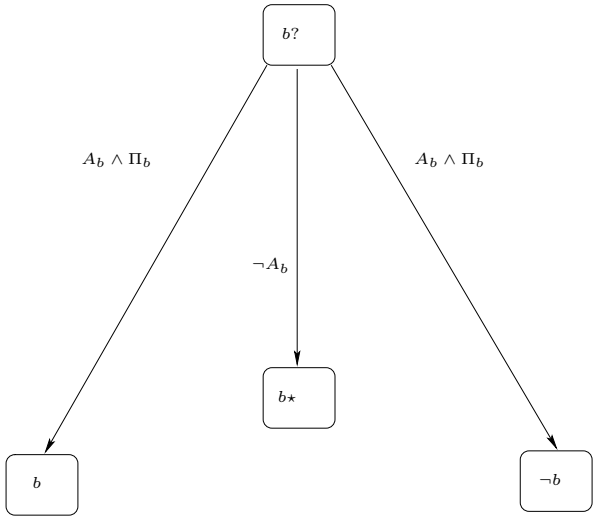


FIG. 7.2 – Modeling boolean tasks with automata.

7.1.3 The Global Model

The whole scheduling problem can be captured by the global automaton representing the mutual exclusion composition of all automata (ordinary tasks and boolean ones). Assuming k machines, the *conflicting* global states, which are tuples of the form $q = (q^1, \cdots, q^n)$, where more than k automata are in their respective active states, are forbidden. The precedence constraints are respected by construction, since an automaton is allowed to make its start transition only when the automata for its predecessors are in their final states. Here, every complete run in this global automaton $\mathcal{A}$ corresponds to a feasible schedule for a given instance, and the set of all runs of the automaton corresponds to all combinations of strategies and instances.

Definition 7.3 (Mutual Exclusion Composition) *Let* $\mathcal{A}^{p_i} = (Q^{p_i}, C^{p_i}, I^{p_i}, \Delta^{p_i}, s^{p_i}, f^{p_i})$ *be the automaton corresponding to each task* p_i *and* $\mathcal{A}^{b_j} = (Q^{b_j}, C^{b_j}, I^{b_j}, \Delta^{b_j}, s^{b_j}, f^{b_j})$ *be the automaton corresponding to each task* b_j*. Their mutual exclusion composition (assuming* M *machines)* $\mathcal{A}^{p_1} \| \ldots \mathcal{A}^{p_n} \| \mathcal{A}^{b_1} \| \ldots \| \mathcal{A}^{b_m}$ *is a timed game automaton* $\mathcal{A} = (Q, C, I, \Delta, s)$ *where* Q *is the restriction of* $Q^{p_1} \times \ldots \times Q^{p_n} \times Q^{b_1} \times \ldots \times Q^{b_m}$ *to non-conflicting states,* $C = C^{p_1} \cup \ldots \cup C^{p_n}$*,* $s = (s^{p_1}, \ldots, s^{p_n}, s^{b_1}, \ldots, s^{b_m})$*, the*

staying condition for a global state $q = (q_1, \ldots, q_n, q_{b_1}, \ldots, q_{b_m})$ *is* $I_q = I^{p_1} \wedge \ldots \wedge I^{p_n} \wedge I^{b_1} \wedge \ldots \wedge I^{b_m}$ *and the transition relation* Δ *can be formalized as follow : for all* $\mathcal{A}^{p_i}$ *task automaton :*

$$\frac{q_i \xrightarrow{\phi_1 \wedge \phi_2}{}^{\rho} q_i' \qquad (q_1, \ldots, q_i, \ldots, q_n, q_{b_1}, \ldots, q_{b_m}) \vDash \phi_1 \wedge q \notin Q_\wedge}{(q_1, \ldots, q_i, \ldots, q_n, q_{b_1}, \ldots, q_{b_m}) \xrightarrow{\phi_2}{}^{\rho} (q_1, \ldots, q_i', \ldots, q_n, q_{b_1}, \ldots, q_{b_m})}$$

and for all $\mathcal{A}^{b_j}$ *boolean automaton :*

$$\frac{q_{b_j} \xrightarrow{\phi_1} q_{b_j}' \qquad (q_1, \ldots, q_n, q_{b_1}, \ldots, q_{b_m}) \vDash \phi_1}{(q_1, \ldots, q_n, q_{b_1}, \ldots, q_{b_j}, \ldots, q_{b_m}) \longrightarrow (q_1, \ldots, q_n, q_{b_1}, \ldots, q_{b_j}' \ldots, q_{b_m})}$$

where $Q_\wedge = \{q \in Q / \exists j \in \overline{1,n} \;\; s.t. \;\; q_{b_j} = b_j? \;\wedge\; q \vDash \Pi_{b_j} \wedge A_{b_j}\}$*, is the set of states where a boolean may change*
$Q_\vee = Q \setminus Q_\wedge$ *represents the scheduler's decisions.*
A state q *is say to be* terminal *if it has no successor.*

Example 7.1 *Figure 7.3 shows a fragment of the global automaton obtained from the CPG of Figure 6.1. Starting from* $(\overline{p}_0, \overline{p}_1)$*, we choose to start* p_0*, leading to state* $(p_0, \overline{p}_1)$*. This state has two continuations, the first is to start* p_1 *which correspond to strategy* s_1*, and the second is to wait for the termination of* p_1 *which correspond to the strategy* s_2*. The termination of* p_0*, which leads to the choice of the value of* b *(dashed arrows), happens at different states in each case. For* s_1*, this happens when* p_1 *occupies one machine and consequently, when* b *is true* p_2 *and* p_3 *can only be executed sequentially, leading to a run of length* 15*. For* s_2 *this happens at state* $\overline{p}_1$ *and both* p_2 *and* p_3 *can be executed in parallel leading to a schedule of length* 13*. note that the fact that the* $start_2$ *and* $start_3$ *are executed one "after" the other is just an artifact of the interleaving semantics ; no time passes between these two transitions and they happen at the same time.*

7.2 Global Model as Game Graph

After having composed tasks automata together with Boolean ones, we obtain a global timed automaton where some of the actions are issued by the scheduler

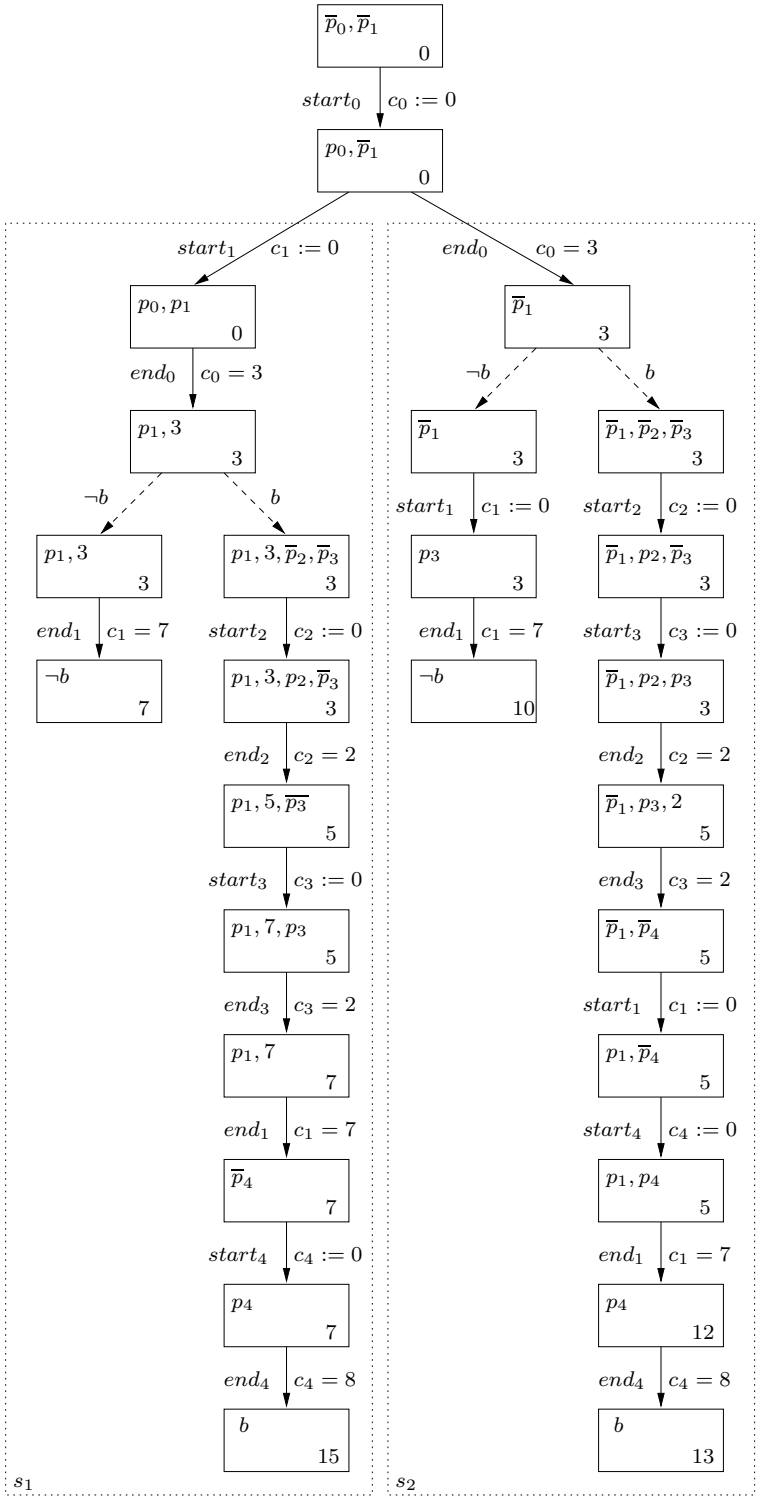


FIG. 7.3 – Part of the global automaton for the example of Figure 6.1. In the states we write only the waiting tasks, the executing tasks and the values of their non zero clocks. The numbers on the lower right corners stand for the total elapsed time to reach the state via a non-lazy run. The branches correspond to the schedules of Figure 6.2.

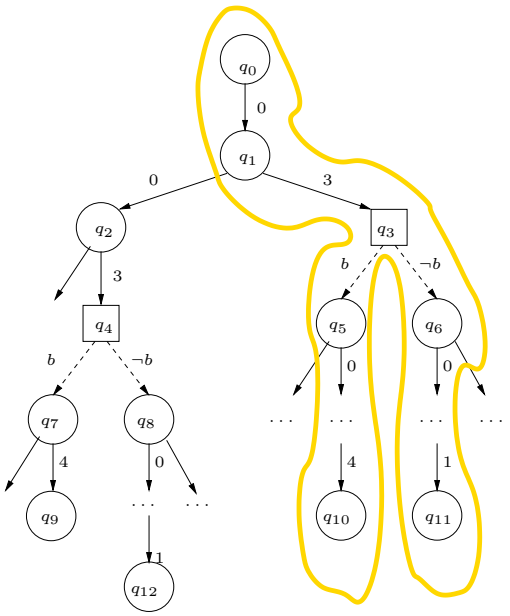


FIG. 7.4 – Part of the game graph for our example, with AND nodes denoted by squares. The sub-graph of strategy s_2 is marked.

(the *start* transitions) and some by the environment (the *branch* transitions). This is a special case of the timed game automaton of [AMPS98]. After restricting this automaton to non-lazy or immediate runs, the automaton is reduced into a kind of a discrete game graph, also known as AND-OR graph [Nil71], [Zha99] or alternating automaton. Such a graph has two types of nodes, the OR nodes which corresponds to global states where the scheduler ("player 1") has to choose an enabled transition, and the AND nodes corresponding to states where the adversary ("player 2") chooses between b and $\neg b$ transitions. Figure 7.4 shows part of the the game graph obtained for our example.

The game automaton can be viewed as a special case of an acyclic game graph called AND/OR graph which is defined as follow :

Definition 7.4 (AND/OR graph for timed game automaton) *Let* $\mathcal{A} = (Q = Q_\vee \cup Q_\wedge, C, I, \Delta)$ *be a timed game automaton ; its corresponding weighted AND/OR graph is the acyclic directed graph* $G = (Q' = Q'_\vee \cup Q'_\wedge, \delta = \delta_\vee \cup \delta_\wedge, w)$ *where* $Q' = Q \times \mathbb{R}^{|C|}$ *is a finite set of nodes partitioned into AND and OR nodes respectively* $Q'_\wedge = \{(q, v_1, \ldots, v_n)/q \in Q_\wedge\}$ *and* $Q'_\vee = \{(q, v_1, \ldots, v_n)/q \in Q_\vee\}$,

$\delta \subseteq Q \times \mathbb{R}^+ \times Q$ is a set of directed edges (also called transitions) partitioned into AND and OR transitions, and w is a weight function on edges. Edges and their corresponding weights are :

$$\frac{q \xrightarrow{\phi_t \rho} q' \in \Delta \quad (v_1, \ldots, v_n) \vDash \phi_t}{(q, v_1, \ldots, v_n) \xrightarrow{0} (q', v_1', \ldots, v_n')}$$

$$v_i' = \begin{cases} v_i & if\ c_i \notin \rho \\ 0 & otherwise. \end{cases}$$

and

$$\frac{\forall \theta', \theta' \leq \theta \Rightarrow I_q(v_1 + \theta', \ldots, v_n + \theta')}{(q, v_1, \ldots, v_n) \xrightarrow{\theta} (q, v_1 + \theta, \ldots, v_n + \theta)}$$

We use the same notations as in chapter 4. The outcome of applying a particular strategy is represented as a binary tree-like partial sub-graph rooted at the initial state in which every OR node has exactly *one successor* and every AND node has *all its successors* (see Figure 7.4). The worst case performance of such a strategy, also known as its *value*, is the length of the longest path in that sub-graph.

The game graph of the automaton we have to deal with doesn't contain any directed cycle. We use G_q to denote the sub-automaton rooted at state q. G_q represent the "residual" game which remains to be played after reaching q. A state $q \in Q$ is said to be terminal if it has no successor ($\sigma(q) = \emptyset$).

Definition 7.5 (partial strategy)
A strategy *is a partial function $s : Q_\vee \to Q$, such that*

$$s(q \in Q_\vee) = q'/q' \in \sigma(q)$$

It can be represented as a tree like sub-graph in which for every $q \in Q_\vee$ all $\delta_\vee$ transitions emanating from q are removed except for $s(q)$.

A partial strategy is a strategy function which is not defined for all nodes in $Q_\vee$

A particular property of the game graphs obtained for CPGs is that all strategies define sub-graphs which are trees with the same type of binary branching.

Definition 7.6 (Path according to a strategy) *is a path* $\pi = q_1, \cdots, q_n$ *such that*

$$\forall i / q_i \in Q_\vee : q_{i+1} = s(q_i)$$

Each time we reach an OR state, we follow the successor indicated by the strategy function.

Definition 7.7 (Cost of a strategy)
The cost of a given strategy s *beginning from a state* q *is the cost to go from* q *which is computed as follow :*

$$\|s\| = \vec{V}(q) = \begin{cases} 0 & \text{if } q \text{ is terminal} \quad (1) \\ w(q, s(q)) + \vec{V}(s(q)) & \text{if } q \in Q_\vee \quad (2) \\ \max_{q' \in \sigma(q)} w(q, q') + \vec{V}(q') & \text{if } q \in Q_\wedge \quad (3) \end{cases}$$

In other words, the cost of s is equal to the cost of the longest path in the tree defined by s. Hence, to compare two given strategies s_1 and s_2, it suffice to compare the costs of their longest paths.

7.3 Non Lazy Strategies

The model described previously is quite inefficient for at least two reasons. The first is that due to the staying condition in the waiting state, the scheduler may decide to wait an arbitrary amount of time before taking its transition to start a given task. We have seen in Chapter 2 that waiting is useless in general, and hence, we can try to understand better which waiting is useless in the context of adversarial scheduling. The second reason is that the parallelism of the CPG can be exploited to understand how the chain decomposition can be applied in this context.

This section will focus on new theoretical results characterizing the domination properties between types of strategies. From these results, we will introduce a new model for our tasks which restrict the model to the non-lazy runs and consequently avoid useless waiting. Moreover, we will extend the Graham bound from the non-adversarial case and show that the greedy approach is a 2-approximation algorithm.

7.3.1 Types of strategies

We generalize definition and results of the Chapter 2 concerning types of schedules to types of strategies.

A strategy is said to be *feasible* if for all instance v of boolean variables, the corresponding schedule is feasible. We say that a set of strategies is *dominant* if it always contains an optimal solution.

Definition 7.8 (Immediate strategy) *An immediate strategy is a feasible strategy where none of the tasks can be locally left shifted in every schedules defined in that strategy except for tasks that activate booleans.*

This definition suggest the following lemma :

Lemma 7.1 *The set of immediate strategies IS is dominant*

Proof : for all instances v of boolean variables, the schedule st_v in the strategy can be transformed in a new schedule $st_v^{'}$ where all termination times of every tasks are at most equal to those in st_v ; hence $C_{max}(st_v^{'}) \leq C_{max}(st_v) \leq \max_v C_{max}(st_v)$. Since the sequence of the boolean variables is preserved in the new strategy $st_v^{'}$ by definition, then the value of $st_v^{'}$ is at worst decreased. ⌟

Definition 7.9 (semi left-shifted strategy) *A semi left-shifted strategy is a feasible strategy in which no global left shift is possible while preserving the logical sequence of boolean variables.*

Note that a global left shift of a tasks which activate a boolean variable may lead to a deterioration of the sequence of the boolean variables, which may lead to another branching structure of the strategy.

Since the branching structure does not change during a global left shift, the following result holds :

Lemma 7.2 *The set of semi left-shifted strategies SLSS is dominant*

The notion of semi left-shifted strategy is not so strict, in the sense that due to preservation of the boolean sequence, a global shift is always possible for tasks which activate booleans without delaying the starting times of the remaining tasks of all sub schedules in that strategy. We try to strengthen the notion of semi left-shifted strategy by introducing another type of strategies which we call *non lazy strategy.*

Definition 7.10 (Non-lazy strategy) *A non-lazy strategy is a feasible strategy in which no local or global shift is possible, nor a left shift which may change the branching structure.*

Lemma 7.3 *The set of non-lazy strategies NLS is dominant*

Proof : this can be proved by showing that every feasible strategy st_L can be transformed into a non-lazy strategy st_{NL} by a sequence of local and global shifts. This means that every complete schedule defined by all instance of the strategy S_L can be globally left shifted without increasing the value of the strategy. Suppose we want to shift a task p_j earlier. If p_j does not activate a boolean task then we are done. Suppose now that p_j activates a boolean task, then the sequence of boolean variables may change[1]. However, its effect is not important because the new schedules obtained by the shift of p_j in every instance are all at least as good as the older ones, consequently, the max of them, which define the cost of the strategy, is at least as good as the previous strategy. ▟

Figure 7.6 shows a global left shift of the task p_2 which changes the activation order of booleans b_1 and b_2.

Theorem 7.1

1. *Non delay strategies* $\subseteq NLS \subseteq SLSS \subseteq IS$
2. *Optimal strategies* $\cap\, NLS \neq \emptyset$

It should be noted that theorem 7.1.2 says that it suffice to explore only the set NLS because it always contain an optimal solution, although some solutions which are optimal may exist outside NLS.

[1]the fact that we can shift a boolean task earlier than another means that they are mutually independent

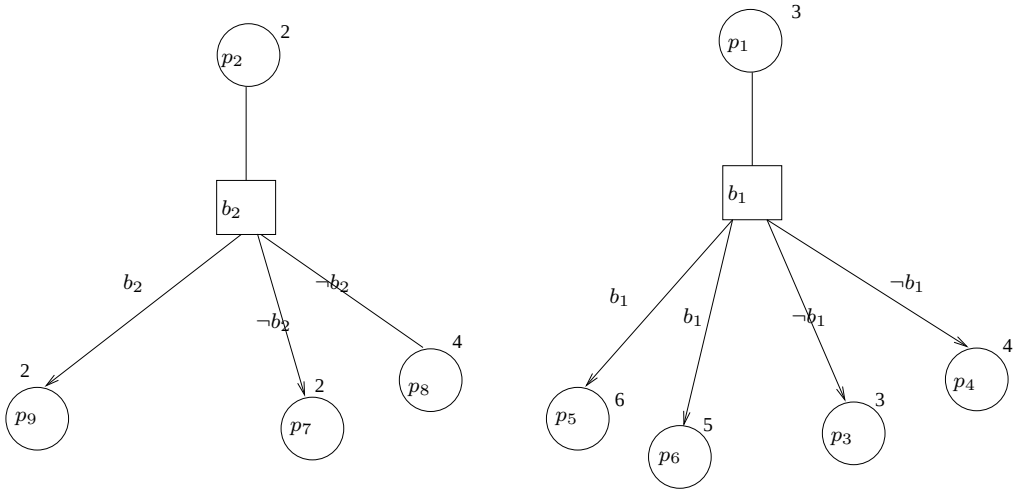


FIG. 7.5 – An example CPG.

7.3.2 Greedy strategies

Let us denote st_v the schedule for an instance v, and $C_{max}(st_v)$ the completion time of st_v. We say that a strategy is greedy if every schedule st_v under all instances in that strategy is a non-delay one. Although the set of greedy (non-delay) strategies are not necessarily optimal, we can show that any greedy approach constitutes a 2-approximation scheme for the CPG scheduling problem.

Theorem 7.2 *Let $C_{max}(st^L)$ be the length of a greedy strategy for the CPG $G = (P, B, \prec, A, d)$. Then*

$$C_{max}(st^L) \leq (2 - 1/m) \times C_{max}(st^{opt})$$

Where $C_{max}(st^{opt})$ is the cost of the worst case optimal strategy

Proof : Let st^{opt}_{v*} be the longest schedule in st^{opt}, where $v*$ stands for the corresponding instance of boolean variables i.e.

$$C_{max}(st^{opt}_{v*}) = \max_{v \in \{0,1\}^n} C_{max}(st^{opt}_{v}),$$

Now let v^L the instance of booleans such that $st^L_{v^L}$ is the longest schedule in st^L ; and let $G = (P_{v^L}, \prec, d)$ be the task graph for the instance v^L of boolean

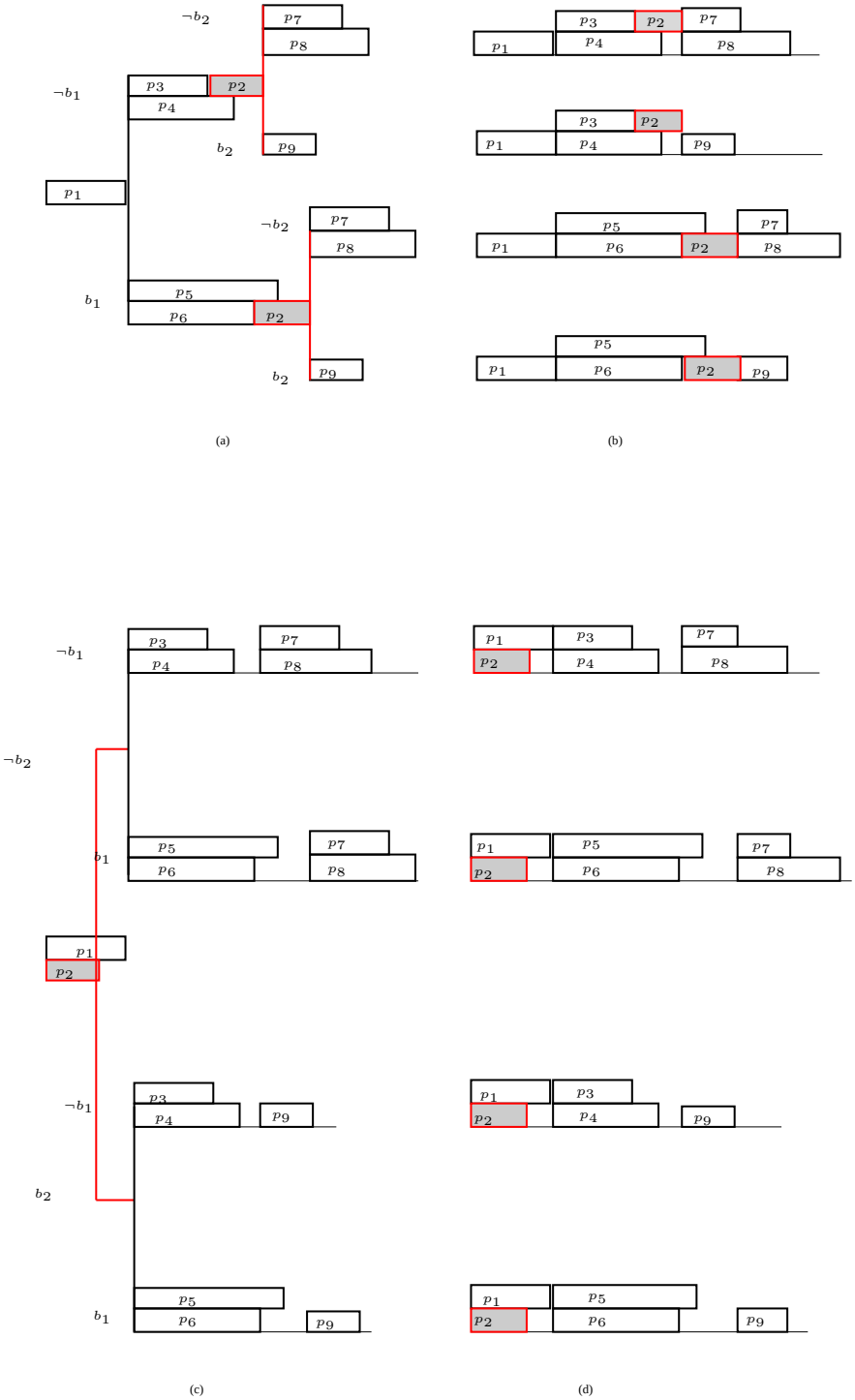


FIG. 7.6 – A possible strategy for CPG of Figure 7.5 and a left shift of task p_2. (a) and (c) are the tree-like schedules; (b) and (d) are the full schedules

variable ; and let tm be the total idle time in $st^L_{v^L}$. By lemma 2.1, we know that there exists a path c in the CPG such that :

$$tm \leq (m-1) \times \sum_{p \in c} d(p)$$

We know that

$$m \times C_{max}(st^L_{v^L}) = tm + \sum_{p \in P_{v^L}} d(p)$$

On the other hand,

$$\sum_{p \in P_{v^L}} d(p) \leq \max_{v \in \{0,1\}^n} \sum_{p \in P_v} d(p)$$

by the work preservation principle we have :

$$\max_{v \in \{0,1\}^n} \sum_{p \in P_v} d(p) \leq m \times C_{max}(st^{opt})$$

hence,

$$\begin{array}{ll} m \times C_{max}(st^L) & = tm + \sum_{p \in P_{v^L}} d(p) \\ & \leq (m-1) \times \sum_{p \in c} d(p) + m \times C_{max}(st^{opt}) \\ & \leq (m-1) \times C_{max}(st^{opt}) + m \times C_{max}(st^{opt}) \\ C_{max}(st^L) & \leq (2 - 1/m) \times C_{max}(st^{opt}) \end{array}$$

Although the set of non-delay strategies doesn't always contain an optimal solution, this theorem claims that any non-delay strategy is guaranteed to be within half to the optimum (the worst case guarantee).

7.3.3 Restricting to non-lazy strategies

Since the set of feasible schedules is infinite, the number of executions in the timed automaton is also uncountable due to the staying condition in the waiting state $\overline{p}$ which is always true. A key results from [AAM06] allows us to consider only a *finite subset* of the runs that we call *non-lazy runs*. This result can be generalized to our problem using lemma 7.3 as follow,

Definition 7.11 (Lazy run) *A lazy run is a run in which at some state the scheduler hesitates some time before starting a task, while the global state remains the same during that waiting period.*

Note that this definition does not mean that the all waiting are useless. For example, the $start_1$ transition from state $(p_0, \overline{p}_1)$ (cf. Figure 7.3) can be taken at any time before condition $c_0 = 3$ becomes true, i.e. anywhere in the interval $[0, 3]$. Such a waiting could be useful if something has changed during this waiting, for example *all machines become blocked* or *the value of a boolean task was revealed.* But if the delayed action is taken in the *same global state*, the waiting is useless and the schedule can be replaced by another schedule of a lesser or equal length in which the lazy action is taken as soon as it is enabled.

Restricting the state space to non-lazy strategies amounts to transforming the timed automaton in order to detect and avoid lazy runs. Whenever a global state has several outgoing transitions, the continuations to consider are those in which a *start* transition is taken immediately, and those in which time (and clock values) advances by the exact amount needed to satisfy the condition for the nearest *end* transition. State $(p_0, \overline{p}_1)$, for example, has two continuation, one is a result of starting p_1 immediately and the other is the result of waiting 3 time units until $c_0 = 3$ and the end_0 transition is taken.

The above observation allows us to consider only a finite subset of runs corresponding to the finite subset of non-lazy schedules. In this work we attempt to strengthen the definition of non-laziness [Abd02] to captures the useless waiting only during the time processing of the modeled task. For example, if the scheduler decides to not start a task p_j at time t when there are available machines, then if there is no blocking situation (all machines becomes occupied) or none of the booleans became activated in the interval $[t, t + d_j]$, then the run is lazy.

The model of Figure 7.7 shows the timed automaton for a task p_j, modeling the laziness detection and avoidance. The automaton associated with each task $p \in P$ is $\mathcal{A}^p = (Q^p, \{c\}, I, \Delta, s, f)$ with $Q = \{p?, p, \overline{p}, \underline{p}, \dot{p}, p_{lazy}\}$ where the initial state is $p?$ and the final state is $\underline{p}$. In this automaton all transitions are taken as soon as possible. The staying conditions are as follow :

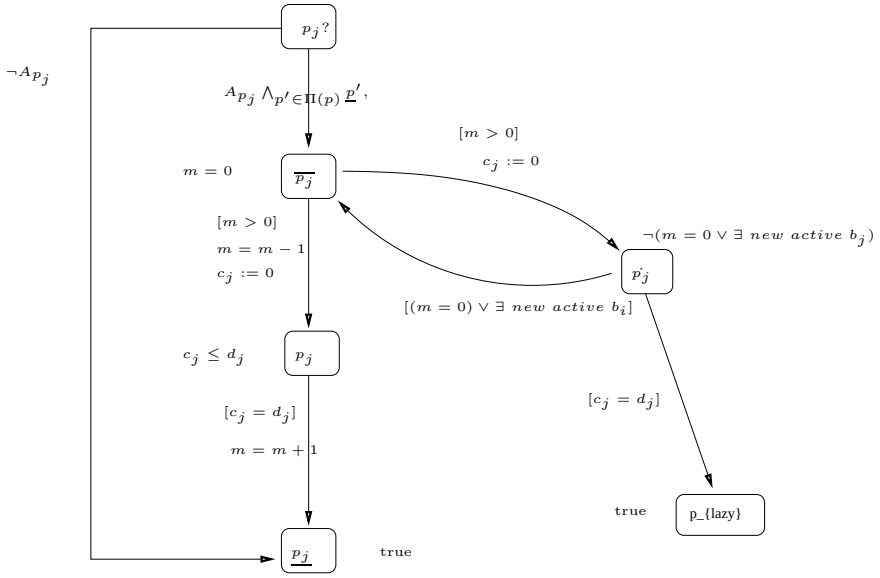


FIG. 7.7 – laziness avoidance.

$$I(p?) \equiv A_p?$$

$$I(\overline{p}) \equiv m = 0$$

$$I(p) \equiv c \leq d(p)$$

$$I(\underline{p}) \equiv \mathbf{true}$$

$$I(\dot{p_j}) \equiv c_j \leq d_j \wedge \neg(\exists\ new\ active\ b_i \vee m = 0)$$

$$I(p_{lazy}) \equiv \mathbf{true}$$

The transition relation Δ consists of the following transitions :

$$activate : \qquad (p?, A_p, \emptyset, \overline{p})$$

$$skip : \qquad (p?, \neg A_p, \emptyset, \underline{p})$$

$$suspend : \quad (\overline{p}, \bigwedge_{p' \in \Pi(p)} \underline{p'}, \{c\}, \dot{p})$$

$$resume: \quad (\dot{p}, \exists\ new\ active\ b_i \vee m = 0, \emptyset, \overline{p})$$

$$cut: \quad (\dot{p}, c = d(p), \emptyset, p_{lazy})$$

$$start: \quad (\overline{p}, \bigwedge_{p' \in \Pi(p)} \underline{p'}, \{c\}, p)$$

$$end: \quad (p, c = d(p), \emptyset, \underline{p})$$

Where m stands for the number of machines which become available at any instant.

As soon as a task automaton reaches a lazy state, the global state is cut.

Since we favor the generation of greedy strategies, the question whether it is better to block task or not is very relevant, but unfortunately not yet answered in this thesis.

7.4 Chain Decomposition

As for the deterministic problem, we can benefit from the chain decomposition when implementing the model. The boolean tasks are considered as usual as well as the boolean dependencies. Here again, the precedence constraints are not completely removed by construction, so we need additional data structure to keep track of the finished tasks at each global state during the composition process. The activation constraints are completely preserved in the chain, so no additional information is needed. As an example, the CPG described of Figure 7.8 can be represented as a special kind of precedence graph on which we can apply the chain decomposition algorithms. Figure 7.9 shows the automaton for each chain. The number of states for each intermediate task is reduced by one since the final state $\underline{p}$ is not always needed.

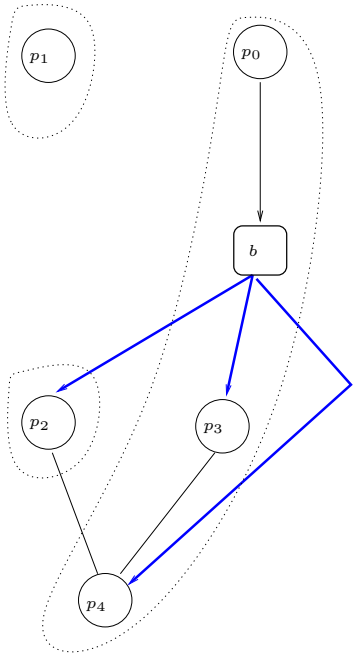


FIG. 7.8 – The new CPG on which we can apply the chain decomposition algorithm.

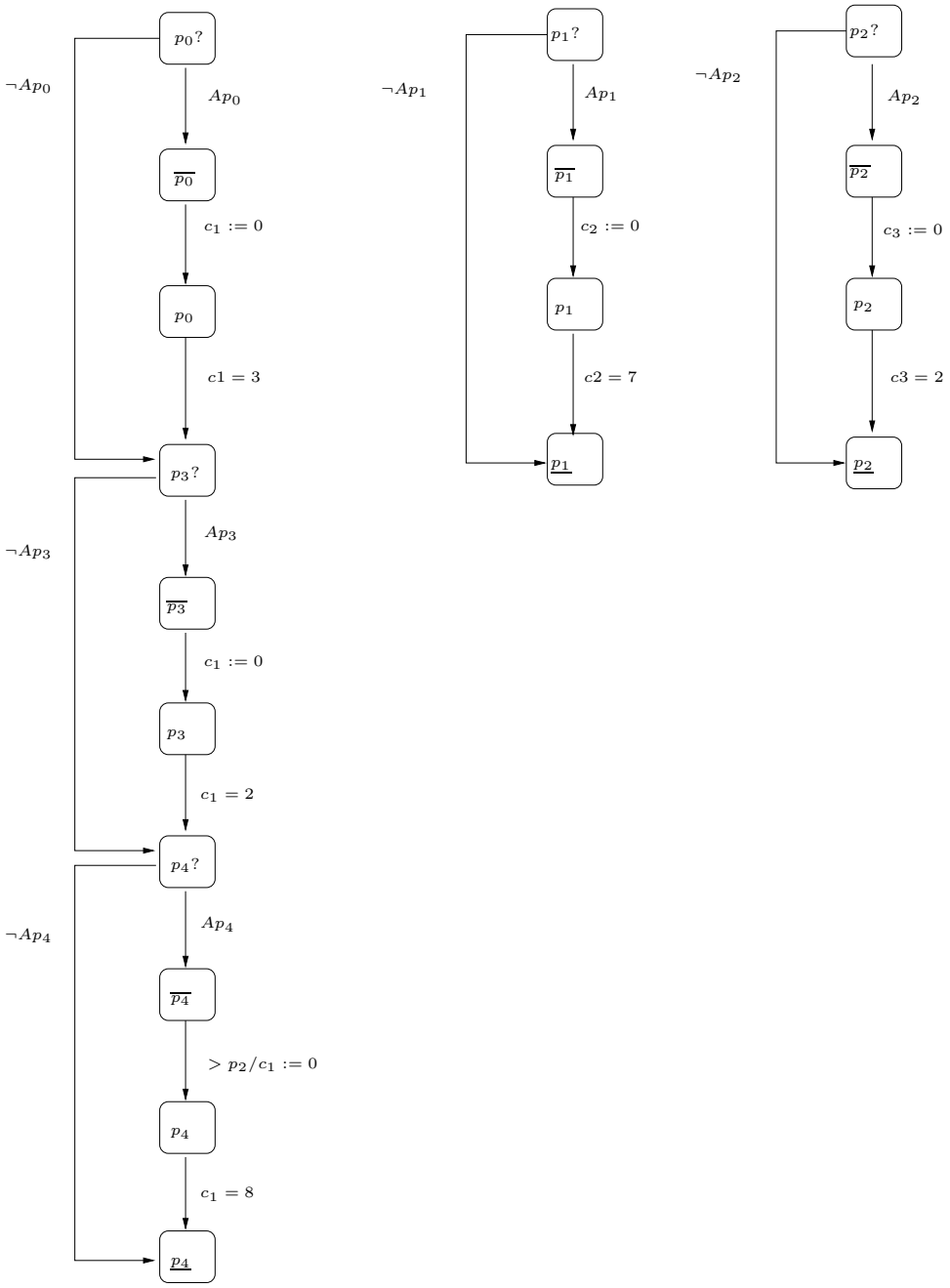


FIG. 7.9 – Timed automata for chains.

Chapitre 8

Shortest Strategies in Game Graphs : Optimal and heuristic Algorithms

In this chapter we will describe exact and approximate techniques for searching game graphs. We investigate the more promising ways to do this by forward search, generating the game graph on the fly. Of course, one can think about backward search procedures to find optimal paths, but they are not of much help because the game graph has to be constructed entirely. Forward procedures avoid this complete construction by pruning useless reachable states. One way to do this is to extend the two exhaustive Breadth First and Depth First algorithms of the previous part. The chapter is organized as follow : section 1 and 2 give formal presentation of the two algorithms's extensions with possible improvements. In section 3 we explain the estimation function and its properties, we will show by the way, the quality of the schedules generated by the heuristic. Finally, we give some experimental results.

All along this chapter, we will consider a game graph $G = (Q, \delta, w)$, representing the global model for the scheduling problem

8.1 Exact algorithms

The computation of the optimal strategy is a by product of computing the *value function* $h : Q \to \mathbb{R}_+$ where $h(q)$ is the value of the best strategy for the residual game $\mathcal{G}_q$. This function is defined recursively as :

$$h(q) = \begin{cases} 0 & \text{if } q \text{ is terminal} \\ \min_{q' \in \sigma(q)} w(q, q') + h(q') & \text{if } q \in Q_\vee \\ \max_{q' \in \sigma(q)} w(q, q') + h(q') & \text{if } q \in Q_\wedge \end{cases}$$

There are various ways to compute h, one of them is the *backward value iteration* procedure, also known as *dynamic programming*, which starts with the final states and propagates values according to the definition of h until $h(q_0)$ is defined, where q_0 is the initial node having no predecessor. This approach suppose the total construction of the game graph which is exponential in the size of the CPG. If we don't want to construct the whole game graph, we can use the forward version of dynamic programming generating the successors forward. This procedure is called Depth-first min-max, and is presented below.

8.1.1 Depth first min-max

The forward depth-first algorithm, which was used in [BKM04], is obtained by invoking $h(q_0)$ and following literally the recursive definition. If the game graph has a tree structure, this procedure has the same complexity as dynamic programming, however on non-tree graphs (i.e. dags), the algorithm can easily become exponential since the same node can be reached via many paths. Consequently we use an algorithm that combines depth-first search with memorization : whenever $h(q)$ is computed for the first time, the result is stored as $E(q)$ and subsequent invocations of $h(q)$ are answered using this value.

Algorithm 8.1 (Depth first min-max)

integer function $h(q)$
if q *is terminal* **return**(0)
elsif $E(q)$ *is defined* **return**$(E(q))$
elsif $q \in Q_\vee$ **then**
begin
  $E :=\infty$
  for *every* σ *such that* $\delta_\vee(q,\sigma)$ *is defined* **do**
    $E' := w(q,\sigma) + h(\delta_\vee(q,\sigma))$
    $E := \min\{E, E'\}$
  $E(q) := E$
  return*(E)*
end
elsif $q \in Q_\wedge$ **then**
begin
  $E :=h(\delta_\wedge(q,\neg b))$
  $E' :=h(\delta_\wedge(q,b))$
    $E := \max\{E, E'\}$
  $E(q) := E$
  return(E)
end

The derivation of a strategy from the value function is standard : we traverse the game graph from q_0, and for every $q \in Q_\vee$ $s(q) = \sigma$ from which the minimum is obtained via the transition σ.

Algorithm 8.1 is (time and space) linear in the size of the game graph but this is not of much help because the game graph by itself is exponential in the size of the CPG. The largest problems that we could solve with this algorithm had up to 20 tasks and 4 Booleans. We have preferred this algorithm over backward dynamic programming because it is more easily amenable to techniques that find optimal or nearly-optimal solutions without exploring the whole graph.

Improvements

Techniques for pruning the search space are based on two related ideas :

– Do not explore paths that can easily be shown not to lead to an improvement of the value function computed so far ; this is usually based on an auxiliary *estimation function* $\bar{\mu}(q)$ which approximates $h(q)$. If this function is *optimistic*, that is, $\bar{\mu}(q) \leq h(q)$ for every q, we need not explore δ-successors of an OR node q satisfying $E \leq w(q, q') + \bar{\mu}(q')$ where E is a value obtained via an already explored successor. We refer to the above condition as a *safe cutting test.*

– Replace the arbitrary order of exploration by a more "intelligent" policy, which explores the more promising successors first. Such a policy can be obtained again using the estimation or another heuristic rule (such as longest task first, most successors first, ...etc.).

The depth first min-max procedure can be seen as a variation of the well known *alpha-beta search* issued from the Artificial Intelligence community [Nil71, Zha99]. However, while the *alpha-beta* procedure uses the cost-to-come to a final state as value for nodes, the depth first min-max uses the cost-to-go from the node in question, which make it more adapted to our problem. The first is a generalization of the Branch and Bound procedures, while the other is a generalization of the forward dynamic programming. The domination test between nodes (see section 8.1.3) in the alpha beta search is reduced to a simple revisiting test in the depth first min max, which is lowest to compute, as well as the safe cutting tests are adapted according to the cost-to-come...etc.
The motivation of using the cost-to-go instead of the standard alpha beta search is that it facilitates the calculation of the cost of each strategy by backward propagation of this potential values during the search, especially in the heuristique search DFSBT (see section 8.2). All along the node of the tree of each strategy, instead of comparing the cost to come to the leafs and take the max between them, we just have to propagate backward the cost-to-go along each node.

8.1.2 Other method : Breadth first à la Dijkstra

The following algorithm is a variant of standard game graph search algorithms. It is an extension of the well known Dijkstra algorithm to game graphs [Mal04]. It explores all reachable partial strategies and finds the optimal worst-case strategy. As usual, it maintains a queue *Frontier*, containing reachable partial strategies that are the frontier of the search process, that is, strategies that have been reached but their successors have not.

Before describing this algorithm, let us begin with some definitions.

Valued node A valued node is a couple (q, T) where q is a state in Q. T is a non-negative number representing the length of the path leading to q.

Successors of a valued node the successors of a valued node are defined

$$Suc(q, T) = (q', T + w(q, q')) \text{ where } q \to q'$$

Macro node A macro node is a set of valued nodes

$$K = \{(q_1, T_1), \cdots, (q_k, T_k)\}.$$

Intuitively, a macro node represents a set of states reachable from the initial state by following a partial strategy under all continuations of AND nodes that have been revealed.

Successor of a macro node A successor of a macro node is obtained by picking an element (q, T) on it and replacing it either by any $Suc(q, T)$ if q is an OR node, or by the set of all its extended successors if it is an AND node. We use $Suc(M)$ to denote the set of successors of M, each being a macro node.

We are now ready to give the algorithm :

Algorithm 8.2 (BF frontier Shortest Path)

$Frontier := \{\{(q_0, 0)\}\}$
repeat forever
  Pick the first non-terminal macro node K *from* $Frontier$
  if K *contains only terminal nodes* **then**
    K *is optimal*
   stop
  else
   for *every* $K' \in Suc(K)$ **do**
     Insert K' *to* $Frontier$
   Remove K *from* $Frontier$

Each terminal macro node $\{(q_1, T_1), \cdots, (q_k, T_k)\}$ represents the outcome of following a particular strategy, where T_i is the length of the schedule obtained by that strategy on all instances of the adversary in the AND nodes. Choosing the macro node which minimizes $max\{T_i\}$ amounts to choosing the worst case optimal strategy. This way, we can sort the $Frontier$ list by increasing order of macro nodes's costs and stop the algorithm as soon as we reaches the first macro node containing only terminal nodes. This avoid us to explore the whole strategy space by forbidding macro nodes with higher cost than those computed so far.
Figure 8.2 shows the space of reachable partial strategies of the AND/OR graph of figure 8.1. The annotation on macro nodes represent the order of exploration. The optimal final macro node is $\{(q_7, 2), (q_{11}, 3)\}$.

This algorithm can be improved by taking into account a "*domination*" relation among macro nodes. Intuitively, a macro node dominates another if any node of the former dominates one of the latter.

8.1.3 Domination relations

Definition 8.1 (definition between nodes) *We say that* (q, T) *dominates* (q', T') *and we note* $(q, T) \leq (q', T')$ *iff*

– q' *is reachable from* q *and*
– $T \leq T'$

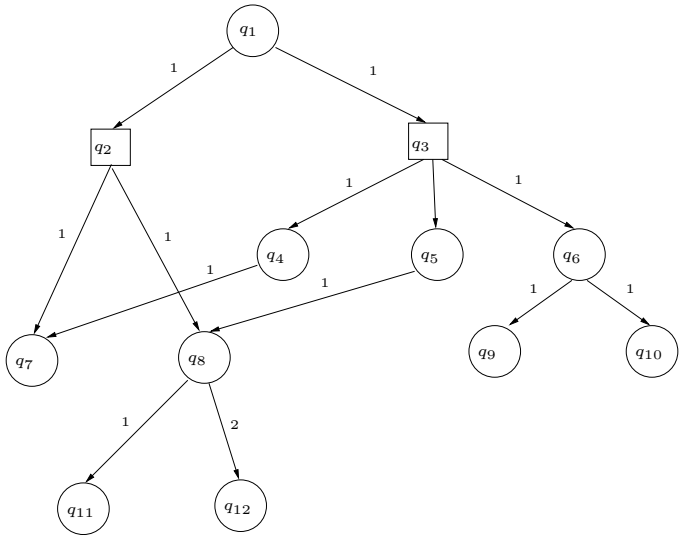


FIG. 8.1 – An AND/OR graph example

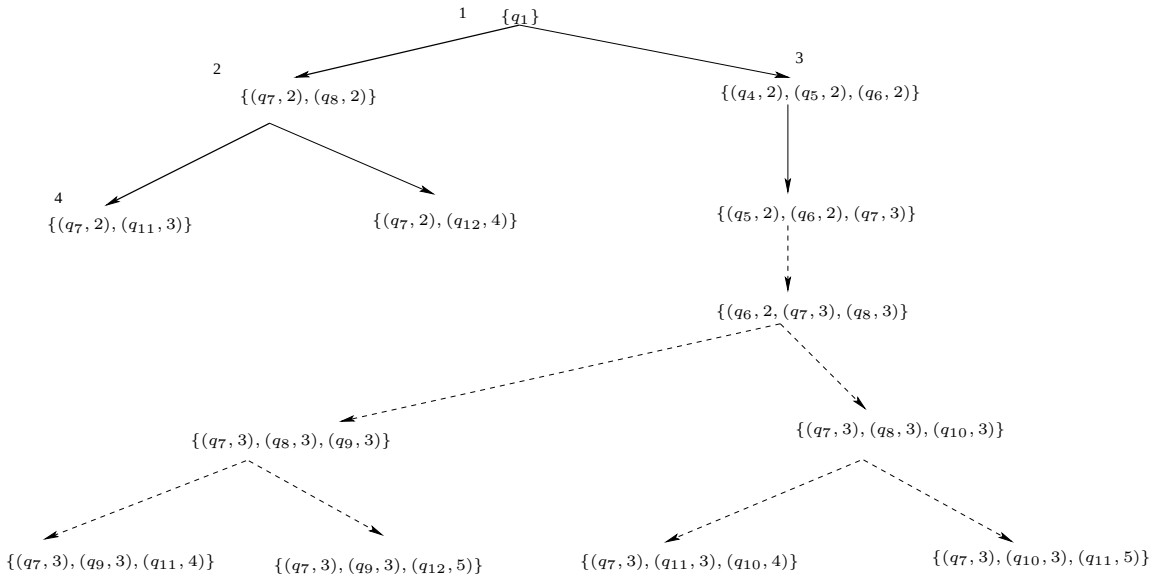


FIG. 8.2 – The space of partial strategies of the graph of figure 8.1

Definition 8.2 (domination between macro nodes) *Let* $K = \{(q_1, T_1), \cdots, (q_k, T_k)\}$ *and* $K' = \{(q'_1, T'_1), \cdots, (q'_k, T'_k)\}$ *be two macro nodes. We say that* K *dominates* K' *and we note* $K \leq K'$ *iff*

$$\forall (q_i, T_i) \in K,\ \exists (q'_j, T'_j) \in K' \text{ such that } (q_i, T_i) \leq (q'_j, T'_j)$$

Moreover, we say that K *and* K' *are uncomparable according to domination and we note* $K \bowtie K^{'}$ *iff*

$$\neg(K \leq K^{'}) \wedge \neg(K^{'} \leq K)$$

We use domination as follows : whenever K and K' are macro nodes in the $Frontier$ and $K \leq K'$, then there is no need to explore K'. This means that the $Frontier$ will be restricted to contain only uncomparable macro nodes.

This algorithm is quite inefficient due to its high time/space complexity. The storage requirements for each macro node grows exponentially in the number of boolean variables. Consequently, the domination relations become very heavy to compute. In addition, the list $Frontier$ is known to increase exponentially in the depth of the graph of macro nodes. All these facts make the algorithm very costly and hence very inefficient when implemented.

8.2 Heuristic : Depth First Search with selective Backtracking

In the deterministic problem, we have presented two heuristic algorithms. There, the time/space complexity are both linear in the size of the graph. It is not the case for the adversarial case. The generalization of the bounded width breadth first heuristic requires to select w macro nodes at each level of the search process, which may reach an exponential size for each macro node. The $DFSBT$ presented below, is supposed to store one strategy at time. This reduces considerably the memory consumption and decreases the time complexity severely compared to the

bounded width heuristic. In addition, while the bounded width heuristic fails to obtain solutions for large instances, the $DFSBT$ can be parameterized so that it is guaranteed to get at least one initial solution. This is why the generalization of $BFS - w$ is excluded from consideration and we have chosen to focus our presentation on the generalization of the $DFSBT$ to the adversarial problem.

Now let us introduce the notations used before giving the algorithm :

q_0 : the initial state (the root of the game graph),
q_{split} : the OR state where we want to update,
q_{choice} : the new choice that we want to make at q_{split}
$S(q)$: the choice made for q. $S(q) \in \delta_\vee(q)$ if q is an OR node
$P(q)$: the parent node for q, if q is in the strategy
$H(q)$: the cost of the strategy starting at q
$T(q)$: the time to reach q in the current strategy

The Algorithm 8.3 is the main routine for $DFSBT$. It searches for an initial strategy by invoking the procedure *computeStrategy* from the initial state q_0 of the game graph. Then, iteratively it tries to improve the current strategy . For doing this, it searches a candidate node for improvement (q_{split}) and computes an alternative strategy for it, following q_{choice}. If this new strategy is better than the old one, then the global strategy is updated. This process continues until some stop condition is met, and it returns the best solution found so far.

Algorithm 8.3 (Dfs with selective backtracking)

Algorithm *DFSBT*
begin
computeStrategy$(q_0, \perp, 0)$
while *(¬stop condition)* **do**
$q_{split} := \perp$,
$q_{choice} := \perp$
$findBacktrackNode(q_0)$
$computeStrategy(q_{choice}, q_{split}, T(q_{split}) + w(q_{split}, q_{choice}))$
if $(H(q_{choice} + w(q_{split}, q_{choice})) < H(q_{split}))$
$S(q_{split}) := q_{choice}$
$P(q_{choice}) := q_{split}$
$propagateUpward(q_{split})$
endif
end while
end

Several stop conditions have been implemented :

- no more improvement : the algorithm stops when all other backtracking nodes have greater estimation than the cost of the best strategy found.
- time bound : the algorithm stops when a time limit is released.
- memory bound : the algorithm stops when no more memory is available.

The procedure *computeStrategy* goes down recursively from a node q, and constructs a complete strategy by choosing always the immediate 'best' successor for OR nodes. During the search process, the cost-to-come $T(q)$ for each explored node are computed forward and the costs-to-go $H(q)$ are propagated backward.

Procedure 8.1 (Computes a strategy beginning from a node)

Algorithm *computeStrategy*(q, p, t)

q : *the current state*
p : *the parent state of* q
t : *the cost to come at* q
begin
$P(q) := p$
$T(q) := t$
if q *is terminal*
  $H(q) := 0$
endif
if $q \in Q_\vee$
  if $S(q) \neq \perp$ *return*
  else
    begin
    $q' := bestOf(\delta_\vee(q))$
    $computeStrategy(q', q, t + w(q, q'))$
    $H(q) := H(q') + w(q, q')$
    $S(q) := q'$
    end
endif
if $q \in Q_\wedge$
  foreach $q' \in \delta_\wedge(q)$
    $computeStrategy(q', q, t + 0)$
  $H(q) := max\{H(q')/q' \in \delta_\wedge(q)\}$
endif
end

Note that each time a node q is encountered, the condition $S(q) \neq \perp$ is checked. This prevents from visiting an already explored node. Revisiting is not necessary because the selection rule is the same,and leads necessarily to the same choices as the ones already defined from q and its successors.

The backtracking rule consists on choosing the most promising node likely to improve the current strategy among all the pending successors of actual nodes in

that strategy. The cost of a strategy from a node q is equal to the cost of the longest path from q in this strategy. Consequently, if we want to improve a given strategy, we need only to improve the costs of nodes contained in the longest path in the current strategy. This is why, each time an AND node q is encountered, the procedure $FindBacktrackNode$ follows with the successor which has the maximal cost equal to $H(q)$ [1].

Procedure 8.2 (Selection of a backtracking edge)

Algorithm *FindBacktrackNode(q))*
begin
if q *is terminal*
  return
if$q \in Q_\wedge$ **then**
  select $q' \in \delta_\wedge(q)$ *s.t.* $H(q) = H(q')$
  $FindBacktrackNode(q')$
endif
if $q \in Q_\vee$ **then**
  select most promising $q' \in \delta_\vee(q) \backslash \{S(q)\}$
  if $q_{choice} = \perp \vee q'$ *is most promising than* q_{choice}
    $q_{split} := q$
    $q_{choice} := q'$
  endif
  $FindBacktrackingNode(S(q))$
endif
end

The procedure $PropagateUpward$ updates the values in the strategy according to the new choice made at the backtracking. It stops as soon as it reaches the same value.

Procedure 8.3 (Backward propagation)

[1] In this work, it is supposed that the AND edges have zero costs

$PropagateUpward(q)$
begin
if $q \in Q_\vee$
  $h := H(S(q)) + w(q, S(q))$
if $q \in Q_\wedge$
  $h := max\{H(q')/q' \in \delta_\wedge(q)\}$
if $h < H(q)$
  $H(q) := h$
  if $P(q) \neq \perp$
    $PropagateUpward(P(q))$
endif
end

8.3 Estimation Functions for Conditional Scheduling

For unconditional precedence graphs, an estimation function that gives a lower bound on the time remaining until termination from a state, can be constructed by first associating with each task p the length $\mu(p)$ of the longest path in the task graph from p to some terminal task. Then, an estimation $\bar{\mu}$ of the value of the global state, is the maximum of μ over all tasks which are waiting or active in this state.

In the conditional setting this is more involved due to precedence constraints between tasks that have *different activation conditions*. Consequently the distance from a task to termination is not a single number but is *instance dependent*. Let $\mu : P \times \{0,1\}^n \rightarrow \mathbb{R}_+$ be a partial function defined over all v such that $A_p(v)$ is true. When $A_p(v)$ is false we use the notation $\mu(p, v) = \perp$. The intended meaning of $\mu(p, v)$ is the total amount of work that needs to be done for instance v before task p has started. Since computation of longest pathsis done (explicitly or implicitly) within the $(\max, +)$-algebra we need to extend these two operations to $\mathbb{R}_+ \cup \{\perp\}$ by letting $r + \perp = \perp$ and $\max\{r, \perp\} = r$ for every $r \in \mathbb{R}_+$.

A simple way to understand this function (although not the most efficient

way to compute it) is the following : for each instance v let G_v be the sub-graph consisting of the tasks whose activation conditions are satisfied by v. If p does not belong to G_v then $\mu(p, v) = \perp$, otherwise let $\mu(p, v)$ be the longest path in G_v from p to termination. This function can be computed backwards on the whole graph, starting from terminal nodes :

$$\mu(p, v) = \begin{cases} \perp & \text{if } A_p(v) = false \\ d(p) & \text{otherwise} \end{cases}$$

and computing for other nodes as

$$\mu(p, v) = d(p) + \max_{p' : p \prec p'} \mu(p', v).$$

This computation can be done symbolically (and off-line) using the syntax of A without necessarily enumerating all instances. From μ we can define an estimation function $\bar{\mu}$ over the states and clock values of the $\mathcal{A}_p$ automaton by letting $\bar{\mu}(p?, c, v) = \bar{\mu}(\overline{p}, c, v) = \mu(p, v)$, $\bar{\mu}(p, c, v) = \mu(p, v) - c$ and $\bar{\mu}(\underline{p}, c, v) = \mu(p, v) - d(p)$.

The function $\bar{\mu}$ is *optimistic* because it takes into account precedence constraints but ignores resource constraints. In other words it assumes sufficiently many machines so that every task can be executed once all of its predecessors have terminated. A complementary way to obtain lower-bounds on schedule length is to ignore precedence constraints and take into account resource constraints, that is, dividing the total amount of work by the number of machines. The estimation

$$\nu_k = \max_v \sum_{p \in P_v} d(p)/m$$

is equally optimistic as it ignores the possibility that a machine can be idle at certain times because no task is enabled. Like μ, estimation ν can be defined for global states of the game automaton by restricting summation to tasks that have not terminated.

Example 8.1 *Figure 8.4 shows the different instances of the CPG of figure 8.3, and the Table 8.1 gives the longest paths under all possible instances of booleans. Now suppose we have the global state* $q = (\underline{p_1}, p_2?)$ *in the global automaton of the CPG of Figure 8.3.* p_1 *has terminated but not* p_2 *and* p_3*. At this state the values of*

b_1 and b_2 are not known yet. Table 8.1 contains the longest path from a task under each instance of boolean variables assuming unit execution times. The estimation for p_2 corresponds to the maximum value of its longest paths among all instances ; namely

$$\bar{\mu}(p, v) = max\{4, 5, 4, 4\}$$

.

When dealing with more than 8 variables, this calculation becomes impractical when using tables like Table 8.1. One way to improve this calculations is to use the *Binary Decision Diagrams (BDDs)*. The estimations for each task are computed off-line from each instance graph as in figure 8.4 and stored as BDDs. Figure 8.5 (a) shows the Shannon decision tree for p_2, and figure 8.5 (b) gives the corresponding Arithmetic Decision Diagram for the task p_2. Note that the decision tree is always exponential in the number of variables ; however, the arithmetic decision diagram can be much smaller due to the simplification and the sharing of isomorphic paths.

Using these estimations in the DFSBT, we can always obtain a greedy initial strategy, which means that the worst case performance of the heuristic is better than 50% of the optimum.

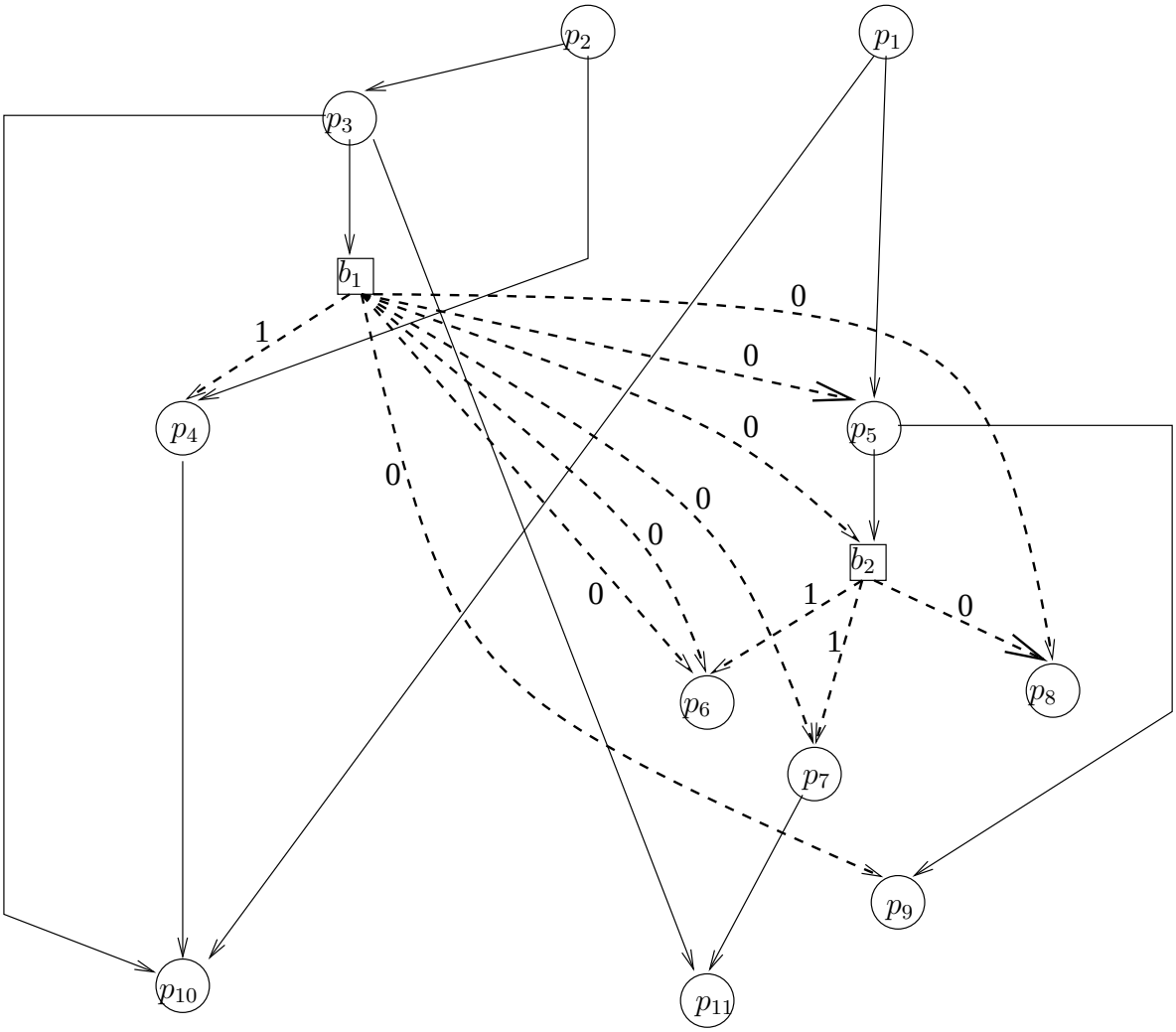


FIG. 8.3 – A CPG with input and output dependencies

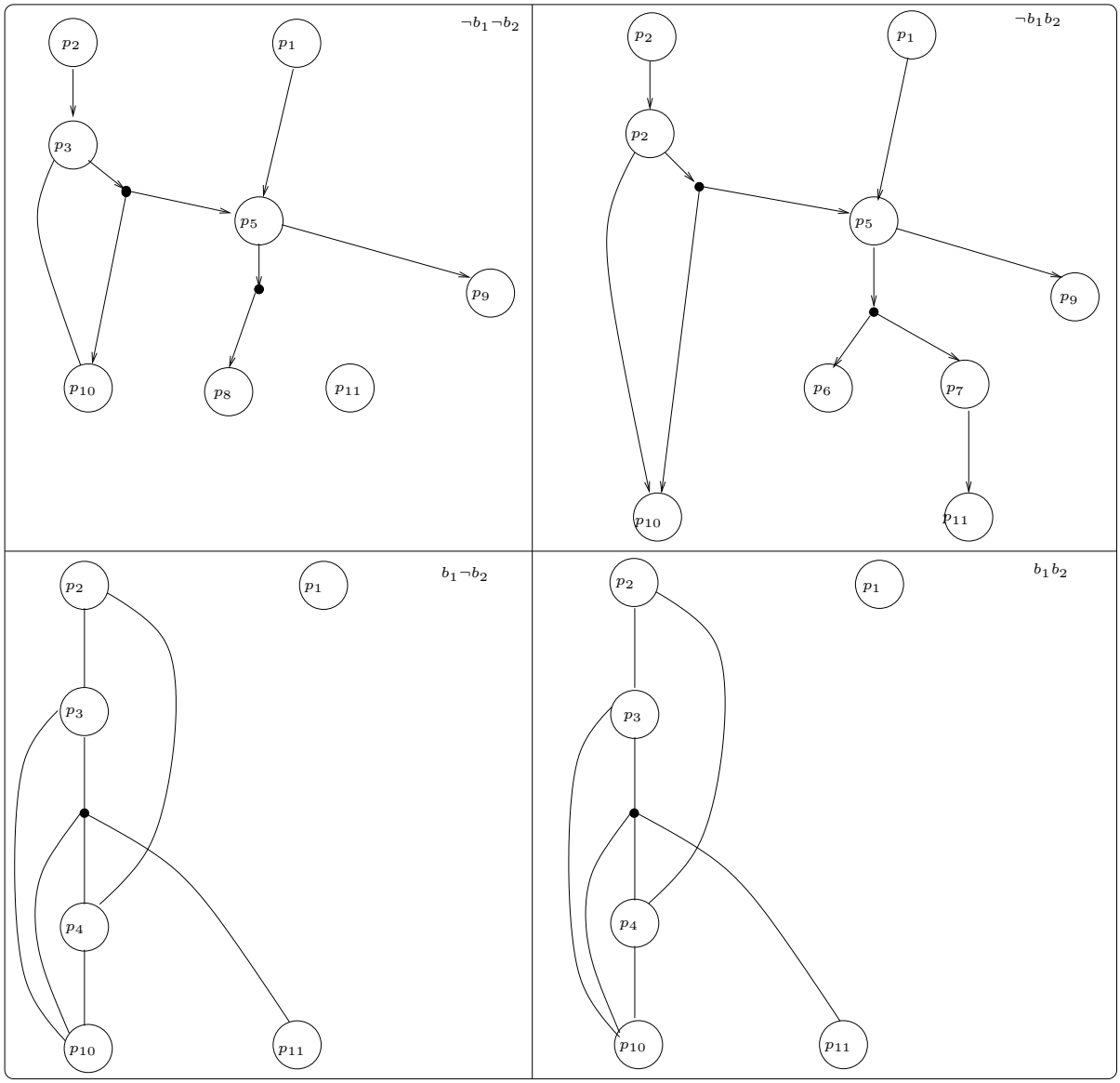


FIG. 8.4 – All instances of the CPG

b_1	b_2	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}
0	0	3	4	3	$\perp$	$\perp$	$\perp$	$\perp$	1	1	1	1
0	1	4	5	4	$\perp$	3	1	2	$\perp$	1	1	1
1	0	1	4	3	2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1	1
1	1	1	4	3	2	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	1	1

TAB. 8.1 –

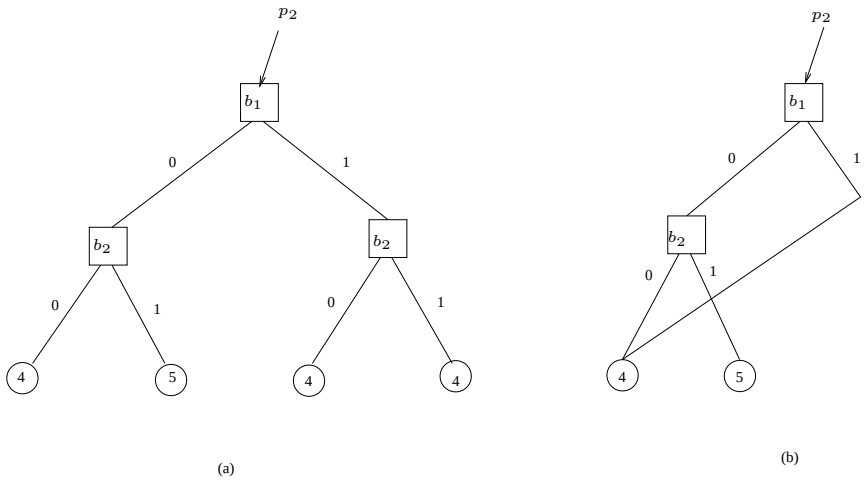


FIG. 8.5 – The longest path estimation of process p_2. (a) The Shannon decision tree and (b) the corresponding Arithmetic Decision Diagram

8.4 Experimental Results

We have implemented a prototype tool that generates random CPGs, translates them into timed automata with chain decomposition, and generates only strict non-lazy runs. It implements all algorithms described inthis thesis. All the results were obtained on an architecture Pentium 1.4 Ghz, 2 Go of memory. Table 8.2 shows results for different CPGs on 3 machines. Applying an exact depth first min-max with a best first search, we could solve problems up to 20 tasks with 4 booleans. Beyond that, we have to use the sub-optimal heuristic.

	2			3			4		
	opt	*time*	*nb st*	*opt*	*time*	*nb st*	*opt*	*time*	*nb st*
8	38	$0''$	79	30	$0''$	218	36	$0''$	758
10	31	$0''$	1388	36	$0''$	152	28	$0''$	281
12	38	$0''$	169	29	$0''$	2668	47	$1''$	1870
14	53	$0''$	1455	59	$2''$	12060	42	$2'10''$	191042
20	71	$8'7''$	2386599	61	$10'45''$	2610496	61	$0''$	1278

TAB. 8.2 – Results for exact dfs on 3 machines. The number of tasks varies from 8 to 20 and the number of booleans is from 2 to 4. *opt* represents the value of the optimum, *time* and *nb st* stands respectively for the time and the number of generated global states.

We have tested our heuristic on large size CPGs Setting the time bound to 1 minute, we could solve problems with up to 200 tasks and 7 booleans as shown in table 8.3. The results of the largest problems we have treated with the time bound was set to 5 minutes are reported in table 8.4. As one can check, the average worst-case deviation from the lower bound is about 1.73%.

		3			4			5		
	α	*best*	*LB*	*imp*	*best*	*LB*	*imp*	*best*	*LB*	*imp*
100,6	0	263	260	339	203	195	303	166	156	290
100,6	0.66	263	260	339	203	195	303	166	156	291
100,7	0	321	316	6000	245	190	904	194	190	206
100,7	0.66	321	316	287000	247	190	266000	200	190	351000
120,7	0	360	354	438	278	266	393	218	213	284
120,7	0.66	360	354	648	279	266	227	218	213	175
100,10	0	474	467	45	383	350	193	347	333	262
100,10	0.66	475	467	2455	383	350	12	349	333	13
150,7	0	428	413	36	325	310	33	255	248	26
150,7	0.66	428	413	43	325	310	12	255	248	26
200,7	0	559	543	8	429	407	6	345	326	7
200,7	0.66	559	543	20000	429	407	19000	345	326	18000

TAB. 8.3 – Results for large CPGs on different machines (column 3, 4 and 5). *best* stands for the best result, *LB* is a lower bound, and *imp* is the number of backtracking points. The time bound execution is 1 minute

	3			4			5		
	best	*LB*	*imp*	*best*	*LB*	*imp*	*best*	*LB*	*imp*
300,10	1391	1371	32	1067	1029	186	898	823	14301
300,12	1410	1385	7	1070	1039	1011	902	831	2822
400,10	1809	1790	13410	1373	1342	2124	1110	1074	6277
500,8	2320	2308	424	1783	1731	8142	1558	1391	6939

TAB. 8.4 – Heuristic results for larger CPGs with 5 minutes time bound.

Chapitre 9

Conclusion

All along this thesis we have considered two problems of scheduling dependent tasks under resource constraints. The first problem is deterministic, whereas the second involves conditional uncertainty. The deterministic task graph scheduling can be solved with existing techniques, most of them based on list scheduling. In this work we have used the timed automata technology in order to evaluate its performance on such problems.

For the deterministic problem, we have first considered the naive model which construct a product of automata, one for each task. By exploiting the structural properties of the partial order in the task graph, using the Dilworth theorem, we have reduced the size of the product by decomposing the task graph into chain cover and building an automaton for each chain in the task graph. This model is very efficient because most of real applications have usually small width compared to the number of tasks.

The non laziness phenomenon was redefined, better formalized and adapted to our problem. Hence, we have proposed a new model based on timed automata with additional features that avoids useless waitings. In this model, the useless waiting is detected during the processing time of the task.

On the algorithmic side, we have investigated several search techniques. Previously, the search defined in [AKM03] was more breadth like. In this work, we attempt to render the search more depth like, and we have proposed the selective backtracking heuristic DFSBT. Moreover, we have shown that using a standard estimation function in a depth first search, the procedure is naturally guided toward

non-delay schedules. This means that, theoretically, the quality of the obtained solution is always better than a standard list scheduling, for which a good worst case performance is guaranteed. In addition, the DFSBT can be parameterized so that an idle time can alway be inserted in the schedule during the backtracking process. Furthermore, a non-lazy strategy that favors waiting is always possible. However, while such a heuristic is known to fail to obtain good schedules in a reasonable amount of time, the greedy guided search can always obtain schedules close to the optimum at a very early stage of the search process.

The generalization to the conditional problem was not straightforward. First we had to define the conditional precedence graphs which capture the whole situation with a well defined semantics. Secondly, we had to verify that all the results for the deterministic case, including the fact the non-lazy schedules achieve the optimum, and that the approximation ratio for Greedy schedules is at most 2, can be transferred to the conditional case. The timed automaton models had to be extended to accomedate for conditional dependencies and the chain decomposition procedure had to be safely adapted for a mixture of ordinary and Boolean tasks.

We have investigated and implemented several forward search procedures for searching the game graph representing the global automaton in order to select the best heuristic. Two exact methods were proposed. The first is based on a generalization of the Bellman principle to game graphs, and the second extends the Dijkstra algorithm. As it turned out that the min-max depth first procedure [BKM04] has a lower complexity than the breadth first one [Mal04], which was not obvious a-priori, we have chosen to generalize the DFSBT heuristic to get sub optimal strategies. This way the DFSBT was parameterized and well tuned, and revealed to be more powerful than the bounded width heuristic due to its low memory requirements.

To guide the search, we have proposed two simple lower bounds that estimate the remaining time from a global state. Instead of using heavy tables to store estimation values, which are always exponential in the number of boolean variables, the storage and computing time requirements were reduced more significantly using a BDD representation. Here again, the search is guided naturally toward non-delay strategies, for which the maximum deviation is theoretically within half to the optimum.

In the next steps of the work, we envisage two major directions. The first aims

to extend the models to treat more complex scheduling with additional constraints, and the other concerns a better understanding of the state space exploration and the problem itself.

To make the model more realistic, we can :

1. Add more complex timing constraints such as relative deadlines or synchronization constraints between tasks. Such constraints are in principle easily included in the TA model, however, important properties concerning non-laziness are lost.
2. Extend the framework to speculative execution.
3. Add new sources of uncertainty such as the combination of the discrete uncertainty treated in this thesis with preemption and temporal uncertainty concerning the duration of atomic tasks.
4. Consider the problem with heterogeneous processors and/or cyclic tasks/activities.
5. Develop a front end for extracting automatically the CPGs via data flow analysis from programs.

Further directions suggest :

1. To understand how to take benefit from the advantage of the DFSBT heuristic in order to include useful waitings in an already explored schedule. Suppose we go down a path with a greedy strategy, then we may discover something about bottlenecks in the current schedule. For example it may be the case that a blocking situation occurs because we have chosen to start the wrong tasks instead of waiting another future enabled one. We can then analyze the reasons of this blocking and backtrack in order to include the useful waiting. Is there a way to get benefit from the already accumulated knowledge ?
2. To establish the complexity of the conditional problem. The problem is NP hard because the deterministic problem is reduced to it ; but is not complete for NP. On the other hand, we know that it is in NEXPTIME since we can check if a given strategy is feasible is in $O(2^n)$, where n is the number of boolean variables. Is it another problem known to be NEXPTIME complete from which it can be reduced ?
3. Construct a parallelized version of the DFSBT.

Conclusion

Tout au long de cette thèse nous avons considéré deux problèmes d'ordonnancement de tâches dépendantes sous contraintes de ressources. Le premier problème est déterministe, alors que le deuxième présente de l'incertitude conditionnelle. L'ordonnancement de graphe de tâches déterministe peut être résolu avec des techniques existantes, la plupart sont basées sur des ordonnancements de listes. Dans ce travail nous avons utilisé la technologie des automates temporisés afin d'évaluer leur performance sur de tels problèmes.

Dans le cas déterministe, nous avons d'abord considéré le modèle naïf qui construit un produit d'automates, un pour chaque tâche. En exploitant les propriétés structurelles de l'ordre partiel dans le graphe de tâches, en utilisant le théorème de Dilworth, nous avons réduit la taille du produit en décomposant le graphe de tâches en chaînes couvrantes et en construisant un automate pour chaque chaîne dans le graphe de tâches. Ce modèle est très efficace car la plupart des applications réelles ont habituellement une petite largeur de graphe en comparaison avec le nombre de tâches.

Le phénomène de la non-paresse a été redéfini, mieux formalisé puis adapté à notre problème. Ainsi, nous avons proposé un nouveau modèle basé sur les automates temporisés avec de nouvelles caractéristiques qui évitent les attentes inutiles. Dans ce modèle, les attentes inutiles sont détectées durant le temps d'exécution de chaque tâche.

D'un point de vue algorithmique, nous avons investigué plusieurs techniques de recherche. Auparavant, la recherche définie dans [AKM03] était plus en largeur. Dans ce travail, nous tentons de rendre la recherche plus en profondeur, ainsi nous

avons proposé l'heuristique profondeur d'abord avec backtrack sélectif (DFSBT). De plus, nous avons montré qu'en utilisant une fonction d'estimation standard dans une recherche d'abord en profondeur, la procédure est guidée naturellement vers des ordonnancement sans attente. Ceci veut dire que, théoriquement, la qualité des solutions obtenues est toujours meilleurs que celle des ordonnancements de listes, pour lesquels une bonne performance pire cas est garantie. D'autre part, l'heuristique DFSBT peut être paramétrée de sorte que des attentes peuvent toujours être insérées dans un ordonnancement donné durant la phase de backtrack. Par conséquent, une stratégie qui préfère attendre est toujours possible. Toutefois, alors qu'il est largement connu que de telles heuristiques sont incapables d'obtenir de bonnes solutions en un temps raisonnable, le DFSBT obtient toujours des solutions très proches de l'optimum au tout début du processus de recherche.

La généralisation au problème conditionnel n'était pas directe. D'abord, nous avons eu à définir le graphe de précédences conditionnel qui capture la totalité de la situation avec une sémantique opérationnelle bien définie. Ensuite, nous avons eu à vérifier que tous les résultats concernant le cas déterministe, y compris le fait que les ordonnancements non-paresseux atteignaient l'optimum, ainsi que la garantie de performance des ordonnancements sans attente est au plus 2, peuvent être généralisés au cas conditionnel. Les modèles d'automates temporisés ont dû être étendus pour être accommodés aux dépendances conditionnelles et la procédure de décomposition en chaînes a dû être adaptée de manière sûre pour une mixture de tâches ordinaires et booléennes.

Nous avons investigué et implémenté plusieurs procédures de recherche en avant pour parcourir un graphe de jeu représentant l'automate global afin de sélectionner la meilleure heuristique. Deux méthodes exactes ont été proposées. La première est basée sur une généralisation du principe de Bellman aux graphes de jeux, et la seconde généralise l'algorithme de Dijkstra. Comme il s'est avéré que la procédure min-max en profondeur d'abord [BKM04] a une complexité inférieure à celle qui va en largeur d'abord [Mal04], ce qui n'était pas évident a-priori, nous avons choisi de généraliser l'heuristique DFSBT pour avoir des stratégies sous optimales. De cette façon, DFSBT a été paramétrée et bien ajustée, et s'est révélée plus avantageuse que l'heuristique à largeur bornée dû à sa faible complexité en mémoire.

Pour guider la recherche, nous avons proposé deux bornes inférieures qui estiment le temps restant à partir d'un état global. Au lieu d'utiliser des tableaux pour stocker les valeurs d'estimations, dont la taille et le parcours sont toujours exponentiels par rapport au nombre de variables booléennes, les besoins en mémoire et en temps ont été réduits d'une façon très significative en utilisant la représentation en BDD (de *Binary Decision Diagrams*). Encore une fois, nous avons pu montrer que la recherche est dirigée naturellement vers des stratégies initiales qui sont sans attente, et pour lesquelles la déviation maximum est théoriquement à moitié de l'optimum.

Bibliographie

[AAM06] Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2) :272–300, 2006.

[Abd02] Y. Abdeddaïm. *Scheduling with Timed Automata.* PhD thesis, INPG Grenoble, November 2002.

[ACD74] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communication of the ACM*, 17(12) :685–690, 1974.

[ACMR03] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller-synthesis techniques to build property-enforcing layers. In *ESOP*, pages 174–188, 2003.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[AGS02] Karine Altisen, Gregor Gößler, and Joseph Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2) :55–84, 2002.

[AKM03] Y. Abdeddaïm, A. Kerbaa, and O. Maler. Task graph scheduling using timed automata. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 237. IEEE Computer Society, 2003.

[AM02] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble,*

France, April 8-12, 2002, Proceedings, volume 2280 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2002.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *In : Proc. IFAC Symposium on System Structure and Control.*, pages 469–474. Elsevier, 1998.

[Bak74] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, 1974.

[Bel57] R.E. Bellman. *Dynamic Programming.* Princeton University press, Princeton, NJ, 1957.

[BEP+96] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling computer and manufacturing processes.* Springer-Verlag New York, Inc., New York, USA, 1996.

[BF01] G. Behrmann and A. Fehnker. Efficient guiding towards cost-optimality in uppaal. In *TACAS 2001 : Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–188, London, UK, 2001. Springer-Verlag.

[BFG+99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF : An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 (Toulouse, France)*, volume 1708 of *LNCS*, pages 307–327. Springer, September 1999.

[BGM02] M. Bozga, S. Graf, and L. Mounier. If-2.0 : A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer, July 2002.

[BKM04] M. Bozga, A. Kerbaa, and O. Maler. Scheduling acyclic branching programs on parallel machines. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004), 5-8 December 2004, Lisbon, Portugal*, pages 208–217. IEEE Computer Society, 2004.

[Bru97] P. Brucker. *Scheduling Algorithms.* Springer, Berlin, 1997.

[CG72] E. G. Coffman and R. L. Graham. Optimal scheduling for two processors systems. *Acta Inf*, 13 :200–213, 1972.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, McGraw-Hill, september 2001.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–165, 1950.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 2000.

[FB73] E. B. Fernández and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedule. *IEEE Transactions on Computers*, C-22(8) :745–751, august 1973.

[Fis73] M. L. Fisher. Optimal solution of scheduling problems using lagrange multipliers : Part i. *Operations Research*, 21(5) :1114–1127, 1973.

[FRS03] S. Felsner, V. Raghavan, and J. Spinrad. Recognition algorithms for orders of small width and graphs of small dilworth number. *Order*, 20(4) :351–364, 2003.

[GJ79] M. R. Garey and D. S. Johnson. *Computer and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[Gra66] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45 :1563–1581, Nov. 1966.

[HC94] Y. Hu and B. S. Carlson. Improved lower bounds for the scheduling optimization problem. In *In Proceedings of the 1994 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 295–298, London, England, June 1994.

[Jr.76] E.G. Coffman Jr. *Scheduling in Computer and Job Shop Systems.* Wiley, New York, 1976.

[JR94] K.K. Jain and V. Rajaraman. Lower and upper bounds on time for multiprocessor optimal schedules. *IEEE Transactions on Parallel and Distributed Systems*, 05(8) :879–886, 1994.

[Ker02] A. Kerbaa. Ordonnancements sur graphe de tâches à l'aide d'automates temporisés. *DEA ISC, UJF-INPG Grenoble*, June 2002.

[KI99a] Y. K. Kwok and A. Ishfaq. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3) :381–422, 1999.

[KI99b] Y. K. Kwok and A. Ishfaq. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4) :406–471, 1999.

[KW00] A. A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs as a unifying design representation in the codesis high-level synthesis system. In *ISSS*, pages 66–72, 2000.

[KW02] A. A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Transactions on Design Automation Electrical Systems.*, 7(3) :380–412, 2002.

[KY03] Ch. Kloukinas and S. Yovine. Synthesis of safe, qos extendible, application specific schedulers for heterogeneous real-time systems. *ecrts*, 00 :287, 2003.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journalm of Software Tools for Technological Transfer*, 1(1-2) :134–152, Dec 1997.

[Mal04] O. Maler. On optimal and sub-optimal control in the presence of adversaries. In *Workshop on Discrete Event Systems (WODES)*, pages 1–12. IFAC, 2004. Invited talk.

[Man67] G. K. Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM*, 14(3) :439–465, July 1967.

[MR02] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. *ecrts*, 00 :241, 2002.

[Nil71] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill, june 1971.

[NY00] P. Niebert and S. Yovine. Computing optimal operation schemes for chemical plants in multi-batch mode. In *Hybrid Systems, Computation and Control*, volume 1790 of *LNCS*. Springer Verlag, March 2000.

[Pin95] M. Pinedo. *Scheduling : Theory, Algorithms and Systems*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[RCG72] C. Ramamoorthy, K. Chandy, and M. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. on Computers*, C-21 :137–146, 1972.

[TK02] T. Tobita and H. Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5) :379–394, 2002.

[TKK00] T. Tobita, M. Kouda, and H. Kasahara. Performance evaluation of minimum execution time multiprocessor scheduling algorithms using standard task graph set. In *PDPTA*, 2000.

[TP78] F. B. Talbot and J. H. Patterson. An efficient integer programming algorithm with network cut for solving resource-constrained scheduling problems. *Management Science.*, 24(11) :1163–1174, 1978.

[Weg99] J. Weglarz. *Project Scheduling - Recent Models, Algorithms and Applications*. Kluwer Academic Publishers, Dordrecht, 1999.

[Yov97] S. Yovine. Kronos : A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1-2) :123–133, 1997.

[Zha99] W. Zhang. *State-Space Search : Algorithms, Complexity, Extensions, and Applications*. Springer, December 1999.

Résumé

Cette thèse développe une méthodologie pour résoudre les problèmes d'ordonnancement de programmes conditionnels où savoir si une tâche doit être exécutée n'est pas connue à l'avance mais dynamiquement. Le modèle utilisé est à base d'automates temporisés représentant l'espace d'états à explorer. Le problème est donc formulé comme le calcul d'une stratégie gagnante (pire cas optimale) dans un jeu contre l'environnement. Dans un premier temps nous étudions le problème d'ordonnancement sur graphes de tâches déterministe puis nous étendons l'étude au problème d'ordonnancement avec incertitude conditionnelle. Pour les deux problèmes nous étudions différentes classes d'ordonnancements et de stratégies pour réduire l'espace d'états, des décompositions en chaînes pour réduire sa taille, puis nous investiguons plusieurs classes d'algorithmes exactes pour en évaluer l'efficacité et à partir desquels nous dérivons de bonnes heuristiques. Des résultats expérimentaux sur plusieurs exemples de benchmarks sont présentés afin de montrer l'efficacité de chaque algorithme et la précision des heuristiques proposées, puis des bornes théoriques sont déduites pour prouver la garantie de performance pire cas de chaque heuristique.

Abstract

In this thesis we develop a methodology for solving conditional scheduling problems where knowing if a task have to be executed is not known in advance but dynamically. The model used is based on timed automata representing the state space to be explored. The problem is formulated as a game against the environment from which we search for a winning strategy (worst case optimal). In the first part we study the deterministic problem of the task graph scheduling and then we extend the framework to the conditional problem. For each problem we study different types of schedules and strategies in order to reduce the state space search, decompositions into chains are proposed to reduce its size, then we investigate several exact algorithms in order to evaluate their efficiency and from which we derive some good heuristics. Experimental results on sets of benchmarks are presented to evaluate the efficiency of each algorithm and the precision of the proposed heuristics, then we deduce theoretical bounds to show the worst case guarantee of each heuristic.