Combiner la programmation par contraintes et l'apprentissage machine pour construire un modèle éco-énergétique pour petits et moyens data centers

Gilles Madi Wamba

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Gilles MADI WAMBA

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire
sous le sceau de l'Université Bretagne Loire

École doctorale : Mathématiques et STIC

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Soutenue le 27 Octobre 2017
Thèse n° : 2017IMTA0045

Combiner la programmation par contraintes et l'apprentissage machine pour construire un modèle éco-énergétique pour petits et moyens data centers.

JURY

Rapporteurs :	**M^me^ Sara BOUCHENAK**, Professeur, INSA Lyon
	M. Arnaud LALLOUET, Professeur, Huawei Technologies France
Examinateurs :	**M. Jean-Marc PIERSON**, Professeur, IRIT Toulouse
	M. Laurent PERRON, Ingénieur de recherche, Google
Directeur de thèse :	**M. Nicolas BELDICEANU**, Professeur, Ecole des Mines - IMT

Remerciements

Je ne saurais commencer la rédaction de ce manuscrit sans *rendre à César ce qui appartient à César*. Cela se traduit par ma profonde gratitude envers ceux qui ont contribué de toute façon que ce soit à l'aboutissement de cette thèse.

En premier lieu, je tiens à remercier mon directeur de thèse Nicolas BELDICEANU, pour la confiance et l'autonomie qu'il m'a accordé dans la réalisation de cette recherche. Sa compétence et sa rigueur scientifique m'ont beaucoup appris. J'ai également beaucoup apprécié sa grande disponibilité tout au long de cette thèse, et cela malgré ses nombreuses charges. Je remercie ensuite Jean-Marc MENAUD et Anne-Cecile ORGERIE pour leur disponibilité.

Merci à Sara BOUCHENAK et Arnaud LALLOUET d'avoir accepté d'en être les rapporteurs de ma thèse, et pour leurs précieux retours qui ont permis d'améliorer la qualité de ce manuscrit. Merci également à Jean-Marc PIERSON et Laurent PERRON d'avoir accepté d'être mes examinateurs.

Mes remerciements vont aussi à mes collègues du département informatique, mention spéciale à l'ensemble de l'équipe TASC. Merci à Xavier, Philippe, Charles et aux deux autres Gilles pour les conseils, la disponibilité, mais aussi la bonne humeur qui a fait en sorte que ces 3 années de recherche n'ont jamais été ennuyantes. Je n'oublie bien évidemment pas mes collègues doctorants Yunbo, Ekaterina, Anicet et Giovanni.

Je remercie également Rodolphe GIROUDEAU et Olivier NAUD qui m'ont donné envie de faire la recherche.

Je tiens enfin à remercier mes frères pour leur soutien indéfectible, ainsi que mes amis du *Beignets Haricots* et du *sprachlernzentrum* pour avoir été une seconde famille pour moi.

Je dédis cette thèse à maman, à papa, à Alex, et à Valery.

Table des matières

Liste des tableaux

Table des figures

1

Introduction

Au cours des dix dernières années, l'usage des technologies de cloud computing a accru de manière significative. Cette augmentation apporte avec elle un problème majeur, la consommation énergétique. En 2010, la quantité d'énergie électrique consommée uniquement par les data center représentait entre 1.1% et 1.5% de la consommation électrique mondiale. Aux États-Unis, ce ratio se situe entre 1.7% et 2.2% [Koo11]. Jusqu'à 30% de la consommation électrique d'un data center est utilisée par les systèmes de refroidissement [PMWV09]. Beloglazov *et al.* [BB10], [BAB12] montrent que la consommation électrique d'un data center est principalement liée à son utilisation de ressources. Dans ce contexte, le projet *EPOC* (Energy Proportional and Opportunistic Computing system réalisé dans le cadre du Cominlabs) propose un prototype de cloud data center, *EpoCloud* [BFG+17]. Le but principal de *EpoCloud* est d'optimiser la consommation électrique d'un cloud data center mono-site connecté à une source énergétique traditionnelle (fossile) et à une source d'énergie renouvelable. Les travaux du projet *EPOC* s'articulent donc autour de trois axes majeurs :

(1) Axe Matériel : Cet axe est chargé de la conception d'un équipement réseau économe en énergie.

(2) Axe Infra-structurel : Cet axe est chargé de la conception d'un système distribué responsable de l'optimisation de la consommation énergétique due à l'exécution des tâches dans le data center.

(3) Axe applicatif : Le rôle est la conception d'un cadriciel qui utilise des techniques de virtualisation et des modèles de cloud computing pour adapter les besoins des applications en fonction des métriques de performance.

Les travaux de cette thèse s'inscrivent dans le cadre du deuxième axe. En combinant l'apprentissage machine et de la programmation par contraintes, le but est de concevoir des modèles de planification de tâches qui minimisent la consommation d'énergie fossile tout en maximisant la consommation d'énergie renouvelable. On parle de planification énergiquement écologique ou *green energy aware scheduling*. Les principales contributions de cette thèses sont les suivantes. Premièrement nous proposons une contrainte globale pour modéliser les problèmes de placement de tâches en tenant compte d'une ressource comme l'énergie verte dont la disponibilité varie au cours du temps. Nous fournissons avec cette contrainte, un algorithme de filtrage borne consistant dont la complexité, de l'ordre de $\mathcal{O}(nm)$ (n étant le nombre de tâches et m le nombres d'intervalle de disponibilité d'énergie verte) est indépendante de la granularité des domaines des tâches en considération. En second lieu, nous proposons deux modèles d'apprentissage machine, l'un basé sur la programmation par contraintes et l'autre à base d'un réseau de neurones pour la prédiction et la génération de traces de charge de travail d'un data center. Après cela, nous proposons une formalisation du problème de planification énergiquement écologique (PPEE) au sein d'un data center. Par la suite nous proposons un modèle global à bases de contraintes qui intègre différents aspects inhérents au PPEE dans un data center et enfin nous proposons une heuristique de recherche dynamique pour le résoudre.

Ces différentes contributions sont présentées en détails dans cette thèse suivant le plan suivant :

- Les Chapitres 2 et 3 présentent les différents concepts de l'état de l'art dont on fait usage tout au long de la thèse. Dans le Chapitre 2, nous introduisons les notions de base en programmation par contraintes et en apprentissage machine. Ces deux domaines étant vastes, nous nous sommes limités aux concepts majeurs nécessaires à une bonne lecture de ce manuscrit. Dans le Chapitre 3 nous présentons des différentes applications issues de l'état de l'art faisant usage de ces deux paradigmes pour résoudre des problématiques de modélisation et de fonctionnement des infrastructures de cloud computing.

- Le Chapitre 4 présente une première contribution de cette thèse, la contrainte TASKINTERSECTION et son algorithme de propagation borne consistant. La contrainte TASKINTERSECTION prend en entrée un ensemble de tâches à durée variable et soumises à des contraintes de précédence, et un ensemble d'intervalles. Son rôle est de minimiser ou de maximiser la somme cumulée de l'intersection des tâches avec les intervalles. L'idée est de placer des tâches sur des intervalles pendant lesquels l'énergie verte est disponible, ou pendant lesquels le coût de l'énergie fossile est moindre. Bien que conçue pour une application dans le contexte des data center, la contrainte TASKINTERSECTION reste une contrainte globale qui peut être utilisée sur de nombreux problèmes de planifications avec prise en compte d'une ressource à coûts. Nous l'avons par ailleurs évalué entres autres sur le problème de résumé vidéo [BBG13, BBG14], améliorant ainsi de façon significative les résultats de l'état de l'art sur ce problème. Cette contribution a fait l'objet d'une publication à **CPAIOR 2016** [WB16].

- Le Chapitre 5 s'articule quant à lui autour des modèles d'apprentissage sur la charge de travail d'un data center. Le cloud computing permet une certaine élasticité vu que les utilisateurs peuvent bénéficier dynamiquement de nouvelles ressources virtuelles lorsque leur charge de travail augmente. Une telle caractéristique nécessite des mécanismes réactifs de résolution des problèmes. Dans le Chapitre 5, nous proposons deux nouveaux modèles de prédiction de la charge de travail, basés sur la programmation par contraintes et les réseaux neuronaux. Nous présentons également deux générateurs de charge de travail qui peuvent aider à étendre un ensemble de données expérimentales afin de tester plus largement les heuristiques d'optimisation des ressources. Nos modèles ont été validés en utilisant de vraies traces d'un petit fournisseur de services cloud. Les deux approches se révèlent complémentaires : les réseaux neuronaux donnent de meilleurs résultats de prédiction, tandis que la programmation par contraintes est plus adaptée à la génération de nouvelles traces. Dans le cadre du

projet EPOC, ces travaux ont fait l'objet d'une publication au journal **Computing 2017** [BFG+17], et ont été publiées dans un article à **SBAC-PAD 2017** [MWLO+17a].

- Le Chapitre 6 propose un modèle et une heuristique de recherche de solution pour optimiser la consommation énergétique d'un data center. Avec la généralisation de l'utilisation des infrastructures de cloud computing, la consommation d'énergie est devenue un problème majeur. Des heuristiques de planification ont été proposées pour optimiser l'utilisation des ressources d'un data center afin de réduire la consommation d'énergie. Le Chapitre 6 formalise ce problème et l'aborde avec une approche différente, en tenant compte de la disponibilité des énergies renouvelables. Premièrement nous formalisons le problème de planification énergiquement écologique (PPEE) au sein d'un data center. Nous proposons par la suite un modèle global à base de programmation par contraintes ainsi qu'une heuristique dédiée pour le résoudre de façon efficace. Le modèle proposé intègre les différents aspects inhérents à la planification dynamique au sein d'un data center : architecture hétérogène des machines physiques, prise en compte de différent type d'applications (i.e., applications interactives et applications par lots), mise en route et extinction des machines physiques et coûts énergétiques associés, interruption/reprise des applications par lots, consommation des ressources CPU et RAM, migration de tâches et coûts énergétiques associés et intégration de la disponibilité d'énergies vertes. Le modèle parvient donc à réduire aussi bien les coûts associés à la consommation énergétique d'un data center que son empreinte carbone. Nous évaluons enfin le modèle par rapport au système état de l'art PIKA [LOM15], sur des traces réelles de charge de travail et des traces d'énergie verte. Cette contribution a été publiée dans un article à **ICPADS 2017** [MWLO+17b].
- Enfin, le Chapitre 7 conclue ce manuscrit par une synthèse des différentes contributions tout en présentant les différentes pistes de perspectives de travail.

En parallèle aux travaux qui constituent l'axe principal de la thèse, nous avons aussi exploré l'usage des automates temporisés pour modéliser et analyser des évènements en temps réel tel que l'évolution de la charge de travail d'un data center ou l'évolution de la quantité d'énergie verte disponible au cours du temps. Les outils de vérification des systèmes temps réel tels qu'UPAAL [BLL+95] présente une limitation majeure sur le type d'opérations prise en compte pour la mise à jour d'un compteur. Ces limitations nous ont amené à développer un mini cadriciel (framework) d'encodage d'automates temporisés par des contraintes. Le cadriciel proposé repose sur la programmation par contraintes pour modéliser et répondre à certaines requêtes relatives aux automates temporisés et automates hybrides linéaires. Grâce à ce cadriciel, nous levons les limitations des outils existants et fournissons une grande souplesse sur le type d'opérations prises en compte pour la mise a jour d'un compteur. Ce cadriciel est présenté en Annexe A

I

État de l’art

2

Techniques de modélisation et apprentissage machine.

Sommaire

La programmation par contraintes est un paradigme qui permet de résoudre des problèmes d'optimisation combinatoire. Dans ce chapitre nous présentons les concepts de base en programmation par contraintes qui seront utilisés tout au long de ce manuscrit. Après avoir introduit la notion de problème de satisfaction de contraintes, nous aborderons les mécanismes de résolution ou de recherche de solution inhérentes au paradigme.

2.1 Notions de base en programmation par contraintes

Cette section introduit d'une part les différentes notions de base en programmation par contraintes et présente d'autre part le mécanisme de résolution des problèmes de satisfaction de contraintes.

2.1.1 Définitions

Définition 1 (Problème de satisfaction de contraintes)**.** *Un problème de satisfaction de contraintes noté CSP (pour constraint satisfaction problem en anglais) [RVBW06] est un triplet $P = (X, D, C)$ où :*

- *X est un tuple de $n > 0$ variables, $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$.*
- *D est un tuple de $n > 0$ domaines, $D = \langle D_0, D_1, \ldots, D_{n-1} \rangle$ tel que $x_i \in D_i$.*
- *C est un tuple de $k > 0$ contraintes $C = \langle C_0, C_1, ..., C_{k-1} \rangle$.*

Nous définissons à présent ce qu'est une contrainte.

Définition 2 (Contrainte)**.** *Soit $P = (X, D, C)$ un problème de satisfaction de contraintes. Une contrainte $C_i \in C$ est une paire $\langle R_i, S_i \rangle$ où S_i est un sous ensemble de X appelé portée de la contrainte et R_i est une relation sur les éléments de S_i. La relation R_i spécifie les tuples de valeurs autorisées dans le produit cartésien des domaines des variables de S_i.*

Définition 3 (Solution d'un problème de satisfaction de contraintes)**.** *Soit $P = (X, D, C)$ un problème de satisfaction de contraintes. Une solution S de P est un tuple $\langle v_0, v_1, ..., v_{n-1} \rangle$ tel que $v_i \in D_i$ et pour chaque contrainte $C_i = \langle R_i, S_i \rangle \in C$, la relation R_i est valide. On dit que la solution $\langle v_0, v_1, ..., v_{n-1} \rangle$ ne viole aucune contrainte.*

La Définition 3 stipule qu'une instanciation de l'ensemble des variables d'un problème de satisfaction de contraintes constitue une solution à ce dernier si et seulement si elle ne viole aucune contrainte du problème. En général, l'espace de recherche d'un CSP est très grand. De ce fait, il n'est pas pratique de chercher une solution en parcourant tout l'espace de recherche. Pour résoudre ce problème, la programmation par contrainte repose sur le mécanisme de propagation de contraintes.

2.1.2 Propagation de contraintes

Soit $P = (X, D, C)$ un problème de satisfaction de contraintes. La propagation par contraintes est un mécanisme qui permet de supprimer du domaine d'une variable les valeurs qui conduisent à coup sûr à la

violation d'une contrainte. Cette suppression de valeurs permet de réduire de façon significative l'espace de recherche. L'Exemple 1 illustre ce processus.

Exemple 1. *Soit le CSP* $P = (X, D, C)$ *où :*

- $X = \{x, y, z\}$.
- $D = \{D(x), D(y), D(z)\}$ *avec* $D(x) = D(y) = D(z) = [1, 4]$.
- $C = \{C_1, C_2\}$ *avec* $C_1 = \langle x < y, \{x, y\}\rangle$ *et* $C_2 = \langle y < z, \{y, z\}\rangle$.

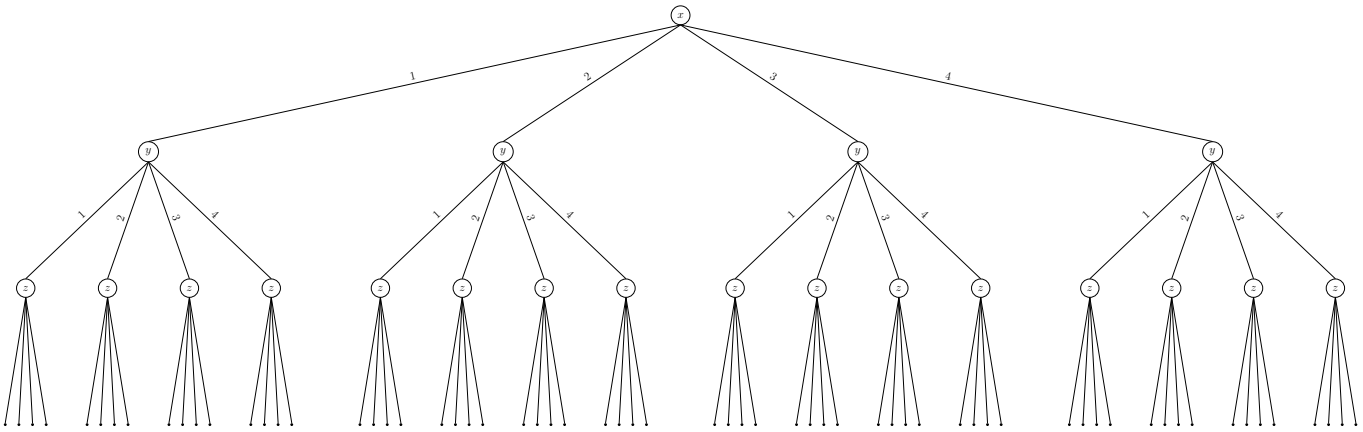


FIGURE 2.1 – Arbre de recherche du CSP de l'Example 1.

La Figure 2.1 présente l'arbre de recherche initial du CSP de l'Exemple 1 sans propager les contraintes. Nous allons maintenant propager les contraintes C_1 *et* C_2 *pour réduire l'espace de recherche de la Figure 2.1 et atteindre un point fixe. Le point fixe est atteint lorsque aucune valeur supplémentaire de ne peut être supprimée du domaine d'une variable par propagation de contraintes. Ce point fixe est atteint par propagation de contraintes suivant l'Algorithme 1.*

Algorithm 1 Recherche d'un point fixe par propagation de contrainte

Données: $P = (X, D, C)$
1: $Q \leftarrow$ Pile des contraintes
2: **TantQue** $Q \neq \emptyset$ **Faire**
3: $c \leftarrow top(Q)$ // Dépiler la contrainte c de Q
4: $c = \langle R, S\rangle$
5: **PourTout** $x \in S$ **Faire**
6: **Si** $\exists v \in Dom(x)$ qui viole la relation R **Alors**
7: $Dom(x) \leftarrow Dom(x)\backslash\{v\}$
8: **PourTout** $c' = \langle R', S'\rangle \in C$ tel que $x \in S'$ **Faire**
9: Empiler la contrainte c' dans Q
10: **Fin Pour**
11: **FinSi**
12: **Fin Pour**
13: **Fin TantQue**

En déroulant l'Algorithme 1 sur le P *on obtient* $D(x) = [1, 2], D(y) = [2, 3]$ *et* $D(z) = [3, 4]$. *La Figure 2.2 présente le nouvel arbre de recherche après propagation des contraintes.*

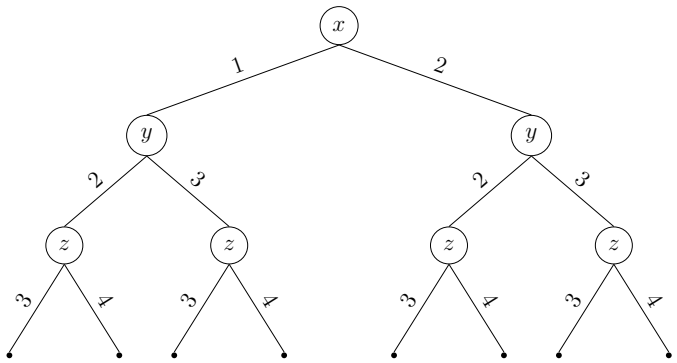


FIGURE 2.2 – Arbre de recherche du CSP de l'Example 1 apres propagation des contraintes

Après propagation des contraintes, le domaine des variables d'un problème de satisfaction de contraintes est laissé dans un état dit *cohérent* ou *consistant* . Il existe plusieurs types de consistance. Dans ce qui suit nous présentons différents types de consistance et en particulier la consistance aux bornes qui est le type de consistance que nous avons utilisé dans les travaux de cette thèse.

2.1.3 Consistance d'arc

Mackworth [Mac77] a dans sa première définition formelle limité la notion de consistance d'arc à des réseaux de contraintes binaires. Dans cette section nous présentons la définition de la consistance d'arc dans sa forme générale appelée *Consistance d'arc généralisée* [Bes06b].

Définition 4 (Consistance d'arc généralisée)**.** *Soit $P = (X, D, C)$ un problème de satisfaction de contraintes. Une contrainte $c = \langle R, S\rangle \in C$ est dite* arc consistante *si pour toute variable $x \in S$, et pour toute valeur $v \in dom(x)$, il existe une instanciation des variables de S telle que $val(x) = v$ satisfaisant la relation R. Cette instanciation est appelée support de la valeur v pour la contrainte c. Le CSP P est dit arc consistant si toutes les contraintes de P sont arc consistantes.*

À partir de la Définition 4, on déduit que pour rendre un réseau de contraintes arc consistant, il suffit de supprimer des domaines des variables toutes les valeurs qui n'ont pas de support pour au moins une contrainte. L'Algorithme 3 appelé *AC3* [BRYZ05] et ses différentes optimisations permettent de rendre un réseau de contraintes arc-consistant.

Algorithm 2 Algorithme pour la consistance d'arc généralisée AC3

Données: $P = (X, D, C)$
1: $Q \leftarrow \{(x_i, x_j) | \exists c = \langle R, S\rangle \in C$ et $x_i, x_j \in S\}$
2: **TantQue** $Q \neq \emptyset$ **Faire**
3: Sélectionner et supprimer un couple (x_i, x_j) de Q.
4: **Si** $REVISE(x_i, x_j)$ **Alors**
5: $Q \leftarrow Q \cup \{(x_k, x_i) | \exists c = \langle R, S\rangle \in C$ avec $x_k, x_i \in S$ et $k \neq j\}$
6: **FinSi**
7: **Fin TantQue**

Algorithm 3 Procédure REVISE pour AC3

Données: (x_i, x_j)
1: $DELETE \leftarrow$ `false`
2: **PourTout** $v_i \in dom(x_i)$ **Faire**
3: **Si** v_i n'a pas de support dans $dom(x_j)$ **Alors**
4: Supprimmer v_i de $dom(x_i)$
5: $DELETE \leftarrow$ `true`
6: **FinSi**
7: **Fin Pour**
8: **Retourner** $DELETE$

2.1.4 Consistance aux bornes

Soit $P = (X, D, C)$ un réseau de contraintes $c = \langle R, S \rangle \in C$ est dite consistante aux bornes (en anglais *bound($\mathbb{Z}$) consistent*) [Bes06b] si et seulement si pour toute variable $x \in S$ ayant pour domaine $dom(x)$, les valeurs $\max(dom(x))$ et $\min(dom(x))$ ont chacune un support par rapport à la contrainte c. Le réseau de contraintes est alors dit consistant aux bornes si toutes ses contraintes le sont. Ce niveau de consistance garanti l'existence d'une solution faisant intervenir les bornes des variables. Les travaux de cette thèse s'articulant autour de la planification des tâches au sein d'un data center, la consistance aux bornes prend tout son sens. Dans ce document, et en particulier au Chapitre 4, sauf mention contraire, le degré de consistance est la consistance aux bornes.

2.1.5 Heuristiques de recherche

Bien que la propagation de contraintes permette de réduire a priori l'espace de recherche d'un problème de satisfaction de contraintes, il est nécessaire d'éviter de parcourir jusqu'aux feuilles les chemins ne menant pas à une solution. En d'autres termes, on doit stopper l'exploration d'un chemin dans l'arbre de recherche aussitôt qu'une instanciation viole une contrainte du problème. Il existe plusieurs algorithmes qui assurent cette propriété pour le parcours de l'arbre de recherche d'un CSP. Dans les travaux de cette thèse, nous avons utilisé le *Forward checking* [BMFL02]. Le principe de l'algorithme de parcours en *forward checking* est de propager chaque instanciation de valeur dans le réseau. Ceci permet à chaque étape de supprimer des domaines des variables non instanciées les valeurs non consistantes avec l'instanciation courante. Le processus est présenté par l'Algorithme 4.

Reste maintenant à déterminer l'ordre dans lequel les variables du problème sont choisies puis instanciées, comment est construit l'arbre de recherche. En effet, selon l'ordre dans lequel les variables d'un problème sont instanciées, on peut détecter plus ou moins rapidement un échec. L'objectif étant de détecter un échec le plus tôt possible, on utilise des heuristiques de recherche qui fixent l'ordre dans lequel les variables sont sélectionnées, et l'ordre dans lequel les valeurs des domaines sont attribuées. Il existe deux principaux types d'heuristique, les heuristiques statiques et les heuristiques dynamiques.

Algorithm 4 Algorithme Forward Checking

PROCEDURE $forward\_checking(X, I)$
Données: X : Ensemble de variables à instancier, I : Instanciation courante
1: **Si** $X = \emptyset$ **Alors**
2: I est une solution
3: **SiNon**
4: Sélectionner $x \in X$
5: **FinSi**
6: choisir $v \in dom(x)$
7: **TantQue** $\neg check\_forward(x, v, X)$ **Faire**
8: $dom(x) \leftarrow dom(x) \backslash \{v\}$
9: choisir $v \in dom(x)$
10: **Fin TantQue**
11: $forward\_checking(X \backslash \{x\}, I \cup \{x \leftarrow v\})$

Algorithm 5 Procédure de vérification $check\_forward$ pour l'algorithme forward check

PROCEDURE $check\_forward(x, v, X)$
Données: X : Ensemble de variables à instancier, x : variable appartenant à X, v : valeur dans $dom(x)$
1: $CONSISTANT \leftarrow$ `true`
2: **PourTout** $x' \in X \backslash \{x\}$ **Faire**
3: **TantQue** $CONSISTANT$ **Faire**
4: **PourTout** $v' \in dom(x')$ **Faire**
5: **Si** l'instanciation $\{x \leftarrow v, x' \leftarrow v'\}$ viole une contrainte **Alors**
6: $dom(x') \leftarrow dom(x') \backslash \{v'\}$
7: **FinSi**
8: **Fin Pour**
9: **Si** $dom(x') = \emptyset$ **Alors** $CONSISTANT \leftarrow$ `false`
10: **FinSi**
11: **Fin TantQue**
12: **Fin Pour**
13: **Retourner** $CONSISTANT$

2.1.5.1 Heuristiques statiques

Le but d'une heuristique fixant l'ordre dans lequel les variables d'un problème sont instanciées est de détecter les échecs au plus tôt. C'est le principe *fail first* introduit par Haralick [HE80]. Ce principe a donné naissance aux heuristiques statiques. Les heuristiques statiques prédéfinissent un ordre sur les variables et les valeurs avant le début de la recherche. La plus simple est l'heuristique *lexicographique* qui fixe un ordre lexicographique sur les variables. Dans cet esprit de détection de l'échec le plus tôt possible, l'ordre lexicographique n'est pas toujours la solution adéquate. Une alternative couramment utilisée est l'heuristique du *degré*. L'heuristique du *degré* fixe l'ordre d'instanciation des variables en fonction du nombre de contraintes les impliquant.

L'usage des heuristiques statiques présente à mon avis un avantage majeur qui n'est pas souvent mentionné dans la littérature. Du fait qu'elles prédéfinissent un ordre avant le début de la recherche, l'arbre de recherche est le même quel que soit l'algorithme de propagation utilisé. Cette propriété permet de comparer deux différents algorithmes de propagation de contraintes sur la résolution d'une même instance de problème. Cette propriété nous permet entre autres de comparer objectivement les algorithmes de propagation que nous proposons dans la partie Contributions II à des reformulations sur des problèmes donnés.

2.1.5.2 Heuristiques dynamiques

Les heuristiques dynamiques spécifient de façon dynamique l'ordre d'instanciation des variables à chaque étape. Le principe *fail first* reste en vigueur. L'heuristique *dom* est un exemple d'heuristique dynamique, elle ordonne les variables à chaque étape de propagation en fonction du nombres de valeurs consistantes avec l'instanciation partielle courante.

Le problème avec *dom* est que si à une étape plusieurs variables ont une même taille de domaine, il devient difficile de choisir la prochaine variable. Pour résoudre ce genre de problème autrement qu'avec un choix aléatoire, Forst et Dechter [FD+95] proposent *dom* + *deg*. L'heuristique *dom* + *deg* fixe l'ordre des variables en fonction de la taille des domaines et en cas d'égalité, elle repose sur le nombre de contraintes impliquant chacune des variables pour les départager.

Dans le Chapitre 6 nous proposeront une heuristique de recherche dynamique pour faciliter la recherche de solution d'un problème de planification énergiquement écologique.

2.2 Notions de base en apprentissage machine

L'apprentissage machine [CMM83] est un ensemble de techniques informatique qui permettent d'apprendre une fonction à partir d'exemples de la dite fonction, ou de classifier en ensemble de données selon un classificateur automatiquement appris. Il existe deux principaux type d'apprentissage, l'apprentissage dit supervisé et l'apprentissage dit non supervisé. Les Sections 2.2.1 et 2.2.2 présentent ces deux types d'apprentissage en s'appuyant sur des algorithmes standards de la littérature. Ces deux types d'apprentissage

sont par ailleurs utilisés dans le Chapitre 5.

2.2.1 Apprentissage supervisé : Réseau de neurones

L'apprentissage supervisé [KZP07] est un ensemble de techniques qui permettent d'apprendre une fonction d'étiquetages à partir d'exemples déjà étiquetés. Les réseaux de neurones permettent de réaliser un tel apprentissage. Dans cette section nous présentons la notion de réseau de neurones ainsi que son principe d'apprentissage et d'inférence.

Un réseau de neurones artificiel est une structure interconnectée de nœuds ou neurones qui lisent une donnée en entrée et calcule un résultat en sorti. Le résultat en sorti prédit une classe ou une valeur cohérente avec la donnée d'entrée. La Figure 2.3 présente un réseau de neurones simple.

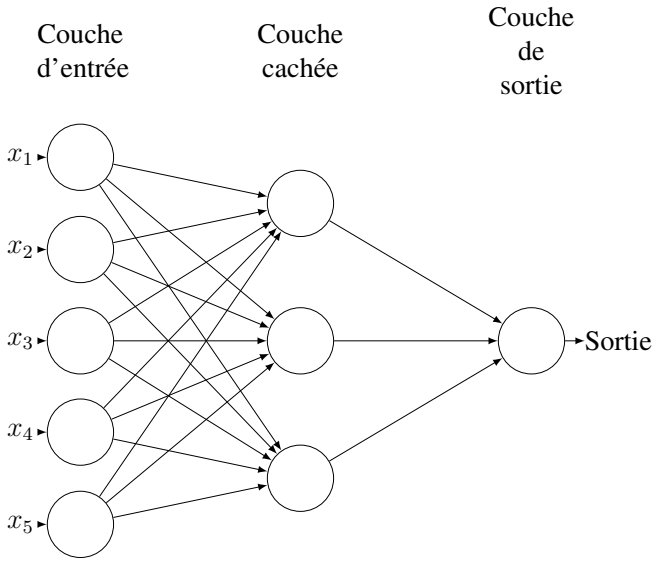


FIGURE 2.3 – Un réseau de neurones simple.

Le réseau de neurones de la Figure 2.3 est structuré comme suit :

- Une couche d'entrée contenant 5 neurones.
- Une seule couche cachée contenant trois neurones.
- Une couche de sortie contenant un seul neurone.

Un réseau de neurones peut avoir plus d'une couche cachée, et le nombre de neurones dans chaque couche peut varier. Chaque arc du réseau de neurones est associé à un *poids* noté w et chaque neurone est associé à une valeur b appelée *biais*. Par ailleurs, chaque réseau de neurones est associé à une fonction

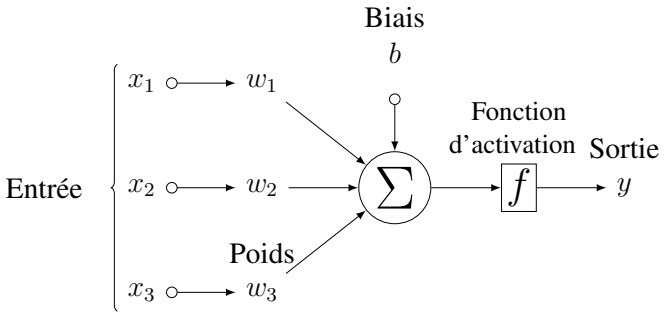


FIGURE 2.4 – Les différents paramètres d'un réseau de neurones.

appelée *fonction d'activation*. La Figure 2.4 présente les différents paramètres associés à un réseau de neurones.

Grâce aux pondérations, aux biais et à la fonction d'activation, le réseau de neurones calcule la sortie y d'une donnée d'entrée $x = (x_1, x_2 \dots x_k)$ suivant l'équation :

$$y = f(\sum_{i=1}^{k} w_i x_i + b_j) \tag{2.1}$$

Dans cette formule, x est un vecteur, w_i est le poids de l'arc connectant la i^{me} composante de x au neurone j de la couche cachée, et b_j est le biais du neurone j de la couche cachée. Par ailleurs, cette formule suppose que le réseau soit constitué d'une seule couche cachée, mais peut tout aussi bien être adaptée pour le cas de réseaux à plusieurs couches cachées.

En général on dénote W la matrice k par m de poids, par B la matrice m par 1 des biais et par X le vecteur d'entrée de taille k.

Ceci permet de simplifier l'équation 2.1 à l'équation 2.2 [Nie15].

$$y = f(WX + B) \tag{2.2}$$

La fonction d'utilisation communément utilisée est la fonction *sigmoïde* $\sigma = \frac{1}{e^{-z}}$ où $z = WX + B$. Dans les travaux de cette thèse et sauf mention contraire, la fonction d'activation f utilisée pour les réseaux de neurones est la fonction sigmoïde σ. Le problème d'apprentissage d'un réseau de neurones consiste à utiliser les données d'apprentissage pour calculer les valeurs des matrices W et B de façon à minimiser une fonction de coût c. Pour résoudre ce problème, l'algorithme couramment utilisé est l'algorithme du gradient [Bot10], en anglais *gradient descent*. La prochaine section présente l'algorithme du gradient.

L'algorithme de gradient descent

Nous rappelons que dans un réseau de neurones, le problème d'apprentissage consiste à trouver les poids W et les biais B de façon à minimiser une fonction de coûts de W et B. Dans notre cas, la fonction de coûts

utilisée est l'erreur quadratique moyenne $c(W, B)$. Dans tout ce qui suit, nous considérons c comme étant une fonction de w et b

L'algorithme du gradient procède en plusieurs étapes. Au début, on fixe des valeurs à w et b. Ces valeurs peuvent être choisies aléatoirement. Nous dénotons w_0 et b_0 les valeurs de w et b à l'étape 0. La prochaine étape consiste à mettre à jour w_0 et b_0 à w_1 et b_1 de telle façon que $c(w_1, b_1) \leq c(w_0, b_0)$. La variation de $c(w_0, b_0)$ à $c(w_1, b_1)$ est Δc et est donnée par la formule :

$$\Delta c = \frac{\delta c}{\delta w}\Delta w + \frac{\delta c}{\delta b}\Delta b. \tag{2.3}$$

avec :

- $\frac{\delta c}{\delta w}$: la dérivée partielle de c par rapport à w
- $\frac{\delta c}{\delta b}$: la dérivée partielle de c par rapport à b
- $\Delta w = w_1 - w_0$
- $\Delta b = b_1 - b_0$
- $\Delta w = w_1 - w_0$
- $\Delta b = b_1 - b_0$

Soit $\Delta v = (\Delta w, \Delta c)$ et $\nabla c = (\frac{\Delta c}{\delta w}, \frac{\Delta c}{\delta b})$ le gradient de c, alors $\Delta c = \nabla c.\Delta v$.

L'idée est de choisir Δv de façon à ce que Δc soit négatif i.e tel que c décroit.

Soit $\Delta v = -\eta \nabla c$ où η est un nombre réel positif, alors $\Delta c = -\eta \|\nabla c\|^2$. Comme $\|\nabla c\|^2$ et η sont positifs, alors Δc est négatif. La valeur η détermine l'ordre de grandeur de Δc, et est appelé *taux d'apprentissage*.

Le gradient ∇c est calculé par l'algorithme de propagation en arrière, en anglais *backpropagation* [AON+09].

2.2.2 Apprentissage non supervisé : K-means

Par opposition à l'apprentissage supervisé, nous présentons ici la notion d'apprentissage dit non supervisé. L'apprentissage non supervisé [HTF09] est un ensemble de techniques d'apprentissage qui permettent de faire des inférences à partir d'un ensemble de données non étiquetés. Les méthodes d'apprentissage non supervisés permettent en général d'apprendre un classificateur à partir des données d'apprentissage. Un des algorithmes d'apprentissage non supervisé les plus populaires est le *K-means* [PM11].

L'algorithme du K-means permet de partitionner un ensemble de données en un nombre k d'ensemble appelés clusters. Pour un ensemble de données $(x_1, x_2, x_3, ..., x_n)$, le paramètre $k(k \leq n)$ spécifiant le nombre de clusters à créer, l'algorithme du K-means procède en trois étapes :

1. Déterminer les coordonnées de k centroïdes correspondant aux k clusters.
2. Calculer la distance de chaque objet x_i aux k centroïdes
3. Associer chaque objet a un cluster unique de façon à minimiser le diamètre du cluster. Le diamètre d'un cluster étant la distance entre les deux objets du cluster les plus éloignés l'un de l'autre.

L'objectif de l'algorithme du K-means [MU13] peut être exprimé par la formule :

$$\arg\min_{s} \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2 \tag{2.4}$$

où, μ_i est le centroïde du cluster S_i.

3

PPC et apprentissage machine pour le cloud computing

Sommaire

Dans ce chapitre, nous présentons les différents travaux de l'état de l'art en programmation par contraintes et en apprentissage machine qui sont corrélés et que nous utiliserons dans la partie Contributions de cette thèse. Le chapitre est organisé comme suit.

- La Section 3.1 présente d'une part un modèle à base de programmation par contraintes pour la planification de tâches dans un data center et d'autre part un ensemble de contraintes temporelles qui nous seront utiles pour la modélisation des traces de charge CPU d'un data center et enfin un gestionnaire de consolidation autonome pour clusters reposant entièrement sur la programmation par contraintes.

- La Section 3.2 quant à elle présente un modèle d'apprentissage machine pour l'apprentissage des traces de charge d'un data center.

3.1 Programmation par contraintes pour le cloud computing

3.1.1 Planification de tâches dans un data center

Dans [HDL11], Hermenier et al. présentent une vision du data center sous forme de *bin packing* dynamique dans lequel les serveurs sont des boîtes dans lesquelles il faut ranger les applications sous différentes contraintes de ressources et de placement. Par ailleurs, avec l'évolution de la disponibilité des ressources au sein d'un data center, il est nécessaire de déplacer les applications. Dans ce contexte, ils introduisent le problème du *bin repacking scheduling*. Le problème du *bin repacking scheduling* inclus une composante *bin packing* et une composante *scheduling*. La composante *bin packing* permet de placer les tâches (encapsulées dans des machines virtuelles) au sein des serveurs en respectant les contraintes de placement et de ressources. Lorsque l'état du système évolue, la composante *scheduling* permet de calculer un ensemble de transitions faisant passer le système d'un état initial à un état final avec un temps minimal. Pour transiter d'un état initial à un état final, chaque machine virtuelle occupe les ressources sur le serveur initial avant et pendant la transition, tandis que l'occupation de ressource au sein du serveur final se fait pendant et après la transition. Nous énonçons à présent la définition formelle du problème du *bin repacking scheduling.*

Définition 5 (Bin repacking scheduling)**.** *Soit $\mathcal{R}$ un ensemble de boîtes $p-$dimensionnelles. Chaque boîte $r \in \mathcal{R}$ a une capacité fixe $B_r \in \mathbb{N}^p$. Soit $\mathcal{J}$ un ensemble d'objets $p-$dimensionnels. Chaque objet $j \in \mathcal{J}$ a une capacité initiale $b_j^o \in \mathcal{N}^p$ et une capacité finale $b_j^f \in \mathcal{N}^p$. L'état initial du système est connu et définit par une correspondance $s_o : \mathcal{J} \to \mathcal{R}$ vérifiant $\sum_{j \in s_o^{-1}(r)} b_j^o \leq B_r$ pour tout $r \in R$, où $s^{-1}(r) \subset \mathcal{R}$ est l'antécédent de $\mathcal{J}$ par s. Pour chaque objet $j \in \mathcal{J}$, la table $\Delta_j \subseteq \mathcal{T} \times \mathcal{R}$ représente l'ensemble des transitions possibles. Chaque élément $\delta = (\tau, r) \in \Delta_j$ indique qu'une transition de type τ peut être appliquée à l'objet j pour le réassigner de sa boîte initiale $s_o(j)$ à la boîte finale r. Par ailleurs, chaque transition $\delta \in \Delta_j$ est associée à une durée de transition $d_\delta \in \mathbb{N}$ et à un poids $w_\delta \in \mathbb{N}$.*

Le problème de bin repacking scheduling consiste à associer à chaque objet $j \in \mathcal{J}$ une transition $\delta(j) = (\tau(j), s_f(j)) \in \Delta_j$ et un temps de début de transition $t_j \in \mathbb{N}$ tel que :

(1) Les contraintes sur les capacités des boîtes soient respectées à tout instant :

$$\sum_{\substack{j \in s_o^{-1}(r) \\ t < t_j + d_{\delta(j)}}} b_j^o + \sum_{\substack{j \in s_f^{-1}(r) \\ t \geq t_j}} b_j^f \leq B_r, \forall r \in \mathcal{R}, \forall t \geq 0, \tag{3.1}$$

(2) et la somme pondérée des temps de complétion est minimisée :

$$\sum_{j \in \mathcal{J}} w_{\delta(j)}(t_j + d_{\delta(j)}), \tag{3.2}$$

ou de prouver qu'un tel packing n'existe pas.

3.1.2 Gestionnaire de consolidation pour clusters

Un cluster est un ensemble de machines appelées nœuds localement inter-connectées pour produire une plus grande puissance de calcul. Pour utiliser efficacement cette puissance de calcul, la consolidation dynamique permet de migrer les tâches au sein des nœuds d'un cluster en fonction de l'évolution des besoins en puissance de calcul de ces dernières. Dans [HLM+09], Hermenier et al. proposent un gestionnaire automatique de consolidation *Entropy* qui prend en compte le problème d'allocation de machines virtuelles aux nœuds d'un cluster ainsi que le problème de migration des machines virtuelles entre les différents nœuds du cluster. Le composant principal de *Entropy* est une boucle de reconfiguration qui fonctionne de la façon suivante :

(Étape 1) Le gestionnaire attend que des capteurs lui informent d'un changement d'état actif à non actif ou inversement de tout machine virtuelle.

(Étape 2) Le gestionnaire essai de calculer un plan de reconfiguration valide à partir de la configuration courante en minimisant le nombre de migrations.

(Étape 3) Lorsqu'un tel plan de reconfiguration est trouvé, il est mis en œuvre si la nouvelle configuration utilise moins de nœuds que la configuration actuelle, ou si la configuration actuelle devient invalide.

L'algorithme de l'Étape 2 procède en deux phases : la première phase calcule le nombre minimum de nœuds nécessaire pour héberger toutes les machines virtuelles, ce problème est appelé le *virtual machine placement problem* (VMPP). La seconde phase une configuration valide qui minimise le temps de reconfiguration, ce problème est appelé le *virtual machine replacement problem* (VMRP). *Entropy* utilise un ensemble des contraintes sur les capacités mémoire et CPU des nœuds pour résoudre le VMPP. Le VMRP est quant à lui résolu en utilisant le modèle du *bin repacking scheduling* [HDL11] présenté à la Section 3.1.1.

Dans la section qui suit, nous présentons les contraintes sur les séries temporelles que nous utiliserons pour modéliser des aspects tel l'évolution de la charge d'un data center ou la disponibilité d'énergie verte au cours du temps.

3.1.3 Les contraintes sur les séries temporelles

De par leur nature chronologique, les séries temporelles sont une structure adéquate pour modéliser et étudier l'évolution de la charge d'un data center ou encore la disponibilité des énergies renouvelables au cours du temps. Dans [BCDS15, ABD+16] Beldiceanu et. al. présentent un ensemble de contraintes sur les séries temporelles. Dans ces travaux, une série temporelle est décrite par les concepts suivants :

- *Signature d'une série temporelle* : La *signature* d'une série temporelle est une séquence d'opérateurs de comparaisons à valeurs dans $\{<, =, >\}$. Chaque élément de la signature d'une série temporelle est obtenu en comparant deux valeurs adjacentes de la série, ce processus étant présenté par l'Exemple 2.

Exemple 2. *Soit* $ts = x_0x_1x_2 \dots x_{n-1}$ *une série temporelle, la signature* $s(ts)$ *de* ts *est donnée par* $s(ts) = s_0s_1s_2 \dots s_{n-2}$ *avec :* $s_i = \begin{cases} \text{`} < \text{'} & \textit{si } x_i < x_{i+1} \\ \text{`} = \text{'} & \textit{si } x_i = x_{i+1} \\ \text{`} > \text{'} & \textit{si } x_i > x_{i+1}. \end{cases}$

- *Motif* : Un *motif* est une forme observable dans le tracé d'une série temporelle. Formellement, un *motif* est décrit par une expression régulière sur l'alphabet $\{<, =, >\}$. Pour détecter l'occurrence d'un motif au sein d'une série temporelle, la signature de cette dernière doit au préalable être calculée. Une occurrence de motif correspond alors à une occurrence maximale au sens de l'inclusion d'une séquence de caractères de la signature qui coïncide avec l'expression régulière du motif en considération. La Table 3.1 présente quelques exemples d'expression régulières correspondant à des motifs pouvant apparaitre dans une série temporelle. La Figure 3.1 présente une représentation visuelle du motif *peak*.

Motif	**Expression régulière**
`increasing`	$<$
`increasing_sequence`	$<(<\mid=)^*<\mid<$
`increasing_terrace`	$<=^+<$
`summit`	$(<\mid(<(=\mid<)^*<))(>\mid(>(=\mid>)^*>))$
`plateau`	$<=^*>$
`proper_plateau`	$<=^+>$
`strictly_increasing_sequence`	$<^+$
`peak`	$<(=\mid<)^*(>\mid=)^*>$
`inflexion`	$<(<\mid=)^*>\mid>(>\mid=)^*<$
`steady`	$=$
`steady_sequence`	$=^+$
`zigzag`	$(<>)^+(<\mid<>)\mid(><)^+(>\mid><)$

TABLE 3.1 – Exemples de motifs et expressions régulières correspondantes.

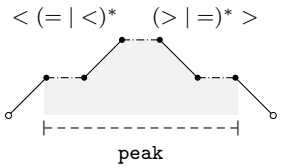


FIGURE 3.1 – Interprétation visuelle du motif peak.

- *Attribut* : Pour une occurrence de motif, un *attribut* est une propriété quantifiable du motif. La *largeur* ou en anglais *the width* est un exemple d'attribut. L'Exemple 3 présente le calcul de la largeur du motif *peak*.

Exemple 3. *Considérons les deux occurrences maximales de peak :*

- $2, 3, 5, 5, 6, 3, 1$

- $1, 5, 5, 5, 5, 5, 5, 1$

tous deux issus de la série temporelle $ts = (4, 4, 2, 2, 3, 5, 5, 6, 3, 1, 1, 5, 5, 5, 5, 5, 5, 1)$. *Sachant que les deux bords ne comptent pas, on a :*

- $width(3, 5, 5, 6, 3) = 5$
- $width(5, 5, 5, 5, 5, 5) = 6$.

- *Agrégateur* : Pour une ou plusieurs occurrences d'un motif p et pour un attribut f de p, un *agrégateur* est une fonction (e.g. $\min$, $\max$, sum) qui agrège les différentes valeurs de l'attribut f correspondant à chaque occurrence du pattern p. L'agrégateur *min* est un exemple, l'Exemple 4 illustre le calcul de l'agrégation *min*.

  Exemple 4. *Soit la série temporelle* $ts = (4, 4, 2, 2, 3, 5, 5, 6, 3, 1, 1, 5, 5, 5, 5, 5, 5, 1)$*, et ses deux occurrences de peak* $2, 3, 5, 5, 6, 3, 1$ *et* $1, 5, 5, 5, 5, 5, 5, 1$ *et l'attribut width. Nous agrégeons la valeur width de chaque occurrence de peak avec l'agrégateur min comme suit :*
  $min\_width\_peak(4, 4, 2, 2, 3, 5, 5, 6, 3, 1, 1, 5, 5, 5, 5, 5, 5, 1) =$
  $\min(width(3, 5, 5, 6, 3), width(5, 5, 5, 5, 5, 5)) = 5$.

- *empreinte* : Pour une série temporelle ts de taille n et un motif p, l'*empreinte* $fp_p(ts)$ du motif p est une séquence de n valeurs qui identifient toutes les occurrences du motif p dans la série temporelle ts. Le calcul de l'empreinte d'un motif est présenté à l'Exemple 5.

  Exemple 5. *Soit la série temporelle* $ts = (4, 4, 2, 2, 3, 5, 5, 6, 3, 1, 1, 5, 5, 5, 5, 5, 5, 1)$*. L'empreinte du motif* $peak$ *sur* ts *est donné par la séquence* $fp_{peak}(ts) = 000011111002222220$*. La séquence* $fp_{peak}(ts)$ *permet d'identifier deux occurrences de* $peaks$ *dans* ts*, en l'occurrence* $3, 5, 5, 6, 3$ *et* $5, 5, 5, 5, 5, 5$.

3.2 Apprentissage machine pour le cloud computing

Dans [IM15], Ismaeel et Miri proposent un modèle de prédiction des besoins en machines virtuelles dans un data center. Le modèle est basé sur du clustering par *k-means* combiné à des *extreme learning machines*. Les *extreme learning machines* désignent un type de réseau de neurones à une seule couche cachée. La spécificité des *extreme learning machines* est que les poids des arcs connectant les neurones de la couche d'entrée à ceux de la couche cachée sont fixés aléatoirement et ne sont jamais mis à jour. L'objectif est d'utiliser les anciennes traces d'utilisation de machines virtuelles pour prédire des besoins futurs. L'algorithme du *k-means* permet de séparer les machines virtuelles en différant clusters en fonction du type de ressource de chaque machine virtuelle (mémoire, CPU, bande passante etc). L'originalité de l'approche proposée est que chaque cluster a un réseau de prédiction qui lui es propre. Le processus se résume en 4 étapes :

(1) L'algorithme *k-means* produit en pré-traitement un ensemble de centres de clusters.

(2) Un nombre adéquat de cluster est choisi.

(3) En fonction du type de prédiction, une fenêtre de temps d'observation est choisie. Les requêtes reçues durant ce temps d'observation sont mise en correspondance avec un cluster.

(4) Une architecture adéquate et des paramètres pour l'*extreme learning machine* sont choisis, il sera utilisé pour prédire les requêtes dans chaque cluster.

II

Contributions

4

Contrainte de précédence et d'intersection de tâches.

Sommaire

4.1 Introduction

Dans de nombreuses applications réelles, il est nécessaire de prendre en considération des ressources à coût variable, ce coût variant en général dans le temps. L'électricité est un bon exemple de ce type de ressource. En effet le coût de l'électricité peut varier d'une période à l'autre [SH10]. En France comme dans plusieurs pays, le coût de l'électricité dépend de la période de la journée (heures pleines et heures creuses). La journée est ainsi partitionnée en heures creuses durant lesquelles le coût est réduit, et en heures pleines durant lesquelles le coût est élevé. Pour réduire les coûts liés à la consommation énergétique, certaines industries exécutent les activités consommatrices d'énergie repassage pendant les heures creusent. Dans ce Chapitre, nous généralisons cette idée à l'ordonnancement des tâches dans un data center. Dans un data center, une tâche est définie à minima par sa date de début, sa durée et sa date de fin. Ces valeurs sont en général reliées par un ensemble de contraintes. De manière simpliste l'objectif de ce travail se résume alors à planifier l'exécution de ces tâches de façon à minimiser les exécutions en périodes critique, et à maximiser l'usage d'énergie verte dont la disponibilité varie tout au long de la journée.

Il existe une extension de la contrainte CUMULATIVE [AB93, BLPN12] qui prend en considération ce type de ressource à coût variable [SH11]. Néanmoins, cette extension nécessite que les tâches aient des durées fixes, et ignore toute relation de précédence entre les tâches. Pour résoudre ce type de problèmes, T. K. Satish Kumar *et al.* [KCK13] proposent un modèle de problème temporel étendu avec des régions tabou. Les régions taboues sont les périodes de temps pendant lesquels aucune tâche ne doit être planifiée. L'algorithme proposé évalue le nombre de tâches planifiées dans une région taboue plutôt que l'intersection totale des tâches avec les régions taboues.

Tout d'abord, ce Chapitre introduit la contrainte TASKINTERSECTION, pour l'expression concise des problèmes de planification avec (1) coût de ressource variable en 0-1, avec (2) tâches à durée variable et avec (3) contraintes de précédence. Ensuite ce chapitre, il propose un algorithme de filtrage de coûts dédié pour la contrainte TASKINTERSECTION. L'algorithme de filtrage présenté est consistant aux bornes -au sens de la borne $\mathbb{Z}$ consistance [Bes06b]- pour des durées fixes.

La contrainte TASKINTERSECTION maintient supérieure ou égale ($o =$ ' $\geq$ '), ou inférieure ou égale ($o =$ ' $\leq$ ') à une variable entière la valeur de l'intersection totale d'un ensemble de tâches à durées variables et sujettes à des contraintes de précédence. Appliquée sur des instances réelles du problème de résumé vidéo [DPFP15, EM+03, BBG13, BBG14], elle permet d'améliorer par plus de 20% la solution.

Dans la pratique, des hypothèses sont faites sur le début, sur la durée et sur la fin d'une tâche. Par exemple, (1) le début d'une tâche est conditionné par la disponibilité des ressources, (2) la durée d'une tâche dépend des propriétés de la ressource, et (3) la fin d'une tâche est sujette à un délai. La contrainte TASKINTERSECTION est compatible avec ces hypothèses et peut être utilisée pour minimiser ou maximiser l'usage d'une ressource. Étant donné une date de début fixe, l'intersection d'une tâche avec un ensemble d'intervalles augmente proportionnellement avec sa durée. En effet, comme les durées des tâches sont variables, on ne peut pas simplement exprimer l'intersection maximale en prenant le dual des intervalles fixes. Dans [Sou05], la fonction de coût à minimiser dépend uniquement de la fin des tâches.

La suite de ce chapitre est organisée de la façon suivante. La Section 4.2 définie la contrainte TASKINTERSECTION. La Section 4.3 énonce et prouve une condition nécessaire et suffisante pour la faisabilité de la contrainte. •Les Sections 4.4 et 4.5 présentent l'algorithme de filtrage et son implémentation. Les

résultats expérimentaux sont présentés à la Section 4.6 et finalement la Section 6.6 conclue.

4.2 Contrainte d'Intersection de Tâches

Définition 6 (Tâche)**.** *Une tâche est caractérisée par ses variables de* début s_t*, de* durée d_t *et de* fin e_t*.*

Définition 7 (Domaine d'une variable)**.** *Le domaine d'une variable entière* var *est noté dom*(var) *et est constituée de l'intervalle* $[\underline{var}, \overline{var}]$ *dans lequel* $\underline{var}$ *et* $\overline{var}$ *dénotent respectivement la plus petite et la plus grande valeur de la variable* var*.*

Définition 8 (Instance réalisable d'une tâche)**.** *Une* instance réalisable *d'une tâche* t *est un triplet* (s_t, d_t, e_t)*,* $s_t, d_t, e_t \in \mathbb{Z}$ *tel que* $d_t > 0$ *et* $s_t + d_t = e_t$*.*

Définition 9 (Tâche normalisée)**.** *Une tâche* t *est dite* normalisé *si et seulement si :*

- $\exists d_t,\ d'_t \in \left[\underline{d_t}, \overline{d_t}\right], e_t,\ e'_t \in \left[\underline{e_t}, \overline{e_t}\right]$ *tel que* $\underline{s_t} + d_t = e_t$ *et* $\overline{s_t} + d'_t = e'_t$*,*
- $\exists s_t,\ s'_t \in \left[\underline{s_t}, \overline{s_t}\right],\ e_t,\ e'_t \in \left[\underline{e_t}, \overline{e_t}\right]$ *tel que* $s_t + \underline{d_t} = e_t$ *et* $s'_t + \overline{d_t} = e'_t$*,*
- $\exists s_t,\ s'_t \in \left[\underline{s_t}, \overline{s_t}\right],\ d_t,\ d'_t \in \left[\underline{e_t}, \overline{e_t}\right]$ *tel que* $s_t + d_t = \underline{e_t}$ *et* $s'_t + d'_t = \overline{e_t}$*.*

Dans tout ce qui suit dans ce chapitre, sauf mention contraire, nous supposons que toutes les tâche sont normalisées.

Définition 10 (Séquence normalisée de tâches)**.** *Une séquence* $\mathcal{T} = (t_0, t_1, \ldots, t_{n-1})$ *de tâches est* normalisée *si et seulement :*

1 *Toutes les tâches de* $\mathcal{T}$ *sont normalisées.*

2 $\forall t \in [0, n-2]$ *:* $\underline{e_t} \leq \underline{s_{t+1}},\ \overline{e_t} \leq \overline{s_{t+1}}$*.*

Définition 11 (Séquence normalisée d'intervalles)**.** *Une séquence d'intervalles* $\mathcal{I} = (r_0, r_1, \ldots, r_{m-1})$*, où chacun des intervalles* r *est caractérisée par deux entiers* ℓ_r *et* u_r *est dite* normalisée *si et seulement si :*

1 $\forall r \in [0, m-1]$ *:* $\ell_r < u_r$ *.*

2 $\forall r \in [0, m-2]$ *:* $u_r < \ell_{r+1}$*.*

Définition 12 (TASKINTERSECTION)**.** *Étant donné une séquence $\mathcal{T}$ de n tâches, une séquence normalisée $\mathcal{I}$ de m intervalles, un opérateur de comparaison $o \in \{\leq, \geq\}$ et une variable entière $inter$, la contrainte* TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, o, inter)$ *est vérifiée si et seulement si les conditions suivantes (4.1), (4.2), (4.3), (4.4), et (4.5) sont toutes vraies :*

$$\forall t \in [0, n-1] : s_t + d_t = e_t \tag{4.1}$$

$$\forall t \in [0, n-2] : e_t \leq s_{t+1} \tag{4.2}$$

$$\forall r \in [0, m-1] : \ell_r < u_r \tag{4.3}$$

$$\forall r \in [0, m-2] : u_r < \ell_{r+1} \tag{4.4}$$

$$\sum_{t=0}^{n-1} \left(\sum_{r=0}^{m-1} \max(\min(e_t, u_r) - \max(s_t, \ell_r), 0) \right) \; o \; inter \; . \tag{4.5}$$

Dans la Définition 12, la valeur $\max(\min(e_t, u_r) - \max(s_t, \ell_r), 0)$ représente la taille de l'intersection de la tâches t avec l'intervalle r ($[s_t, e_t] \cap [\ell_r, u_r]$).

Exemple 6. *Considérons la contrainte* TASKINTERSECTION$\left(\begin{array}{c} (\langle s_0, d_0, e_0 \rangle, \langle s_1, d_1, e_1 \rangle, \langle s_2, d_2, e_2 \rangle), \\ ([5, 9], [23, 25], [30, 40]), \;\; \leq, \;\; inter \end{array} \right)$ *où :*

$$\left\{ \begin{array}{lll} dom(s_0) = [2, 8], & dom(d_0) = [3, 15], & dom(e_0) = [11, 17], \\ dom(s_1) = [18, 23], & dom(d_1) = [5, 6], & dom(e_1) = [23, 29], \\ dom(s_2) = [31, 40], & dom(d_2) = [4, 5], & dom(e_2) = [35, 45], \end{array} \right.$$

et $dom(inter) = [0, 5]$.

La Figure 4.1 donne une solution vérifiant cette contrainte. Les rectangles représentent les trois tâches instanciées de façon à ce que l'intersection totale avec les intervalles fixés soit égale à $9 - 5 = 4$ *et appartienne au domaine de la variable d'intersection.*

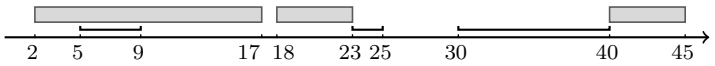


FIGURE 4.1 – Une solution pour la contrainte de l'Exemple 6 avec une intersection totale de 4.

Proposition 1. *Une réformulation de la contrainte* TASKINTERSECTION *est obtenue en reécrivant la relation* $\sum_{t=0}^{n-1}\left(\sum_{r=0}^{m-1}\max(\min(e_t,u_r)-\max(s_t,\ell_r),0)\right)\ o\ inter$ *comme une contrainte (i.e. la somme de* $n\cdot m$ *termes où chacun des termes représente l'intersection entre une tâche donnée et un intervalle donné). Nous utiliserons cette réformulation à la Section 4.6 à des fins d'évaluation.*

L'algorithme de filtrage pésenté dans ce chapitre maintient la consistance aux bornes au sens bounds($\mathbb{Z}$) consistency [Bes06b]. En supposant que le domaine de toutes les variables ne contient aucun trou, la consistance aux bornes assure que les valeurs min et max de chaque variable font partie d'une solution pour la contrainte.

Sans perte de généralité, l'opérateur de comparaison o est maintenant fixé à "$\leq$".

4.3 Faisabilité de la Contrainte d'Intersection de Tâches

En supposant que les domaines ne contiennent pas de trous, cette section présente une condition nécessaire et suffisante pour la faisabilité de la contrainte TASKINTERSECTION. Premièrement, la Proposition 2 donne une borne minimale réalisable pour l'intersection de toutes les tâches avec tous les intervalles. La borne minimale réalisable est obtenue en construisant une séquence $\mathcal{T}$ de tâches fixées vérifiant la Definition 10. En second lieu, en utilisant cette borne minimale, la Proposition 3 énonce une condition nécessaire et suffisante pour la faisabilité de la contrainte TASKINTERSECTION.

4.3.1 Borne Réalisables pour l'Intersection Totale

Pour construire une borne minimale réalisable, nous procédons de la façon suivante :

Étape 1 Pour toute date de début potentielle s de la première tâche de $\mathcal{T}$ (i.e. tâche 0), on calcule l'intersection minimale entre les intervalles $\mathcal{I}$ et toutes les instances réalisables de la tâche 0 pour lesquelles la variable de début s_0 est fixée à la valeur s.

Étape 2 Ensuite, après le calcul de l'intersection minimale de chacune des tâches $0,1,\ldots,t-1$ avec les intervalles de $\mathcal{I}$, on calcule l'intersection minimale cumulée de toutes les tâches $0,1,\ldots,t$ avec $\mathcal{I}$, pour toutes les différentes dates possibles de début de la tâche t.

Étape 3 L'intersection minimale totale des tâches $0,1,\ldots,n-1$ avec les intervalles $\mathcal{I}$ est obtenue en considérant la dernière tâche t_{n-1}.

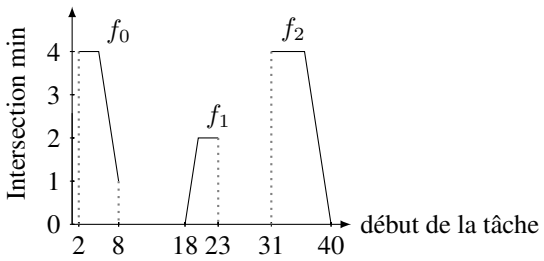


FIGURE 4.2 – Intersection minimale de chaque tâche.

Nous détaillons à présent ces trois étapes de calcul. Un point clé à considérer concerne la prise en compte directe du fait que la durée soit variable ainsi que la contrainte reliant le début, la durée et la fin de chaque tâche.

Étape 1 : Intersection minimale d'une seule tâche

Sachant que la durée d'une tâche est variable, pour un début potentiel $s \in \text{dom}(s_t)$ d'une tâche t, il peut exister plus d'une instanciation réalisable de t avec $s_t = s$. A chacune de ces instanciations correspond une valeur d'intersection de la tâche t avec les intervalles $\mathcal{I}$. La fonction f_t donne le minimum de ces intersections :

$$f_t(s) = \min_{d \in \text{dom}(d_t), e \in \text{dom}(e_t) | s+d=e} \left(\sum_{r=0}^{m-1} \max\left(\min(e, u_r) - \max(s, \ell_r), 0\right) \right)$$

Étape 2 : Intersection minimale des tâches $0, 1, \ldots, t$

Pour une tâche t (avec $t \in [0, n-1]$) et un début potentiel $s \in \left[\underline{s_t}, \overline{s_t}\right]$, il pourrait exister plus d'une instanciation réalisable de la séquence de tâches $0, 1, \ldots, t$ avec $s_t = s$. À chacune de ces instanciations correspond une valeur pour l'intersection des tâches $0, 1, \ldots, t$ avec $\mathcal{I}$. La fonction g_t donne le minimum de ces intersections. Elle est définie par la Proposition 2.

Proposition 2. *Pour une tâche t et un début $s \in \left[\underline{s_t}, \overline{s_t}\right]$, l'*intersection minimale $g_t(s)$ *des tâches $0, 1, \ldots, t$, où $s_t = s$, avec les intervalles de $\mathcal{I}$ est donnée par :*

$$g_t(s) = \begin{cases} f_0(s), & \textit{Si } t = 0, \\ f_t(s) + \min_{v_{t-1} \in [\underline{s_{t-1}}, \min(s - \underline{d_{t-1}}, \overline{s_{t-1}})]} g_{t-1}(v_{t-1}) & \textit{Sinon.} \end{cases}$$

Exemple 7. *La Figure 4.2 présente les courbes des fonctions f_t, pour chacune des tâches t de l'Exemple 6.*

Exemple 8. *La Figure 4.3 présente les courbes des fonctions g_t et la Figure 4.2 présente les courbes des fonctions f_t, pour chacune des tâches t de l'Exemple 6.*

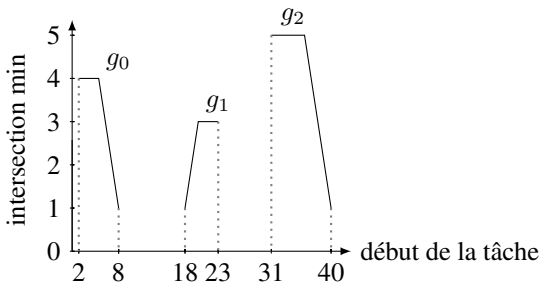


FIGURE 4.3 – $g_t, (0 \leq t \leq 2)$: intersection minimale des tâches $0, 1, \ldots, t$.

FIGURE 4.4 – Functions d'intersection f_t et $g_t (0 \leq t \leq 2)$ sur tous les intervalles fixes

Étape 3 : Intersection minimale totale

La fonction $g_t(s), s \in \left[\underline{s_t}, \overline{s_t}\right]$ calcule l'intersection minimale de toutes les tâches $0, 1, \ldots, t$ avec les intervalles fixés $\mathcal{I}$, en supposant que $s_t = s$. En conséquence, pour calculer une borne minimale pour l'intersection totale de toutes les n tâches, il est nécessaire d'évaluer $\min_{s \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]} (g_{n-1}(s))$.

Exemple 9. *De la Figure 4.3 de l'Exemple 8, on tire que* $\min_{s \in [31,40]} g_2(s) = g_2(40) = 1$. *La borne minimale de l'intersection totale des tâches* $0, 1$ *et* 2 *est donc* 1 *et est atteinte pour l'instanciation suivante :*

- $s_0 = 8, d_0 = 3, e_0 = 11$
- $s_1 = 18, d_1 = 5, e_1 = 23$
- $s_2 = 40, d_4 = 3, e_0 = 45$

4.3.2 Condition Nécessaire et Suffisante pour la Faisabilité

En se basant sur la fonction g_t introduite à la Proposition 2 de la Section 4.3.1, cette section présente une condition nécessaire et suffisante pour la faisabilité de la contrainte TASKINTERSECTION.

Proposition 3 (Condition nécessaire et suffisante)**.** *Etant donné une séquence* $\mathcal{T}$ *de* n *tâches et une séquence* $\mathcal{I}$ *de* m *intervalles, la condition nécessaire et suffisante pour la validité de la contrainte* TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$ *est :*

$$\min_{s \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]} g_{n-1}(s) \leq \overline{inter}. \tag{4.6}$$

La nécessite de cette condition découle de la définition de g_n. La preuve de la suffisance est réalisée de façon constructive. Elle consiste à construire une solution pour la contrainteTASKINTERSECTION.

Nous procédons comme suit. Tout d'abord, nous introduisons les Lemmes 1 et 2 au sujet de la caractérisation d'une durée de tâche adéquate pour minimiser l'intersection. Ensuite, le Lemme 3 montre comment

construire une solution pour la contrainte TASKINTERSECTION. Ce processus de construction de solution est illustré dans l'Exemple 10.

Notation 1. *Étant donné une séquence $\mathcal{I}$ d'intervalles et une tâche t ayant ses variables de début, de durée et de fin fixées respectivement à s, d et e, $f_t(s,d,e)$ dénote l'intersection de la tâche t avec les intervalles de $\mathcal{I}$.*

Lemme 1. *Soit $\mathcal{I}$ un ensemble de m intervalles et soit t une tâche. Étant donnée deux instanciations réalisables (s,d,e) et (s,d',e') de la tâche t où $d \leq d'$, on a $f(s,d,e) \leq f(s,d',e')$.*

Démonstration.

$$\begin{aligned} d \leq d' &\Rightarrow e \leq e' \ \text{(car } e = s+d \text{ et } e' = s+d') \\ &\Rightarrow \sum_{r=0}^{m-1} \max(\min(e,u_r) - \max(s,\ell_r), 0) \leq \sum_{r=0}^{m-1} \max(\min(e',u_r) - \max(s,\ell_r), 0) \\ &\Rightarrow f(s,d,e) \leq f(s,d',e'). \end{aligned}$$

□

Notation 2. *Etant donné une tâche t, la durée minimale possible des instances faisables de la tâche t débutant à l'instant s est dénoté d_s^{min}. A cette durée miniminimale réalisable d_s^{min} correspond une fin minimale qu'on dénote e_s^{min} ($s + d_s^{min}$).*

Lemme 2. *Étant donné une tâche t, la plus petite durée possible d_s^{min} des instanciations réalisables de la tâche t débutant à s est donnée par* $\max(\underline{d_t}, \underline{e_t} - s)$.

Démonstration. On note que $s + \underline{d_t} \leq \overline{e_t}$, car la tâche t est supposée normalisée.

1 Si $s + \underline{d_t} \geq \underline{e_t}$ alors $\underline{d_t}$ est une durée réalisable pour le début s, et $\underline{d_t} = \max(\underline{d_t}, \underline{e_t} - s)$.

2 Sinon si $s + \underline{d_t} < \underline{e_t}$, alors on doit étendre la durée minimum $\underline{d_t}$ d'au moins $\delta = \underline{e_t} - (s + \underline{d_t})$ pour atteindre la fin au plus tôt $\underline{e_t}$. Ceci implique une durée minimum de $\underline{d_t} + \delta = \underline{e_t} - s$, qui est égale à $\max(\underline{d_t}, \underline{e_t} - s)$.

□

Le prochain Lemme montre comment construire une solution réalisable de la contrainte TASKINTERSECTION.

Lemme 3. *Soit $\mathcal{T}$ une séquence de n tâches et soit $\mathcal{I}$ une séquence de m intervalles. Soit $(\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}}), (\alpha_{n-2}, d^{min}_{\alpha_{n-2}}, e^{min}_{\alpha_{n-2}}), \ldots, (\alpha_0, d^{min}_{\alpha_0}, e^{min}_{\alpha_0})$ une instanciation des tâches $n-1, n-2, \ldots, 0$ de $\mathcal{T}$, où α_t est la plus grande valeur telle que :*

$$\begin{cases} f_t(\alpha_t) = \min_{s \in [\underline{s_t}, \overline{s_t}]} f_t(s), & \text{si } t = n-1, \\ f_t(\alpha_t) = \min_{s \in [\underline{s_t}, \min(\alpha_{t+1} - \underline{d_t}, \overline{s_t})]} f_t(s) & \text{sinon.} \end{cases}$$

Si

$$\min_{\alpha_{n-1} \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]} g_{n-1}(\alpha_{n-1}) \leq \overline{inter} \tag{4.7}$$

Alors $(\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}}), (\alpha_{n-2}, d^{min}_{\alpha_{n-2}}, e^{min}_{\alpha_{n-2}}), \ldots, (\alpha_0, d^{min}_{\alpha_0}, e^{min}_{\alpha_0})$ est une solution réalisable pour la contrainte TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$.

Démonstration. La preuve du Lemme 3 se fait en deux etapes.

Premièrement, on montre par récurrence sur les indices des tâches que $(\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}}), (\alpha_{n-2}, d^{min}_{\alpha_{n-2}}, e^{min}_{\alpha_{n-2}}), \ldots, (\alpha_0, d^{min}_{\alpha_0}, e^{min}_{\alpha_0})$ est une instanciation réalisable des tâches $n-1, n-2, \ldots, 0$.

Ensuite, on montre que si l'inégalité (4.7) est vérifiée, alors l'intersection de ces tâches fixées avec les intervalles de $\mathcal{I}$ est inférieure ou égale à $\overline{inter}$.

(1)

- $[t = n-1]$
  Par hypothèse $\alpha_{n-1} \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]$. Et par le Lemme 2
  $d^{min}_{\alpha_{n-1}} \in \left[\underline{d_{n-1}}, \overline{d_{n-1}}\right]$. Par définition de $d^{min}_{\alpha_{n-1}}$, $e^{min}_{\alpha_{n-1}} \in \left[\underline{e_{n-1}}, \overline{e_{n-1}}\right]$.
  Alors $(\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}})$ est une instance réalisable de la tâche $n-1$.

- $[t < n-1]$
  Supposons que $(\alpha_t, d^{min}_{\alpha_t}, e^{min}_{\alpha_t}), (\alpha_{t+1}, d^{min}_{\alpha_{t+1}}, e^{min}_{\alpha_{t+1}}), \ldots, (\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}})$ soit une intance réalisable des tâches $t, t+1, \ldots, n-1$.

  Pour montrer que $(\alpha_{t-1}, d^{min}_{\alpha_{t-1}}, e^{min}_{\alpha_{t-1}}), (\alpha_t, d^{min}_{\alpha_t}, e^{min}_{\alpha_t}), \ldots, (\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}})$ est une instance réalisable des tâches $t-1, t, \ldots, n-1$, on doit montrer que :
  (a) l'instance $(\alpha_{t-1}, d^{min}_{\alpha_{t-1}}, e^{min}_{\alpha_{t-1}})$ est réalisable et

(b) $\alpha_{t-1} + d^{min}_{\alpha_{t-1}} \leq \alpha_t$.

(a) Puisque $\alpha_{t-1} \in [\underline{s_{t-1}}, \min(\alpha_t - \underline{d_{t-1}}, \overline{s_{t-1}})] \subseteq \left[\underline{s_{t-1}}, \overline{s_{t-1}}\right]$ alors $d^{min}_{\alpha_{t-1}} \in \left[\underline{d_{t-1}}, \overline{d_{t-1}}\right]$ et $e^{min}_{\alpha_{t-1}} \in \left[\underline{e_{t-1}}, \overline{e_{t-1}}\right]$.
Donc $(\alpha_{t-1}, d^{min}_{\alpha_{t-1}}, e^{min}_{\alpha_{t-1}})$ est réalisable.

(b) Puisque $\alpha_{t-1} \in [\underline{s_{t-1}}, \min(\alpha_t - \underline{d_{t-1}}, \overline{s_{t-1}})]$, alors $\alpha_{t-1} \leq \min(\alpha_t - \underline{d_{t-1}}, \overline{s_{t-1}})$. Il s'en suit que $\alpha_{t-1} + d^{min}_{\alpha_{t-1}} \leq \alpha_t$.

(2) Par construction,

$$\sum_{t=0}^{n-1} \left(f(\alpha_t, d^{min}_{\alpha_t}, e^{min}_{\alpha_t}) \right) = \min_{\alpha_{n-1} \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]} g_{n-1}(\alpha_{n-1})$$

Ainsi

$$\min_{\alpha_{n-1} \in \left[\underline{s_{n-1}}, \overline{s_{n-1}}\right]} g_{n-1}(\alpha_{n-1}) \leq \overline{inter} \Rightarrow \sum_{t=0}^{n-1} \left(f(\alpha_t, d^{min}_{\alpha_t}, e^{min}_{\alpha_t}) \right) \leq \overline{inter}$$

i.e. $(\alpha_{n-1}, d^{min}_{\alpha_{n-1}}, e^{min}_{\alpha_{n-1}}), (\alpha_{n-2}, d^{min}_{\alpha_{n-2}}, e^{min}_{\alpha_{n-2}}), \ldots, (\alpha_0, d^{min}_{\alpha_0}, e^{min}_{\alpha_0})$ est une solution réalisable de TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$.

□

La Proposition 3 découle directement du Lemme 3.

Exemple 10. *Dans le contexte de l'Exemple 9, cet example illustre comment construire une solution pour la contrainte* TASKINTERSECTION. *On a* $\min_{s \in [31,40]} g_2(s) = g_2(40) = 1$, *on cherche les instanciations* $(s_0, d_0, e_0), (s_1, d_1, e_1)$ *et* (s_2, d_2, e_2) *telles que* $f(s_0, d_0, e_0) + f(s_1, d_1, e_1) + f(s_2, d_2, e_2) = g_2(40) = 1$.

$t = 2$: $\min_{s \in [31,40]} f_2(s) = f_2(40)$ *i.e.* $\alpha_2 = 40$. *On calcule donc* $d^{min}_{\alpha_2}$ *et* $e^{min}_{\alpha_2}$ *qui sont* 4 *et* 44.

$t = 1$: *De la courbe* f_1 *dans la Figure 4.2, on a* $\min_{s \in [18,23]} f_1(s) = f_1(18) = 0$ *i.e.* $\alpha_1 = 18$. *On calcule donc* $d^{min}_{\alpha_1}$ *et* $e^{min}_{\alpha_1}$ *qui sont* 5 *et* 23.

$t = 0$: *De la courbe* f_0 *dans la Figure 4.2, on a* $\min_{s \in [2,8]} f_0(s) = f_0(8) = 1$ *i.e.* $\alpha_0 = 8$. *On calcule donc* $d^{min}_{\alpha_0}$ *et* $e^{min}_{\alpha_1}$ *qui sont* 3 *et* 11.

Donc $\{(8,3,11),(18,5,23),(40,4,44)\}$ *est une solution, avec* $f(8,3,11)+f(18,5,23)+f(40,4,44) = \min_{s\in[\underline{s_3},\overline{s_3}]} g_3(s) = g_3(40) = 1$.

4.4 Algorithme de Filtrage de la Contrainte TASKINTERSECTION

Cette section montre comment filtrer les domaines des variables de début et de fin des tâches de façon à obtenir les dates réalisables de début au plus tôt (respectivement au plus tard) et de fin au plus tôt (respectivement au plus tard) pour chacune des tâches en fonction de l'intersection maximale autorisée. Par ailleurs, on ajuste la valeur minimale d'intersection à une valeur faisable et on ajuste le domaine de la variable de durée des tâches par normalisation des tâches. Tous les mécanismes de filtrage dérivent de la condition nécessaire et suffisante présentée à la Section 4.3, qui est maintenue par un pré traitement. Nous décrivons en premier l'ensemble des valeurs à filtrer et caractérisons par la suite l'algorithme de filtrage correspondant.

4.4.1 Caractérisation des Valeurs à Filtrer

Cette section présente trois propositions 4, 5 et 6 qui décrivent les ensembles de valeurs à filtrer du domaine de la variable $inter$ et des domaines des variables de début et de fin de chaque tâche.

Proposition 4. *La plus petite valeur réalisable de la variable d'intersection* $inter$ *est* $\min_{s\in[\underline{s_{n-1}},\overline{s_{n-1}}]} g_{n-1}(s)$.

Démonstration. La preuve découle de la Proposition 2. □

Pour décider si une valeur s peut être filtrée ou non du domaine $[\underline{s_t},\overline{s_t}]$ de la variable de début s_t d'une tâche t, (avec $t \in [0, n-1]$), nous devons d'abord introduire la notion d'inverse d'une contrainte TASKINTERSECTION, ainsi que les notions d'intersection minimale préfixe et suffixe en fonction du début d'une tâche (Lemmes 4 et 5). On utilise ensuite ces intersections minimales préfixe et suffixe pour évaluer l'intersection minimale totale de toutes les tâches avec les intervalles fixés, étant donnée une date de début ou de fin potentielle (Lemmes 6 et 7).

Définition 13 (Inverse d'une contrainte TASKINTERSECTION). *Soit* $\mathcal{T} = (s_0,d_0,e_0),\ldots,(s_{n-1},d_{n-1},e_{n-1})$ *une séquence de* n *tâches, et* $\mathcal{I} = [\ell_0,u_0],\ldots,[\ell_{m-1},u_{m-1}]$ *une séquence de* m *intervalles fixes. Etant donné la contrainte* TASKINTERSECTION$(\mathcal{T},\mathcal{I},\leq,inter)$*, on définit son* inverse *par la contrainte* TASKINTERSECTION$(\mathcal{T}',\mathcal{I}',\leq,inter)$*, où à chaque tâche* (s_t,d_t,e_t) *de* $\mathcal{T}$ *(avec* $t\in[0,n-1]$*) correspond la tâche* $t' = (s'_t,d'_t,e'_t)$ *de* $\mathcal{T}'$ *définie par*

- $s_t{}' = \overline{e_{n-1}} - e_t$
- $d_t{}' = d_t$

- $e_t{}' = \overline{e_{n-1}} - s_t$.

et à chaque intervalle $[\ell_r, u_r]$ *de* $\mathcal{I}$ *(avec* $r \in [0, m-1]$*) correspond un intervalle* $[\ell'_r, u'_r]$ *de* $\mathcal{I}'$ *défini par*

- $\ell'_r = \overline{e_{n-1}} - u_{m-1-r}$
- $u'_r = \overline{e_{n-1}} - \ell_{m-1-r}$.

Lemme 4 (Intersection minimale préfixe/suffixe en fonction du début d'une tâche). *L'*intersection minimale préfixe *(resp.* suffixe*)* en fonction du début d'une tâche t *de* $\mathcal{T}$ *notée* $\underline{P_t^{start}}(s)$ *(resp.* $\underline{S_t^{start}}(s)$*) est l'intersection minimale des tâches* $0, 1, \ldots, t$ *(resp.* $t, t+1, \ldots, n-1$*) avec les* m *intervalles de* $\mathcal{I}$*, pourvu que la tâche* t *ait sa variable de début fixée à* $s \in \left[\underline{s_t}, \overline{s_t}\right]$*. On a* $\underline{P_t^{start}}(s) = g_t(s)$ *et* $\underline{S_t^{start}}(s) = g_{t'}(s')$ *où* $s' = \overline{e_{n-1}} - e_s^{min}$.

Démonstration. La preuve découle de la définition de la fonction g (voir Proposition 2). □

Lemme 5 (Intersection minimale préfixe/suffixe en fonction de la fin d'une tâche). *L'*intersection minimale préfixe *(resp.* suffixe*)* en fonction de la fin d'une tâche t *de* $\mathcal{T}$ *notée* $\underline{P_t^{end}}(e)$ *(resp.* $\underline{S_t^{end}}(e)$*) est l'intersection minimale des tâches* $0, 1, \ldots, t$ *(resp.* $t, t+1, \ldots, n-1$*) avec les* m *intervalles de* $\mathcal{I}$*, pourvu que la tâche* t *ait sa variable de fin fixée à* $e \in \left[\underline{e_t}, \overline{e_t}\right]$*. On a* $\underline{P_t^{end}}(e) = \underline{P_{t'}^{start}}(e')$ *et* $\underline{S_t^{end}}(e) = \underline{S_{t'}^{start}}(e')$ *où* $e' = \overline{e_{n-1}} - e$.

Démonstration. La preuve découle de la définition de l'inverse de la contrainte TASKINTERSECTION. □

Lemme 6. *Pour une contrainte* TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$*, l'intersection mimimale de toutes les* n *tâches avec les intervalles de* $\mathcal{I}$*, pourvu que la tâche* t *ait sa variable d'origine fixée à* $s \in \left[\underline{s_t}, \overline{s_t}\right]$*, est égale à* $\underline{P_t^{start}}(s) + \underline{S_t^{start}}(s) - f_t(s)$ *et est notée* $m_{s_t=s}^{start}$.

Démonstration. Soit $s \in \left[\underline{s_t}, \overline{s_t}\right]$ et supposons que la tâche t débute à l'instant s.

- L'intersection minimale des tâches $0, 1, \ldots, t$ est donnée par $\underline{P_t^{start}}(s)$
  et l'intersection minimale des tâches $t, t+1, \ldots, n-1$ est donnée par $\underline{S_t^{start}}(s)$.

- Comme la contribution $f_t(s)$ de la tâche t est prise en compte deux fois, (dans $\underline{P_t^{start}}(s)$ et dans $\underline{S_t^{start}}(s)$) on la soustrait une fois. On a donc :
  $m_{s_t=s}^{start} = \underline{P_t^{start}}(s) + \underline{S_t^{start}}(s) - f_t(s)$.

□

Lemme 7. *Pour une contrainte* TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$ *l'intersection minimale des* n *tâches avec les intervalles de* $\mathcal{I}$*, pourvu que la tâche* t *ait sa variable de fin fixée à* $e \in [\underline{e_t}, \overline{e_t}]$*, est égale à* $\underline{P_t^{end}}(e) + \underline{S_t^{end}}(e) - f_{t'}(e')$ *où* $e' = \overline{e_{n-1}} - e$ *et est notée* $m^{end}_{e_t=e}$*.*

Démonstration. La preuve est similaire à la preuve du Lemme 6. □

Proposition 5. *Pour toute tâche* $t \in [0, n-1]$*), la plus petite valeur réalisable de* s_t *est* α_{s_t} *telle que* $\forall s \in [\underline{s_t}, \alpha_{s_t} - 1], m^{start}_{s_t=s} > \overline{inter}$ *et* $m^{start}_{s_t=\alpha_{s_t}} \leq \overline{inter}$*. De façon similaire, la plus grande valeur réalisable de* s_t *est* β_{s_t} *telle que* $\forall s \in [\beta_{s_t} + 1, \overline{s_t}], m^{start}_{s_t=s} > \overline{inter}$ *et* $m^{start}_{s_t=\beta_{s_t}} \leq \overline{inter}$*.*

Démonstration. La preuve découle du Lemme 6. □

Proposition 6. *Pour toute tâche* t *(avec* $t \in [0, n-1]$*) la plus petite valeur réalisable de* e_t *est* α_{e_t} *telle que* $\forall e \in [\underline{e_t}, \alpha_{e_t} - 1], m^{end}_{e_t=e} > \overline{inter}$ *et* $m^{end}_{e_t=\alpha_{e_t}} \leq \overline{inter}$*. De façon similaire, la plus grande valeur réalisable de* e_t *est* β_{e_t} *telle que* $\forall e \in [\beta_{e_t} + 1, \overline{e_t}], m^{end}_{e_t=e} > \overline{inter}$ *et* $m^{end}_{e_t=\beta_{e_t}} \leq \overline{inter}$*.*

Démonstration. La preuve découle du Lemme 7. □

Exemple 11. *Dans le cadre de l'Exemple 6, cet exemple montre comment utiliser les Propositions 4, 5 et 6 pour filtrer la variable* $inter$ *et les variables de début et de fin de chaque tâche. Par lecture de la Figure 4.3, on a* $\min_{s \in [\underline{s_2}, \overline{s_2}]} g_2(s) = g_2(40) = 1$*. La Proposition 4, permet de filtrer la valeur* 0 *du domaine de* $inter$*, i.e. dom*$(inter) = [1, 5]$*. En appliquant les Propositions 5 and 6 et en normalisant les tâches, on ajuste la valeur minimale de* s_0 *à* 6*, la valeur maximale de* d_0 *à* 11*, la valeur minimale de* s_2 *a* 38 *et la valeur minimale de* e_2 *à* 42*.*

4.4.2 Caractérisation du Filtrage

La Proposition 7 de cette section montre comment la borne($\mathbb{Z}$) consistance est atteinte sur les variables de début et de fin des tâches d'une contrainte TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$ lorsque les durées sont fixées.

Proposition 7. *En supposant les durés fixées, l'application des Propositions 4 et 5 sur la contrainte* TASKINTERSECTION$(\mathcal{T}, \mathcal{I}, \leq, inter)$ *la rend borne($\mathbb{Z}$) consistante par rapport aux variables de début et de fin des tâches.*

Démonstration. Tout d'abord, la Proposition 4 assure la réalisabilité de la contrainte TASKINTERSECTION. Ensuite les quantités α_{s_t} et β_{s_t} de la Proposition 5 sont respectivement la plus petite et la plus grande valeur réalisable pour la variable de début de la tâche t. □

4.5 Implémentation

L'algorithme de filtrage de la contrainte TASKINTERSECTION est décomposé en 3 partie :

- Une première partie évalue les fonctions f_t introduites à l'étape 1 de la Section 4.3.1.
- Une seconde partie calcule les fonctions g_t introduites à l'étape 2 de la Section 4.3.1.
- Une troisième partie utilise la fonction g_t pour le filtrer les filtrer les domaines :

  (1) de la variable d'intersection de la contrainte TASKINTERSECTION en fonction de la Proposition 4, et (2) des variables de début et de fin de chaque tâche en fonction des Propositions 5 et 6.

Cette section présente les algorithmes efficaces pour le calcul de f_t et g_t pour chacune des tâches.

Calcul de f_t

En utilisant deux intuitions clés, cet algorithme calcule une courbe continue par morceaux donnant la valeur de f_t, l'intersection minimale de la tâche t avec les intervalles fixes, pour un début de la tâche t à $s \in \left[\underline{s_t}, \overline{s_t}\right]$. La difficulté du calcul de f_t est double.

1 A chaque étape, on doit maintenir la contrainte de faisabilité $s_t + d_t = e_t$.

2 Pour avoir une complexité indépendante de la taille des domaines, on veut éviter d'itérer sur chaque valeur de $\text{dom}(s_t)$. En effet, la complexité ne doit pas dépendre de la granularité choisie au niveau temporel.

La première intuition est que, si la date de début d'une tâche varie d'une unité d'un instant $s \in \left[\underline{s_t}, \overline{s_t}\right]$ au prochain instant $s+1 \in \left[\underline{s_t}, \overline{s_t}\right]$, alors l'intersection minimale de la tâche t varie aussi d'au plus une unité. En d'autres termes, $|f_t(s) - f_t(s+1)| \leq 1$. Cette intuition se traduit par la représentation de la courbe de f_t par une droite ayant une pente comprise entre -1, 0 ou 1. L'algorithme crée une partition $\mathcal{P} = (p_0, p_1, \ldots, p_k)$ de $\left[\underline{s_t}, \overline{s_t}\right]$, avec $\underline{s_t} = p_0 < p_1 < \cdots < p_k = \overline{s_t} + 1$ et $k \geq 0$, telle que $f_t|_{[p_i, p_{i+1}[}$, la restriction de f_t dans $[p_i, p_{i+1}[$, soit strictement croissante (pente égale à 1), strictement décroissante (pente égale à -1) ou constante (pente égale à 0), et pour tout couple d'intervalles consécutifs $[p_i, p_{i+1}[$ et $[p_{i+1}, p_{i+2}[$ les fonctions $f_t|_{[p_i, p_{i+1}[}$ et $f_t|_{[p_{i+1}, p_{i+2}[}$ n'aient pas la même pente.

La seconde intuition pour trouver la fin d'un sous intervalle p_{i+1} de la partition est la suivante : Il existe $\delta_i \in \mathbb{N}$ tel que $f_t|_{[p_i, p_i+\delta_i[}(p_i + \delta_i) \neq f_t|_{[p_i+\delta_i, p_{i+2}[}(p_i + \delta_i)$. La valeur de p_{i+1} est donc donnée par $p_i + \delta_i$. Pour calculer δ_i nous introduisons d'abord trois quantités $\delta_{i_s}, \delta_{i_d}$ et δ_{i_e} que nous définissons à présent :

(1) Quand p_i appartient à un intervalle de $\mathcal{I}$, δ_{i_s} est la distance de p_i à la fin de cet intervalle (Figure 4.5), sinon δ_{i_s} est la distance de p_i au début du prochain intervalle lorqu'il existe (Figure 4.6), ou $+\infty$ lorsqu'il n'existe pas (Figure 4.7).

$$\delta_{i_s} = \begin{cases} u_r - p_i & \text{si} \quad in(p_i) \ \wedge \ \ell_r \leq p_i < u_r, & (a) \\ \ell_r - p_i & \text{si} \ \neg in(p_i) \ \wedge \ \text{l'intervalle } r \text{ est le premier intervalle à la droite de } p_i, & (b) \\ +\infty & \text{si} \ \neg in(p_i) \ \wedge \ \nexists r \mid \ell_r > p_i. & (c) \end{cases}$$

Où $in(p_i)$ est la fonction qui retourne `true` s'il existe un intervalle fixe r contenant p_i, i.e. tel que $\ell_r \leq p_i < u_r$, et `false` dans le cas contraire.

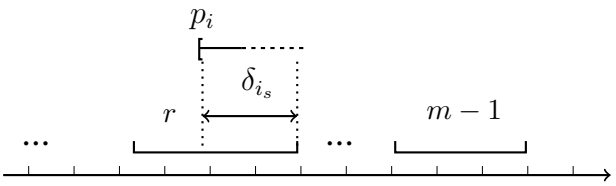


FIGURE 4.5 – Illustration de la position (a) et valeur correspondante de δ_{i_s}. On a $\ell_r \leq p_i < u_r$, δ_{i_s} est donc la distance de p_i à la fin de l'intervalle r.

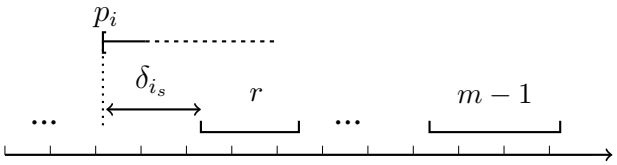


FIGURE 4.6 – Illustration de la position (b) et valeur correspondante de δ_{i_s}. On a $\neg in(p_i)$ et l'intervalle r est le premier intervalle à la droite de p_i, δ_{i_s} est donc la distance de p_i au début de l'intervalle r.

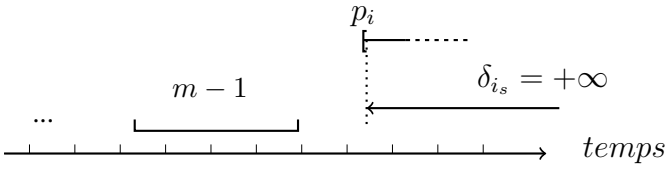


FIGURE 4.7 – Illustration de la position (c) et valeur correspondante de δ_{i_s}. On a $\neg in(p_i)$ et il n'existe pas d'intervalle après p_i, on fixe donc la valeur de δ_{i_s} à $+\infty$.

(2) δ_{i_d} est la différence entre $d_{p_i}^{min}$ et $\underline{d_t}$: $\delta_{i_d} = d_{p_i}^{min} - \underline{d_t}$.

(3) Si $e_{p_i}^{min}$ est inclus dans un intervalle de $\mathcal{I}$, alors δ_{i_e} est la distance entre $e_{p_i}^{min}$ et la fin du l'intervalle en question (d) ; dans le cas contraire δ_{i_e} est la distance entre $e_{p_i}^{min}$ et le début du prochain intervalle s'il existe (e), ou $+\infty$ dans le cas contraire (f).

$$\delta_{i_e} = \begin{cases} u_r - e_{p_i}^{min} & \text{si} \quad in(e_{p_i}^{min}) \wedge \ell_r \leq e_{p_i}^{min} < u_r, & (d) \\ \ell_r - e_{p_i}^{min} & \text{si} \ \neg in(e_{p_i}^{min}) \wedge r \text{ est le premier intervalle apres } e_{p_i}^{min}, & (e) \\ +\infty & \text{si} \ \neg in(e_{p_i}^{min}) \wedge \nexists r \mid \ell_r > e_{p_i}^{min}. & (f) \end{cases}$$

	$in(p_i) = \texttt{true}$	$in(p_i) = \texttt{false}$
$in(e^{min}_{p_i}) = \texttt{true}$	-1 if $d^{min}_{p_i} = \underline{e_t} - p_i$ 0 if $d^{min}_{p_i} = \underline{\underline{d_t}}$	0 if $d^{min}_{p_i} = \underline{e_t} - p_i$ 1 if $d^{min}_{p_i} = \underline{\underline{d_t}}$
$in(e^{min}_{p_i}) = \texttt{false}$	-1	0

TABLE 4.1 – Différentes valeurs pour la pente de $f_t|_{[p_i,p_{i+1}[}$ en fonction des positions de p_i et $e^{min}_{p_i}$, données par les valeurs de $in(p_i)$ et $in(e^{min}_{p_i})$ et en fonction des valeurs de $d^{min}_{p_i}$

La valeur de δ_i est donnée par $\min(\delta_{i_s}, \delta_{i_e})$ ou par $\min(\delta_{i_s}, \delta_{i_d})$ selon que $d^{min}_{p_i}$ soit égal à $\underline{\underline{d_t}}$ ou à $\underline{e_t} - p_i$:

$$\delta_i = \begin{cases} \min(\delta_{i_s}, \delta_{i_e}) & \text{si} \quad d^{min}_{p_i} = \underline{\underline{d_t}}, \\ \min(\delta_{i_s}, \delta_{i_d}) & \text{sinon} \quad (\text{si } d^{min}_{p_i} = \underline{e_t} - p_i). \end{cases}$$

Après avoir créé la partition $\mathcal{P}$ l'algorithme calcule $f_t|_{[p_i,p_{i+1}[}$, la restriction de f_t dans $[p_i, p_{i+1}[$. Pour ce faire, la valeur de $f_t|_{[p_i,p_{i+1}[}(p_i) = f_t(p_i)$ est calculée de façon explicite et utilisée avec la pente de $f_t|_{[p_i,p_{i+1}[}$. La Table 4.1 présente les différentes valeurs de la pente $f_t|_{[p_i,p_{i+1}[}$ en fonction des positions de p_i, $e^{min}_{p_i}$ et $d^{min}_{p_i}$.

Une fois la partition créée et la pente de $f_t|_{[p_i,p_{i+1}[}$ connue pour chaque sous intervalle $[p_i, p_{i+1}[$ de la partition, il est facile d'obtenir la constante de l'équation de droite de la courbe de f_t dans $[p_i, p_{i+1}[$ en connaissant la valeur de f_t au point p_i. Ce processus est illustré dans l'Exemple 12.

Exemple 12. *On illustre l'algorithme esquissé à la Section 4.5 pour le calcul des fonctions f_0, f_1 et f_2, l'intersection minimale des tâches 0, 1 et 2 de l'Exemple 6 avec les intervalles $[5, 9]$, $[23, 25]$, $[30, 40]$.*

1) Pour la tâche 0, on calcule tout d'abord les valeurs de $p_0, d^{min}_{p_0}, e^{min}_{p_0}, in(p_0)$ et $in(e^{min}_{p_0})$.

$p_0 = \underline{s_0} = 2,$
$d^{min}_{p_0} = \max(\underline{e_0} - p_0, \underline{d_0}) = \max(11 - 2, 3) = 9,$
$e^{min}_{p_0} = p_0 + d^{min}_{p_0} = 2 + 9 = 11,$
$in(p_0) = \texttt{false}$ (*l'intervallei_0 est le premier intervalle à la gauche de* p_0),
$in(e^{min}_{p_0}) = \texttt{true}$ ($e^{min}_{p_0}$ *est inclu dans l'intervallei_0*),
Comme $d^{min}_{p_0} = \underline{e_0} - p_0$, *alors on a* $\delta_0 = \min(\delta_{0_s}, \delta_{0_d})$,
$\delta_{0_s} = \ell_0 - p_0 = 5 - 2 = 3, \delta_{0_d} = d^{min}_{p_0} - \underline{d_0} = 9 - 3 = 6,$ *d'où* $\delta_0 = 3,$
$p_1 = p_0 + \delta_0 = 2 + 3 = 5,$
Comme $in(p_0) = \texttt{false}$ *et* $in(e^{min}_{p_0}) = \texttt{true}$, *alors* $slope_0 = 0.$

2) On calcule la valeur de $f_0(p_0) = f_0(2) = 4$ de façon explicite.

3) L'équation de $f_0|_{[p_0,p_1[}$ *est donnée par* $f_0|_{[p_0,p_1[}(s) = s \cdot slope_0 + (f(p_0) - p_0 \cdot slope_0)$. *Donc* $f_0|_{[2,5[}(s) = 4$.

On procède de façon itérative jusqu'à la condition d'arrêt $p_k = \overline{s_0} + 1$ *et on répète le processus pour les autres tâches. Les résultats sont présentés dans les Tables 4.2, 4.3 et 4.4. Ces résultats coïncident avec les courbes* f_0, f_1 *et* f_2 *de la Figure 4.2 que nous avons placée à coté des tableaux.*

TABLE 4.2 – Tâche 0 : $s_0 = [2, 8]$

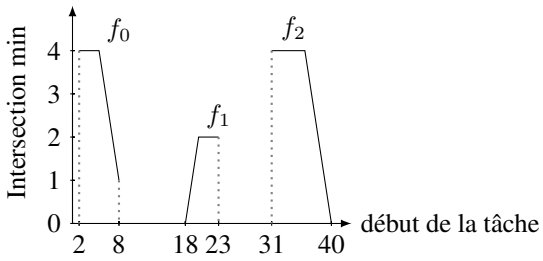
Sous intervalles	$[2, 5[$	$[5, 9[$
Pente	0	-1
Constante	4	9

TABLE 4.3 – Tâche 1 : $s_1 = [18, 23]$

Sous intervalles	$[18, 20[$	$[20, 24[$
Pente	1	0
Constante	18	2

TABLE 4.4 – Tâche 2 : $s_2 = [31, 40]$

Sous intervalles	$[31, 36[$	$[36, 41[$
Pente	0	-1
Constante	4	40



Proposition 8. *Soient* n *tâches et* m *intervalles, la complexité au pire des cas du calcul de toutes les fonctions* f_t *(avec* $t \in [0, n-1]$*) est de l'ordre de* $\mathcal{O}(nm)$.

Démonstration. Pour une tâche t, la complexité de l'algorithme est donnée par le nombre k de sous intervalles de la partition $\mathcal{P} = (p_0, p_1, \ldots, p_k)$ de $\left[\underline{s_t}, \overline{s_t}\right]$. Pour tout $p_i, p_{i+1} \in \mathcal{P}$, $\exists \delta_i$ tel que $p_{i+1} = p_i + \delta_i$. Soit un intervalle r. Pour tout $p_i \in \mathcal{P}$, δ_i est soit égal à $\min(\delta_{i_s}, \delta_{i_e})$ soit égal à $\min(\delta_{i_s}, \delta_{i_d})$, i.e. $\delta_i \in \{u_r - p_i, \ell_r - p_i, u_r - e^{min}_{p_i}, \ell_r - e^{min}_{p_i}, \delta_{i_d}\}$. δ_{i_d} ne dépend pas de l'intervalle r et peut prendre tout au plus 2 valeurs : 0 si $d^{min}_{p_i} = \underline{d_t}$ et $\underline{e_t} - p_i - \underline{d_t}$ si $d^{min}_{p_i} = \underline{e_t} - p_i$. Comme le nombre total d'intervalles est m la complexité de calcul de f_t est de l'ordre de $4m+2$. La complexité totale pour n tâches est donc $\mathcal{O}(nm)$. □

L'algorithme en pseudocode pour le calcul de f_t pour une tâche t est présenté dans l'Algorithme 6.

Algorithm 6 Algorithme de calcul de l'intersection minimale d'une tâche

PROCEDURE f_t

INITIALISATION

$s_{in} \leftarrow false$; $e_{in} \leftarrow false$; $p_i \leftarrow \underline{s_t}$; $p_{i+1} \leftarrow p_i$; $e^{min}_{p_i} \leftarrow \underline{e}$; $inter \leftarrow 0$; $i \leftarrow 0$; $n \leftarrow 0$

TantQue $i < m \wedge u_i \leq p_i$ **Faire**

  $i \leftarrow i + 1$

Fin TantQue // Cherche le premier intervalle avec lequel l'intersection est non nulle

$j \leftarrow i$;

TantQue $j < m \wedge u_j \leq e^{min}_{p_i}$ **Faire**

  $inter \leftarrow inter + max(min(e^{min}_{p_i}, u_j) - \max(p_i, \ell_j), 0)$

  $j \leftarrow j + 1$; // Tous les intervalles se terminant avant $e^{min}_{p_i}$

Fin TantQue

Si $j < m$ **Alors**

  $inter \leftarrow inter + max(min(e^{min}_{p_i}, u_j) - max(p_i, \ell_j), 0)$

FinSi // Le prochain intervalle qui se termine après $e^{min}_{p_i}$

TantQue $(p_{i+1} < \overline{s_t} \vee n = 0) \wedge i < m$ **Faire**

  Si $n = 0$ **Alors**

    $p_i \leftarrow \underline{s_t}$

    $d^{min}_{p_i} \leftarrow \max(\underline{d_t}, \underline{e_t} - p_i)$

    $e^{min}_{p_i} \leftarrow p_i + d^{min}_{p_i}$

  FinSi

  $f_{low}[n] \leftarrow p_i$

  $\delta_d \leftarrow d^{min}_{p_i} - \underline{d_t}$

  $s_{in} \leftarrow \ell_i \leq p_i \wedge p_i \leq u_i$

  $e_{in} \leftarrow j < m \wedge \ell_j \leq e^{min}_{p_i} \wedge e^{min}_{p_i} \leq u_j$

  Si $d^{min}_{p_i} \neq \underline{d_t}$ **Alors**

    Si $!s_{in}$ **Alors**

      $\delta_s \leftarrow \ell_i - p_i$

      $\delta \leftarrow \min(\delta_s, \delta_d)$

      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$

      $slope \leftarrow 0$

      $cst \leftarrow inter - slope * p_i$

    SiNon

      $\delta_s \leftarrow u_i - p_i$

      $\delta \leftarrow \min(\delta_s, \delta_d)$

      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$

      $slope \leftarrow -1$

      $cst \leftarrow inter - slope * p_i$

    FinSi

SiNon
  Si $!s_{in}$ **Alors**
    $\delta_s \leftarrow \ell_i - p_i$
    Si e_{in} **Alors**
      $\delta_e \leftarrow u_j - e^{min}_{p_i}$
      $\delta \leftarrow \min(\delta_s, \delta_e)$
      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$
      $slope \leftarrow 1$
      $cst \leftarrow inter - slope * p_i$
    SiNon
      Si $j < m$ **Alors**
        $\delta_e \leftarrow \ell_j - e^{min}_{p_i}$
      SiNon
        $\delta_e \leftarrow \overline{s_t} - p_i$
      FinSi
      $\delta \leftarrow \min(\delta_s, \delta_e)$
      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$
      $slope \leftarrow 0$
      $cst \leftarrow inter - slope * p_i$
    FinSi
  SiNon
    $\delta_s \leftarrow u_i - p_i$
    Si e_{in} **Alors**
      $\delta_e \leftarrow u_j - e^{min}_{p_i}$
      $\delta \leftarrow \min(\delta_s, \delta_e)$
      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$
      $slope \leftarrow 0$
      $cst \leftarrow inter - slope * p_i$
    SiNon
      Si $j < m$ **Alors**
        $\delta_e \leftarrow \ell_j - e^{min}_{p_i}$
      SiNon
        $\delta_e \leftarrow \overline{s_t} - p_i$
      FinSi
      $\delta \leftarrow \min(\delta_s, \delta_e)$
      $p_{i+1} \leftarrow \min(\overline{s_t}, p_i + \delta)$
      $slope \leftarrow -1$
      $cst \leftarrow inter - slope * p_i$
    FinSi
  FinSi
FinSi

78: $p_i \leftarrow p_{i+1}$
79: $d_{p_i}^{min} \leftarrow \max(\underline{d_t}, \underline{e_t} - p_i)$
80: $e_{p_i}^{min} \leftarrow p_i + d_{p_i}^{min}$
81: $i \leftarrow i + ((p_i \geq u_i)?1 : 0)$
82: **Si** $j < m$ **Alors**
83: $j \leftarrow j + ((e_{p_i}^{min} \geq u_j)?1 : 0)$
84: **FinSi**
85: $inter \leftarrow inter + delta * slope$
86: **Si** $n > 0 \wedge f_{slope}[n-1] = slope$ **Alors**
87: $f_{up}[n-1] \leftarrow p_{i+1}$
88: **SiNon**
89: $f_{up}[n] \leftarrow p_{i+1}$
90: $f_{slope}[n] \leftarrow slope$
91: $f_{cst}[n] \leftarrow cst$
92: $++n$
93: **FinSi**
94: **Fin TantQue**
95: **Si** $(p_{i+1} < \overline{s_t}) \vee (n = 0 \wedge i \geq m)$ **Alors**
96: $f_{low}[n] \leftarrow p_i$
97: $f_{up}[n] \leftarrow \overline{s_t}$
98: $f_{slope}[n] \leftarrow 0$
99: $f_{cst}[n] \leftarrow 0$
100: $++n$
101: **FinSi**
102: **Retourner** n

Calcul de g_t

La section 4.3.1 énonce et prouve la formule pour calculer l'intersection minimale des tâches 0 à t, en supposant que la tâche t débute à $s \in \left[\underline{s_t}, \overline{s_t}\right]$. Nous présentons à présent l'algorithme de calcul de la fonction g_t pour toute tâche t.

$$g_t(s) = \begin{cases} f_0(s), & \text{Si } t = 0, \\ f_t(s) + \min_{v_{t-1} \in [\underline{s_{t-1}}, \min(s - \underline{d_{t-1}}, \overline{s_{t-1}})]} g_{t-1}(v_{t-1}) & \text{sinon.} \end{cases}$$

On réécrit $g_t = f_t + h_t$ où $h_t(s)$ est la fonction qui donne le minimum de la fonction g_{t-1} dans l'intervalle $[\underline{s_{t-1}}, \min(s - \underline{d_{t-1}}, \overline{s_{t-1}})]$. La fonction $h_t(s)$ est définie par :

$$h_t(s) = \begin{cases} 0, & \text{Si } t = 0, \\ \min_{v_{t-1} \in [\underline{s_{t-1}}, \min(s - \underline{d_{t-1}}, \overline{s_{t-1}})]} g_{t-1}(v_{t-1}) & \text{Otherwise} \end{cases}$$

L'algorithme en pseudo code pour le calcul de h_t pour toute tâche t est présenté dans l'Algorithme 7.

4.6 Évaluation

On a implémenté les algorithmes de la Section 4.5 dans Choco [FP15]. Les expérimentations on été exécutées sur un processeur Intel i7 (2.93GHz) sous Mac OS X Yosemite. Nous avons conduit deux types d'expérimentations : Dans un premier temps, nous avons comparé la contrainte TASKINTERSECTION avec sa réformulation (présentée à la Proposition 1) sur des instances de problème générées aléatoirement et disponibles à [MW15]. Dans un second temps, nous avons évalué la contrainte TASKINTERSECTION dans le contexte du problème de résumé vidéo [EM+03, BBG13, BBG14] sur les instances réelles de [DPFP15].

4.6.1 Évaluation Comparative de la Contrainte TASKINTERSECTION avec sa Reformulation

On génère aléatoirement des instances de 50 tâches et 100 intervalles chacune. Pour chaque instance ainsi aléatoirement générée, on utilise la condition nécessaire et suffisante énoncée à la Section 3 pour obtenir la borne minimale de l'intersection totale. On fixe par la suite la variable $inter$ à cette borne minimale, et enfin on essaie de chercher une solution. Par la suite on relaxe de plus en plus la valeur maximale de la variable $inter$ en l'incrémentant par un pourcentage de sa borne minimale. Avec ce processus, on crée 11 configurations : $\forall i \in [0, 10]$, la configuration i correspond à une relaxation de la variable $inter$ par $100 - 10 \cdot i$ pour cent de sa borne inférieure.

Pour chaque configuration, on génère 43 instances sur lesquelles on exécute les tests avec une limite de 10 minutes. Premièrement on effectue un test dit de robustesse. Ce test évalue de combien il est difficile pour chacune des deux approches de trouver une solution pour les 43 instances de chaque configuration dans la limite des 10 minutes. Dans un second temps, on calcule le temps moyen nécessaire (dans la limite des 10 minutes) pour trouver une solution dans chaque configuration.

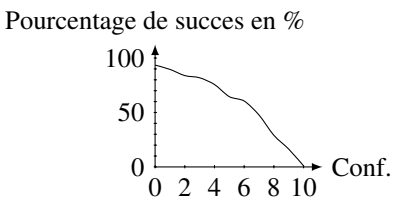
Notre algorithme parvient à trouver une solution à chaque instance de chaque configuration dans la limite de temps allouée. La reformulation cependant trouve de moins en moins de solutions quand la variable $inter$ est de moins en moins relaxée. Ceci se traduit par la courbe décroissante présentée à la Figure 4.8a. Les Figures 4.8b et 4.8c présentent les temps moyens nécessaires pour trouver une solution. De ces figures, on observe que le temps moyen nécessaire à la réformulation pour parvenir à une solution augmente de façon significative quand la variable $inter$ se voit relaxée par une plus petite valeur. Notre algorithme au contraire se comporte mieux, diminuant son temps de calcul quand la variable $inter$ est moins relaxée. L'explication tient au fait que, plus on restreint la variable $inter$, plus on a de valeurs qui sont

Algorithm 7 Procédure de calcul de h

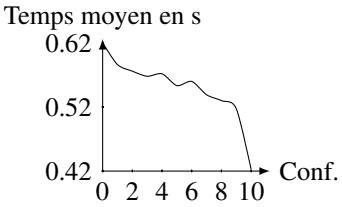
PROCEDURE Calcul de h

// Cette procédure calcule h pour toutes les n tâches.

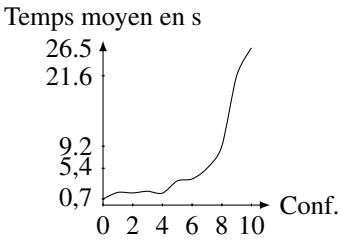
Pour t de 0 à $n-1$ **Faire**
  Pour j de 0 à $m-1$ **Faire**
    $h_{up}[t,j] = f_{up}[t,j]$
    $h_{low}[t,j] = f_{low}[t,j]$
    Si $f_{slope}[t,j] = 0$ **Alors**
      $h_{slope}[t,j] = 0$
      Si $j = 0$ **Alors**
        $h_{cst}[t,j] = f_{cst}[t,j]$
      SiNon
        Si $j > 0 \wedge h_{cst}[t,j-1] \leq f_{cst}[t,j]$ **Alors**
          $h_{cst}[t,j] = h_{cst}[t,j-1]$
        SiNon
          $h_{cst}[t,j] = f_{cst}[t,j]$
        FinSi
      FinSi
    SiNon Si $f_{slope}[t,j] < 0$ **Alors**
      Si $j = 0$ **Alors**
        $h_{slope}[t,j] = f_{slope}[t,j]$
        $h_{cst}[t,j] = f_{cst}[t,j]$
      SiNon
        Si $f_{slope}[t,j] * f_{up}[i,j] + f_{cst}[t,j] > h_{cst}[t,j-1]$ **Alors**
          $h_{cst}[t,j] = h_{cst}[t,j-1]$
          $h_{slope}[t,j] = 0$
        SiNon
          $h_{up}[t,j-1] = f_{cst}^{-1}(h_{cst}[t,j-1])$
          $h_{low}[t,j] = h_{up}[t,j-1] + 1$
          $h_{slope}[t,j] = f_{slope}[t,j]$
          $h_{cst}[t,j] = f_{cst}[t,j]$
        FinSi
      FinSi
    SiNon Si $f_{slope}[t,j] > 0$ **Alors**
      Si $j = 0$ **Alors**
        $h_{slope}[t,j] = 0$
        $h_{cst}[t,j] = f_{slope}[t,j] * f_{low}[t,j] + f_{slope}[t,j]$
      SiNon
        $h_{cst}[t,j] = h_{cst}[t,j-1]$
        $h_{slope}[t,j] = 0$
      FinSi
    FinSi
  Fin Pour
Fin Pour



(a) Robustesse de la réformulation de la contrainte TASKINTERSECTION constraint.



(b) Algorithme de filtrage de TASKINTERSECTION.



(c) Reformulation de la contrainte TASKINTERSECTION.

FIGURE 4.8 – Évaluation de la contrainte TASKINTERSECTION avec sa reformulation pour chacune des configuration.

filtrées des variables de début et de fin des tâches en vue de satisfaire la condition nécessaire et suffisante de la contrainte TASKINTERSECTION.

4.6.2 Évaluation sur des Instances réelles du Problème de Résumé Vidéo

Le problème du résumé vidéo [BBG13, BBG14] consiste à extraire sous certaines contraintes des segments de vidéo à partir d'une vidéo source. En utilisant les relations d'Allen [All83], Derrien *et al.* proposent la contrainte EXISTALLEN [DPFP15] avec un propagateur qu'ils utilisent par la suite dans une application qui génère des résumés vidéo pour des matchs de tennis. Une phase de pré-traitement extrait un ensemble d'attributs de la vidéo source (e.g. jeu, applaudissements, paroles, couleur dominante) sous forme d'intervalles. Le problème par la suite se résume alors à sélectionner des segments de vidéo qui constitueront le résumé de la vidéo. La sélection doit se faire de façon à maximiser les applaudissements en respectant les contraintes ci-dessous :

(1a) un segment ne doit pas intersecter un intervalle de parole,

(1b) un segment ne doit pas intersecter un intervalle de jeu,

(2) chaque segment sélectionné doit contenir un intervalle d'applaudissement,

(3) la cardinalité de l'intersection entre les segments et les intervalles couleur dominante ne doit pas excéder un tiers de la taille du résumé.

Comme point de départ, nous utilisons le modèle présenté dans [DPFP15] que nous modifions pour utiliser la contrainte TASKINTERSECTION cen lieu et place des contraintes (2) et (3). Le résumé doit avoir une durée totale comprise entre quatre et cinq minutes, et doit être composée de dix segments de vidéo dont la durée varie pour chacun de 10 à 120 secondes. Nous exécutons notre modèle sur les trois instances réelles disponibles fournies par Boukadida *et al.* [BBG14]. Pour garantir l'exploration du même arbre de recherche que celui présenté à la Figure 2 de [DPFP15], nous considérons une heuristique de recherche statique. L'heuristique sélectionne les variables en suivant l'ordre lexicographique, et affecte les valeurs par ordre croisant sur les domaines des variables. Les résultats de l'évaluation comparative sont présentés à la Figure 4.9 :

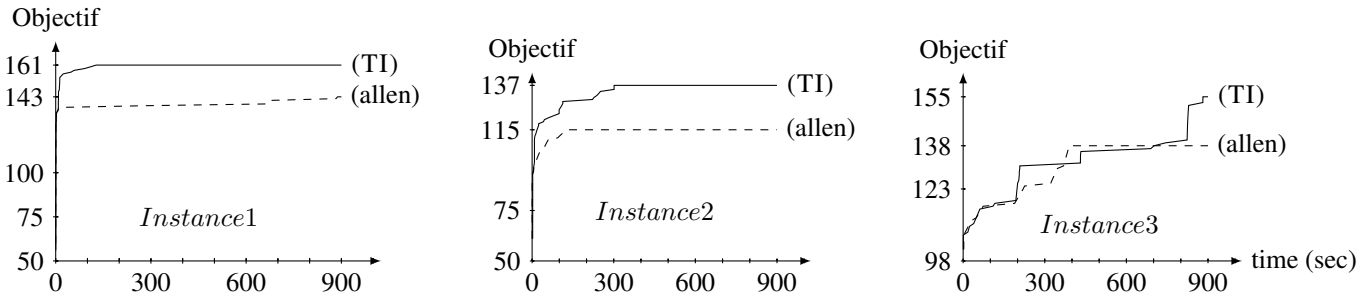


FIGURE 4.9 – Évaluation de la contribution du modèle qui inclus la contrainte TASKINTERSECTION (TI) par rapport au modèle présenté dans [DPFP15] (allen) sur le problème de résumé vidéo ; Les courbes présentent l'évolution de l'objectif, i.e. la durée totale des applaudissements, en fonction du temps de traitement avec une limite de 900 secondes .

- Pour les instances *Instance1* et *Instance2* la contrainte TASKINTERSECTION trouve immédiatement une meilleure solution et l'améliore par la suite par jusqu'à 25% sur chacune des deux instances par rapport aux meilleures solutions trouvées par EXISTALLEN.
- Pour la dernière instance en revanche, les deux algorithmes trouvent des solutions de qualité similaire au début de la recherche. Cependant à l'issue du temps alloué de recherche de $900s$, la contrainte TASKINTERSECTION améliore la qualité de la meilleure solution trouvée par EXISTALLEN jusqu'à 23%.

4.7 Conclusion

Dans ce chapitre, nous avons introduit la contrainte TASKINTERSECTION pour des problèmes d'ordonnancement de tâches à durées variables, sujettes à une chaîne de précédences et avec une ressource a coût variable en $0-1$ sur le temps. Nous fournissons une borne minimale réalisable pour l'intersection minimale et un algorithme de filtrage basé sur cette borne. Nous évaluons par la suite cette contrainte vis-à-vis de sa reformulation sur des instances de problème générées aléatoirement. Les bons résultats de cette évaluation rendent la contrainte TASKINTERSECTION adéquate pour modéliser les problèmes d'ordonnancement de tâches dans un data center avec une ressource énergétique au coût variable. Cependant la contrainte TASKINTERSECTION est définie de façon assez généraliste pour pouvoir s'adapter à plusieurs autres problèmes de la vie réelle, comme le problème de résumé vidéo. Nous avons évalué la contrainte TASKINTERSECTION sur un problème réel de résumé vidéo. La contrainte TASKINTERSECTION permet d'améliorer par jusqu'à 25% les meilleurs résultats de l'état de l'art [DPFP15]. Ce travail a fait l'objet d'une publication à CPAIOR 2016 [WB16].

5

Modèles de génération et de prédiction pour la charge des data centers.

Sommaire

5.1 Introduction

Une manière commune d'optimiser l'utilisation des ressources consiste à utiliser la prédiction de la charge de travail des data center [IM16]. Dès lors, de nombreux travaux s'articulent autour de la prédiction des dites charges de travail.

Dans in [IM15] et [IM16], Ismaeel *et al.* utilisent des "extreme learning machines", un type de réseau de neurones avec une seule couche de nœuds cachée, dont les coefficients connectant cette couche à la couche d'entrée est fixée une fois pour toute, et de façon aléatoire [HZS06]. Dans leurs travaux, ils assimilent la charge de travail d'un data center au nombre de machines virtuelles exécutées. Dès lors, prédire la charge de travail du data center consiste à prédire le nombre de machines virtuelles exécutée dans le data center dans un futur proche. L'approche que nous proposons dans ce chapitre consiste a assimiler la charge de travail d'un data center à son usage CPU qu'on modélise par des séries temporelles. Dans notre contexte, cette approche est adéquate car les spécifications des serveurs du data center mentionnent directement les capacités CPU ; par ailleurs, cette modélisation permet de prendre en compte des techniques d'économie d'économie d'énergie telles que la surcharge CPU [BFG+16].

Une autre difficulté rencontrée dans la conception d'heuristiques d'optimisation de ressources pour un data center est la disponibilité de traces réelles de charges de CPU. De telles traces sont nécessaires à la validation des algorithmes proposés. Cependant elles ne sont en général pas disponibles, ou du moins pas en quantité suffisante, pour des raisons évidentes de confidentialité. Motivés par cet état des faits, nous présentons dans ce contexte deux modèles complémentaires, l'un basé sur des réseaux de neurones, et l'autre sur la programmation par contraintes pour résoudre ces difficultés.

La contribution de ce chapitre est donc double.

- Pour prédire la charge CPU d'un data center, nous présentons et comparons deux modèles d'apprentissage machine respectivement basés sur la programmation par contraintes et sur les réseaux de neurones.

- Pour étendre tout ensemble de données de charge de travail réel, nous présentons un générateur de traces de charge de travail. Le générateur utilise un modèle de contraintes appris pour générer des séries temporelles similaires aux traces de charge de travail réelles.

La suite du chapitre est organisée comme suit : La Section 5.2 présente les charge de travail réelles utilisées tout au long du chapitre. La Section 5.3 présente les modèles d'apprentissage machine et la Section 5.4 quant à elle décrit le modèle de génération de traces. Enfin la Section 6.6 conclue ce chapitre.

5.2 Présentation des traces de charge réelles utilisées

Dans le cadre de ces travaux, nous avons établi un partenariat avec la PME française EasyVirt, spécialisée dans l'analyse des data center virtualisés. Dans une partie de ses activités, EasyVirt déploie des sondes logicielles dans l'infrastructure de ses clients et archive les données dans une base de données MySQL. Ces sondes collectent la consommation des ressources système des serveurs physiques et des machines virtuelles. Pour des raisons de confidentialité, les entreprises pour lesquelles les données ont été collectées ne sont pas explicitement mentionnées. De même, nous ne pouvons pas distribuer les données brutes collectées. Par contre, l'analyse réalisée peut être communiquée. Pour ce travail, nous avons sélectionné une trace représentative d'un centre de données de taille moyenne (50 serveurs physiques pour 1000 VM).

Plus précisément, à partir d'un compte en lecture seule sur VMwareVCenter, la solution récupère des informations statiques et dynamiques. Les informations statiques collectées sont : l'architecture Datacenter / Cluster / Server / VM, l'information statique des serveurs physiques (modèle de serveur, modèle de CPU, fréquence de CPU, nombre de noyaux, RAM disponible, etc.), informations de machine virtuelle statique (vCPU (processeur virtuel), mémoire allouée, mémoire réservée, statut VMware Tools, taille VMDK, etc.). Diverses ressources sont surveillées : processeur, mémoire, réseau et disque. Cette surveillance est effectuée toutes les 30 secondes, pour ne pas affecter les performances de VMwareVCenter. Les données sont stockées dans une base de données relationnelle classique (MySQL).

La trace sélectionnée représente six mois consécutifs d'activité, pour une base de données d'environ 2 Go. Tous les résultats présentés ci-dessous sont basés sur ces traces.

5.3 Modèles de prédiction de la charge des data centers

Pour concevoir une heuristique d'optimisation de l'usage des ressources d'un data center, nous avons besoin de savoir par avance l'évolution des besoins en ressources dans le temps. Pour ce faire, nous avons besoin d'un modèle de prédiction de la charge de travail basée sur des données de traces historiques. Cette section présente deux approches complémentaires pour la construction d'un tel modèle. La première approche utilise la programmation par contraintes est présentée dans la Section 5.3.2 tandis que la seconde utilise un réseau de neurones et est présentée dans la Section 5.3.3.

5.3.1 Classification des charges de travail

Du fait que notre jeu de données soit conséquent (50 serveurs physiques pendant 6 mois), il peut présenter plusieurs comportements différents les uns des autres. Dans une étape préliminaire, nous faisons du clustering. Le clustering est une technique beaucoup utilisée en analyse de données pour regrouper des données issues d'un ensemble par sous-ensembles similaires. Dans notre cas, le clustering permet de classer les différentes traces par type d'activité dans le data center, l'algorithme utilisé est le K-Means [PM11].

5.3.2 Modèle à base de programmation par contraintes.

Étant donné un ensemble de traces réelles sous forme de séries temporelles, nous procédons de la façon suivante :

- *Étape de clustering* : Du fait que notre jeu de données soit conséquent (50 serveurs physiques pendant 6 mois), il peut présenter plusieurs comportements différents les uns des autres. Comme étape préliminaire, nous faisons du clustering. Le clustering est une technique beaucoup utilisée en analyse de données pour regrouper des données issues d'un ensemble par paquets similaires. Dans notre cas, le clustering permet de classer les différentes traces par type d'activité dans le data center, l'algorithme utilisé est le K-Means [PM11]. Ceci est motivé par le fait que la charge d'un data center dépend de plusieurs paramètres tel que le type de jour (e.g en semaine, weekend, jour férié) ou le type de services fournis à des périodes spécifiques.
- *Étape hors ligne* : De chaque cluster cl_i, on extrait des propriétés clé qui correspondent aux caractéristiques type des séries temporelles (e.g. le plus grand pic, le nombre de pics etc...). En utilisant ces caractéristiques, on construit un modèle $m(cl_i)$ pour chaque cluster cl_i. Cette étape réalisée hors ligne est *l'étape d'apprentissage*. En fin, le *modèle de prédiction* est donné par $\cup_{i=0}^{p-1} m(cl_i)$, l'union de tous les modèles $m(cl_i)$ de chaque cluster cl_i.
- *Étape en temps réel et en ligne* : A tout instant t, le modèle de prédiction doit être capable de prédire en temps réel l'évolution de la charge de travail correspondant à l'instant $t + \epsilon$, où ϵ est un petit laps de temps.

Le reste de cette section est structuré comme suit : (1) nous présentons en détails l'étape d'apprentissage et (2) nous montrons comment le modèle construit à l'étape d'apprentissage est utilisé pour faire des prédictions en temps réel.

5.3.2.1 Étape de pré-traitement "hors ligne" : Étape d'apprentissage

En utilisant des techniques de programmation par contraintes, nous analysons les données d'entrée correspondant aux traces de la charge de travail pour extraire des propriétés pertinentes. Pour chaque cluster cl_i un modèle $m(cl_i)$ est construit après une analyse en 3 étapes de cl_i. Pour chaque motif p d'intérêt et pour chaque série temporelle ts en entrée, nous procédons comme suit :

(1) Le nombre d'occurrences du motif p dans la série temporelle ts est calculé.

(2) L'empreinte $fp_p(ts)$ du motif p est calculée.

(3) Les différentes valeurs d'agrégation des attributs du motif p sont calculées.

A la fin, tous les résultats pour tous les motifs d'intérêts sont réunis et analysés pour extraire les plages de variation des valeurs pour chaque caractéristique. Parmi ces plages, nous sélectionnons les plus pertinents pour former le modèle $m(cl_i)$ du cluster cl_i.

Exemple 13. *Considérons le cluster cl contenant les séries temporelles suivantes, chacune de taille 15 :*

```
ts_1 = 5 5 4 4 6 3 7 8 9 6 3 3 1 1 1.
ts_2 = 4 3 1 1 4 2 8 5 6 2 6 5 9 8 9.
ts_3 = 3 4 5 6 6 5 6 3 4 4 2 2 7 6 9.
```

Considérons le motif $peak$, l'attribut $width$, et les agrégateurs sum et max.

Des étapes (1) et (2) on obtient :

```
(nb_peak,2,[0,0,0,0,1,0,2,2,2,2,2,2,0,0,0]).
(nb_peak,5,[0,0,0,0,1,0,2,0,3,0,4,0,5,0,0]).
(nb_peak,4,[0,1,1,1,1,0,2,0,3,3,0,0,4,0,0]).
```

Chaque ligne de la forme $(nb\_motif, val, empr_{motif}(ts_i))$ donne le nombre val d'occurrences du motif $motif$ dans la série temporelle ts_i et l'empreinte $empr_{motif}(ts_i)$ du motif dans la série temporelle. Par exemple ts_1 contient deux occurrences de $peak$ localisées aux positions 5 et $7-12$.

De l'étape (3) on obtient la base de faits suivante, de la forme $(aggr\_attr\_motif, val, empr_{pat}(ts_i))$. Chaque fait donne la valeur val de l'agrégation $aggr$ des différentes valeurs de l'attribut $attr$ pour chaque occurrence du motif $motif$ dans la série temporelle ts_i.

```
(max_width_peak,6,[0,0,0,0,1,0,2,2,2,2,2,2,0,0,0]).
(max_width_peak,1,[0,0,0,0,1,0,2,0,3,0,4,0,5,0,0]).
(max_width_peak,4,[0,1,1,1,1,0,2,0,3,3,0,0,4,0,0]).
(sum_width_peak,7,[0,0,0,0,1,0,2,2,2,2,2,2,0,0,0]).
(sum_width_peak,5,[0,0,0,0,1,0,2,0,3,0,4,0,5,0,0]).
(sum_width_peak,8,[0,1,1,1,1,0,2,0,3,3,0,0,4,0,0]).
```

Nous réunissons ensuite ces faits pour obtenir une nouvelle base de faits de la forme $range(r, motif, attr, min, max)$, où $attr$ est un attribut du motif. min (resp. max) est la plus petite (resp. la plus grande) valeur prise par l'attribut $attr$ sur chaque série temporelle du cluster cl. En fin r est la plage obtenue en soustrayant min de max.

```
range(3, peak,nb_peak,2,5).
range(5, peak,max_width_peak,1,6).
range(3, peak,sum_width_peak,5,8).
```

Pour un attribut donné, la plage indique l'amplitude de variation des valeurs prises par l'attribut. On peut alors sélectionner les attributs d'intérêt en fixant une valeur maximum de plage acceptée.

Exemple 14. *Si on fixe la plage max à* 4 *(i.e.* $max\_range = 4$*) alors le modèle* $m(cl)$*du cluster* cl *est :*

```
m(cl) = {
range(3, peak,nb_peak,2,5),
range(3, peak,sum_width_peak,5,8).}
```

Une série temporelle ts *est alors compatible au modèle* $m(cl)$ *du cluster* cl *si et seulement si les contraintes suivantes sont toutes vérifiées :*

- $2 \leq nb\_peak(ts) \leq 5$
- $5 \leq sum\_width\_peak(ts) \leq 8$

5.3.2.2 Étape en temps réel : Étape de prédiction

Dans cette section, nous présentons comment le modèle construit pendant l'étape hors ligne est utilisé pour prédire en temps réel la prochaine valeur d'une série temporelle.

Nous introduisons d'abord la notion de préfixe d'une série temporelle :

Définition 14 (Préfixe d'une série temporelle)**.** *Étant donné un index* $t < n$ *(où* n *est la longueur de la série temporelle) le* préfixe de série temporelle *induit par* t *et noté* $pref(t)$ *est la série temporelle* $x_0 x_1 \dots x_t$ *qui représente la charge de travail du créneau horaire* 0 *au créneau* t*. Par abus de langage, dans tout ce qui suit nous utiliserons l'expression préfixe en lieu et place de préfixe de série temporelle.*

La prédiction est réalisée en 3 étapes. Étant donné un préfixe $pref(t)$, nous déterminons dans un premier temps à quel cluster il appartient. La seconde étape prédit un intervalle de valeurs potentielle pour le préfixe $pref(t)$ au créneau $t + \epsilon$. La troisième et dernière étape consiste à raffiner cet intervalle pour améliorer la précision de la prédiction.

(1) Premièrement on vérifie la compatibilité de chaque série temporelle ts avec chaque modèle de cluster $m(cl)$. La série temporelle ts est *compatible* avec le cluster cl_i si la valeur pour chaque attribut de ts (i.e nombre d'occurrence de chaque motif, empreinte de chaque motif, agrégations de valeurs) est comprise dans la plage correspondante du modèle $m(cl_i)$ du cluster cl_i.

(2) Pour chaque cluster compatible cl, on calcule l'intervalle de valeurs potentielles correspondant au créneau $(t + \epsilon)$. Cet intervalle est dénoté $I_{cl}(t + \epsilon)$ et est tel que : $\forall ts \in cl$, on a $ts(t + \epsilon) \in I_{cl}(t + \epsilon)$. Par la suite on fait l'union de tous les intervalles $I_{cl}(t + \epsilon)$ qu'on dénote $I(t + \epsilon)$.

(3) Enfin on réduit la taille de l'intervalle $I(t+\epsilon)$. Pour ce faire, on considère la série centrale de chaque cluster compatible et on calcule l'empreinte des motifs $strictly\_increasing\_sequence$ et $strictly\_decreasing\_sequence$. La série centrale d'un cluster est la série temporelle correspondant au centroïde du cluster. Ceci nous permet d'identifier les occurrences de ces motifs, cette information permet de filtrer :

- la borne min de $I(t+\epsilon)$ s'il y'a occurrence du motif $strictly\_increasing\_sequence$
- la borne max de $I(t+\epsilon)$ s'il y'a occurrence du motif $strictly\_decreasing\_sequence$

5.3.2.3 Évaluation du modèle à base de programmation par contraintes

Pour évaluer ce modèle, nous avons utilisé un jeu de séries réelles de 500 traces réelle de charge sous forme de séries temporelles, ces traces sont décrites à la Section 5.2. Les traces ont été partitionnées en 5 clusters (en utilisant la méthode présentée à la Section 5.3.1) de cardinalités respectives $142, 79, 142, 106,$ et 31. Nous avons par ailleurs séparé les données de chaque cluster en deux catégories : une sur laquelle l'apprentissage se fera, et l'autre qui sera utilisée pour la prédiction. Les données d'apprentissage représente 70% tandis que les données qui seront utilisées pour évaluer la prédiction, encore appelées données de test représentent 30% restant.

Les traces brutes ont une longueur de 1008 chacune, chaque heure étant représentée par 42 créneaux. Pour des raisons pratiques mais aussi pour éliminer le bruit, nous avons réduit cette résolution à une seule valeur pour chaque heure en utilisant la technique des moyennes glissantes, ainsi chaque créneau horaire représente une heure d'activité en utilisant . Chaque série temporelle est désormais de longueur 24 et associée à un cluster.

Pour chaque préfixe de taille k (avec $k \in [3, 23]$) issu des séries temporelles de test, on réalise les évaluations suivantes.

(1) La première expérimentation présentée à la Figure 5.1 évalue le pourcentage de cas pour lesquels il existe au moins un cluster compatible avec les préfixe.

(2) La seconde expérimentation présentée à la Figure 5.2 évalue le pourcentage de cas ou la $(k+\epsilon)^{me}$ valeur de la série temporelle de test est comprise dans l'intervalle prédit.

Des courbes présentées aux Figures 5.1 et 5.2, on observe que la qualité de la prédiction dépend de la première partie sur la compatibilité des clusters. Ceci se traduit par les pentes similaires des deux courbes. On va mesurer la qualité de la prédiction en utilisant l'erreur quadratique moyenne.

Définition 15. *L'*erreur quadratique moyenne *est une mesure standard utilisée pour évaluer l'estimation d'une valeur inconnue. Étant donné un échantillon de k prédictions $pred(1), pred(2) \ldots, pred(k)$, et un ensemble de k valeurs observées $val(1), val(2) \ldots, val(k)$, l'erreur quadratique moyenne de la prédiction est donné par :*

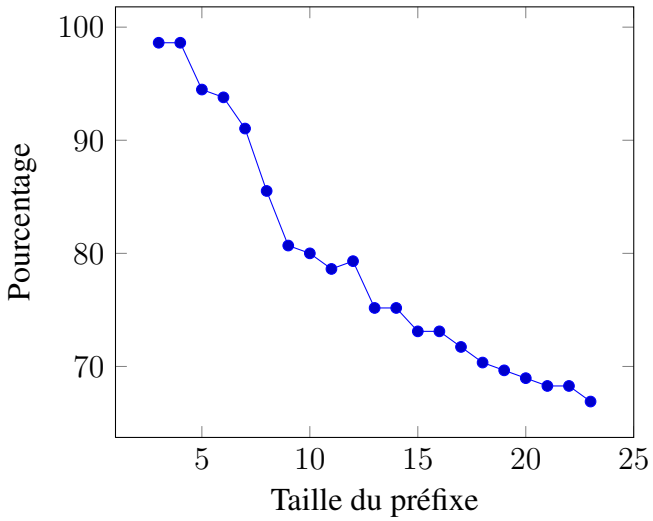


FIGURE 5.1 – Pourcentage des cas où il existe au moins un cluster compatible avec le préfixe.

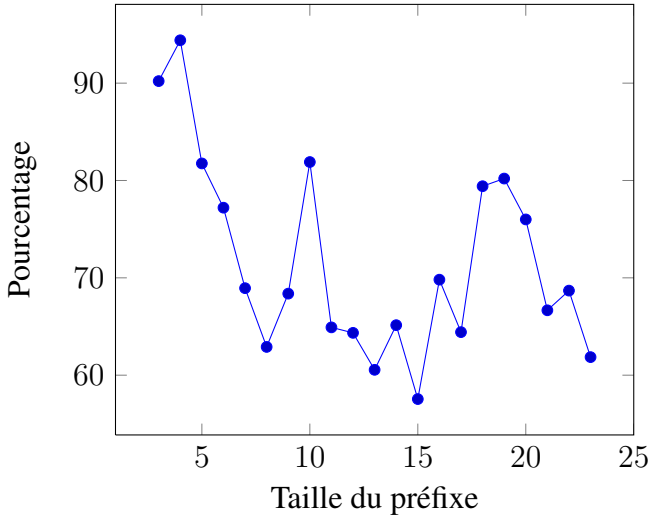


FIGURE 5.2 – Pourcentage de cas ou la $(k+\epsilon)^{me}$ valeur de la série temporelle de test appartient à l'intervalle $I(t+1)$ prédit.

$EQM = \frac{1}{k}\sum_{i=1}^{k}\left(pred(k) - val(k)\right)^2$ *Dans ce qui suit, nous considérons la racine de l'erreur quadratique moyenne donnée par :*

$$REQM = \sqrt{\frac{1}{k}\sum_{i=1}^{k}\left(pred(k) - val(k)\right)^2}$$

Pour calculer la racine de l'erreur quadratique moyenne (REQM) de notre modèle de prédiction à base de programmation par contraintes, nous avons considéré le centre de l'intervalle $I(t+1)$ prédit comme étant la valeur observée, pour chaque prédiction. . La Figure 5.3 présente la REQM de la prédiction pour chaque de taille de préfixe de 3 à 23.

La valeur moyenne de la REQM pour toutes les longueurs de préfixe est d'environ 2800. Ceci est satisfaisant puisque les valeurs des séries temporelles utilisées vont de 3000 à 29000. Cependant, la qualité de la prédiction repose sur la qualité de la classification des séries temporelles en clusters. Cela signifie que les résultats pourraient empirer si de nouvelles séries temporelles ajoutées système ne sont pas correctement

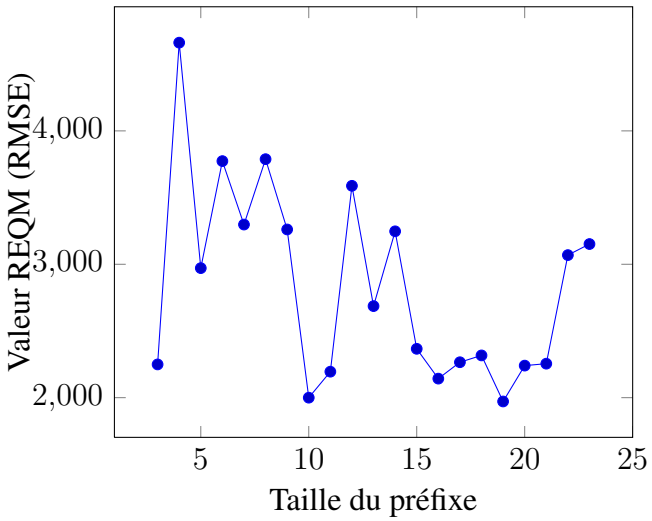


FIGURE 5.3 – Évaluation de la REQM de la prédiction

classées.

Pour surmonter ces problèmes, nous avons élaboré un nouveau modèle basé sur les réseaux de neurones. Le modèle est présenté dans la Section 5.3.3 et ne nécessite pas de phase de clustering.

5.3.3 Apprentissage sur des séries temporelles avec un réseau de neurones

Cette section présente la conception de notre réseau de neurones ainsi que la façon dont il est entrainé pour faire des prédictions sur les valeurs futures des séries temporelles.

L'idée centrale derrière l'entrainement d'un réseau de neurones qui prédit les valeurs futures des séries chronologiques est d'apprendre un mappage qui prend en entrée un préfixe et retourne la prochaine valeur du préfixe.

Étant donné un ensemble de données réelles d'apprentissage de la forme $(X = pref(t), a(X))$ où $pref(t)$ est une série temporelle de taille t et $a(X)$ est la valeur de cette série temporelle au prochain créneau horaire $t+1$, le réseau entrainé doit pouvoir prédire une valeur $y(X)$ proche de $a(X)$. Formellement, ceci est fait en résolvant un problème de minimisation. Soit $C(W,B) = \frac{1}{2n}\sum_X \|y(X) - a(X)\|^2$ une fonction de coût où :

- n Est le nombre de données d'apprentissage.
- W est la matrice des coefficients.
- B la matrice des biais.
- $y(X) = f(WX + B)$ est le résultat en sortie du réseau de neurones avec la fonction d'activation f avec X en entrée.

La fonction C est l'erreur quadratique moyenne. De par sa définition, C devient petit quand $y(X)$ tend vers $a(X)$. L'objectif est donc de trouver les valeurs des matrices W et B qui minimisent C.

Étant donné que l'entrée de notre réseau neuronal est un préfixe, nous avons conçu notre réseau pour être compatible avec la longueur maximale qu'un préfixe puisse prendre dans notre application. Comme nous avons affaire à des séries temporelles de longueur de 24 qui représente la charge de travail quotidienne d'un data center, nous concevons notre réseau avec 24 neurones dans la couche d'entrée. Lorsque nous voulons alimenter le réseau avec une série temporelle de longueur de $t < 24$, nous ajoutons des -1 au préfixe pour obtenir une série temporelle de taille 24. De plus, comme nous voulons que notre réseau prédise une seule valeur pour chaque préfixe, nous mettons un seul neurone en sortie.

Pour calculer le gradient ∇c nous avons utilisé l'algorithme de backpropagation [AON+09], un algorithme couramment utilisé en apprentissage machine.

La section suivante présente l'évaluation de notre modèle de prédiction basé sur le réseau neuronal.

5.3.3.1 Évaluation du modèle à base de réseau de neurones

Pour l'évaluation de ce modèle, nous avons utilisé les mêmes données que dans le cas du modèle de programmation de contraintes (Section 5.3.2.3). Pour entrainer le réseau, nous extrayons des préfixes de longueur de 3 à 23 de chaque série temporelle issue des données d'apprentissage, et alimentons le réseau avec un couple comprenant ces préfixes et la prochaine valeur de la série temporelle. Le nombre de données d'apprentissage est donc artificiellement augmenté car pour chaque série temporelle de taille 24, on extrait 22 différents préfixes de tailles allant de 3 à 23.

Le réseau de neurones a 24 neurones dans sa couche d'entrée, un neurone dans sa couche de sortie et une seule couche cachée. Le nombre de neurones de la couche cachée ainsi que le taux d'apprentissage η ont été fixés empiriquement à 65 et 0.2 respectivement.

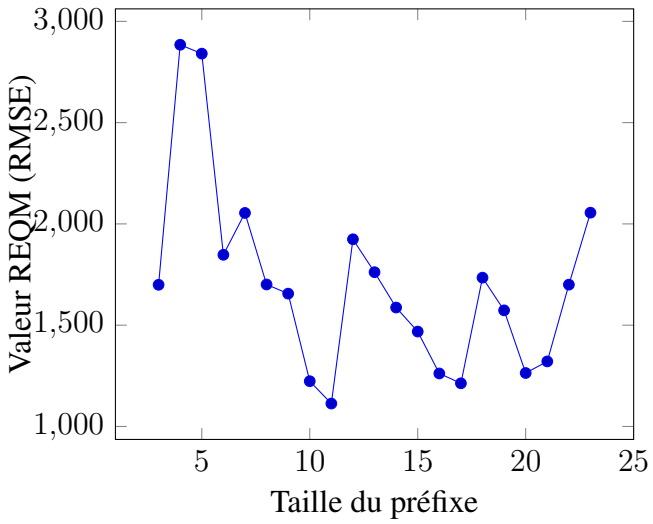


FIGURE 5.4 – REQM de la prédiction avec un réseau de neurones.

La Figure 5.4 présente le REQM de la prédiction en fonction de la taille de préfixe. La qualité de la prédiction avec ce modèle de réseau de neurones est deux fois meilleure que celle de la prédiction avec le modèle à base de contraintes temporelles.

Bien que le modèle basé sur le réseau neuronal donne de meilleurs résultats que le modèle basé sur la programmation par contraintes, les Figures 5.3 et 5.4, montrent que les deux modèles suivent une évolution similaire en fonction de la taille des préfixes. L'explication tient au fait que le réseau neuronal réussisse à apprendre les motifs des séries temporelles d'apprentissage. Les sommets de la courbe de REQM correspondent aux créneaux où plusieurs motifs candidats ont été identifiés. Les creux de la courbe de REQM quant à eux correspondent aux créneaux où le motif courant du préfixe a clairement été identifié. Quand le motif est identifié, la prédiction est plus précise. Ces observations restent vraies dans le cas des sommets et des creux observés dans la courbe de REQM de la prédiction du modèle à base de programmation par contraintes de la Figure 5.3.

Nous rappelons que le but de modèle est de prédire en temps réel la charge de travail d'un data center. En temps réel signifie que la prédiction doit être faite assez rapidement pour pouvoir être exploitée par une heuristique d'optimisation de ressources tel que celle présentée au Chapitre 6. Le temps est donc un paramètre à prendre en considération pour un tel modèle. Par ailleurs nous voulons aussi montrer que le modèle passe bien à l'échelle. Pour évaluer ce passage à l'échelle, nous avons augmenté la résolution des traces, de 24 valeurs par heure à 41 valeurs par heure. Ces benchmarks ont été effectués sur un ordinateur exécutant Mac OS 10.10.5 Yosemite, avec 16 Go de mémoire et un processeur Intel Core i7 à 2,93 GHz.

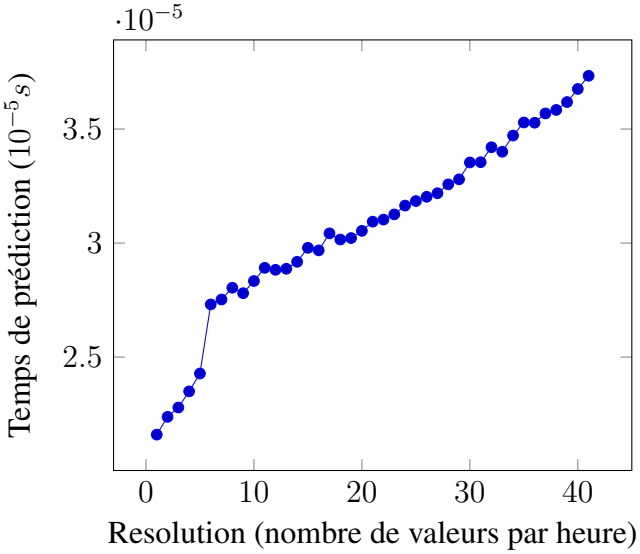


FIGURE 5.5 – Temps de prédiction du réseau de neurones.

A partir de la Figure 5.5, nous pouvons observer qu'avec notre modèle le temps de prédiction est inférieur à $1ms$, et ceux même avec des séries temporelles de 41 valeurs par heure i.e des séries de taille 984. Cette rapidité est due à la complexité linéaire de l'algorithme de propagation en avant utilisée dans un réseau de neurones pour produire un résultat. Le nombre de multiplications nécessaires pour calculer la sortie de la fonction d'activation de chaque neurone est linéaire avec le nombre de neurones. La complexité est donc $\mathcal{O}(k * 65) = \mathcal{O}(k)$ où k est le nombre de neurones de la couche d'entrée, dans notre cas, k est la durée de la série temporelle. Ces résultats contrastent avec le temps nécessaire à l'apprentissage de ce même réseau neuronal.

La Figure 5.6 montre que le temps nécessaire pour entrainer le réseau de neurones va au-delà de une

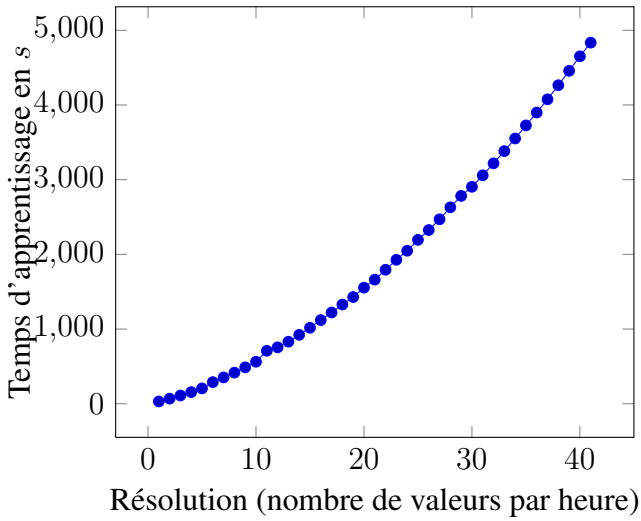


FIGURE 5.6 – Temps d'apprentissage du réseau de neurones.

heure pour les séries temporelles de taille supérieure à 35, i.e des séries de taille supérieure à 875. Ceci est du à la complexité de l'algorithme de backpropagation linéaire au nombre d'étapes d'entrainement, à la taille de l'échantillon d'apprentissage, et au nombre de neurones. Le fait que le temps d'apprentissage aille au delà de l'heure ne représente pas un problème car l'apprentissage n'est pas réalisée en temps réel, l'étape d'apprentissage est réalisée en pré-traitement hors ligne, sans contraintes sur le temps.

Ces résultats conduisent à la section 5.4 sur l'utilisation du modèle de programmation de contraintes pour générer de nouvelles traces de charge de travail.

5.4 Génération de traces de la charge d'un data center

Dans les Sections 5.3.2 et 5.3.3, nous avons présenté et évalué deux modèles de prédiction. D'une part, les résultats ont établi que le modèle avec le réseau de neurones fonctionne mieux que le modèle de programmation des contraintes et est également bien adapté pour la prédiction en temps réel car il ne nécessite aucune classification des données.

Cependant, l'avantage du modèle de programmation de contraintes est qu'il apprend un ensemble de contraintes qui caractérisent les séries temporelles de chaque cluster. Ce même modèle peut alors être utilisé pour générer des séries temporelles compatibles avec un cluster donné. Les données réelles sur la charge de travail d'un data center ne sont généralement pas disponibles en grande quantité. Un tel générateur est très utile car il peut générer autant de données que nécessaire. Les données générées ont les mêmes caractéristiques que les données réelles d'apprentissage. Les deux modèles sont donc complémentaires.

Cette section décrit comment de nouvelles traces peuvent être générées en deux étapes à partir d'une poignée de données réelles. La première étape construit un *problème de satisfaction de contraintes (CSP)* à partir des données réelles disponibles, et la seconde étape résout ce problème pour générer de nouvelles traces.

5.4.1 Construction du CSP et génération de nouvelles traces

Soit $P = (X, D, C)$ le CSP à construire pour générer de nouvelles traces. On définit X, D et C comme suit :

- [L'ensemble des variables X] Nous fixons le nombre n de variables à la taille des traces disponibles. Dans notre cas $n = 24$. i.e $X = \langle x_0, x_1, \ldots, x_{23} \rangle$.

- [L'ensemble des domaines D] Soit m la plus petite valeur et M la plus grande valeur prise par les traces réelles de l'ensemble d'apprentissage. m et M sont tels que, pour toute série temporelle ts issue de tout cluster, et pour tout index $t \leq 23$, la t^{me} valeur de ts est inclus dans $[m, M]$. $D = \langle D_0, D_1, \ldots, D_{23} \rangle$ où $D_0 = D_1 = \cdots = D_{23} = [m, M]$.

- [L'ensemble des contraintes C] Dans la Section 5.3.2 nous avons présenté la construction d'un modèle de contraintes capturant les caractéristiques des traces de charge de travail. Nous rappelons que ledit modèle $m(cl)$ d'un cluster cl est de la forme :

```
m(cl_i) = {
range(3, peak,nb_peak,2,5),
range(3, peak,sum_width_peak,5,8).}
```

  Pour faciliter la lecture, cet exemple de modèle ne considère que deux attributs d'un seul motif. Ce modèle, nous donne les informations suivantes :

  - Toutes les séries temporelles du cluster cl comprennent au moins 2 et au plus 5 peaks, de sorte que l'amplitude de variation du nombre de pics est de 3.
  - En additionnant les longueurs de tous les pics présents dans chaque série temporelle de cl, on obtient une valeur comprise entre 5 et 8.

  Nous traduisons cette information en contraintes, notre problème comporte donc deux contraintes sur l'ensemble des variables :

  - $C_0 = \langle R_0, X \rangle$ avec $nb\_peak(R_0, \langle x_0, x_1, \ldots, x_{23} \rangle)$ et $R_0 \in [2, 5]$,
  - $C_1 = \langle R_1, X \rangle$
    avec $sum\_width\_peak(R_1, \langle x_0, x_1, \ldots, x_{23} \rangle)$ et $R_1 \in [5, 8]$.

En utilisant un solveur de contraintes, on trouve les solutions $S_i = \langle v_{i,0}, v_{i,0}, ..., v_{i,23} \rangle$ du CSP C. Chaque solution est une série temporelle de taille 24 ayant les mêmes caractéristiques et motifs appris des traces réelles.

La section suivante présente une évaluation de la qualité des séries temporelles obtenues par ce modèle.

5.4.2 Évaluation du générateur de traces à base de CSP

Pour ces benchmarks, nous avons utilisé SICStus Prolog Solver [CWA+88], sur un ordinateur tournant sous Mac OS 10.10.5 Yosemite, avec 16G0 de mémoire et un processeur Intel Core i7 à 2.93 GHz.

Nous avons évalué le temps nécessaire au CSP pour générer de nouvelles séries chronologiques en fonction de la taille de la série chronologique. La Figure 5.7 montre que pour toute résolution de 1 à 41 valeurs par heure, le temps nécessaire pour générer une série temporelle de taille correspondante, soit de taille allant de 24 jusqu'à 984 est inférieur à une seconde. Cette bonne performance s'explique par le paradigme de propagation de contraintes [Bes06a] qui est utilisé pour résoudre le modèle CSP et générer des solutions.

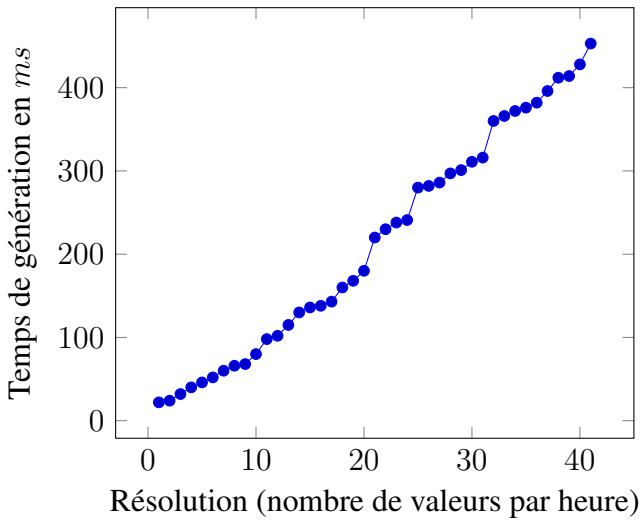


FIGURE 5.7 – Temps de génération de nouvelles séries temporelles à partir du CSP

5.4.3 Générateur de traces à base de réseau de neurones

Nous avons aussi utilisé le réseau de neurones pour générer de nouvelles séries temporelles. Pour ce faire, nous partons d'un petit préfixe, c'est à dire un préfixe de taille 3 issu d'une trace réelle. Par la suite on utilise le réseau de neurones pour prédire la prochaine valeur, on obtient un préfixe de taille 4 et on répète l'opération jusqu'à obtenir une série temporelle de taille désirée. Ce processus est présenté par l'Algorithm 8

Comme prévu, aucune des séries temporelles générées avec le réseau de neurones ne respecte toutes les caractéristiques apprises. Un autre inconvénient dans la génération de séries temporelles avec le réseau de neurones est qu'avec ce dernier, il faut obligatoirement commencer par un petit préfixe non vide qui est par la suite complété de proche en proche pour former une nouvelle série temporelle. Le modèle CSP quant à lui peut rapidement et à partir de zéro, générer des séries de temporelles entières qui vérifient toutes les contraintes apprises.

Algorithm 8 Génération de nouvelles séries temporelles à partir du réseau de neurones.

$L_i \leftarrow Ensemble\ d'apprentissage$
$T_i \leftarrow Ensemble\ de\ test$
$n_i \leftarrow Rseau\ de; neurones$
$S_i \leftarrow Ensemble\ de\ sries\ temporelles\ gnres$
Taille désirée : N // Chaque série temporelle est de taille N
$n_i.train(L_i)$ // Entrainer le réseau de neurone
Pour $st \in T_i$ **Faire**
  $p \leftarrow pref(st, 3)$ // p est le préfixe de taille 3 issue de st
  TantQue $length(p) < N$ **Faire**
    $y \leftarrow n.predict(p)$ // Prédiction de la prochaine valeur
    $p \leftarrow p \bigoplus y$ // Concaténation de y à p
  Fin TantQue
  $S_i \leftarrow p$
Fin Pour

5.5 Conclusion

Dans ce chapitre, nous avons proposé deux modèles de prédiction de charge de travail pour les infrastructures Cloud. Ces deux modèles, respectivement basés sur la programmation par contraintes et les réseaux de neurones, font de la prédiction sur l'utilisation de la charge CPU des serveurs physiques dans un data center Cloud. Nous fournissons également un générateur de trace efficace basé sur la programmation par contraintes et ne nécessitant qu'une petite quantité de traces réelles. Un tel générateur peut résoudre les problèmes de disponibilité des traces de charge de travail réelles nécessaires pour la validation d'heuristiques d'optimisation. Tandis que les réseaux neuronaux présentent des capacités de prédiction plus élevées, les techniques de programmation des contraintes semblent plus adaptées à la génération de traces, ce qui rend les deux techniques complémentaires. Notre travail futur comprend la fourniture d'un site web pour accéder à des ensembles de données à la demande, produits par notre générateur utilisant diverses charges de travail réelles qui ne peuvent être rendues directement accessibles au public. Les résultats de ce chapitre font partie d'une publication au journal Computing [BFG+16].

6

Planification énergiquement écologique.

Sommaire

6.1 Introduction et travaux connexes

Au fil des années, la consommation d'énergie est devenue une préoccupation majeure dans le domaine des technologies de l'information. Amazon rapporte que les coûts liés à la consommation d'énergie sur une période de 3 ans sont supérieurs à 40% du coût global de ses data center [Ham09]. Dans [Bar05], Barrosso montre que, au cours de la durée de vie d'un data center, les dépenses liées à la consommation d'énergie peuvent facilement dépasser le coût du matériel. Dans la littérature, la plupart des travaux qui s'attellent à réduire les coûts énergétiques d'un data center proposent une réduction de la consommation globale d'énergie [CMOS13b], [CMOS13a]. Dans ce chapitre, nous abordons le problème avec une vision différente. Outre l'aspect économique, la consommation massive d'énergie a également une répercussion sur l'environnement, car l'énergie principalement utilisée est de type fossile et provient de sources polluantes. Nous proposons non seulement de réduire la consommation d'énergie pour aborder le problème économique, mais aussi de maximiser la consommation d'énergie verte pour faire face au problème environnemental.

La consommation d'énergie d'un système comprend une partie statique et une partie dynamique [OAL14]. La partie statique est liée à la taille du système et au type de matériel utilisé tandis que la partie dynamique est elle liée à l'utilisation des ressources. Ce chapitre se concentre sur la partie dynamique. Dans [LOM15], Li et al. proposent un cadriciel novateur appelé oPportunistic schedullIng broKer infrAstructure (PIKA). Le cadriciel PIKA permet de réduire la consommation d'énergie fossile d'un data center de petite ou de moyenne taille. Le cadriciel PIKA est composé de 9 modules.

- **User** : C'est le point d'entrée, il fournit la liste des jobs à exécuter.
- **Predictor** : Il prédit la quantité d'énergie verte disponible à court terme,
- **Broker** : Il est responsable de la mise en route ou de l'extinction dynamique des machines physiques en fonction de la quantité d'énergie verte.
- **Gap** : Il détermine le nombre de machines physiques à allumer ou à éteindre en fonction de la quantité d'énergie verte
- **Job pool** : C'est la structure dans laquelle sont placés les jobs en fonction de leurs natures.
- **Waiting queue** : C'est a queue dans laquelle sont placés les job en attente d'exécution.
- **Consolidation decision** : Ce module est responsable de la consolidation des machines virtuelles.
- **Opportunistic scheduling** : Ce module est responsable du calcul des placements, migration, interruptions des jobs.
- **PM pool** : C'est l'ensemble des machines physique du data center.

En fonction de la quantité d'énergie renouvelable disponible à chaque créneaux horaire, le module *Gap* détermine le nombre de machines physiques à allumer ou à éteindre. Le module *Opportunistic scheduling* quant à lui planifie les applications qui peuvent être exécutées et reporte celles qui ne peuvent pas. Ce module est aussi responsable de la migration des applications d'une machine physique à une autre. Cependant, cette planification opportuniste est plus réactive que pro-active.

Dans ce chapitre, nous formalisons dans un premier temps le problème de planification énergiquement écologique. Ensuite nous proposons un modèle global de programmation par contraintes ainsi qu'une heuristique de recherche pour optimiser de façon efficace la consommation d'énergie d'un data center. Le modèle proposé prend en compte de manière intégrée chacun des aspects suivants :

- Prise en compte des applications par lots et des applications interactives,
- Prise en compte des applications dont les besoins en ressources varient dans le temps,
- Migration des applications d'un serveur physique à un autre,
- Coûts énergétiques des migrations d'applications,
- Pause/Reprise des applications par lots,
- compatibilité avec un data center hétérogène i.e. dont les ressources varient d'un serveur physique à un autre,
- Mise en route et extinction des serveurs physiques,
- Consommation énergétique supplémentaire lors de la mise en route et de l'extinction des serveurs physiques,
- Gestions des ressources des serveurs physiques,
- Prise en compte de la disponibilité de l'énergie verte.

Les principaux avantages de la programmation par contraintes incluent un certain niveau de flexibilité qui fait défaut à PIKA. Grace à cette caractéristique, le modèle proposé est aussi bien compatible avec des contraintes métier inhérentes à la gestion des infrastructures de data center, qu'avec des contraintes de préférences utilisateur. La Figure 6.1 illustre ces différentes facettes du contexte.

Enfin, nous montrons que notre modèle optimise de façon significative la consommation énergétique globale d'un data center. Le reste du chapitre est organisé comme suit : la Section 6.2 décrit le problème et donne les définitions nécessaires. La Section 6.3 formalise le PPEE et présente notre modèle. La Section 6.4 présente l'heuristique de recherche qui permet de trouver efficacement de bonnes solution au PPEE. En fin la Section 6.5 évalue notre contribution en utilisant des traces réelles de charge de travail d'un data center ainsi que de disponibilité d'énergie verte. La Section 6.6 conclue ce travail.

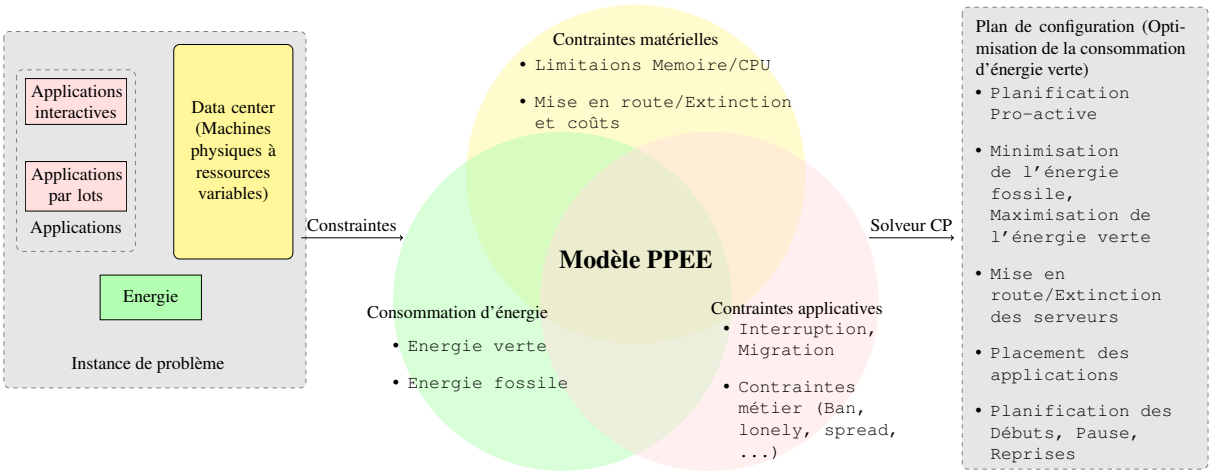


FIGURE 6.1 – Le modèle PPEE prend simultanément en compte les différents aspects du problème et calcule un plan de configuration global qui minimise la consommation d'énergie fossile en maximisant la consommation d'énergie verte du data center.

6.2 Description du problème et définitions

Nous considérons le problème de placement d'applications au sein d'un petit/moyen data center à ressources limitées sur un horizon temporel fixé. Nous présentons dans [BFG+15] le type de data center cible. Le placement des applications réalisé doit maximiser la consommation d'énergies renouvelables. Nous rappelons qu'un data center est constitué de un ou plusieurs serveurs physiques qui hébergent les applications. Nous dans ce travail, nous considérons deux types d'applications que nous introduisons à la Section 6.2.1.

6.2.1 Définitions

Définition 16 (Applications active)**.** *Une application active est une application exécutée tout au long de l'horizon temporel sans interruption. Chaque application active est décrite par :*

- *Sa consommation Mémoire. C'est la quantité constante de mémoire consommée par l'application dans le serveur qui l'héberge.*
- *Son usage CPU. C'est la puissance de traitement réquisitionnée par un serveur pour exécuter l'application. Cette puissance peut varier d'un créneau horaire à l'autre tout au long de l'horizon temporel.*
- *Son coût de migration. C'est la consommation d'énergie supplémentaire nécessaire pour migrer une application d'un serveur à un autre.*

Une application active de consommation mémoire $\mathbf{mem_i}$*, d'usage CPU* $\mathbf{cpu_i}$ *et ayant un coût de migration* $\mathbf{migr_i}$ *est dénotée* $\mathbf{a_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{migr_i})$.

Définition 17 (Application par lots)**.** *Une application par lots est une application dont l'exécution peut être interrompue, puis reprise. Un créneau horaire pendant lequel une application par lots s'exécute est appelé créneau d'exécution. Chaque application par lots est caractérisée par :*

- *Sa consommation mémoire. La consommation mémoire d'une application par lots est constante.*
- *Son usage CPU. L'usage CPU d'une application par lots est constante.*
- *Sa durée.*
- *Son coût de migration.*
- *Son Slack. Le temps total pendant lequel une application par lots est interrompue ne doit pas dépasser son slack.*

Une application par lots de consommation mémoire $\mathbf{mem_i}$*, d'usage CPU* $\mathbf{cpu_i}$*, de durée* d_i*, ayant un coût de migration* $\mathbf{migr_i}$ *et un slack* s_i *est dénotée* $\mathbf{b_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{d_i}, \mathbf{migr_i}, \mathbf{s_i})$.

Définition 18 (Serveur)**.** *Nous définissons un serveur dans ce problème comme étant une machine qui héberge les applications. Il se caractérise par :*

- *Sa limite mémoire. C'est la quantité totale de mémoire qui est partagée entre les applications hébergées.*
- *Sa limite CPU. C'est la quantité totale de puissance de calcul qui est partagée entre les applications hébergées.*
- *Son énergie d'allumage (respectivement d'extinction). C'est la quantité d'énergie requise pour allumer (respectivement éteindre) le serveur.*
- *Sa relation de charge/consommation qui relie la charge CPU du serveur à sa consommation d'énergie.*
- *Son id. C'est un nombre entier unique qui identifie chaque serveur.*

Un serveur de mémoire $\mathbf{MEM_i}$*, de limite CPU* $\mathbf{CPU_i}$*, ayant une énergie d'allumage* $\mathbf{E_{on_i}}$*, une énergie d'extinction* $\mathbf{E_{off_i}}$*, une relation de charge / consommation* r_i *et un id* id_i *est dénotée* $\mathbf{s_i}(\mathbf{MEM_i}, \mathbf{CPU_i}, \mathbf{E_{on_i}}, \mathbf{E_{off_i}}, \mathbf{r_i}, \mathbf{id_i})$.

L'objectif est de calculer un plan de configuration énergiquement écologique pour le centre de données. La prochaine définition introduit la notion de *plan de configuration*.

Définition 19 (plan de configuration)**.** *Étant donné un horizon temporel, un ensemble d'applications interactives, un ensemble d'applications lots et un data center constitué de un ou plusieurs serveurs physiques, un plan de configuration est une planification répondant aux exigences suivantes :*

- *Il spécifie pour chaque application active le serveur sûr lequel il est exécuté à chaque instant de l'horizon temporel. Une migration se produit lorsque l'hôte d'une application active change d'un créneau horaire au suivant.*
- *Il spécifie pour chaque application par lots les dates de début, de fin, d'interruption, et de reprise. Le plan de configuration doit aussi spécifier pour chaque application par lots le serveur dans lequel il s'est exécuté à chaque créneau. Une migration se produit lorsque l'hôte d'une application par lots change d'un créneau d'exécution au suivant.*
- *Pour chaque serveur du data center, le plan de configuration spécifie les différentes dates de mise en route ou d'extinction.*

Le plan de configuration est énergiquement efficace lorsqu'il maximise la consommation d'énergie verte du centre de données. En effet, la disponibilité de l'énergie verte fluctue au fil du temps. Un exemple typique est l'énergie solaire qui est plus disponible pendant les périodes ensoleillées.

La Section 6.3 formalise ce problème en un problème de programmation par contraintes.

6.3 Modélisation du problème

Dans cette section, nous proposons un modèle à base de PPC pour résoudre le problème de planification énergiquement écologique au sein de data centers virtualisés. Avant de détailler les différents aspects du modèle, nous motivons l'usage de la PPC pour un tel problème.

6.3.1 La programmation par contraintes pour le problème de planification énergiquement écologique au sein d'un data center virtualisé.

La programmation par contraintes est une solution appropriée pour résoudre la plupart des problèmes de planification [BLPN12]. Dans le cas d'un problème de planification énergiquement écologique, la programmation par contraintes est bien adaptée car elle fournit une bonne expressivité pour modéliser chaque composant du problème. La programmation par contraintes est assez flexible. Cette flexibilité permet d'ajouter toute nouvelle contrainte à un modèle existant. Dans [HDL11], Hermenier et al. présente $quatre$ contraintes latérales qui peuvent être requises par l'administrateur d'un système ou par les utilisateurs pour restreindre le placement des applications dans les serveurs.

- **Ban**. A un instant t donné, l'administrateur du système pourrait avoir besoin d'éteindre un serveur pour exécuter des opérations de maintenance. La contrainte *Ban* permet de modéliser ce besoin.
- **Spread**. Pour être robuste en termes de tolérance aux pannes matérielles, une application peut utiliser la réplication. L'application et sa (ou ses) réplication(s) doivent alors être exécutées sur des serveurs différents.
- **Lonely**. Pour certaines raisons et sous certaines circonstances, une application pourrait nécessiter d'être exécutée à elle toute seule sur un serveur.
- **Capacity**. Parfois, en raison de certaines ressources partagées, il peut être utile de limiter le nombre d'applications exécutées sûr un même serveur. La contrainte de capacité est utilisée pour s'assurer que le nombre d'applications hébergées sur un serveur donné est inférieur à une limite donnée.

La liste de ces contraintes latérales est non exhaustive. Outre de la bonne expressivité et la flexibilité, le processus de résolution d'un problème en PPC par des algorithmes de propagation de contraintes [RVBW06] fait en sorte que toutes les contraintes du problème soient satisfaites. Nous formalisons à présent la définition du problème de planification énergiquement écologique.

Définition 20 (Problème de planification énergiquement écologique)**.** *Une instance du problème de planification énergiquement écologique (PPEE) est un tuple* $P = \{S, A, B, T, E\}$ *où :*

- S *est un ensemble de serveurs.*
- A *est un ensemble d'applications interactives.*
- B *est un ensemble d'applications par lots.*
- T *est l'horizon temporel. C'est un entier qui spécifie le nombre de créneaux horaires à considérer.*
- E *est une série temporelle de longueur* T *qui spécifie la quantité d'énergie renouvelable disponible à chaque créneau horaire.*

Dans les Sections 6.3.2 à 6.3.6 nous présentons la formalisation de chaque aspect d'un ***PPEE*** *sous forme d'un problème de PPC pour lequel nous recherchons des solutions. Nous utiliserons l'Exemple illustratif 15 pour couvrir les différents aspects de notre modèle.*

Exemple 15. *Tout au long de ce chapitre, nous utiliserons l'exemple suivant de PPEE.* $P = \{S, A, B, T, E\}$ *où :*

- $S = \{s_0(3, 50, 3, 3, r, 0), s_1(3, 50, 3, 3, r, 0)\}$ *où* $r(cpu\_load) = 10 + cpu\_load$
- $A = \{a_o(1, 20, 2)\}$
- $B = \{b_0(2, 20, 3, 1, 5), b_1(2, 20, 3, 1, 5)\}$

- $T = 10h$
- $E = 10, 10, 10, 90, 90, 10, 10, 10, 48, 43$

Un plan de plan de configuration énergiquement écologique pour ce PPEE est donné ci-dessous. Ce plan spécifie pour chaque créneau horaire de l'horizon toutes les actions réalisées.

- ***Créneau*** 0 : *Le serveur s_0 est mis en route et l'application interactive a_0 est lancée sur ce serveur.*
- ***Créneau*** 1 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0.*
- ***Créneau*** 2 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0.*
- ***Créneau*** 3 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0, l'application par lots b_0 est lancée sur s_0. Le serveur s_1 est mis en route et l'application par lots b_1 est lancée sur ce serveur.*
- ***Créneau*** 4 : *L'application interactive a_0 et l'application par lots b_0 sont toujours en exécution sur s_0 tandis que l'application par lots b_1 est en cours d'exécution sur s_1.*
- ***Créneau*** 5 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0. Les deux applications par lots b_0 et b_1 sont interrompues et le serveur s_1 est éteint.*
- ***Créneau*** 6 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0.*
- ***Créneau*** 7 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0.*
- ***Créneau*** 8 : *L'application interactive a_0 est toujours en cours d'exécution sur s_0 et l'application par lots b_0 est reprise sur s_0.*
- ***Créneau*** 9 : *L'application interactive a_0 est toujours en exécution sur s_0 et l'application par lots b_1 est migrée de s_1 à s_0 et reprise.*

6.3.2 Mise en route et extinction d'un serveur

Pour modéliser le fait de mettre un serveur en route ou bien de l'éteindre, nous devons considérer deux aspects. Premièrement, nous devons prendre en compte le coût énergétique inhérent à chacune de ces deux actions, et deuxièmement nous devons nous assurer qu'aucune application ne soit planifiée sur un serveur éteint. Ces deux besoins se traduisent par les deux contraintes suivantes.

6.3.2.1 coût energétique de mise en route et d'extinction

Comme on le voit dans la définition d'un serveur à la Section 6.2.1, chaque serveur dispose d'un coût énergétique de mise en route et d'extinction. Soit $P = \{S, A, B, T, E\}$ un PPEE. Pour chaque serveur $\mathbf{S_i}(\mathbf{MEM_i}, \mathbf{CPU_i}, \mathbf{E_{on_i}}, \mathbf{E_{off_i}}, \mathbf{r_i}, \mathbf{id_i}) \in S$, nous utilisons un ensemble de $T + 1$ variables booléennes $ON_{(i,t)}$ indiquant à chaque créneau horaire $t, t \in [-1, T-1]$ si le serveur est allumé ou éteint. $ON_{(i,-1)} = false$ indique que le serveur S_i est initialement éteint.

Allumer ou éteindre un serveur au créneau t consomme une certaine quantité d'énergie supplémentaire. Pour chaque serveur $\mathbf{S_i}(\mathbf{MEM_i}, \mathbf{CPU_i}, \mathbf{E_{on_i}}, \mathbf{E_{off_i}}, \mathbf{r_i}, \mathbf{id_i})$, la modélisation de ces coûts énergétiques supplémentaires repose sur T variables de type "tâche de mise en route". Avant de plus détailler ces variables, nous introduisons d'abord la notion de tâche.

Définition 21 (Task)**.** *Une tâche est un tuple* $task_i(s_i, d_i, e_i, < r_i >, h_i)$ *où :*

- s_i *est le début de la tâche,*
- d_i *est la durée de la tâche,*
- e_i *est la fin de la tâche,*
- $< r_i >$ *est la liste des ressources requises par la tâche,*
- h_i *est le serveur hébergeant la tâche.*

Pour un serveur $\mathbf{s_i}(\mathbf{MEM_i}, \mathbf{CPU_i}, \mathbf{E_{on_i}}, \mathbf{E_{off_i}}, \mathbf{r_i}, \mathbf{id_i})$, la consommation énergétique de sa tâche de mise en route au créneau t vaut :

- E_{on_i} si on met en route s_i au créneau t.
- E_{off_i} si on éteint s_i au créneau t.
- 0 si aucune action de mise en route ou d'extinction n'est effectuée au créneau t.

Cette consommation électrique est donnée par la variable $switch\_energy_{(i,t)}$ suivant la formule suivante :

$$switch\_energy_{(i,t)} = E_{on_i} * (ON_{(i,t-1)} < ON_{(i,t)}) + E_{off_i} * (ON_{(i,t-1)} > ON_{(i,t)}). \quad (6.1)$$

où un terme tel que $ON_{(i,t-1)} > ON_{(i,t)}$ est interprété comme une variable en $0 - 1$ qui prenant la valeur 1 (resp. 0) si la condition correspondante est satisfaite (resp. n'est pas satisfaite).

Exemple 16. *D'après le plan de configuration donné à l'Exemple 15 on a :*

- $switch\_energy_{0,0} = 3$ *car le serveur* s_0 *est allumé à* $t = 0$,

- $switch\_energy_{1,3} = 3$ *car le serveur* s_1 *est allumé à* $t = 3$,
- $switch\_energy_{1,5} = 3$ *car le serveur* s_1 *est éteint à* $t = 5$,
- $switch\_energy_{0,t} = 0$ *pour tout* $t \neq 0$ *et* $switch\_energy_{1,t} = 0$ *pour tout* $t \in \{0, 1, 2, 4, 6, 7, 8, 9\}$

6.3.2.2 Disponibilité d'un serveur

Du fait que chaque application consomme une quantité non nulle de mémoire sur son serveur hôte, il est alors impossible de placer une application sur un serveur qui ne dispose pas de la mémoire suffisante. Donc pour rendre un serveur indisponible, il suffit de remplir artificiellement toute sa mémoire de façon à avoir 0 mémoire de disponible. Pour ce faire, nous créons des tâches de mémoire dont le rôle se limite à remplir la mémoire du serveur hôte quand ce dernier est éteint. Pour éviter d'avoir à créer T tâches supplémentaire pour accomplir cette mission, nous ajoutons simplement une ressource de type mémoire aux tâches de mise en route introduites à la Section 6.3.2.1. La consommation mémoire $m_{i,t}$ d'une tâche de mise en route $\mathbf{switch_{(i,t)}}$ d'un serveur $s_i(MEM_i, CPU_i, E_{on_i}, E_{off_i}, r_i, id_i)$ au créneau t est donnée par :

$$mem_{i,t} = MEM_i * (ON_{(i,t)} = 0). \tag{6.2}$$

Les tâches de mise en route du modèle sont dès lors indexées par l'index i du serveur et par le créneau horaire t, on les note par $\mathbf{switch_{(i,t)}}(\mathbf{t}, \mathbf{1}, \mathbf{t+1}, < \mathbf{energy_{i,t}}, \mathbf{mem_{i,t}} >, \mathbf{i})$.

La Section 6.3.3 ci-après présente comment modéliser les applications actives et les applications par lots en utilisant la notion de tâche introduite à la Définition 21.

6.3.3 Modélisation des applications

La partie la plus difficile d'un PPEE est le modèle des applications. Le modèle doit tenir compte des exigences suivantes :

- [L'hôte]. A chaque créneau t, chaque application active et chaque application par lots non interrompu devrait être placée sur un serveur hôte.
- [Migration]. Les applications actives et par lots peuvent être déplacées d'un serveur à un autre. Ces migrations ont un coût énergétique qui doit être pris en considération. Nous présentons en détail la prise en compte de ces coûts de migration à la Section 6.3.4.
- Les applications interactives ne peuvent pas être interrompues.
- Les applications par lots sont interruptibles des lors que l'interruption ne viole pas la contrainte de slack i.e tant que la somme des interruptions ne dépasse pas une constante donnée.

A fin de répondre à ces exigences, nous modélisons chaque application active $\mathbf{a_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{migr_i})$ (resp. chaque application par lots $\mathbf{b_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{d_i}, \mathbf{migr_i}, \mathbf{s_i})$) par un ensemble S_{a_i} (resp. S_{b_i}) de tâches.

6.3.3.1 Applications interactives

Du fait que les applications interactives soient non interruptibles, chaque application active $\mathbf{a_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{migr_i})$ est modélisée par l'ensemble S_{a_i} de T tâches dénoté $\mathbf{sub_{(a_i,t)}}(\mathbf{t}, \mathbf{1}, \mathbf{t+1}, < \mathbf{mem_{i,t}}, \mathbf{cpu_{i,t}}, \mathbf{energy_{i,t}} >, \mathbf{h_{i,t}})$, avec t de 0 à $T-1$ où :

- $mem_{i,t}$ (resp. $cpu_{i,t}$) est la consommation mémoire (resp. l'usage CPU) de la tâche $\mathbf{a_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{migr_i})$ au créneau t.
- $h_{i,t}$ est le serveur qui héberge $\mathbf{a_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{migr_i})$ au créneau t.
- $energy_{i,t}$ est la quantité d'énergie supplémentaire consommée par le serveur h pour exécuter $sub_{(a_i,t)}$.

6.3.3.2 Applications par lots

Du fait que la durée totale des applications par lots soit fixée, chaque application par lots $\mathbf{b_i}(\mathbf{mem_i}, \mathbf{cpu_i}, \mathbf{d_i}, \mathbf{migr_i}, \mathbf{s_i})$ est modélisée par un ensemble S_{b_i} de d_i tâches dénoté $\mathbf{sub_{(b_i,j)}}(\mathbf{s_{(b_i,j)}}, \mathbf{1}, \mathbf{e_{(b_i,j)}}, < \mathbf{mem_{i,j}}, \mathbf{cpu_{i,j}}, \mathbf{energy_{i,j}} >, \mathbf{h_{i,j}})$. Où $j \in [0, d-1]$. Comme les applications par lots peuvent être interrompues, $s_{(b_i,j)}$ et $e_{(b_i,j)}$ sont des variables dont le domaine est $[0, T-1]$ sous les contraintes suivantes :

$$\forall j \in [1, d_i - 1], s_{(b_i,j)} \leq e_{(b_i,j-1)} \tag{6.3}$$

La contrainte de slack est posée comme suit :

$$e_{(b_i,d_i-1)} - s_{(b_i,0)} - d_i \leq slack. \tag{6.4}$$

Pour les deux types d'applications, la variable $energy_{i,t}$ spécifie la quantité d'énergie additionnelle dont a besoin un serveur pour l'exécution. Sa valeur change d'un serveur à l'autre. Les détails sur l'instanciation de sa valeur sont données à la Section 6.3.6.

Pour garantir qu'un serveur sur lequel est planifié une tâche au créneau t soit en marche à cet instant, nous ajoutons les contraintes suivantes pour chaque tâche active et chaque tâche par lots.

$$\forall sub_{(a_i,t)}(t, 1, t+1, < mem_{i,t}, cpu_{i,t}, energy_{i,t} >, h_{i,t}) \in S_{a_i}, h_{i,t} = s \Rightarrow ON_{(s,t)} = 1 \quad (6.5)$$

$$\forall sub_{(b_i,j)}(j, 1, j+1, < mem_{i,j}, cpu_{i,j}, energy_{i,j} >, h_{i,j}) \in S_{b_i}, h_{i,j} = s \Rightarrow ON_{(s,t)} = 1 \quad (6.6)$$

6.3.4 Coûts de migration

Cette section présente la modélisation des coûts de migration des applications interactives et par lots.

Pour toute application active $\mathbf{a_i(mem_i, cpu_i, migr_i)}$ et toute tâche $\mathbf{sub_{(a_i,t)}(t, 1, t+1, < mem_{i,t}, cpu_{i,t}, energy_{i,t} >, h_{(i,t)})} \in S_{a_i}$, nous créons une tâche de migration $\mathbf{migr_{(a_i,t)}(t, 1, t+1, < migr\_energy_{i,t} >, h_{(i,t)})}$ où $migr\_energy_{i,t}$ est la quantité d'énergie supplémentaire consommée en cas de migration de l'application active a_i d'un serveur à un autre au créneau t. sa valeur est donnée par :

$$migr\_energy_{i,t} = \begin{cases} 0 & t = 0 \\ migr_i * (h_{(i,t)} \neq h_{(i,t-1)}) & t > 0 \end{cases}$$

De façon similaire, nous créons des tâches de migration $\mathbf{migr_{(b_i,j)}}$ pour les applications par lots.

Exemple 17. *Avec le plan de configuration donné à l'Exemple 15 on a :*

- $migr\_energy_{b_1,9} = 1$ *car l'application par lots* b_1 *est migrée du serveur* s_1*au serveur* s_0 *au créneau* 9.

6.3.5 Cosommation mémoire et usage CPU

Chaque serveur dispose des ressources mémoire et CPU limitées. Une application peut être planifiée sur un serveur $s_i \in S$ au créneau t si et seulement si le serveur s_i dispose d'assez de ressources à cet instant pour satisfaire les besoin de l'application en question. Pour prendre en considération ces contraintes nous introduisons la contrainte *cumulative*. [ABD+16] [LBC12].

Définition 22 (Contrainte cumulative)**.** *Étant donné un ensemble de tâches à ressources, la contrainte cumulative garantie qu'à chaque instant* t*, la valeur cumulée des besoins en ressources des tâches planifiées à* t *n'excède pas une limite donnée.*

La définition standard de la contrainte cumulative suppose un seul hôte à ressources limitées pour les tâches. Nous utiliserons plutôt la contrainte *cumulatives* [BC02], une généralisation de la contrainte cumulative qui suppose un ou plusieurs hôtes à ressources limitées et variables d'un hôte à l'autre. i.e. un ou plusieurs serveurs à capacités pouvant être différentes. La signature de la contrainte *cumulatives* est :

`cumulatives(Tasks,Machines)`

où :

- $Tasks$ est un ensemble de tâches mono-ressource de la forme $task(start, duration, end, ressource, host)$.
- $Machines$ est un ensemble d'hôtes à ressource limitée de la forme $machine(id, resource\_limit)$.

L'hôte d'une tâche est l'identifiant du serveur sur lequel elle est planifiée. Avant d'entrer dans les détails de l'utilisation de la contrainte *cumulatives* dans le cas de la consommation mémoire et de l'usage CPU, nous introduisons d'abord la notion de *restriction d'une tâche sur une ressource*.

Définition 23 (Restriction d'une tâche)**.** *La restriction d'une tâche* $t(start, duration, end, < r_1, r_2, \dots r_n >, host)$ *à la ressource* $r \in < r_1, r_2, \dots r_n >$ *une tâche donnée par* $t\_r(start, duration, end, r, host)$.

Pour gérer les ressources mémoire et CPU, nous utiliserons respectivement les contraintes *cumulatives*, $cumulatives(Memory\_Tasks, Memory\_Machines)$ et $cumulatives(CPU\_Tasks, CPU\_Machines)$ où :

- $Memory\_Tasks$ est l'ensemble formé par les tâches interactives, les tâches par lots et les tâches de mise en route restreintes sur la ressource mémoire.
- CPU_Tasks est l'ensemble formé par les tâches interactives, les tâches par lots, les tâches de mise en route restreintes sur la ressource CPU ainsi qu'un ensemble tâches complémentaires utilisées pour le calcul de la consommation énergétique de chaque serveur à chaque créneau. Les détails sur ces tâches complémentaires ainsi que leur usage pour le calcul de la consommation énergétique sont décrits dans la Section 6.3.6
- $Memory\_Machines$ et $CPU\_Machines$ sont deux ensembles de machines. Ils sont construits comme suit :

  1 $Memory\_Machines = \{machine\_mem(id_i, MEM_i)$ tel que $\exists s_i(MEM_i, CPU_i, E_{on_i}, E_{off_i}, r_i, id_i) \in S\}$

  2 $CPU\_Machines = \{machine\_cpu(id_i, CPU_i)$ tel que $\exists s_i(MEM_i, CPU_i, E_{on_i}, E_{off_i}, r_i, id_i) \in S\}$

Exemple 18. *Considérons le plan de configuration donné à l'Exemple 15. La Figure 6.2 illustre comment la contrainte cumulatives prend en considération les limites RAM de chaque serveur physique à chaque créneau de l'horizon.*

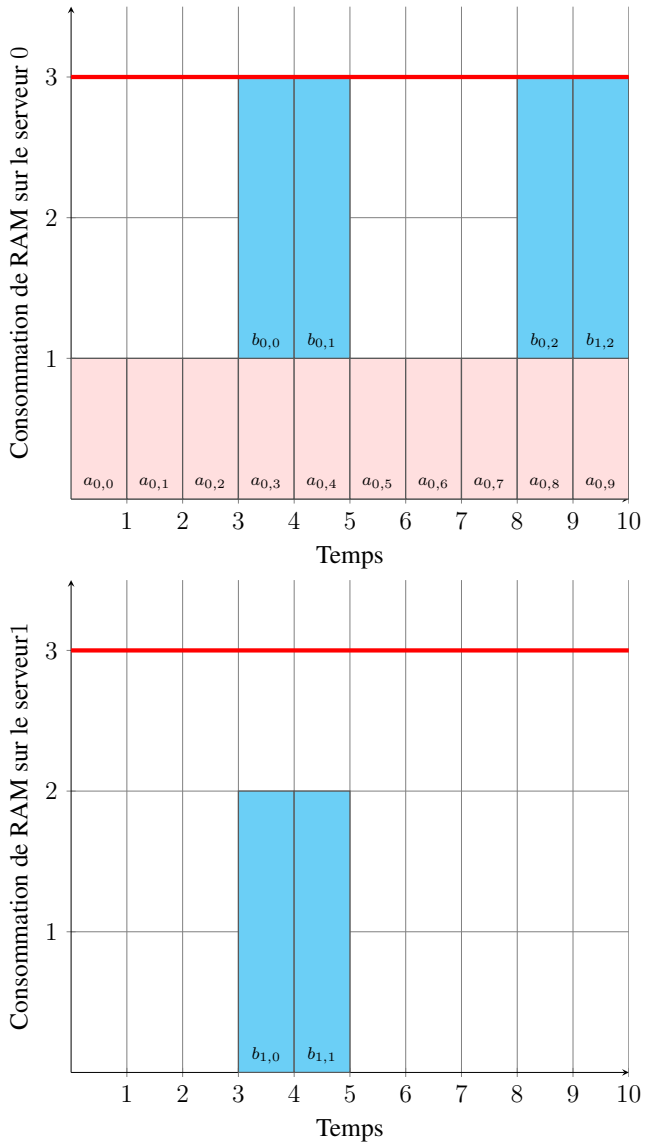


FIGURE 6.2 – Contrainte *cumulatives* pour la ressource RAM

6.3.6 Minimisation de la consommation d'énergie fossile

Avant de détailler comment minimiser la consommation d'énergie fossile d'un data center, nous spécifions dans un premier temps le calcul de la consommation énergétique d'une application active ou d'une application par lots. Soit $\mathbf{s_i}(\mathbf{MEM_i}, \mathbf{CPU_i}, \mathbf{E_{on_i}}, \mathbf{E_{off_i}}, \mathbf{r_i}, \mathbf{id_i})$ un serveur. On suppose que s_i soit déjà allumé à l'instant $t \in [0, T-1]$ et qu'aucune migration de tâche ne soit planifiée pour l'instant t. La consommation énergétique e_t de s_i à l'instant t est obtenu à partir de la relation r par :

$$e_t = r(load_{s_i,t}), \tag{6.7}$$

où $load_{s_i,t}$ est le pourcentage de charge CPU de s_i à l'instant t. Par conséquent, pour connaître la consommation énergétique d'une tâche active (resp. par lots) $\mathbf{sub_{(a_i,t)}}(\mathbf{t}, \mathbf{1}, \mathbf{t+1}, < \mathbf{mem_{i,t}}, \mathbf{cpu_{i,t}}, \mathbf{energy_{i,t}} >, \mathbf{h_{i,t}})$ (resp. $\mathbf{sub_{t,1,t+1,<mem_{i,j},cpu_{i,j},energy_{i,j}>,h_{i,j}}}$) à l'instant t, nous devons premièrement connaître la capacité CPU de l'hôte $h_{i,t}$ (resp. $h_{i,j}$) sur lequel elle est planifiée au créneau t. Le problème vient du fait que $h_{i,t}$ (resp. $h_{i,j}$) n'est pas initialement fixée, et vu que les capacités CPU des serveurs ne sont pas identiques (Data center hétérogène), il n'est pas possible de

connaitre par avance la capacité cpu du serveur hôte. Pour régler ce problème, nous utilisons la contrainte $element$ [VHC88].

Définition 24 (Contrainte élément)**.** *Soient une liste L d'entiers, deux variables entières $index$ et V. La contrainte $element(index, L, V)$ garantie que V soit $index^{ième}$ élément de la liste L i.e. $V = L_{index}$.*

Soit $CPU\_Capacities$ la liste des capacités CPU des serveurs. La liste est classée par ordre croissant des identifiants de serveurs. Pour tout instant $t \in [0, T-1]$ et pour toute tâche active (resp. par lots) $\mathbf{sub_{(a_i,t)}(t, 1, t+1, < mem_{i,t}, cpu_{i,t}, energy_{i,t} >, h_{i,t})}$ (resp. $\mathbf{sub_{(b_j,j)}(s_{(b_i,j)}, 1, e_{(b_i,j)}, < mem_{i,j}, cpu_{i,j}, energy_{i,j} >, h_{i,j})}$) on pose les contraintes :

$$element(h, CPU\_Capacities, HostCapacity) \tag{6.8}$$

La consommation énergétique $energy$ de la tâche est alors donnée par :

$$energy = r\left(\frac{cpu * 100}{HostCapacity}\right) \tag{6.9}$$

En fonction du modèle choisi pour le calcul de la consommation énergétique d'un serveur en fonction de sa charge CPU, la relation r peut être une fonction en escalier. La conséquence directe est qu'à chaque instant t, l'énergie consommée par un serveur hébergeant un certain nombre de tâches ne peut pas être calculée en sommant simplement la consommation énergétique de chaque tâche. Ceci se traduit par l'Inégalité 6.10.

$$r(\sum_i load_{i,t}) \neq \sum_i r(load_{i,t}) \tag{6.10}$$

Dès lors, pour calculer la consommation énergétique d'un serveur à un moment donné, nous devons appliquer sa charge totale à la relation r. Un problème subsiste néanmoins. Avec la contrainte *cumulatives*, il n'y a aucun moyen direct pour connaître la charge des machines, on a juste la garantie que chaque machine n'est pas chargée au delà de ses capacités. Pour contourner ce problème, nous utilisons un ensemble de $T \times Horizon$ tâche factices qu'on appelle *tâches complémentaires*. Le rôle de ces dernières est de totalement remplir la ressource CPU de chaque machine à chaque instant. Pour cela on procède comme suit :

Pour chaque machine hôte h et pour chaque instant $t \in [0, Horizon - 1]$ on définit une tâche complémentaire $comp_{h,t}(t, 1, t+1, < cpu_{com_{h,t}} >, h)$ où :

- t est le début, la durée est 1 et la fin est $t + 1$.
- $cpu_{com_{h,t}}$ est l'usage cpu. C'et une variable qui est fixée uniquement après que toutes les tâches affectées à l'hôte h à l'instant t soient connues. Elle prend alors la valeur de l'espace cpu encore disponible sur l'hôte à cet instant.

- h est l'identifiant de l'hôte.

Dans ce qui suit, nous présentons comment utiliser ces tâches complémentaires pour minimiser la consommation d'énergie fossile du data center. Mais auparavant, nous montrons comment calculer la consommation énergétique d'un serveur à un instant t donné.

Pour chaque serveur hôte h et pour chaque instant $t \in [0, Horizon]$, il existe par construction un tâche complémentaire $comp_{h,t}(t, 1, t+1, < cpu_{com_{h,t}} >, h)$. Par définition de cette tâche complémentaire, la charge CPU du serveur h à l'instant t est donnée par :

$$load(h,t) = \frac{CPU - cpu_{com_{h,t}}}{CPU} \tag{6.11}$$

où CPU est la capacité CPU de l'hôte h. La consommation énergétique de l'hôte h à l'instant t est alors donnée par l'Équation 6.12.

$$energy(h,t) = r(load(h,t)) \tag{6.12}$$

La valeur $energy(h,t)$ prend uniquement en considération la charge CPU de l'hôte. Pour calculer la consommation effective du data center, il est nécessaire de prendre en considération toutes les opérations qui ont un coût énergétique, en l'occurrence les migrations de tâches d'un serveur à un autre et les mise en route/extinctions de serveurs. On obtient donc les contraintes suivantes.

L'Équation 6.13 donne la consommation énergétique totale du data center à l'instant t.

$$E_t = \sum_h energy(h,t) + \sum_a migr\_energy_{(a_i,t)} + \sum_b migr\_energy_{(b_i,t)} + \sum_{h,t} switch\_energy_{(h,t)} \tag{6.13}$$

Où :

- $migr\_energy_{(a_i,t)}$ (respectivement $migr\_energy_{(b_i,t)}$) est la quantité d'énergie consommée pour toute migration de la tâche active a (respectivement de la tâche par lot b) d'un hôte à un autre à l'instant t.
- $switch\_energy_{(h,t)}$ est la quantité d'énergie pour mettre en route ou pour éteindre l'hôte h à l'instant t.

l'Égalité 6.14 contraint la quantité totale d'énergie fossile consommée par le data center à l'instant t.

$$B_t = \max(0, E_t - G_t) \tag{6.14}$$

où G_t est la quantité d'énergie verte disponible à l'instant t. L'Équation 6.15 donne enfin la quantité totale d'énergie fossile consommée par le data center durant toute la période fixée par l'horizon temporel.

$$B = \sum_{t=0}^{T-1} B_t \tag{6.15}$$

Exemple 19. *La Figure 6.3 montre la consommation énergétique globale du data center avec le plan de configuration de l'Exemple 15. Comme le montre la figure, les applications par lots sont mises en exécution de préférence pendant les périodes où l'énergie verte est suffisamment disponible, il en va de même pour les migration et les mises sous tensions des serveurs. Ceci est rendu possible par l'heuristique que nous présentons à la Section 6.4.*

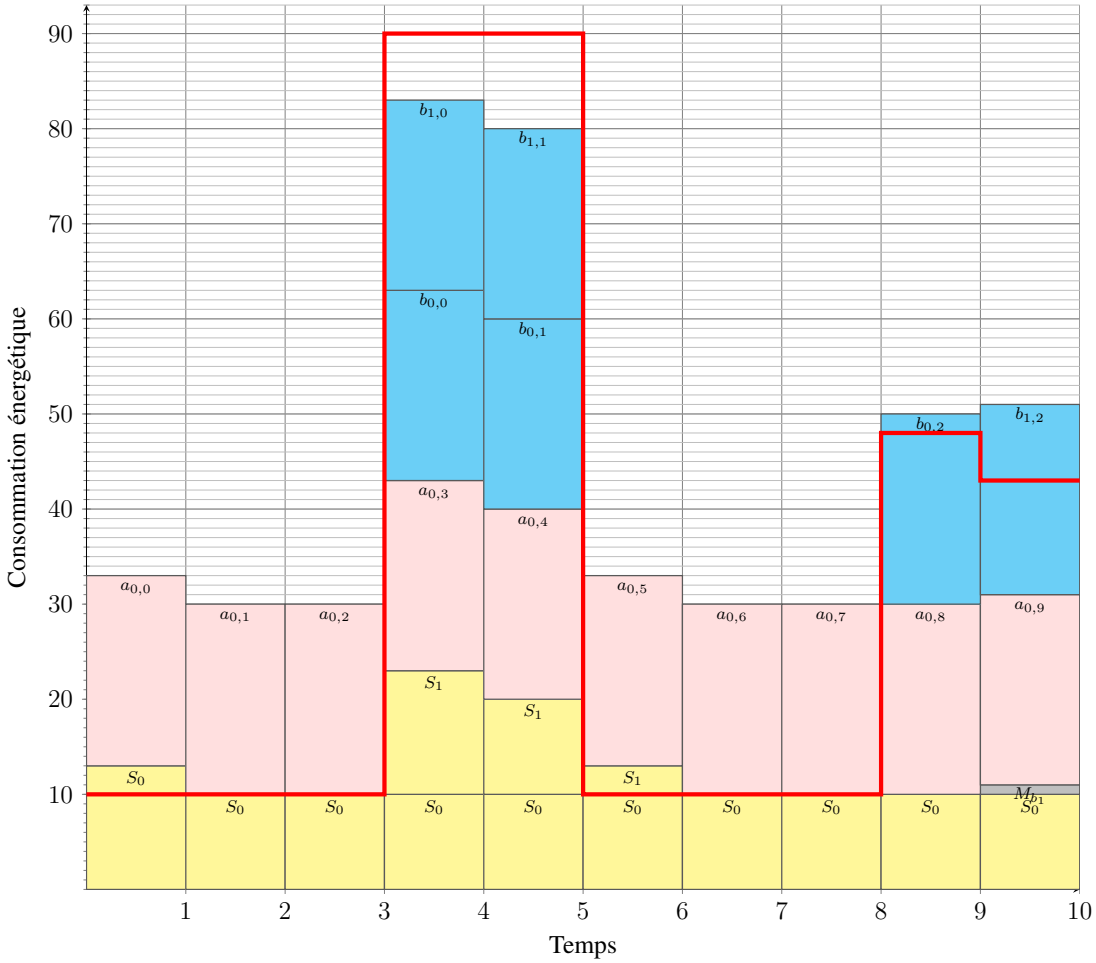


FIGURE 6.3 – Consommation énergétique du plan de configuration. La courbe rouge indique la quantité d'énergie verte disponible à chaque créneau

Un plan de configuration énergiquement efficace est un plan maximisant la consommation d'énergie verte en tout minimisant la consommation d'énergie fossile tout au long de l'horizon. L'objectif global du PPEE est alors de minimiser la valeur de B.

6.4 Résolution d'un PPEE

En fonction de la taille du problème, le temps de recherche d'une solution optimale (et la preuve d'optimalité) à un problème de satisfaction de contraintes peut être relativement long. Pour réduire ce temps de

recherche, il est important d'avoir une stratégie de recherche dédiée. Il existe deux principaux types d'heuristiques, les heuristiques statiques et les heuristiques dynamiques. Les heuristiques statiques définissent un ordre fixe dans lequel les variables sont instanciées. L'ordre d'assignation des variables avec une heuristique dynamique peut varier d'une branche à l'autre de l'arbre de recherche.

L'heuristique que nous proposons est dynamique, et se concentre sur les variables de début des applications par lots. Une fois ces variables fixées, les autres variables du problème sont fixées par propagation. L'heuristique que nous proposons ne cible que les variables de début des applications par lots car les variables de début des tâches interactives sont déjà fixés vu que ces tâches ne sont pas interruptibles.

Du fait qu'elles soient interruptibles, les variables de début et de fin des tâches par lots sont flexibles. L'intuition de l'heuristique est de planifier à chaque fois le début d'une tâche par lots sur les créneaux pendant lesquels la disponibilité en énergie verte est importante. Pour ce faire, nous procédons comme suit.

6.4.0.1 Choix des variables

Soit $b(mem, cpu, d, migr, slack)$ une application par lots modélisées par l'ensemble S_b de tâches par lots de la forme $sub_{(b,i)}(s_{(b,i)}, 1, e_{(b,i)}, < mem, cpu, energy >, h)$. L'ordre dans lequel les variables de début des tâches par lots de l'ensemble S_b sont fixées est $\mathbf{s_{(b,0)}}, \mathbf{s_{(b,1)}}, \ldots, \mathbf{s_{(b,d-1)}}$. Cet ordre permet lors de la recherche de prendre en considération la contrainte de précédence présentée à l'équation 6.3 pendant la recherche.

6.4.0.2 Choix des valeurs

Cette partie de la stratégie concerne l'ordre dans lequel des valeurs sont choisies dans le domaine de chaque variable de début. Soit $s_{(b,i)}$ la variable de début de $sub_{(b,i)}$. On a $dom(s_{(b,i)}) = [0, T-1]$. Suivant l'intuition, l'idée est d'essayer de fixer $s_{(b,i)}$ à une valeur $t \in dom(s_{(b,i)})$ qui représente un créneau durant lequel la disponibilité en énergie verte est maximale. Ceci garanti une consommation maximale de l'énergie verte. Nous procédons en deux étapes :

1 **Étape de tri**. Ceci est la première étape, elle est exécutée chaque fois qu'on doit instancier une variable de début. Soit $E = e_0 e_1 \ldots e_{T-1}$ l'énergie verte d'un PPEE où $E(t) = e_t, t \in [0, T-1]$ est la quantité d'énergie verte disponible au créneau t. On trie les valeurs e_t en ordre décroissant et on obtient la liste $SortedE = e_{(0,t_0)}, e_{(1,t_1)}, \ldots, e_{(T-1,t_{T-1})}$ telle que $\forall e_{(i,t_i)} \in SortedE, E(t_i) = e_{t_i} = e_{(i,t_i)}$

   On obtient ainsi l'ordre dans lequel les valeurs sont choisies de $dom(s_{(b,i)})$, en l'occurrence $t_0, t_1, \ldots, t_{T-1}$.

2 **Étape de mise à jour**. Cette étape est exécutée immédiatement après que la variable de début d'une tâche ait été fixée. Soit une tâche par lots $sub_{(b,i)}(s_{(b,i)}, 1, e_{(b,i)}, < mem, cpu, energy >, h)$ et supposons que $s_{(b,i)}$ soit fixé à $t \in [0, T-1]$. Comme l'exécution de $sub_{(b,i)}$ consomme une certaine

quantité d'énergie, cette étape met à jour la quantité d'énergie disponible au créneau t. La mise à jour est faite en décrémentant $E(t)$ par $energy$. La contrainte suivante effectue cette mise à jour.

$$E \leftarrow e_0, e_1, \ldots, e_t - energy, e_{t+1}, \ldots, e_{T-1} \tag{6.16}$$

Les étapes de *tri* et de *mise à jour* présentées dans cette Section illustrent l'idée générale de l'heuristique pour résoudre un PPEE. Cependant, en pratique le modèle pour le calcul de la consommation énergétique d'un serveur utilise une relation r qui est en général une fonction escalier. Dans la Section 6.5, nous allons donc adapter ces étapes de tri et de mise à jour pour mieux s'accorder aux instances réelles évaluées.

6.5 Evaluation

Dans cette section, nous évaluons les performances de notre modèle en termes de ratio de consommation énergétique. L'objectif est de mesurer l'efficacité du modèle dans un petit/moyen data center. Ceci justifie la nécessité d'exécuter les simulations sur des traces réelles de charge de travail. Comme stipulé dans [BFG+17], il n'est pas aisé de disposer des traces réelles d'activité d'un data center. Pour cette évaluation, nous avons étudié des traces anonymisées fournies par EasyVirt. Nous avons utilisé 3 ensembles de données réelles pour en faire 3 instances réelles d'évaluation. Chaque ensemble de données recense l'activité d'un service d'hébergement de machines virtuelles consistant en 55 serveurs physiques sur une période de 12 heures. Les traces recensent les requêtes des clients pour des machines virtuelles avec des besoins en CPU et RAM spécifiés. Dans le contexte d'un data center virtualisé, les machines virtuelles sont utilisées pour encapsuler les applications qui sont exécutées sur les machines physiques. Avant de présenter les résultats de l'évaluation, nous donnons d'abord les détails sur les jeux de données utilisés ainsi que sur le modèle énergétique utilisé pour calculer la consommation électrique.

6.5.1 Description des jeux de données

Chacune des trois traces réelles est organisée comme suit :

(1) 55 serveurs où chaque serveur est caractérisé par une quantité limité de ressources CPU et RAM.

(2) Un ensemble de 567 applications interactives à encapsuler dans des machines virtuelles. Comme précisé à la section 6.3.3, chaque application active est exécutée tout au long de l'horizon temporel sans interruption. Dans notre cas, l'horizon temporel est de 12. Nous pouvons observer à partir des traces que que les besoins en ressources CPU et RAM des applications interactives peuvent varier d'un instant à l'autre. Ceci se traduit par $6804 = 567 \times 12$ tâches à gérer. Les coûts de migration en termes de consommation énergétique sont aussi spécifiés.

(3) Un ensemble de 2268 applications par lots à encapsuler dans des machines virtuelles. Chaque application par lot est caractérisée par une durée d'exécution fixe, une valeur de slack, et des besoins en

Nombre de cœurs actifs	0	1	2	3	4	5	6	7	8	9	10	11	12
Puissance (Watts)	97	128	150	158	165	171	177	185	195	200	204	212	220

TABLE 6.1 – Consommation énergétique expérimentale d'un nœud à 12 cœurs en fonction du nombre de cœurs actifs.

ressources CPU et RAM pouvant aussi varier d'un instant à l'autre. Ceci se traduit par plus de 6500 tâches par lots a gérer.

(4) L'énergie renouvelable. Chaque jeu de données inclue une série temporelle de taille 12. Cette série temporelle est la trace d'enregistrement de la quantité d'énergie renouvelable disponible chaque heure. Cette période de 12 heures coïncide avec la période de 12 d'exécution des applications.

6.5.2 Modèle énergétique

Comme présenté à la Section 6.2.1, chaque serveur est caractérisé par une relation r qui permet de calculer sa consommation énergétique en fonction de sa charge. Dans [LOM15], Li et al. ont mené des expérimentations sur le nœud Taurus sur le site lyonnais de Grid'5000. Chaque nœud a 12 cœurs et chaque cœur représente 8.3% de l'utilisation CPU totale. Ils sont arrivés à la conclusion que la consommation globale d'énergie du nœud dépend du nombre de cœurs actifs. La Table 6.1 présente ces résultats.

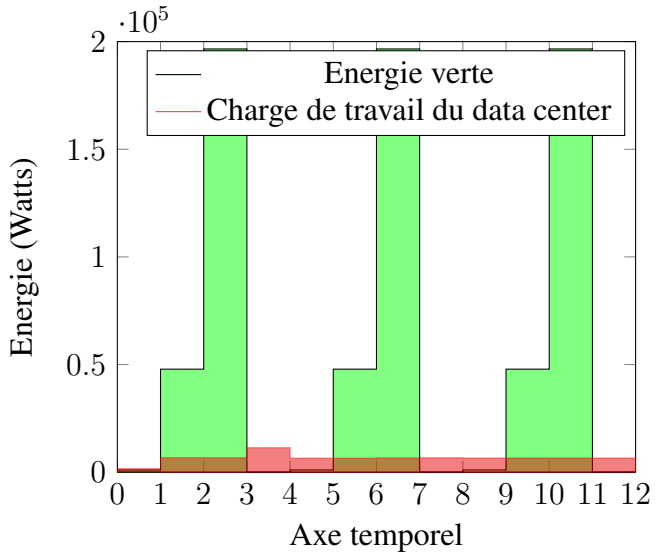
Suivant l'idée du *PowerModel* de *CloudSim*[CRB+11], nous utilisons la Table 6.1 pour implémenter la procédure *getPower()* qui calcule la consommation énergétique d'un serveur en fonction de sa charge CPU. Le pseudo-code de la procédure *getPower()* est donné dans l'Algorithme 9.

Algorithm 9 Procédure getPower() contraint la consommation énergétique d'un serveur en fonction de sa charge CPU

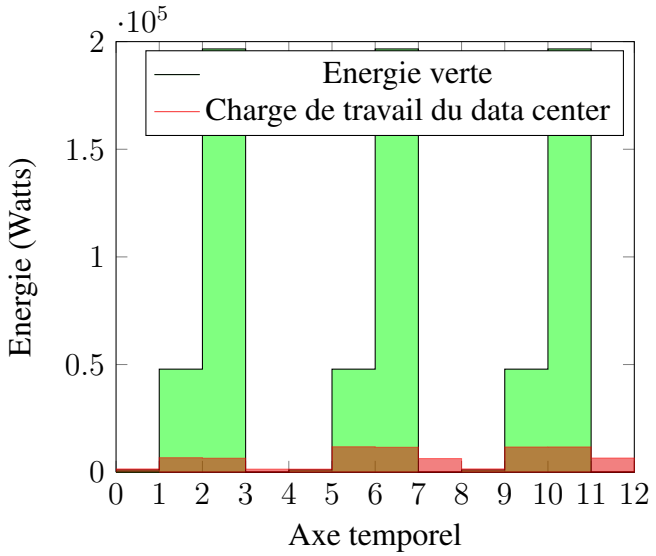
Données: $CPU\_Capacity$ // La capacité CPU du serveur
Données: CPU_Used // La quantité de CPU utilisée
$Power\_table \leftarrow [97, 128, 150, 158, 165, 171, 177, 185, 195, 200, 204, 212, 220]$
$Step \leftarrow \frac{1}{12} \times CPU\_Capacity$
$A \leftarrow \frac{1}{Step+1} \times CPU\_Used$
$B \leftarrow \min(A+1, 13)$
$element(A, Power\_table, E_A)$ // La contrainte *élément* [ABD+16]
$element(B, Power\_table, E_B)$
$Energy \leftarrow E_A + ((E_B - E_A) \times CPU\_Used \times 10)/(CPU\_Capacity \times 10)$
Retourner $Energy$

Dans le pseudo-code de l'Algorithme 9, le symbole # indique que l'instruction devant laquelle il est placé est une contrainte. Nous montrons à présent comment nous adaptons les étapes de **tri** et de **mise à jour** de l'heuristique pour la rendre compatible avec ce modèle énergétique.

1 **Étape de tri**. Pendant la phase de tri, nous n'allons pas totalement ordonner les valeurs de quantité d'énergie verte disponible e_t. La raison est que, comme $97Watts$ est la consommation énergétique minimale d'un serveur en marche, on veut juste s'assurer qu'il y ait au moins $97Watts$ d'énergie



(a) Algorithme *first fit*



(b) Modèle PPEE avec heuristique

FIGURE 6.4 – Planification de la charge de travail en fonction de la disponibilité d'énergie verte.

verte de disponible à l'instant t sur lequel on veut placer une tâche. Dans le cas contraire, on cherche le prochain créneau horaire disposant de cette quantité d'énergie verte et on effectue un échange d'indice entre les deux instants. La tâche en cours d'instanciation sera alors fixée au moment où on est sûr de disposer d'assez d'énergie verte pour l'exécuter. Si aucun créneau ne dispose d'assez d'énergie verte, alors aucun échange d'indice n'est fait, et la tâche est placée au plus tôt. Comme certaines tâches sont liées par des relations de précédence, l'avantage de ce tri partiel est de permettre d'instancier les tâches d'une chaine de précédences le plus tôt possible.

2 **Étape de mis à jour**. La modification principale de l'étape de mise à jour est due au fait que le modèle énergétique choisi pour cette évaluation est une fonction escalier. Et donc, exécuter une tâche par lots $sub_{(b,i)}(s_{(b,i)}, 1, e_{(b,i)}, < mem, cpu, energy >, h)$ au créneau t ne décrémente pas de $energy$ l'énergie renouvelable disponible à t. Nous avons empiriquement fixé cette valeur à 8. l'Équation 6.17 devient alors :

$$E \leftarrow e_0, e_1, \ldots, e_t - 8, e_{t+1}, \ldots, e_{T-1} \tag{6.17}$$

Pour évaluer notre modèle et pour suivre le même protocole que dans [LOM15], nous avons comparé les résultats à ceux obtenu en utilisant un algorithme de type *first fit* de référence. Ces deux implémentations sont faites en SICStus Prolog [Ca14].

Les Figures 6.4a et 6.4b montrent comment notre modèle parvient à déplacer une quantité considérable de charge de travail des créneaux pauvres en énergie verte vers des créneaux plus riches en énergie verte.

Les Tables 6.2, 6.3 et 6.4 présentent dans les détails comment notre modèle de PPEE avec l'heuristique réduit considérablement la consommation d'énergie fossile tout en démultipliant la consommation d'énergie verte.

	E. Totale	E. Fossile	E. Verte
First Fit	78811	39019	39792
Modèle PPEE + Heur	78694	18500	60194
	-0.14%	**-52.58%**	**+51.27%**

TABLE 6.2 – Gains énergétiques (W) sur la première instance réelle.

	E. Totale	E. Fossile	E. Verte
First Fit	78309	38699	39610
Modèle PPEE + Heur	78374	18196	60178
	+0.08%	**-51.98%**	**+51.92%**

TABLE 6.3 – Gains énergétiques (W) sur la deuxième instance réelle.

	E. Totale	E. Fossile	E. Verte
First Fit	79100	36195	42905
Modèle PPEE + Heur	78868	14670	64198
	-0.29%	**-59.46%**	**+49.62%**

TABLE 6.4 – Gains énergétiques (W) sur la troisième instance réelle.

En moyenne, comparé à l'algorithme *First Fit* de référence, notre modèle réduit par 55% la quantité d'énergie fossile consommée par le data center, et augmente par 50.9% la consommation d'énergie verte. Bien que PIKA [LOM15] présente un plus grand ratio d'intégration d'énergie verte comparée à l'algorithme *First Fit* de référence, il le fait avec deux inconvénients majeurs. Premièrement, contrairement à PIKA [LOM15], notre modèle planifie toutes les tâches pas plus tard que l'algorithme de référence. Cet avantage est dû au fait que l'horizon temporel est fixé, de sorte qu'aucune tâche ne peut être programmée au-delà. Si la contrainte de l'horizon temporel est supprimée, il sera possible d'optimiser d'avantage la consommation. Par exemple, la quantité de charge de travail visible au créneau 11 de la Figure 6.4b pourrait être reporté à un emplacement ultérieur disposant de plus d'énergie verte. Mais cela entraînera par la suite que les tâches soient terminées plus tard qu'avec l'algorithme de référence comme c'est le cas avec PIKA. Deuxièmement, tandis que le modèle GEASP diminue légèrement la consommation totale d'énergie (Brown + Green), PIKA de son côté l'augmente jusqu'à 31%.

6.6 Conclusion

Dans le cloud computing la consommation d'énergie et les problèmes environnementaux sont devenus une préoccupation majeure au cours des dernières années. Dans ce chapitre, nous avons introduit le problème de planification énergiquement écologique pour optimiser la consommation d'énergie d'un data center de petite et moyenne taille. En utilisant notre modèle pour résoudre le PPEE, nous avons pu optimiser la consommation d'énergie d'un data center de petite/moyenne taille sur trois plans. D'abord, nous avons légèrement

diminué sa consommation globale d'énergie, puis nous avons considérablement réduit la consommation d'énergie fossile, et enfin nous avons considérablement augmenté sa consommation d'énergie verte. Pour atteindre ces résultats, nous nous sommes principalement focalisé sur les applications par lots dont l'exécution peut parfois être retardée pour attendre une période avec une forte disponibilité d'énergie verte. La prochaine étape de travail consiste à optimiser la consommation d'énergie des applications interactives, en adaptant la qualité du service (QoS) en fonction de la disponibilité énergétique verte [HALP16].

7

Conclusion

Du fait que l'électricité devienne un facteur limitant dans le déploiement des data center, la tendance actuelle s'articule autour des problématiques de réduction de leur consommation énergétique. Les travaux d'optimisations peuvent être de deux ordres :

- **Statique** : Sur ce plan, il est question de concevoir des matériels moins énergivores.
- **Dynamique** : Sur ce plan il est question d'optimiser l'utilisation des ressources de façon à diminuer la consommation d'énergie.

Dans cette thèse, nous avons attaqué le problème sous l'angle dynamique. Cependant, au lieu de se limiter à simplement réduire la consommation globale d'énergie, nous avons également suivi la piste consistant à intégrer directement les énergies renouvelables dans le modèle.

L'énergie verte est par nature une énergie intermittente. Ainsi nous avons commencé par étudier les problèmes d'ordonnancement tenant compte d'une ressource à coût variable. Les travaux existant dans la littérature ne se prêtent pas au contexte des data center dans lequel il faut tenir compte à minima des deux contraintes suivantes :

(1) Les tâches ont des durées variables. En effet, une tâche s'exécute plus ou moins rapidement selon la configuration de l'environnement d'exécution. Par ailleurs le temps d'exécution d'une tâche peut varier lorsque pour des raisons diverses, l'administrateur du système décide de modifier la qualité du service (QoS) [HALP16].

(2) La précédence entre les tâches. Dans certains cas, les résultats de traitement d'une tâches constituent les données d'entrée d'une autre. Dans ce contexte il est donc impératif de respecter l'ordre d'exécution des tâches ainsi liées.

Nous avons donc proposé la contrainte TASKINTERSECTION ainsi qu'un algorithme de propagation basé sur la programmation dynamique. La contrainte TASKINTERSECTION prend en compte un ensemble de tâches à durée variable, et un ensemble d'intervalles qui représentent la ressource dont le coût est soit 0 ou 1, le but étant de minimiser le cumul des intersections de tâches avec les intervalles. Cette contribution a fait l'objet d'une publication à **CPAIOR 2016** [WB16]. Une perspective à cette contribution est de généraliser la chaine de précédence à l'arborescence de précédence. On obtiendrait ainsi un modèle ou certaines tâches peuvent être exécutées en concomitance et d'autres non.

Nous avons par la suite utilisé de la programmation par contraintes et des techniques issues de l'apprentissage machine pour faire de l'apprentissage à des fins de prédiction et de génération des charges de travail d'un data center. La prédiction de la charge de travail d'un data center permet de développer des heuristiques d'optimisation de ressources comme celle présentée au Chapitre 6. En général, les traces réelles de charge des data center sont disponibles en très petites quantités voir pas du tout. Quand bien même elles le sont, elles ne prêtent pas toujours au contexte d'étude. Cette contribution permet dans ces cas de valider les algorithmes et expérimentations qui prennent en paramètre la charge de travail d'un data center. Dans le cadre du projet EPOC, ces travaux ont fait l'objet d'une publication au journal **Computing 2017** [BFG+17], et ont été publiées dans un article à **SBAC-PAD 2017** [MWLO+17a]. Notre travail futur lié à cette contribution comprend la fourniture d'un site Web pour accéder à des ensembles de données à la demande produits par notre générateur utilisant diverses charges de travail réelles qui ne peuvent être rendues directement accessibles au public.

Nous avons enfin formalisé le problème de planification énergiquement écologique au sein d'un data center que nous avons ensuite modélisé en utilisant la programmation par contraintes. Pour résoudre ce problème nous avons ensuite proposé une heuristique dédiée. Les résultats montrent que l'on est capable de doubler la consommation en énergie renouvelable, de diviser par deux la consommation en énergie fossile, sans toutefois augmenter la consommation globale, et sans non plus retarder plus qu'il ne faut l'exécution des tâches dans un data center. Cette contribution a été publiée dans un article à **ICPADS 2017** [MWLO+17b]. La perspective de ce travail permettra d'optimiser d'avantage la consommation électrique d'un data center en jouant sur la qualité de service (QoS) des applications interactives.

Bien que spécifiques au contexte des data center, les contributions de cette thèse sont formulées de façon à pouvoir trouver des applications dans d'autres domaines. C'est ainsi qu'en utilisant la contraintes TASKINTERSECTION par exemple, nous avons pu améliorer de façon considérables les résultats de l'état de l'art sur le problème de résumé vidéo [BBG13, BBG14].

Encodage d'automates temporisés par des contraintes d'automate.

A.1 Introduction

Pour modéliser et analyser des systèmes temps réels, il existe des outils tel que UPAAL [BLL+95]. Ces outils permettent de modéliser des automates dont les valeurs des compteurs varient en général par simple incrémentation ou décrémentation. Ceci devient rapidement une limitation lorsqu'on a besoin de faire évoluer la valeur d'un compteur d'un état à un autre au moyen d'une opération arithmétique telle que *min, max, modulo*... Le but de ce travail est de lever cette limitation. Pour cela, nous proposons un cadriciel de modélisation d'automates temporisés par des contraintes d'automate. Cette modélisation permettra ensuite d'utiliser des outils et algorithmes de la programmation par contraintes [BCDP05, BCP04] pour répondre à certaines requêtes sur les automates. Les requêtes traitées sont les suivantes :

- Accessibilité d'un état,
- Temps minimum/maximum nécessaire pour atteindre un état acceptant,
- Calcul du produit de plusieurs automates hybrides linéaires,

Partant d'exemples d'automates temporisés, nous proposons une modélisation par contraintes et nous fournissons à chaque fois les codes Prolog compatibles avec le solveur de contraintes SICStus Prolog [Ca14]. Ce chapitre s'articule autour de deux grandes sections. La Section A.2 relative aux automates temporisés simples et la Section A.3 relative aux automates hybrides linéaires. Chacune des Sections présente les principes et les grandes lignes de l'encodage et donne des exemples de requêtes prises en compte.

A.2 Automates temporisés

Un automate temporisé [Alu99] est un automate fini étendu à l'aide d'un ensemble fini d'horloges. Les gardes ou contraintes d'horloge sont des contraintes qui autorisent ou interdisent le franchissement d'une transition en fonction des valeurs des horloges. Dans cette Section nous partons d'exemples d'automates temporisés pour illustrer notre modèle d'encodage basé sur la programmation par contraintes.

A.2.1 Encodage d'un automate temporisé

Dans notre modélisation, nous introduisons une nouvelle variable δ. Cette variable exprime le temps passé dans un état avant le franchissement d'une transition. Ainsi, étant donné un mot $abbc$ de l'alphabet $\mathbb{A} = \{a, b, c\}$, la lecture du symbole a à partir d'un état initial i se fait après un laps de temps δ_1 et la transition correspondante est franchie (en supposant que les conditions de franchissement de la transition soient respectées). La lecture du symbole b se fera aussi après un temps δ_2 qui désigne le temps passé sur le prochain état $i + 1$ et ainsi de suite.

Le mot $abbc$, associé à une séquence de laps de temps $\delta_1\delta_2\delta_3\delta_4$ est un mot temporisé que nous notons $(a, \delta_1)(b, \delta_2)(b, \delta_3)(c, \delta_4)$

La contrainte que nous modélisons doit être équivalente (voir Définition 25) à l'automate de départ.

Définition 25. ***Équivalence automate et contrainte d'automate***
Une contrainte d'automate est dite équivalente à un automate temporisé si et seulement si pour tout $n > 0$, l'ensemble de tous les mots de longueur n acceptés par l'automate et l'ensemble de tous les $n - uplets$ qui ne violent pas la contrainte contiennent exactement les mêmes éléments.

Exemple 20. *L'automate de la Figure A.1 est un automate temporisé à 4 états. Un des objectifs que nous nous sommes fixé pour la modélisation est de ne pas augmenter le nombre d'états de l'automate initial. Notre modélisation donne lieu à la contrainte d'automate de la Figure A.2.*

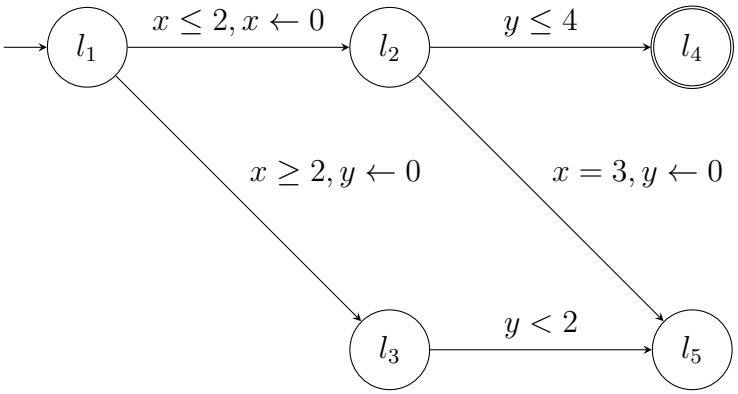


FIGURE A.1 – Automate temporisé 1.

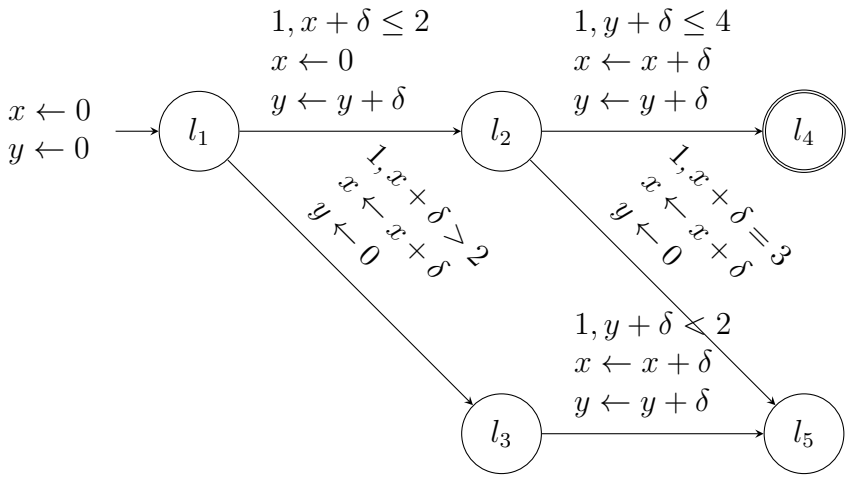


FIGURE A.2 – Automate contrainte correspondant à l'automate temporisé de la Figure A.1.

Exemple 21. *L'automate de la Figure A.3 contrairement à celui de la Figure A.1 contiens un invariant, sur l'état l_1. Cet invariant donne la condition sous laquelle il est possible de rester sur l'état l_1. Nous*

avons exprimé cet invariant sous forme de condition sur δ et x et/ou y, que nous rajoutons à la liste des conditions portées par chaque transition issue de l'état l_1. Cette transformation nous permet de maintenir l'équivalence entre l'automate et la contrainte d'automate.

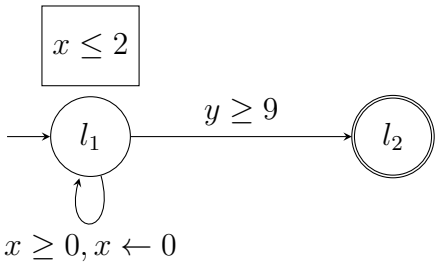


FIGURE A.3 – Automate temporisé 2

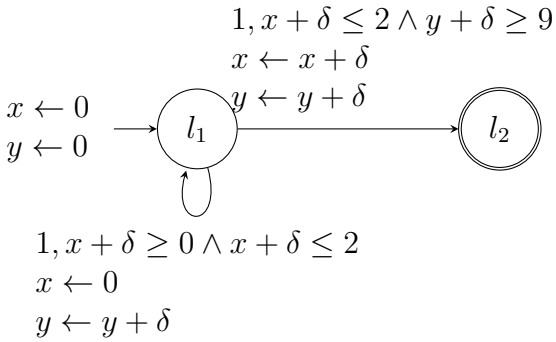


FIGURE A.4 – Automate contrainte correspondant à l'automate temporisé de la Figure A.3.

Remarque 1. *Les automates des Figures A.1 et A.3 ne consomment aucun symbole en franchissant les transitions. Cependant, dans le cas d'un automate à contraintes il faut obligatoirement consommer un symbole pour franchir une transition. Dans notre modèle par contraintes, nous avons choisi le symbole* 1 *qui est consommé sur toutes les transitions.*

Exemple 22. *Dans cet exemple, nous partons d'un automate qui à la différence des automates des Figures A.1 et A.3 consomme des symboles sur les transitions. La logique de la transformation reste la même, à cette différence que le symbole consommé n'est plus un symbole bidon. Les mots reconnus par l'automate de la Figure A.5 sont formés dans l'alphabet $\mathbb{A} = \{a, b, c, d\}$. Notre modèle ne pouvant prendre en compte que des variables entières, nous avons fait une correspondance une à une entre les symboles de l'alphabet $\mathbb{A} = \{a, b, c, d\}$ et ceux de l'ensemble $\{1, 2, 3, 4\}$.*

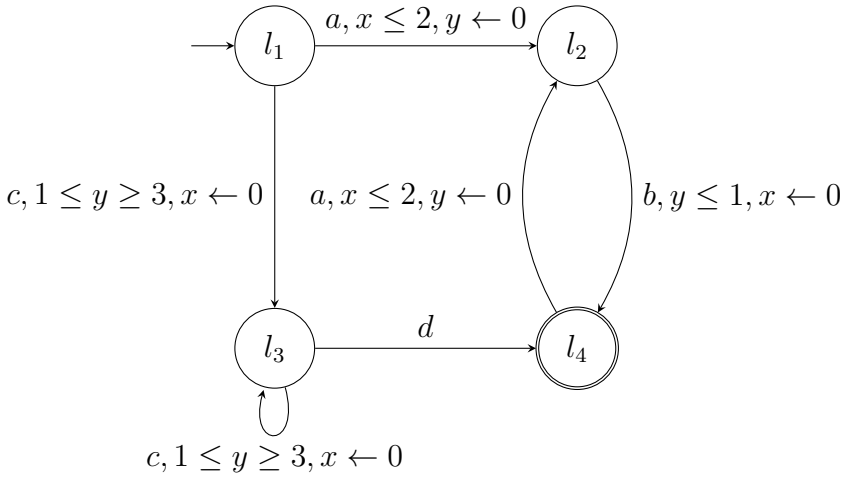


FIGURE A.5 – Automate temporisé 3.

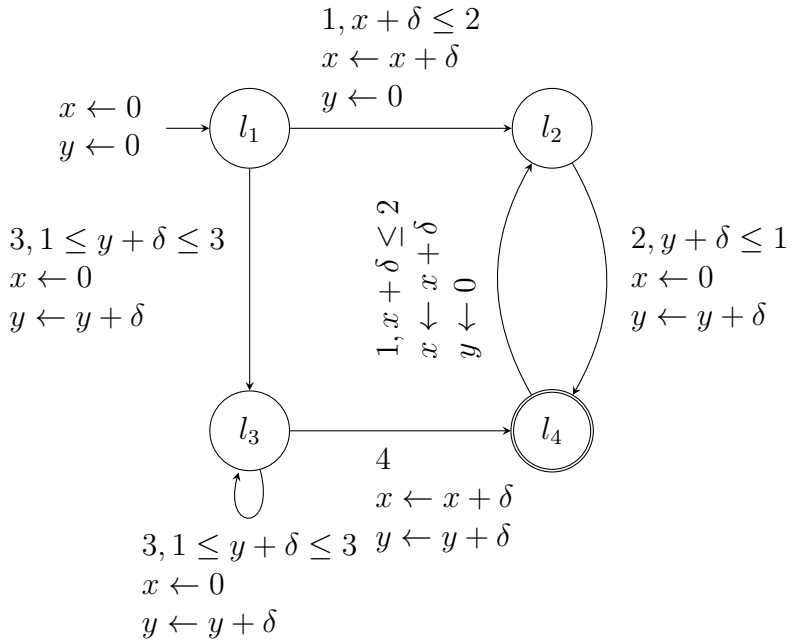


FIGURE A.6 – Automate contrainte correspondant à l'automate temporisé de la Figure A.5.

A.2.2 Code prolog

Pour les 3 exemples présentés, nous donnons dans cette section le code Prolog ainsi que des exemples de requêtes que nous pouvons exécuter.

A.2.3 Automate 1

Le code prolog correspondant pour l'automate de la Figure A.2 est donné ci-dessous.

```
a1(N) :-
    Max = 10,
    length(LDelta,N),
    domain(LDelta,0,Max),
    length(L,N),
```

```
    domain(L,1,1),
    N1 is N+1,
    length(LStates,N1),
    domain(LStates, 1,5),
    creates_counters(N1,LCounters),
    automaton(LDelta, Delta, L,
            [source(l1),sink(l4)],
            [arc(l1,1,l2,(X+Delta#=<2 -> [0,Y+Delta])),
            arc(l1,1,l3,(X+Delta#>2 -> [X + Delta,0])),
            arc(l2,1,l4,(Y+Delta#=<4 -> [X+Delta,Y+Delta])),
            arc(l2,1,l5,(X+Delta#=3 -> [X+Delta,0])),
            arc(l3,1,l5,(Y+Delta#<2 -> [X+Delta,Y+Delta]))],
            [X,Y],[0,0],[XF,YF],[state([l1-1,l2-2, l3-3, l4-4, l5-5],
            LStates),
            counterseq(LCounters)]),
    labeling([],LDelta),
    write(t(L, LStates,LCounters,LDelta,XF,YF)), nl,
    fail.

creates_counters(0, []).
creates_counters(N, [[C1,C2]|R]) :-
    N > 0,
    C1 in 0..20,
    C2 in 0..20,
    N1 is N-1,
    creates_counters(N1, R).
```

Avec SICStus Prolog, nous pouvons exécuter la requête :

```
?- a1(2).
```

Cette requêtes retourne l'ensemble des mots temporisés de longueur 2 acceptés par l'automate,ainsi que les différents états franchis, et l'évolution des compteurs.

```
t([1,1],[1,2,4],[[0,0],[0,0],[0,0]],[0,0],0,0)
t([1,1],[1,2,4],[[0,0],[0,0],[1,1]],[0,1],1,1)
t([1,1],[1,2,4],[[0,0],[0,0],[2,2]],[0,2],2,2)
t([1,1],[1,2,4],[[0,0],[0,0],[3,3]],[0,3],3,3)
t([1,1],[1,2,4],[[0,0],[0,0],[4,4]],[0,4],4,4)
t([1,1],[1,2,4],[[0,0],[0,1],[0,1]],[1,0],0,1)
t([1,1],[1,2,4],[[0,0],[0,1],[1,2]],[1,1],1,2)
t([1,1],[1,2,4],[[0,0],[0,1],[2,3]],[1,2],2,3)
t([1,1],[1,2,4],[[0,0],[0,1],[3,4]],[1,3],3,4)
t([1,1],[1,2,4],[[0,0],[0,2],[0,2]],[2,0],0,2)
t([1,1],[1,2,4],[[0,0],[0,2],[1,3]],[2,1],1,3)
t([1,1],[1,2,4],[[0,0],[0,2],[2,4]],[2,2],2,4)
```

Dans ce résultat, le seul mot de longueur 2 accepté par l'automate est le mot 11, les différents états atteints pour lire ce mot sont les états 1 (état initial), 2 et 4. On a ensuite la séquence d'évolution des compteurs ainsi que des δ et les valeurs finales des compteurs.

Sur ce même automate, la requête :

```
?- a1(3).
```

retourne :

```
no
```

Car cet automate ne peut accepter de mot de longueur 3.

A.2.4 Automate 2

Le code prolog correspondant pour l'automate de la Figure A.4 est donné ci-dessous.

```
a2(N) :-
    Max = 10,
    length(LDelta,N),
    domain(LDelta,0,Max),
    length(L,N),
    domain(L,1,1),
    N1 is N+1,
    length(LStates,N1),
    domain(LStates, 1,2),
    creates_counters(N1,LCounters),
    automaton(LDelta, Delta, L,
            [source(l1),sink(l2)],
            [arc(l1,1,l1,(X+Delta#>=0 #/\ X+Delta #=< 2 -> [0,Y+Delta])),
            arc(l1,1,l2,(Y+Delta#>=9 #/\ X+Delta #=< 2 -> [X+Delta,Y+Delta]))
            ],
            [X,Y],[0,0],[XF,YF],[state([l1-1,l2-2],LStates),
            counterseq(LCounters)]),
    labeling([],LDelta),
    write(t(L, LStates,LCounters,LDelta,XF,YF)), nl,
    fail.

creates_counters(0, []).
creates_counters(N, [[C1,C2]|R]) :-
    N > 0,
    C1 in 0..20,
    C2 in 0..20,
    N1 is N-1,
    creates_counters(N1, R).
```

Ce code est similaire à celui de la Section A.2.4.

A.2.5 Automate 3

Le code prolog correspondant pour l'automate de la Figure A.4 est donné ci-dessous.

```
a3(N) :-
    Max = 10,
    length(LDelta,N),
```

```
    domain(LDelta,0,Max),
    length(L,N),
    domain(L,1,4),
    N1 is N+1,
    length(LStates,N1),
    domain(LStates, 1,4),
    creates_counters(N1,LCounters),
    automaton(LDelta, Delta, L,
            [source(l1),sink(l4)],
            [arc(l1,1,l2,(X+Delta#=<2 -> [X+Delta,0])),
            arc(l1,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta])),
            arc(l3,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta])),
            arc(l3,4,l4,(Delta#=<MAX -> [X + Delta,Y+Delta])),
            arc(l2,2,l4,(Y+Delta#=<1 -> [0,Y+Delta])),
            arc(l4,1,l2,(X+Delta#=<2 -> [X+Delta,0]))],
            [X,Y],[0,0],[XF,YF],[state([l1-1,l2-2, l3-3, l4-4],LStates),
            counterseq(LCounters)]),
    create_letter_delta(L,LDelta,SymbolDelta),
    labeling([],SymbolDelta),
    write(t(L, LStates,LCounters,LDelta,XF,YF)), nl,
    fail.

create_letter_delta([],[],[]).
create_letter_delta([Symbol|R],[Delta|S],[Symbol,Delta|T]) :-
    create_letter_delta(R,S,T).

creates_counters(0, []).
creates_counters(N, [[C1,C2]|R]) :-
    N > 0,
    C1 in 0..20,
    C2 in 0..20,
    N1 is N-1,
    creates_counters(N1, R).
```

Ce code est similaire à celui de des Sections A.2.4 et A.2.3 à la différence que les symboles ne sont pas uniquement des 1.

Remarque 2. *Il est possible de modifier ces codes de façon à pouvoir exécuter des requêtes avec en paramètre une séquence de caractères (un mot) pour savoir s'il existe une séquence de δ telle que ce mot soit accepté par l'automate, ou avec en paramètre une séquence de δ pour savoir s'il existe une séquence de caractères qui combinée à la séquence des δ soit acceptée par l'automate. Il est aussi possible de passer en paramètre une séquence de caractères et une séquence de δ à une requête pour savoir si le mot temporisé est accepté ou non et avoir le cas échéant les informations sur les différents états franchis, les évolutions des compteur...*

Exemple 23. *Un question fréquente quand on travaille sur des automates est l'accessibilité d'un état donné. Pour répondre à ce type de question en utilisant des contraintes, nous allons modifier l'automate contrainte de la Figure A.5 pour prendre en compte le fait que l'on se donne une séquence de taille maximum fixée.*

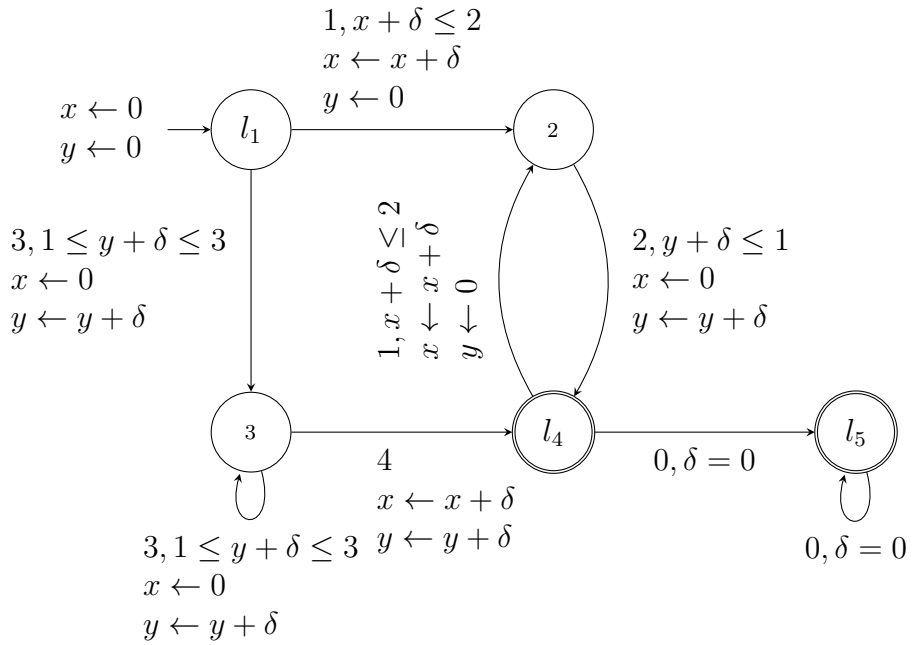


FIGURE A.7 – Modification de l'automate contrainte de la Figure A.6.

Dans ce nouvel automate contrainte, nous avons rajouté un état final, le l_5. Cet état est accessible à partir de l'état final de l'ancienne contrainte l_4 après consommation du nouveau symbole 0 et sans aucune condition ni mise à jour des horloges. De même nous avons créé une boucle sur ce nouvel état, qui permet d'y rester en consommant le symbole 0 et sans aucune condition ni mise à jour des horloges.

Le symbole 0 est un symbole que nous rajoutons à l'alphabet. Pour un $n \in \mathbb{N}$ donné, le but de cette modification est de permettre à la contrainte de considérer tous les mots de longueurs inférieure ou égale à n. Dans les faits, les mots trouvés par la contrainte sont tous de taille n, ils se terminent cependant par une concaténation du symbole 0. La taille du mot réel est donc obtenu en faisant $n-$ le nombre d'occurrences du symbole 0 en fin du mot.

Ainsi pour obtenir le nombre nous allons rajouter à cette contrainte d'automate la contrainte $atleast$ [BCR12].

La contrainte $atleast(N, VARIABLES, VALUE)$ prend 3 paramètres, un entier N, une collection de variables entières $VARIABLES$ et une valeur entière $VALUE$. La contrainte est vérifiée si le nombre d'occurrences de $VALUE$ dans la collection $VARIABLES$ est supérieure ou égale à N.

Le code Prolog correspondant à cette modélisation est donné ci-dessous :

```
a4(N,S) :-
    Max = 10,
    length(LDelta,N),
    domain(LDelta,0,Max),
    length(L,N),
    domain(L,0,4),
    N1 is N+1,
    length(LStates,N1),
    domain(LStates, 1,5),
    creates_counters(N1,LCounters),
    automaton(LDelta, Delta, L,
            [source(l1),sink(l4),sink(l5)],
            [arc(l1,1,l2,(X+Delta#=<2 -> [X+Delta,0])),
            arc(l1,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta])),
            arc(l3,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta])),
```

```
            arc(l3,4,l4,(Delta#=<MAX -> [X + Delta,Y+Delta])),
            arc(l2,2,l4,(Y+Delta#=<1 -> [0,Y+Delta])),
            arc(l4,1,l2,(X+Delta#=<2 -> [X+Delta,0])),
            arc(l4,0,l5, (Delta#=0 -> [X,Y])),
            arc(l5,0,l5, (Delta#=0 -> [X,Y]))],
            [X,Y],[0,0],[XF,YF],[state([l1-1,l2-2, l3-3, l4-4, l5-5],
            LStates),
            counterseq(LCounters)]),
    create_letter_delta(L,LDelta,SymbolDelta),
    count(S,LStates,#>=,1),
    labeling([],SymbolDelta),
    write(t(L, LStates,LCounters,LDelta,XF,YF)), nl,
    fail.

create_letter_delta([],[],[]).
create_letter_delta([Symbol|R],[Delta|S],[Symbol,Delta|T]) :-
    create_letter_delta(R,S,T).

creates_counters(0, []).
creates_counters(N, [[C1,C2]|R]) :-
    N > 0,
    C1 in 0..20,
    C2 in 0..20,
    N1 is N-1,
    creates_counters(N1, R).
```

Pour trouver tous les mots de taille inférieure ou égale à 3 *qui permettent d'atteindre l'état* l_2*, on exécute la requête :*

```
?- a4(3,2).
```

qui produit le résultat suivant :

```
t([1,2,0],[1,2,4,5],[[0,0],[0,0],[0,0],[0,0]],[0,0,0],0,0)
t([1,2,0],[1,2,4,5],[[0,0],[0,0],[0,1],[0,1]],[0,1,0],0,1)
t([1,2,0],[1,2,4,5],[[0,0],[1,0],[0,0],[0,0]],[1,0,0],0,0)
t([1,2,0],[1,2,4,5],[[0,0],[1,0],[0,1],[0,1]],[1,1,0],0,1)
t([1,2,0],[1,2,4,5],[[0,0],[2,0],[0,0],[0,0]],[2,0,0],0,0)
t([1,2,0],[1,2,4,5],[[0,0],[2,0],[0,1],[0,1]],[2,1,0],0,1)
```

Le seul mot de longueur inférieure ou égale à 3 *qui permet d'atteindre l'état* l_2 *est* 12*, car on ne compte pas le symbole* 0*.*

Exemple 24. *Dans cet exemple nous modélisons la question : quel est le temps minimum pour atteindre un état acceptant donné.*

Pour cela, nous rajoutons à la modélisation de l'Exemple 23 une troisième horloge K *qui n'est jamais réinitialisée. Il suffit alors de prendre le résultat minimisant la valeur finale de* K*.*

```
a5(N,S) :-
    Max = 10,
    length(LDelta,N),
    domain(LDelta,0,Max),
    length(L,N),
    domain(L,0,4),
    N1 is N+1,
    length(LStates,N1),
    domain(LStates, 1,5),
    creates_counters(N1,3,LCounters),
    automaton(LDelta, Delta, L,
            [source(l1),sink(l4),sink(l5)],
            [arc(l1,1,l2,(X+Delta#=<2 -> [X+Delta,0,K+Delta])),
            arc(l1,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta,K+Delta]
            )),
            arc(l3,3,l3,(Y+Delta#=<3 #/\ Y+Delta#>=1 -> [0,Y+Delta,K+Delta]
            )),
            arc(l3,4,l4,(Delta#=<MAX -> [X + Delta,Y+Delta,K+Delta])),
            arc(l2,2,l4,(Y+Delta#=<1 -> [0,Y+Delta,K+Delta])),
            arc(l4,1,l2,(X+Delta#=<2 -> [X+Delta,0,K+Delta])),
            arc(l4,0,l5, (Delta#=0 -> [X,Y,K+Delta])),
            arc(l5,0,l5, (Delta#=0 -> [X,Y,K+Delta]))],
            [X,Y,K],[0,0,0],[XF,YF,KF],[state([l1-1,l2-2, l3-3, l4-4, l5-5],
            LStates),
            counterseq(LCounters)]),
    create_letter_delta(L,LDelta,SymbolDelta),
    count(S,LStates,#>=,1),
    minimize(labeling([],SymbolDelta),KF),
    write(t(L, LStates,LCounters,LDelta,XF,YF,KF)), nl.
```

Pour trouver tous le temps minimum nécessaire pour atteindre l'état l_2, on peut par exemple exécute la requête :

```
?- a5(3,2).
```

qui produit le résultat suivant :

```
t([1,2,0],[1,2,4,5],[[0,0,0],[0,0,0],[0,0,0],[0,0,0]],[0,0,0],0,0,0)
```

Le temps minimum pour atteindre l'état l_2 est 0.

A.3 Automates hybrides linéaires.

Dans cette section, nous nous intéressons aux automates hybrides linéaires. Les automates hybrides linéaires diffèrent des automates temporisés par la présence sur chaque localité d'un vecteur d'activité qui associe à chaque variable de X (ensemble des horloges) une pente à valeur dans $\mathbb{Z}$. En partant d'un exemple, nous allons à la Section A.3.1 montrer comment nous modélisons un AHL par des contraintes d'automate.

La modélisation de certains problèmes réels par des AHL peut être complexe. La méthode généralement adoptée est de modéliser individuellement des sous parties du problème par des AHL, dont on fera ensuite le produit. Le calcul du produit de plusieurs n'est pas trivial, il en résulte souvent un automate à plusieurs états et transitions coûteux en mémoire. Dans la section A.3.3 nous allons simuler le fonctionnement du produit de plusieurs AHL, nous évitant ainsi d'avoir à construire ce produit.

A.3.1 Encodage d'un AHL

Dans cette modélisation, nous reprenons le principe présenté à la Section A.2.1, avec la variable δ qui représente le temps passé dans une localité. Dans le cas des AHL, nous avons des vecteurs d'activité dans chaque état, ces vecteurs indiquent comment évoluent les horloges dans ces états.

Considérons une localité l d'un AHL A à deux horloges x, y et dans laquelle nous avons le vecteur d'activité $\begin{pmatrix} \dot{x} = k_1 \\ \dot{y} = k_2 \end{pmatrix}$, $k_1, k_2 \in \mathbb{Z}$. Soit $t = (l, g, e, R, l')$ une transition de la localité l à une localité l' avec pour garde g, e une lettre de l'alphabet, et $R \subseteq X$ l'ensemble des compteurs réinitialisés après franchissement. Le vecteur d'activité s'interprète de la façon suivante. Après franchissement de t, les valeurs des compteurs x et y sont mis à jour de la façon suivante :

- Si $x \in R$ alors $x \leftarrow 0$ sinon $x \leftarrow x + k_1.\delta$.
- Si $y \in R$ alors $y \leftarrow 0$ sinon $y \leftarrow x + k_2.\delta$.

Exemple 25. ***Le bruleur à gaz***

Considérons un brûleur à gaz [Lar03] qui fuit de temps à autre. Supposons que toute fuite est détectée et résorbée en moins de $1s$ *et qu'un délai minimum de* $30s$ *sépare deux fuites. On souhaite maintenant vérifier que le temps total de fuite est au plus* $1/20$ *du temps total durant tout intervalle supérieur à* $60s$*. On peut modéliser un tel système par l'automate hybride linéaire de la Figure A.8. La localité* q_0 *correspond à l'état où le système fuit. L'horloge* y *mesure le temps total de fonctionnement, l'horloge* x *mesure le temps de présence dans chacune des localités (elle est remise à zéro à chaque changement d'état) et* z *mesure le temps total de fuite (elle n'est active que dans la localité* q_0*). La propriété à vérifier est donc* $(y \geq 60) \Rightarrow (20z \leq y)$.

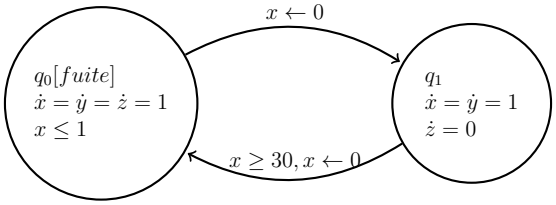


FIGURE A.8 – Automate temporisé pour un brûleur à gaz.

Notre modélisation de cet AHL par une contrainte d'automate est donnée en Figure A.9.

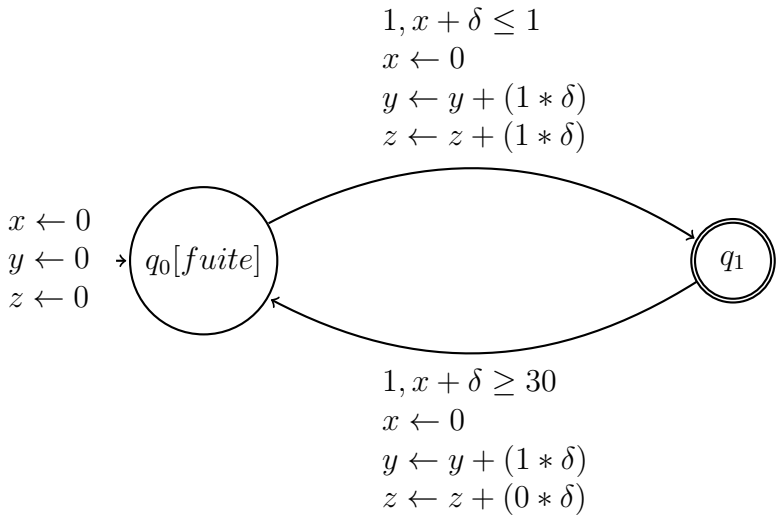


FIGURE A.9 – Automate contrainte correspondant à l'AHL du bruleur à gaz la Figure A.8.

A.3.2 Modélisation d'un AHL à coûts

Il est possible d'ajouter à un AHL une variable de coûts, cette variable qui dépend du temps passé dans chaque état donne le coût total nécessaire à la lecture d'un mot. Le fait que la variable de coût dépende des différents temps passés dans chaque état implique que le coût d'un mot dépend des différents états visités pour la lecture du mot, un mot peut donc avoir plusieurs coûts selon la suite de transitions franchises pour le lire. Il sera donc possible de chercher la suite de transitions lisant ce mot tout en minimisant le coût total. La modélisation de ces automates par des contraintes est très similaire à celle d'un AHL classique, pour prendre en compte le coût, nous rajoutons une variable c, un compteur ayant lui aussi une pente dans chaque état, la pente indiquant comment il évolue en fonction du temps passé dans un état. Cette variable de coût n'est cependant jamais réinitialisée.
Avec l'exemple du bruleur à gaz vu ci-dessus, l'horloge y de l'automate de la Figure A.9 peut être utilisée pour donner le coût d'un mot, car elle n'est jamais réinitialisée.

A.3.3 Simulation du produit de plusieurs AHL

Dans cette section nous nous intéressons au produit de plusieurs automates. Si dans certains cas, le calcul du produit de deux ou plusieurs automates peux se faire à la main, cette tâche devient rapidement complexe.

La démarche que nous proposons consiste non pas à calculer ce produit, mais plutôt à modifier légèrement les automates en considération de façon à pouvoir simuler le fonctionnement de l'automate produit sur un mot temporisé.

Remarque 3. *La procédure que nous présentons suppose que les compteurs soient indépendants d'un automate à l'autre.*

Pour décrire notre démarche, nous avons généré deux AHL (Figures A.10 et A.11) relativement simples dont le produit calculé à la main est donné en Figure A.12. Une fois ce produit calculé, il est facile de savoir si un mot temporisé donné est accepté ou pas. Considérons les trois mots temporisés ci-dessous que nous passons à l'automate produit de de $AHL1$ et $AHL2$:

- $m_1 = (c, 1)(c, 2)(a, 2)$
- $m_2 = (c, 0)(c, 0)(a, 4)$
- $m_3 = (c, 1)(c, 2)(a, 1)$

- Le mot m_1 n'est pas accepté par l'automate car à la lecture du symbole a, le compteur x_2 vaut 5 et ne satisfait pas la garde de la transition de la localité (p_0, q_0) vers (p_2, q_2).
- Le mot m_2 n'est lui aussi pas accepté car à la lecture du symbole a, le compteur x_1 vaut 4 et ne peut donc satisfaire la garde de la transition de la localité (p_0, q_0) vers (p_2, q_2).
- Le mot m_3 est quant à lui accepté par l'automate.

Dans cet exemple, il était trivial de simuler le comportement de l'automate produit car nous avons au préalable calculé ce produit. En général il ne sera pas toujours évident de calculer le produit de deux ou plusieurs AHL, nous allons à présent présenter une méthode pour simuler le comportement du produit de plusieurs AHL sans devoir au préalable le calculer.

Notre méthode passe par deux étapes de pré-traitement :

1 Premièrement, on va rajouter une boucle sur chaque localité de chaque automate, la boucle lit un symbole bidon α et a pour garde l'invariant de la localité en considération. Le symbole α est rajouté aux alphabets associés à chacun des AHL. On obtient les AHL des Figures A.13 et A.14.

2 La deuxième étape consiste à générer à partir du mot qu'on veux faire lire à l'automate résultant du produit, plusieurs mots qui seront lus individuellement et de façon synchrone par chaque automate du produit ayant été modifié à l'étape 1. soit M le mot en considération. Nous allons créer autant de mots que d'automates dans le produit et la taille de chaque mot est égale à la taille de M. Pour tout AHL i du produit, on crée le mot $M(AHL_i)$ de la façon suivante :

- $M(AHL_i) = \epsilon$ Au début, $M(AHL_i)$ est le mot vide.
- Pour chaque symbole a_j de la gauche à la droite de M avec δ_j son temps associé, on a :
$$M(AHL_i) = \begin{cases} M(AHL_i) + (a_j, \delta_j) & \text{si } a_j \in \mathbb{A}(AHL_i) \\ M(AHL_i) + (\alpha, \delta_j) & \text{si } a_j \notin \mathbb{A}(AHL_i). \end{cases}$$
où $+$ est l'opérateur de concaténation et $\mathbb{A}(AHLi)$ est l'alphabet de l'AHL i.

Nous allons dérouler cette étape sur les mots temporisés $m_1, m_2, m3$ ci-dessus. Les Symboles qui apparaissent dans l'AHL_1 sont a, b et c, nous déduisons donc son alphabet, $\mathbb{A}(AHL_1) = \{a, b, c\}$, de même nous avons $\mathbb{A}(AHL_2) = \{a, d\}$.

- $m_1 = (c, 1)(c, 2)(a, 2)$
On obtient les mot $m_1(AHL_1) = (c, 1)(c, 2)(a, 2)$ et $m_1(AHL_2) = (\alpha, 1)(\alpha, 2)(a, 2)$
- $m_2 = (c, 0)(c, 0)(a, 4)$
On obtient les mot $m_2(AHL_1) = (c, 0)(c, 0)(a, 4)$ et $m_2(AHL_2) = (\alpha, 0)(\alpha, 0)(a, 4)$
- $m_3 = (c, 1)(c, 2)(a, 1)$
On obtient les mot $m_3(AHL_1) = (c, 1)(c, 2)(a, 1)$ et $m_3(AHL_2) = (\alpha, 1)(\alpha, 2)(a, 1)$

Après cette étape de pré-traitement, on peut simuler le comportement de l'automate résultant du produit sur un mot. Le mot M est accepté par le produit des automates $AHL1, ..., AHLn$ si et seulement si chacun des mots $M(AHL_j)$ $1 \leq j \leq n$ est accepté par l'AHL_j modifié à l'étape 1 de pré-traitement. Vérifions à nouveau Le comportement de l'automate résultant du produit des AHL des Figures A.10 et A.11 sur les mots m_1, m_2 et m_3.

- $m_1 = (c,1)(c,2)(a,2)$

  Le mot $m_1(AHL_1) = (c,1)(c,2)(a,2)$ est accepté par l'automate de la Figure A.13 tandis $m_1(AHL_2) = (\alpha,1)(\alpha,2)(a,2)$ ne l'est pas par l'Automate de la Figure A.14. On conclut donc que m_1 n'est pas accepté par l'automate résultant du produit de AHL_1 et AHL_2.

- $m_2 = (c,0)(c,0)(a,4)$
  Le mot $m_2(AHL_2) = (\alpha,0)(\alpha,0)(a,4)$ est accepté par l'automate de la Figure A.14 tandis que $m_2(AHL_1) = (c,0)(c,0)(a,4)$ ne l'est pas par l'automate de la Figure A.13. On conclut donc que m_2 n'est pas accepté par l'automate résultant du produit de AHL_1 et AHL_2.

- $m_3 = (c,1)(c,2)(a,1)$
  Les mots $m_3(AHL_1) = (c,1)(c,2)(a,1)$ et $m_3(AHL2) = (\alpha,1)(\alpha,2)(a,1)$ sont tous deux acceptés par les automates des Figures A.13 et A.14 respectivement. On conclut donc que m_3 est accepté par l'automate résultant du produit de AHL_1 et AHL_2.

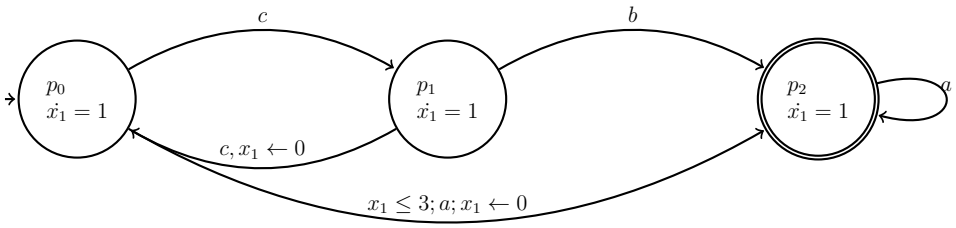


FIGURE A.10 – AHL_1.

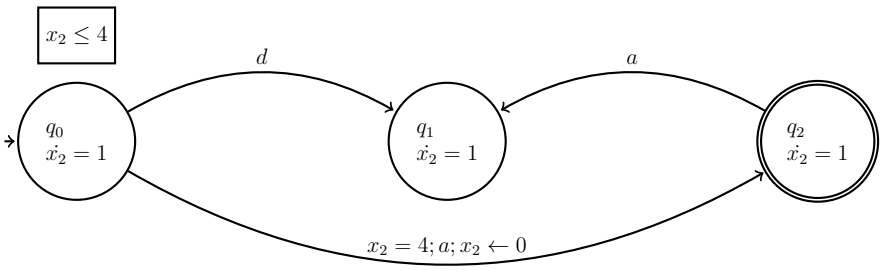


FIGURE A.11 – AHL_2.

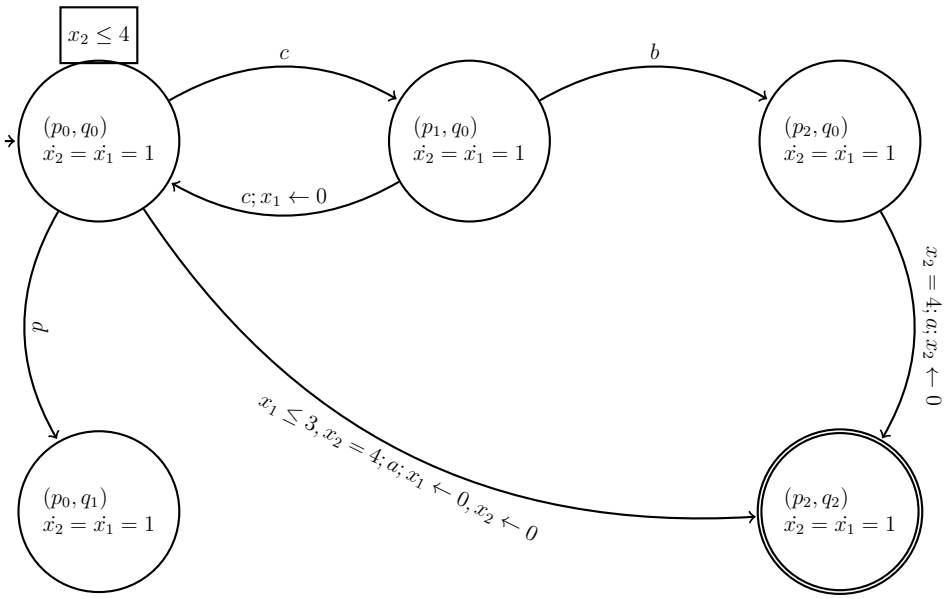


FIGURE A.12 – Produit des AHL 1 et 2.

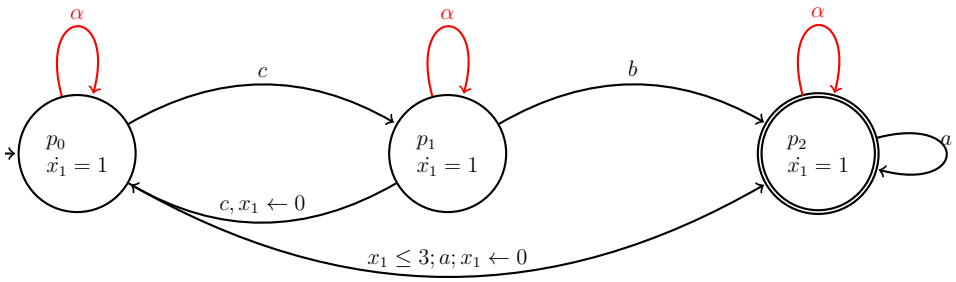


FIGURE A.13 – AHL_1 après modification.

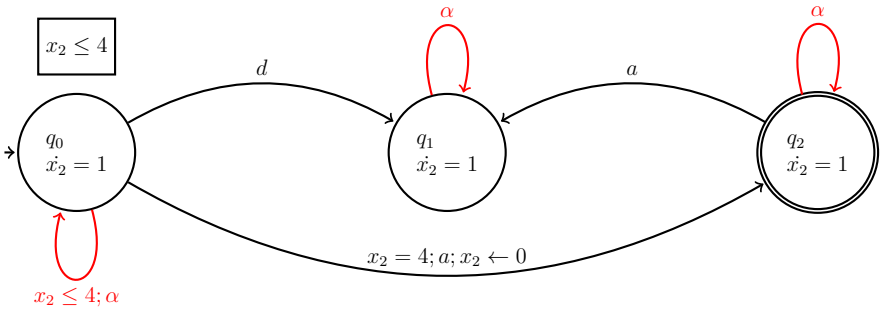


FIGURE A.14 – AHL_2 après modification.

A.3.4 Code prolog

Nous présentons dans cette section le code prolog permettant de simuler le produit des deux automates des Figures A.10 et A.11. Le code combine les versions modifiées des AHL exprimés sous forme de contraintes d'automates. Nous avons aussi rajouté une contrainte de signature qui à partir d'un mot temporisé donné, va générer selon la méthode décrite plus haut deux mots (signatures) qui seront lus(es) par les deux automates.

```
a7(VARS, LDelta) :-
    ALPHABET_LIST_1 = [1,2,3,4],
```

```
    ALPHABET_LIST_2 = [1,2,5],
    Max = 10,
    length(VARS,N),
    N1 is N+1,
    length(LStates_1,N1),
    domain(LStates_1, 1,3),
    creates_counters(N1,1,LCounters_1),
    list_to_fdset(ALPHABET_LIST_1,ALPHABET_SET_1),
    list_to_fdset(ALPHABET_LIST_2,ALPHABET_SET_2),
    composition_signature(VARS, SIGN_1, SIGN_2, ALPHABET_SET_1,
    ALPHABET_SET_2),
    automaton(LDelta, Delta, SIGN_1,
            [source(l1),sink(l3)],
            [arc(l1,4,l2,[X + Delta]),
            arc(l2,1,l2,[X + Delta]),
            arc(l1,1,l1,[X + Delta]),
            arc(l3,1,l3,[X + Delta]),
            arc(l2,3,l3,[X + Delta]),
            arc(l2,2,l3,[X + Delta]),
            arc(l2,4,l1,[0]),
            arc(l1,2,l3,(X+Delta#=<3 -> [0]))],
            [X],[0],[XF],[state([l1-1,l2-2, l3-3],LStates_1),
             counterseq(LCounters_1)]),

    length(LStates_2, N1),
    domain(LStates_2, 1,3),
    creates_counters(N1,1,LCounters_2),
    automaton(LDelta, Delta, SIGN_2,
            [source(p1),sink(p3)],
            [arc(p1,5,p2,[Y + Delta]),
            arc(p3,2,p2,[Y + Delta]),
            arc(p1,2,p3,(Y+Delta#=4 -> [0])),
            arc(p1,1,p1,(Y+Delta#=<4 -> [Y + Delta])),
            arc(p3,1,p3,[Y + Delta]),
            arc(p2,1,p2,[Y + Delta])],
            [Y],[0],[YF],[state([p1-1,p2-2, p3-3],LStates_2),
            counterseq(LCounters_2)]),

    create_LState(LStates_1, LStates_2, LStates),
    labeling([],LDelta),
    write(t(SIGN_1, SIGN_2, LStates, XF, YF)), nl,
    fail.



composition_signature([], [], [], _, _).
composition_signature([VAR|VARs], [S|Ss], [T|Tt],
 ALPHABET_SET_1, ALPHABET_SET_2) :-
    VAR in_set ALPHABET_SET_1 #<=> S #= VAR,
    #\ VAR in_set ALPHABET_SET_1 #<=> S #= 1,
```

```
    VAR in_set ALPHABET_SET_2 #<=> T #= VAR,
    #\ VAR in_set ALPHABET_SET_2 #<=> T #= 1,
    composition_signature(VARs, Ss, Tt, ALPHABET_SET_1, ALPHABET_SET_2).


%Function to combine the states of the 2 automata
create_LState([],[],[]).
create_LState([S|Ss],[T|Tt],[[P,Q]|R]) :-
    P #= S,
    Q #= T,
    create_LState(Ss,Tt,R).
```

Une requête sur cette modélisation a la forme $?-a7(VARS, LDelta)$. où $VARS$ représente la liste des symboles et $LDelta$ la liste des δ constituant le mot temporisé à tester. Par ailleurs, nous avons fait une correspondance une à une entre les symboles des alphabets $\mathbb{A}(AHL_2) \cup \mathbb{A}(AHL_2) = \{a, b, c, d\}$ et ceux de l'ensemble $\{2, 3, 4, 5\}$, et entre le symbole α et 1. Nous allons donc reprendre nos 3 mots m_1, m_2 et m_3.

- Pour tester le mot $m_1 = (c, 1)(c, 2)(a, 2)$, on exécute la requête :

  ```
  ?- a7([4,4,2], [1,2,2]).
  ```

  La requête retourne :

  ```
  no
  ```

  Le mot m_1 n'est donc pas accepté.

- Pour tester le mot $m_2 = (c, 0)(c, 0)(a, 4)$, on exécute la requête :

  ```
  ?- a7([4,4,2], [0,0,4]).
  ```

  La requête retourne :

  ```
  no
  ```

  Le mot m_2 n'est donc pas accepté.

- Pour tester le mot $m_3 = (c, 1)(c, 2)(a, 1)$, on exécute la requête :

  ```
  ?- a7([4,4,2], [1,2,1]).
  ```

  La requête retourne :

  ```
  t([4,4,2],[1,1,2],[[1,1],[2,1],[1,1],[3,3]],0,0)
  ```

  Le mot m_3 donc accepté et la réponse de la requête donne les signatures crées, les différents états visités par les automates pour lire le mot ainsi que les horloges finales.

A.4 Conclusion

Dans cette annexe, nous avons présenté un mini cadriciel d'encodage d'automates temporisés et hybrides linéaires. Le cadriciel proposé, reposant sur la programmation par contraintes permet de lever certaines limitations rencontrées avec des outils classiques d'analyse de systèmes temps réels. Les perspectives de ce travail consistent à créer une couche d'abstraction qui permettra de modéliser des automates dans un langage similaire à celui utilisé par UPAAL.

Bibliographie

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7) :57–73, 1993. 38

[ABD+16] Ekaterina Arafailova, Nicolas Beldiceanu, Rémi Douence, Mats Carlsson, Pierre Flener, María Andreína Francisco Rodríguez, Justin Pearson, and Helmut Simonis. Global Constraint Catalog, Volume II, Time-Series Constraints. *CoRR*, abs/1609.08925, 2016. 31, 88, 96

[All83] James Frederick Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11) :832–843, 1983. 59

[Alu99] Rajeev Alur. Timed automata. In *International Conference on Computer Aided Verification*, pages 8–22. Springer, 1999. 103

[AON+09] Mutasem Khalil Sari Alsmadi, Khairuddin Bin Omar, Shahrul Azman Noah, et al. Back propagation algorithm : the best algorithm among the multi-layer perceptron algorithm. *IJCSNS International Journal of Computer Science and Network Security*, 9(4) :378–383, 2009. 26, 70

[BAB12] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5) :755–768, 2012. 11

[Bar05] Luiz André Barroso. The price of performance. *Queue*, 3(7) :48–53, 2005. 78

[BB10] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *IEEE/ACM international conference on cluster, cloud and grid computing (CCGrid)*, pages 826–831, 2010. 11

[BBG13] Sid-Ahmed Berrani, Haykel Boukadida, and Patrick Gros. Constraint satisfaction programming for video summarization. In *Multimedia (ISM), 2013 IEEE International Symposium on*, pages 195–202. IEEE, 2013. 12, 38, 57, 59, 102

[BBG14] Haykel Boukadida, Sid-Ahmed Berrani, and Patrick Gros. A novel modeling for video summarization using constraint satisfaction programming. In *Advances in Visual Computing*, pages 208–219. Springer, 2014. 12, 38, 57, 59, 102

[BC02] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *International Conference on Principles and Practice of Constraint Programming*, pages 63–79. Springer, 2002. 88

[BCDP05] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of Global Constraints Based on Constraint Checkers. *Constraints*, 10(3), 2005. 103

[BCDS15] Nicolas Beldiceanu, Mats Carlsson, Rémi Douence, and Helmut Simonis. Using finite transducers for describing and synthesising structural time-series constraints. *Constraints*, pages 1–19, 2015. 31

[BCP04] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving Filtering Algorithms from Constraint Checkers. In M. G. Wallace, editor, *Principles and Practice of Constraint Programming (CP'2004)*, volume 3258 of *LNCS*, pages 107–122. Springer-Verlag, 2004. Preprint available as SICS Tech Report T2004-08, soda.swedish-ict.se/2346/1/SICS-T--2004-08--SE.pdf. 103

[BCR12] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global Constraint Catalog, 2nd Edition (revision a). Technical Report T2012-03, Swedish Institute of Computer Science, 2012. Available at http://soda.swedish-ict.se/5195/1/T2012-03.pdf. 110

[Bes06a] Christian Bessiere. Constraint propagation. *Foundations of Artificial Intelligence*, 2 :29–83, 2006. 74

[Bes06b] Christian Bessière. Constraint propagation."Handbook of constraint programming", 2006, chapter 3, 29-83, 2006. 20, 21, 38, 41

[BFG+15] Nicolas Beldiceanu, Barbara Dumas Feris, Philippe Gravey, Sabbir Hasan, Claude Jard, Thomas Ledoux, Yunbo Li, Didier Lime, Gilles Madi-Wamba, Jean-Marc Menaud, et al. The epoc project : Energy proportional and opportunistic computing system. In *Smart Cities and Green ICT Systems (SMARTGREENS), 2015 International Conference on*, pages 1–7. IEEE, 2015. 80

[BFG+16] Nicolas Beldiceanu, Bárbara Dumas Feris, Philippe Gravey, Sabbir Hasan, Claude Jard, Thomas Ledoux, Yunbo Li, Didier Lime, Gilles Madi-Wamba, Jean-Marc Menaud, Pascal Morel, Michel Morvan, Marie-Laure Moulinard, Anne-Cécile Orgerie, Jean-Louis Pazat, Olivier Roux, and Ammar Sharaiha. Towards energy-proportional clouds partially powered by renewable energy. *Computing*, pages 1–20, 2016. 62, 75

[BFG+17] Nicolas Beldiceanu, Bárbara Dumas Feris, Philippe Gravey, Sabbir Hasan, Claude Jard, Thomas Ledoux, Yunbo Li, Didier Lime, Gilles Madi-Wamba, Jean-Marc Menaud, et al. Towards energy-proportional clouds partially powered by renewable energy. *Computing*, 99(1) :3–22, 2017. 11, 13, 95, 102

[BLL+95] J Bengtsson, K Larsen, F Larsson, P Pettersson, and W Yi. Upaall a tool suite for automatic verification of real-time systems, inproccedings of the 4th dimacs workshop on verification and control of hybrid systems'. *Lecture Notes in Computer Science, Springer-Verlag*, 1995. 13, 103

[BLPN12] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling : applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012. 38, 82

[BMFL02] Christian Bessière, Pedro Meseguer, Eugene C Freuder, and Javier Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141(1-2) :205–224, 2002. 21

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010. 25

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland HC Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005. 20

[Ca14] Mats Carlsson and al. *SICStus Prolog User´s Manual*. SICS Swedish ICT AB, Mai 2014. 97, 103

[CMM83] Jaime G Carbonell, Ryszard S Michalski, and Tom M Mitchell. An overview of machine learning. In *Machine learning*, pages 3–23. Springer, 1983. 23

[CMOS13a] Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Bin packing with linear usage costs–an application to energy management in data centres. In *International Conference on Principles and Practice of Constraint Programming*, pages 47–62. Springer, 2013. 78

[CMOS13b] Hadrien Cambazard, Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Constraint programming based large neighbourhood search for energy minimisation in data centres. In *International Conference on Grid Economics and Business Models*, pages 44–59. Springer, 2013. 78

[CRB+11] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software : Practice and experience*, 41(1) :23–50, 2011. 96

[CWA+88] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog user's manual*, volume 3. Swedish Institute of Computer Science Kista, Sweden, 1988. 74

[DPFP15] Alban Derrien, Charles Prud'Homme, Jean-Guillaume Fages, and Thierry Petit. A global constraint for a tractable class of temporal optimization problems. *Constraints Programming*, pages 105–120, 2015. 9, 38, 57, 59, 60

[EM+03] Ahmet Ekin, Rajiv Mehrotra, et al. Automatic soccer video analysis and summarization. *Image Processing, IEEE Transactions on*, 12(7) :796–807, 2003. 38, 57

[FD+95] Daniel Frost, Rina Dechter, et al. Look-ahead value ordering for constraint satisfaction problems. In *IJCAI (1)*, pages 572–578, 1995. 23

[FP15] Jean-Guillaume Fages and Charles Prud'homme. A free and open-source java library for constraint programming. `http://choco.emn.fr/`, 2015. 57

[HALP16] Md Sabbir Hasan, Frederico Alvares, Thomas Ledoux, and Jean-Louis Pazat. Enabling green energy awareness in interactive cloud application. In *IEEE International Conference on Cloud Computing Technology and Science 2016*, 2016. 99, 101

[Ham09] James Hamilton. Cooperative expendable micro-slice servers (cems) : low cost, low power servers for internet-scale services. In *Conference on Innovative Data Systems Research (CIDR'09)(January 2009)*. Citeseer, 2009. 78

[HDL11] Fabien Hermenier, Sophie Demassey, and Xavier Lorca. Bin repacking scheduling in virtualized datacenters. In *International Conference on Principles and Practice of Constraint Programming*, pages 27–41. Springer, 2011. 30, 31, 82

[HE80] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3) :263–313, 1980. 23

[HLM+09] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy : a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009. 31

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Unsupervised learning. In *The elements of statistical learning*, pages 485–585. Springer, 2009. 26

[HZS06] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine : theory and applications. *Neurocomputing*, 70(1) :489–501, 2006. 62

[IM15] Salam Ismaeel and Ali Miri. Using ELM techniques to predict data centre VM requests. In *IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 80–86, 2015. 33, 62

[IM16] Salam Ismaeel and Ali Miri. Multivariate Time Series ELM for Cloud Data Centre Workload Prediction. In *International Conference on Human-Computer Interaction (HCI). Theory, Design, Development and Practice*, pages 565–576. Springer, 2016. 62

[KCK13] TK Satish Kumar, Marcello Cirillo, and Sven Koenig. Simple temporal problems with taboo regions. In *AAAI*. Citeseer, 2013. 38

[Koo11] J. Koomey. Growth in Data Center Electricity Use 2005 to 2010. Analytics Press, Aug 2011. 11

[KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning : A review of classification techniques, 2007. 24

[Lar03] François Laroussinie. *Automates temporisés et hybrides Modélisation et vérification*, 2003. Available at http://www.liafa.jussieu.fr/~francoisl/PUBLIS/fl-etr2003.pdf. 113

[LBC12] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 439–454. Springer, 2012. 88

[LOM15] Yunbo Li, Anne-Cécile Orgerie, and Jean-Marc Menaud. Opportunistic scheduling in clouds partially powered by green energy. In *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*, pages 448–455. IEEE, 2015. 13, 78, 96, 97, 98

[Mac77] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1) :99–118, 1977. 20

[MU13] Ismail Bin Mohamad and Dauda Usman. Standardization and its effects on k-means clustering algorithm. *Research Journal of Applied Sciences, Engineering and Technology*, 6(17) :3299–3303, 2013. 27

[MW15] Gilles Madi Wamba. Random generated instances of the taskintersection problem. https://www.dropbox.com/sh/uwvn86rx7mxebty/AADyUAdnEWdOkmC8Xkcyjg3Ua?dl=0, 2015. 57

[MWLO+17a] Gilles Madi Wamba, Yunbo Li, Anne-Cécile Orgerie, Nicolas Beldiceanu, and Jean-Marc Menaud. Cloud workload prediction and generation models. In *SBAC-PAD : International Symposium on Computer Architecture and High Performance Computing*, Campinas, Brazil, October 2017. 13, 102

[MWLO+17b] Gilles Madi Wamba, Yunbo Li, Anne-Cécile Orgerie, Nicolas Beldiceanu, and Jean-Marc Menaud. Green energy aware scheduling problem in virtualized datacenters. In *ICPADS 2017*, Shenzen, China, December 2017. 13, 102

[Nie15] Michael A Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. 25

[OAL14] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computing Surveys (CSUR)*, 46(4) :47, 2014. 78

[PM11] Vaishali R Patel and Rupa G Mehta. Impact of outlier removal and normalization approach in modified k-means clustering algorithm. *IJCSI International Journal of Computer Science Issues*, 8(5), 2011. 26, 63, 64

[PMWV09] Steven Pelley, David Meisner, Thomas F Wenisch, and James W VanGilder. Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*, 2009. 11

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. 18, 83

[SH10] Helmut Simonis and Tarik Hadzic. A family of resource constraints for energy cost aware scheduling. In *Third International Workshop on Constraint Reasoning and Optimization for Computational Sustainability, St. Andrews, Scotland, UK (September 2010)*, 2010. 38

[SH11] Helmut Simonis and Tarik Hadzic. A resource cost aware cumulative. In *Recent Advances in Constraints*, pages 76–89. Springer, 2011. 38

[Sou05] Francis Sourd. Optimal timing of a sequence of tasks with general completion costs. *European Journal of Operational Research*, 165(1) :82–96, 2005. 38

[VHC88] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity : An experience with ai and or techniques. In *AAAI*, pages 660–664, 1988. 91

[WB16] Gilles Madi Wamba and Nicolas Beldiceanu. The taskintersection constraint. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 246–261. Springer, 2016. 12, 60, 101

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Gilles MADI WAMBA

Combiner la programmation par contraintes et l'apprentissage machine pour construire un modèle éco-énergétique pour petits et moyens data centers.

Combining constraint programming and machine learning to come up with an energy aware model for small/medium size data centers.

Résumé

Au cours de la dernière décennie les technologies de cloud computing ont connu un essor considérable se traduisant par la montée en flèche de la consommation électrique des data center. L'ampleur du problème a motivé de nombreux travaux de recherche s'articulant autour de solutions de réduction statique ou dynamique de l'enveloppe globale de consommation électrique d'un data center. L'objectif de cette thèse est d'intégrer les sources d'énergie renouvelables dans les modèles d'optimisation dynamique d'énergie dans un data center. Pour cela nous utilisons la programmation par contraintes ainsi que des techniques d'apprentissage machine. Dans un premier temps, nous proposons une contrainte globale d'intersection de tâches tenant compte d'une ressource à coûts variables. Dans un second temps, nous proposons deux modèles d'apprentissage pour la prédiction de la charge de travail d'un data center et pour la génération de telles courbes. Enfin, nous formalisons le problème de planification énergiquement écologique (PPEE) et proposons un modèle global à base de PPC ainsi qu'une heuristique de recherche pour le résoudre efficacement. Le modèle proposé intègre les différents aspects inhérents au problème de planification dynamique dans un data center : machines physiques hétérogènes, types d'applications variés (i.e., applications interactives et applications par lots), opérations et coûts énergétiques de mise en route et d'extinction des machines physiques, interruption/reprise des applications par lots, consommation des ressources CPU et RAM des applications, migration des tâches et coûts énergétiques relatifs aux migrations, prédiction de la disponibilité d'énergie verte, consommation énergétique variable des machines physiques.

Mots clés

Programmation par contraintes, cloud computing, apprentissage machine, planification de tâches.

Abstract

Over the last decade, cloud computing technologies have considerably grown, this translates into a surge in data center power consumption. The magnitude of the problem has motivated numerous research studies around static or dynamic solutions to reduce the overall energy consumption of a data center. The aim of this thesis is to integrate renewable energy sources into dynamic energy optimization models in a data center. For this we use constraint programming as well as machine learning techniques. First, we propose a global constraint for tasks intersection that takes into account a ressource with variable cost. Second, we propose two learning models for the prediction of the workload of a data center and for the generation of such curves. Finally, we formalize the green energy aware scheduling problem (GEASP) and propose a global model based on constraint programming as well as a search heuristic to solve it efficiently. The proposed model integrates the various aspects inherent to the dynamic planning problem in a data center : heterogeneous physical machines, various application types (i.e., ractive applications and batch applications), actions and energetic costs of turning ON/OFF physical machine, interrupting/resuming batch applications, CPU and RAM ressource consumption of applications, migration of tasks and energy costs related to the migrations, prediction of green energy availability, variable energy consumption of physical machines.

Key Words

Constraint programming, Cloud computing, Machine learning, task scheduling.