Chapitre 9

Implémentation dans le solveur de contraintes choco

Nous présentons nos principales contributions au solveur de contraintes choco*. La plupart concernent le développement d'un module d'ordonnancement. Nous évoquerons brièvement d'autres contributions sur les problèmes de placement et certains mécanismes internes du solveur. Finalement, nous examinerons les résultats obtenus sur les problèmes d'ordonnancement par* choco *lors d'une compétition de solveurs. En complément, les annexes A et B décrivent respectivement des outils de développement et d'expérimentation et plusieurs cas d'utilisation.*

Sommaire

Le succès de la programmation par contraintes est dû à la simplicité de la modélisation grâce à un langage déclaratif de haut niveau, mais aussi à la disponibilité de solveurs de contraintes efficaces. L'architecture, l'implémentation et l'évaluation de ces systèmes relèvent encore du domaine de la re-

cherche [171] tant il est difficile de trouver un équilibre entre la simplicité, l'efficacité et la flexibilité. Par conséquent, nous discutons dans ce chapitre de nos principales contributions au solveur choco, évaluées dans les chapitres précédents et à la fin de celui-ci. Nous insisterons sur le langage déclaratif proposé pour l'ordonnancement sous contraintes, et nous évoquerons succinctement certains choix d'architecture et d'implémentation.

Le solveur choco est une librairie Java pour la programmation par contraintes. Il est basé sur un moteur de propagation réagissant à des évènements et des structures de données réversibles. Le solveur choco est un logiciel libre distribué sous une **licence BSD** et hébergé sur sourceforge.net.

L'organisation de ce chapitre est la suivante. Les sections 9.1 et 9.2 introduisent les deux éléments essentiels de choco que sont le Model et le Solver à travers le prisme de l'ordonnancement sous contraintes. La section 9.3 détaille les principales contraintes implémentées durant cette thèse et maintenant disponibles dans choco. La section 9.4 présente une procédure de construction automatique du graphe disjonctif généralisé (voir section 3.4) à partir d'un modèle de base défini par l'utilisateur. Finalement, la section 9.5 récapitule les résultats de l'édition 2009 de la *CSP solver competition* sur les problèmes d'ordonnancement. Ce chapitre est axé sur l'ordonnancement sous contraintes et nous invitons le lecteur à consulter http://choco.mines-nantes.fr pour de plus amples informations. La documentation officielle est disponible à cet adresse.

9.1 Le modèle

Le Model, accompagné du Solver, est l'un des éléments fondamentaux de choco. Il permet de décrire simplement un problème de manière déclarative. Son rôle principal est d'enregistrer les variables et contraintes d'un modèle. Nous introduisons dans cette section les variables et contraintes utiles à la modélisation de problèmes d'ordonnancement ou de placement en une dimension.
On accède généralement à l'API pour construire un Model par un *import statique*.

```
1  import static choco.Choco.*;
   Model model = new CPModel(); //Modèle PPC
```

9.1.1 Variables

Une variable est définie par son type (entier, réel, ensemble), un nom et les valeurs de son domaine. On peut spécifier certaines options pour imposer une représentation du domaine dans le Solver, par exemple un domaine borné ou énuméré pour une variable entière. Un choix judicieux des types des domaines en fonction de leur taille et des contraintes du modèle portant sur leur variable a une influence critique sur l'efficacité du solveur. Une combinaison de variables entières par des opérateurs (arithmétiques et logiques) est représentée par la classe IntegerExpressionVariable. On ajoute explicitement des variables au modèle grâce aux méthodes :

```
1  addVariable(String options, Variable v);
   addVariables(String options, Variable... v);
```

Nous rappelons la signification de deux options pour les variables entières pour l'optimisation et l'ordonnancement :
Options.V_OBJECTIVE déclare une Variable comme objectif (optimisation).
Options.V_MAKESPAN déclare une IntegerVariable comme la date d'achèvement de l'ordonnancement, c'est-à-dire de *toutes* les tâches du solveur.
L'interface de création des variables simples est détaillée dans la documentation officielle.

La classe MultipleVariables représente une variable complexe, c'est-à-dire un conteneur d'objets Variable à qui sont associés une sémantique. Une tâche non préemptive définie en section 3.1 est représentée par un objet TaskVariable héritant de la classe MultipleVariables. Sa sémantique est définie dans l'interface ITaskVariable qui donne accès à ces variables entières (IntegerVariable) : sa date de début (start), sa date de fin (end) et sa durée positive (duration). Une tâche non préemptive est dite consistante si ses variables entières vérifient la relation : start + duration = end.
Nous avons justifié empiriquement en section 3.1 le choix d'utiliser trois variables et non deux (par

exemple `start` et `duration`). Une limitation de `choco` réduit encore l'intérêt d'utiliser une représentation à deux variables. En effet, la modélisation de la troisième variable par une expression arithmétique (`IntegerExpressionVariable`) est insuffisante, car ces objets ne peuvent pas appartenir au scope d'une contrainte globale. Dans ce cas, un utilisateur doit, lui-même, définir, gérer l'unicité et assurer la consistance de la troisième variable ce qui réduit la lisibilité du modèle et augmente le risque d'erreur. Le tableau 9.1 décrit l'API pour la création d'une ou plusieurs `TaskVariable` en précisant explicitement les variables entières ou les fenêtres de temps des tâches. *Nous supposons que l'origine des temps est l'instant $t = 0$ et que les durées des tâches sont positives. Il n'y a aucune garantie sur la résolution d'un modèle ne respectant pas ces hypothèses.*

return type : `TaskVariable`
`makeTaskVar(String name, IntegerVariable start, IntegerVariable end, IntegerVariable duration, String... options)`
`makeTaskVar(String name, IntegerVariable start, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int binf, int bsup, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int binf, int bsup, int duration, String... options)`
`makeTaskVar(String name, int bsup, IntegerVariable duration, String... options)`
`makeTaskVar(String name, int bsup, int duration, String... options)`
return type : `TaskVariable[]`
`makeTaskVarArray(String prefix, IntegerVariable[] starts, IntegerVariable[] ends, IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name, int binf[], int bsup[], IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name,int binf, int bsup, IntegerVariable[] durations, String... options)`
`makeTaskVarArray(String name, int binf, int bsup, int[] durations, String... options)`
return type : `TaskVariable[][]`
`makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[][] durations, String... options)`
`makeTaskVarArray(String name, int binf, int bsup, int[][] durations, String... options)`

TABLE 9.1 – Méthodes pour la création d'une ou plusieurs tâches (`TaskVariable`).

9.1.2 Contraintes

Pour un solveur non commercial, `choco` propose un catalogue étoffé de contraintes simples et globales. Une contrainte porte sur une ou plusieurs variables et restreint les valeurs qu'elles peuvent prendre simultanément en fonction d'une relation logique. Le scope d'une contrainte simple contient un nombre prédéterminé de variables contrairement à celui des contraintes globales. Une contrainte globale représente un sous-problème pour lequel elle réalise généralement des réductions de domaine supplémentaires par rapport à un modèle décomposé logiquement équivalent. De plus, l'utilisateur peut facilement définir de nouvelles contraintes génériques ou dédiées à un problème. L'API offre différentes classes abstraites dont peuvent hériter les contraintes utilisateurs.
On poste une contrainte dans le modèle par le biais d'une des méthodes de la classe `Model` :

```
1 addConstraint(Constraint c);
  addConstraints(String options, Constraint... c);
```

L'ajout d'une contrainte au modèle ajoute automatiquement toutes ses variables, c'est-à-dire il n'est pas nécessaire de les ajouter explicitement. Les *options* disponibles pour une contrainte dépendent de son type et déterminent, par exemple, le ou les algorithmes de filtrage appliqués pendant la propagation.

Au début de cette thèse, la notion de tâche n'existait pas et la seule contrainte d'ordonnancement était une contrainte `cumulativeMax` définie sur des variables entières et avec des hauteurs positives. Nous

donnons ici une liste des principales contraintes implémentées dans le cadre de cette thèse :
Temporelles `precedence`, `precedenceDisjoint`, `precedenceReified`, `precedenceImplied`, `disjoint`.
Partage de ressource `cumulative`, `disjunctive`.
Allocation de ressource `useResources`.
Sac à dos `pack`.
Ensemblistes `min`, `max` (voir documentation officielle).
La section 9.3 donne une description détaillée des APIs et algorithmes de filtrage.
La liste des contraintes actuellement disponibles en `choco` est accessible par la javadoc. Certaines contraintes développées parallèlement par d'autres contributeurs traitent de problématiques voisines. On peut citer les contraintes `equation` et `knapsackProblem` [73] pour les problèmes de sac à dos, la contrainte `geost` [172] pour les problèmes de partage de ressource ou de placement géométrique et enfin `regular`, `costRegular` [173] et `multiCostRegular` [174] pour les problèmes d'emploi du temps.

9.2 Le solveur

Les fonctionnalités principales offertes par l'interface `Solver` sont la lecture du `Model` et la configuration d'un algorithme de recherche.

9.2.1 Lecture du modèle

Il est fortement recommandé de lire le modèle une seule fois après sa définition complète. La lecture du modèle est décomposée en deux étapes, la lecture des variables et la lecture des contraintes. La création d'un `Solver` PPC et la lecture d'un modèle sont réalisées de la manière suivante.

```
1 Solver solver = new CPSolver();
solver.read(model);
```

9.2.1.1 Lecture des variables

Le solveur parcourt les variables du modèle et construit les variables et domaines spécifiques du solveur. Le solveur utilise trois types de variables simples, les variables entières (`IntVar`), ensemblistes (`SetVar`) et réelles (`RealVar`). Le type de domaine est déterminé en fonction des options ou automatiquement.

Les variables du modèle et du solveur sont des entités distinctes. Les variables du solveur sont une représentation des variables du modèle. Ainsi, la notion de valeur d'une variable est définie uniquement dans le solveur. On accède à la variable du solveur à partir d'une variable du modèle grâce à la méthode du `Solver` : `getVar(Variable v)`. Par exemple, on accède à une tâche du solveur par la méthode `getVar( TaskVariable v)` qui renvoie un objet `TaskVar`.

```
1   Model model = new CPModel();
    TaskVariable t = makeTaskVat("T", 0, 100, 5); // model variable
    model.addvariable(t);
    Solver solver = new CPSolver();
5   solver.read(model);
    TaskVar tOnSolver = solver.getVar(x); // solver variable
```

La classe `TaskVar` implémente les interfaces suivantes :
- `Var` donne accès au réseau de contraintes et à l'état de la variable.
- `ITaskVariable` donne accès aux variables entières de la tâche.
- `ITask` donne accès à la durée et la fenêtre de temps de la tâche.

Le tableau 9.2 donne un aperçu des méthodes publiques de la classe `TaskVar`.

9.2.1.2 Lecture des contraintes

Après la lecture des variables, le solveur parcourt les contraintes du modèle pour créer les contraintes du solveur qui encapsulent des algorithmes de filtrage réagissant à différents évènements sur les variables

Méthode	Description
`isInstantiated()`	indique si la tâche est ordonnancée (`Var`)
`start(), end(), duration()`	renvoient ses variables entières (`ITaskVariable`)
`getEST(), getLST(), getECT(), getLCT()`	renvoient les informations sur la fenêtre de temps (`ITask`)
`getMinDuration(), getMaxDuration()`	renvoient les informations sur la durée (`ITask`)

TABLE 9.2 – Aperçu des méthodes publiques de la classe `TaskVar`.

(modifications des domaines) ou du solveur (propagation). Les contraintes du solveur sont ajoutées à son réseau de contraintes.

La consistance de tâche est assurée si la tâche appartient au moins à une contrainte de partage de ressource (`cumulative` ou `disjunctive`). Dans le cas contraire, le solveur ajoute automatiquement une contrainte arithmétique assurant la consistance de tâche.
La variable représentant le `makespan` peut être définie par l'utilisateur ou créée automatiquement si nécessaire (en présence de contraintes de partage de ressource). La consistance du `makespan` est assurée par l'ajout automatique de la contrainte : `makespan` $= \max_{\texttt{tasks}}(\texttt{tasks}[i].\texttt{end})$ où `tasks` contient toutes les tâches du solveur.

L'option `Options.S_MULTIPLE_READINGS` du `Solver` indique que le modèle est lu plusieurs fois. Cette situation se produit par exemple quand on veut propager itérativement des contraintes. Dans ce cas, les contraintes de consistance de tâche et du `makespan` doivent être ajoutées explicitement grâce à un unique appel aux méthodes `CPSolver.postTaskConsistencyConstraints()` et `CPsolver.postMakespanConstraint()`.

9.2.2 Stratégie de branchement

Un ingrédient essentiel d'une approche PPC est une stratégie de branchement adaptée. Au cours de cette thèse, nous avons longuement discuté des algorithmes de recherche et stratégies de branchement utilisés pour résoudre les problèmes d'atelier, et plus généralement les problèmes d'ordonnancement disjonctif, mais aussi des problèmes de fournées et placement. En `choco`, la construction de l'arbre de recherche repose sur une séquence d'objets de branchement qui jouent le rôle des *goals* intermédiaires en programmation logique. L'utilisateur peut définir la séquence d'objets de branchement utilisée lors de la construction de l'arbre de recherche. Outre son large catalogue de contraintes, `choco` propose de nombreuses stratégies de branchement et heuristiques de sélection pour tous les types de variables simples (entière, tâche, réelle) et complexes (tâches). Un axe important de son développement consiste à élaborer des interfaces et classes abstraites pour que les utilisateurs soient capables de développer rapidement leurs propres stratégies et heuristiques. Ces points sont abondamment discutés dans la documentation et nous ne reviendrons pas dessus.

À l'exception notable de la stratégie de branchement de Brucker *et al.* [104] pour les problèmes d'atelier, toutes les stratégies de branchement et heuristiques de sélection présentées dans cette thèse sont maintenant disponibles dans `choco`.

Par souci de simplicité et de concision, nous ne rentrerons pas dans les détails de leur implémentation mais nous mentionnerons néanmoins certains aspects importants. Nos principales contributions concernent la stratégie de branchement *setTimes*, l'heuristique de sélection de disjonction *profile* introduites en section 6.1.4 ainsi que toutes les heuristiques basées sur les degrés (voir sections 2.3 et 7.1.2). Nous proposons des heuristiques de sélection de disjonction et d'arbitrage basées sur la fonction $preserved(x \leq y)$ entre deux variables temporelles proposée par Laborie [122]. Ces dernières n'ont pas été évaluées pendant cette thèse, car nous avons eu cette idée pendant la rédaction du manuscrit.
Plusieurs heuristiques d'arbitrage basées sur l'interface `OrderingValSelector` sont compatibles avec toutes les heuristiques de sélection de disjonction : *minVal*, *random*, *centroid*, *minPreserved* et *maxPreserved*.
Le constructeur de la classe `SetTimes` accepte un objet implémentant l'interface `TaskVarSelector` ou `Comparator<ITask>` qui détermine l'ordre d'exploration des branches (on ordonnance au plus tôt une tâche différente dans chaque branche).
Les heuristiques basées sur les domaines et les degrés sont nombreuses : $(dom \mid slack \mid preserved)/(deg$

| *ddeg* | *wdeg*). Pour chacune, nous proposons le choix entre une stratégie de branchement par *standard labeling* binaire ($x = v \vee x \neq v$) ou n-aire (une valeur différente dans chaque branche) qui influence l'apprentissage des poids. Ainsi, leur implémentation est relativement complexe. Des résolutions successives d'un même modèle par ces heuristiques sont d'ailleurs un test intéressant pour vérifier le déterminisme d'un solveur. Leur implémentation repose sur l'observation suivant : l'évaluation de la condition logique $dom_1 \div deg_1 < dom_2 \div deg_2$ est beaucoup plus lente que celle de la condition équivalente $dom_1 \times deg_2 < dom_2 \times deg_1$. L'implémentation évite soigneusement cet écueil en utilisant des objets héritant de la classe abstraite `AbstractIntVarRatioSelector`. Ensuite, à chaque point de choix, le calcul du degré pondéré de toutes les variables non instanciées est très coûteux. Nous proposons encore des variantes pour les heuristiques (*dom* | *slack* | *preserved*)/*wdeg* où le calcul du degré pondéré est incrémental, approximatif (centré sur les décisions de branchement) et encore expérimental. Le formalisme autour de ces variantes n'est pas complètement défini mais elles obtiennent de très bons résultats expérimentaux.

La classe `BranchingFactory` facilite l'utilisation de toutes ces heuristiques en proposant des méthodes pour la construction d'objets de branchement (par défaut ou personnalisé) sans se soucier des objets intermédiaires. La lecture de son code source constitue une base d'exemples pour mettre en place ses propres « recettes ».

9.2.3 Redémarrages

Nous montrons comment régler les redémarrages et enregistrer des *nogoods* (voir section 6.2.3). Les redémarrages sont intéressants quand la stratégie de recherche intègre des éléments d'apprentissage (*wdeg* et *impact*), ou de randomisation. Les méthodes suivantes appartiennent à la classe `Solver`.
On actionne les redémarrages après la découverte d'une solution (optimisation) :

```
setRestart(boolean restart);
```

On actionne la stratégie de Walsh :

```
setGeometricRestart(int base, double grow);
setGeometricRestart(int base, double grow, int restartLimit);
```

On actionne la stratégie de Luby :

```
setLubyRestart(int base);
setLubyRestart(int base, int grow);
setLubyRestart(int base, int grow, int restartLimit);
```

Le paramètre `base` est le facteur d'échelle, `grow` est le facteur géométrique et `restartLimit` est une limite sur le nombre de redémarrages. On peut configurer ces paramètres en passant directement par l'objet `Configuration` (héritant de la classe `Properties`) du `Solver`.
Par défaut, la longueur d'une exécution dépend du nombre de backtracks mais il est possible d'utiliser une autre mesure (limite, temps, nœuds ...) en modifiant directement la valeur de la propriété `Configuration.RESTART_POLICY_LIMIT`.
Le listing suivant donne un exemple d'activation des redémarrages :

```
CPSolver s = new CPSolver();
s.read(model);
s.setLubyRestart(50, 2, 100);
//s.setGeometricRestart(14, 1.5d);
     s.setFirstSolution(true);
s.attachGoal(BranchingFactory.domWDeg(s));
s.generateSearchStrategy();
s.launch();
```

L'enregistrement des *nogoods* lors d'un redémarrage est déclenché en modifiant la valeur de la propriété `Configuration.NOGOOD_RECORDING_FROM_RESTART` ou par un appel à la méthode :

```
setRecordNogoodFromRestart(boolean recordNogoodFromRestart);
```

À l'heure actuelle, l'enregistrement des *nogoods* est limité aux branchements portant sur des variables booléennes. Plus précisément, la dernière branche explorée avant le redémarrage est tronquée au premier point de choix portant sur une variable non booléenne. Supposons que notre stratégie de branchement soit composée de deux objets de branchement B_{bool} et B_{int} portant respectivement sur des variables booléennes et entières. Il est possible d'enregistrer des *nogoods* pour la séquence de branchement B_{bool}, B_{int} mais pas pour la séquence B_{int}, B_{bool}.

Notes sur l'implémentation des redémarrages et des nogoods Les *nogoods* sont enregistrés avant un redémarrage. Pour que les *nogoods* soient complets, il est souhaitable que les redémarrages aient lieu avant la création d'un nouveau nœud. Cependant, le mécanisme doit rester assez flexible pour admettre des requêtes de redémarrage provenant de sources diverses. Cette étape a engendré deux modifications majeures du mécanisme interne de `choco` : (a) la boucle de recherche (b) la gestion des limites. Un redémarrage est un mouvement dans la boucle de recherche. On peut le déclencher brutalement en (a) modifiant directement le champ public représentant le prochain mouvement ou (b) déclenchant une `ContradictionException` particulière. Cependant, il est préférable de définir sa propre stratégie (universelle) grâce à l'interface `UniversalRestartStrategy`. Le gestionnaire de limites applique alors cette stratégie de manière sûre et efficace.

9.2.4 Résolution d'un modèle

Les méthodes du tableau 9.3 lancent la résolution d'un problème. Historiquement, `choco` a toujours utilisé une procédure d'optimisation top-down (voir section 2.4). Nous proposons depuis peu d'autres procédures d'optimisation (bottom-up et destructive-lower-bound) et un mécanisme basique de *shaving* (voir section 2.2.4). Ces nouvelles procédures peuvent être activées dans l'objet `Configuration`. Cependant, leur implémentation n'est pas aussi éprouvée que celle de top-down.

Méthode `Solver`	Description
`solve()`	calcule la première solution du modèle, si le modèle est satisfiable.
`solveAll()`	calcule toutes les solutions du modèle, si le modèle est satisfiable.
`minimize(Var obj, boolean restart )`	calcule une solution minimisant la variable objectif `obj`.
`maximize(Var obj, boolean restart )`	calcule une solution maximisant la variable objectif `obj`.
`isFeasible()`	indique le statut du modèle, c'est-à-dire si le modèle est satisfiable, insatisfiable ou si la résolution n'a pas commencé.
`propagate()`	propage les contraintes du modèle jusqu'à ce qu'un point fixe soit atteint ou déclenche une `ContradictionException` si une inconsistance est détectée.

TABLE 9.3 – Méthodes de résolution du `Solver`.

9.3 Contraintes de choco

Dans cette section, nous donnons une description détaillée des contraintes intégrées dans `choco` au cours de cette thèse. La contrainte `sequenceEDD` définie au chapitre 8 n'est pas livrée dans `choco`, car nous la jugeons trop dédiée à notre problème de fournées.

L'implémentation des contraintes de partage de ressource `cumulative` et `disjunctive` repose sur la classe abstraite `AbstractResourceSConstraint` qui définit l'ordre des variables et assure la consistance de tâche. Une tâche `TaskVar` est représentée dans une contrainte de partage de ressource par un objet implémentant l'interface `IRTask` qui donne accès aux autres variables associées à l'exécution de la tâche

sur la ressource (hauteur, usage) et permet de modifier leur domaine de manière transparente, c'est-à-dire sans se soucier de la contrainte représentant la ressource (gestion du réseau de contraintes et de la propagation). Nous rappelons que toutes les tâches d'une ressource régulière sont obligatoires alors que certaines tâches sont optionnelles sur une ressource alternative. Une tâche optionnelle est dite *effective* lorsque la ressource lui a été allouée (une tâche obligatoire est toujours effective).

L'implémentation des contraintes temporelles repose sur la classe abstraite `AbstractPrecedenceSConstraint` qui implémente l'interface `ITemporalRelation`. Elle impose un ordre des variables ainsi qu'une gestion commune des évènements lors de la propagation. Les heuristiques de sélection basées sur le degré pondéré acceptent indifféremment des variables entières ou des objets implémentant l'interface `ITemporalRelation`. Des contraintes temporelles dont le scope ne contient que des variables entières aussi conservées pour assurer la compatibilité avec les versions antérieures de `choco`.

Un module de visualisation des contraintes est disponible par le biais de la fabrique `ChocoChartFactory`. Ce module est basé sur la librairie `jfreechart` pour visualiser les contraintes `cumulative`, `disjunctive` et `pack`. La construction du graphe disjonctif généralisé (voir section 9.4) permet une visualisation du sous-réseau des contraintes `precedence` et `precedenceDisjoint` basée sur la libraire `Graphviz`.

9.3.1 cumulative

Signature `cumulative(String name, TaskVariable[] tasks,IntegerVariable[] heights, IntegerVariable [] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String ... options)`
`tasks` sont les tâches exécutées sur la ressource (recommandé $n \geq 3$).
`heights` sont les hauteurs des tâches . Les hauteurs négatives sont autorisées (producteur-consommateur)
`usages` indiquent si les tâches sont *effectives* sur la ressource (booléens).
`consumption` est la consommation minimale requise sur la ressource.
`capacity` est la capacité de la ressource (recommandé ≥ 2).
`uppBound` est l'horizon de planification pour la ressource (obligatoire ≥ 0).

Description Nous modifions légèrement la définition de la fonction de Dirac $\delta_i(t)$ des moments $\mathcal{M}_i$ d'une tâche préemptive `tasks`$[i]$ introduite en section 3.1 pour refléter son exécution *effective* sur la ressource :

$$\delta_i(t) = \begin{cases} \texttt{usages}[i] & \text{si } \texttt{tasks}[i].\texttt{start} \leq t < \texttt{tasks}[i].\texttt{end} \\ 0 & \text{sinon} \end{cases}$$

La contrainte `cumulative` impose que la hauteur totale des tâches en cours d'exécution à chaque instant t soit inférieure à la capacité de la ressource (9.1) et supérieure à la consommation minimale lorsqu'au moins une tâche s'exécute (9.2). Elle assure aussi la consistance des tâches non préemptives (9.3). Finalement, elle vérifie la compatibilité des dates de fin des tâches effectives avec son horizon (9.4).

$$\sum_{\texttt{tasks}} \delta_i(t) \times \texttt{heights}[i] \leq \texttt{capacity} \quad (\forall \text{ instant } t) \tag{9.1}$$

$$\sum_{\texttt{tasks}} \delta_i(t) \times \texttt{heights}[i] \geq \texttt{consumption} \times \max_{\texttt{tasks}}(\delta_i(t)) \quad (\forall \text{ instant } t) \tag{9.2}$$

$$\texttt{tasks}[i].\texttt{start} + \texttt{tasks}[i].\texttt{duration} = \texttt{tasks}[i].\texttt{end} \quad (\forall \text{ tâche } i) \tag{9.3}$$

$$\texttt{usages}[i] \times \texttt{tasks}[i].\texttt{end} \leq \texttt{uppBound} \quad (\forall \text{ tâche } i) \tag{9.4}$$

Algorithme de filtrage La section 3.3.2 décrit plus précisément les algorithmes de filtrage évoqués ci-dessous. L'algorithme de filtrage par défaut (on ne peut pas le désactiver) est une implémentation fidèle de l'algorithme à balayage proposé par Beldiceanu et Carlsson [54] qui gère les hauteurs négatives et les tâches optionnelles. Cependant, nous supprimons les règles de filtrage associées à leur hypothèse sur l'allocation de ressource : il existe plusieurs ressources cumulatives et chaque tâche doit être exécutée sur exactement une de ces ressources. En `choco`, l'allocation des ressources est gérée par des contraintes `useResources`.

Lorsque les hauteurs sont positives, les options permettent d'appliquer des algorithmes de filtrage supplémentaires (*task intervals* ou *edge finding*) pour la satisfaction de la contrainte de capacité (9.1). Actuellement, ces algorithmes supplémentaires ne raisonnent pas sur la contrainte de consommation minimale et ignorent purement et simplement les tâches optionnelles.

Options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut uniquement : algorithme à balayage [54]
`C_CUMUL_TI` implémentation « maison » des raisonnements énergétiques (*task intervals* ou *overload checking*) en $O(n \log n)$ basée sur une variante des $\Theta - tree$ de Vìlim.
`C_CUMUL_STI` implémentation classique des raisonnements énergétiques en $O(n^2)$. Les inférences sont plus fortes que précédemment puisque le calcul des énergies intègre les tâches à cheval sur un intervalle de temps (sur la droite).
`C_CUMUL_EF` implémentation de l'*edge finding* en $O(n^2)$ basée sur l'algorithme CalcEF proposé par Mercier et Hentenryck [56] (le calcul de R est paresseux).
`C_CUMUL_VEF` implémentation de l'*edge finding* (*en développement*) pour une ressource alternative proposée par Vilím [57, 175].

Notes sur l'API On peut éventuellement nommer une ressource pour faciliter la lecture du modèle, par exemple avec `Solver.pretty()`.
Le nombre de tâches optionnelles est défini par la longueur du tableau `usages` (`usages.length` $\leq$ `tasks.length`). Si le paramètre `usages` est `null`, alors la ressource est régulière. En fait, les `usages.length` dernières tâches du tableau `tasks` sont optionnelles et les `tasks.length` − `usages.length` premières sont obligatoires. On peut bien sûr ignorer ce mécanisme en mélangeant les variables booléennes et constantes dans le tableau `usages`. Cependant, il est conseillé de respecter cette convention, car l'implémentation de l'interface `IRTask` est différente selon que la tâche soit obligatoire ou optionnelle.
Si l'utilisateur ne précise pas d'horizon de planification pour la ressource (`uppBound` = `null`) alors on le remplace par la variable `makespan` représentant la date d'achèvement de l'ordonnancement, en la créant automatiquement si nécessaire.
Les méthodes `cumulativeMax` considèrent que la consommation minimale est nulle (`consumption` = `null` ou 0). Les méthodes `cumulativeMin` considèrent que la capacité est infinie (`capacity` = `null`).

Exemple Un exemple d'utilisation de la contrainte `cumulative` est présenté en section B.2.

9.3.2 disjunctive

Signature `disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)`
`tasks` sont les tâches exécutées sur la ressource (recommandé $n \geq 3$).
`usages` indiquent si les tâches sont *effectives* sur la ressource (booléens).
`uppBound` est l'horizon de la ressource (obligatoire ≥ 0). .

Description La contrainte `disjunctive` est un cas particulier de la contrainte `cumulative` où les hauteurs et la capacité sont unitaires alors que la consommation minimale requise est nulle. La ressource n'exécute au plus une tâche à chaque instant, c'est-à-dire que les tâches ne se chevauchent pas.

Algorithme de filtrages Les règles de filtrage décrites en section 3.3.1 sont activées par les options. Nous avons implémenté les algorithmes de filtrage proposés par Vilím [43], Vilím *et al.* [44], Kuhnert [52], Vilím [150], Vilím *et al.* [176] qui changent selon le caractère régulier ou alternatif de la ressource.
Une règle atteint un point fixe local lorsqu'elle ne peut plus réaliser de déduction (ceci peut nécessiter plusieurs exécutions de la règle). La propagation des règles dans la contrainte doit atteindre un point fixe global qui est atteint lorsque toutes les règles ont atteint leur point fixe local.
Vilìm a proposé un algorithme de propagation où les règles atteignent successivement leur point fixe local jusqu'à ce que le point fixe global soit atteint dans la boucle principale. La boucle principale par défaut applique chaque règle une seule fois (une règle n'atteint pas forcément son point fixe local) jusqu'à ce que

le point fixe global soit atteint. Ce choix est motivé par des raisons spécifiques à notre implémentation, car il tente de réduire le nombre d'opérations de tri des arbres binaires équilibrés maintenus dans la contrainte.

options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut : `C_DISJ_NFNL, C_DISJ_DP, C_DISJ_EF`.
`C_DISJ_OC` *overload checking* en $O(n \log n)$.
`C_DISJ_NFNL` *not first/not last* en $O(n \log n)$.
`C_DISJ_DP` *detectable precedence* en $O(n \log n)$.
`C_DISJ_EF` *edge finding* en $O(n \log n)$.
`C_DISJ_VF` applique l'algorithme de propagation des règles de Vilìm.

Notes sur l'API La sémantique des paramètres de l'API est identique à celle de la contrainte `cumulative`.
La méthode `Reformulation.disjunctive(TaskVariable[] clique, String... boolvarOptions)` décompose la contrainte en une clique de disjonctions grâce à `precedenceDisjoint` : $\texttt{clique}[i] \simeq \texttt{clique}[j],\ \forall\, i < j$.
La méthode `forbiddenIntervals(String name, TaskVariable[] tasks)` poste une contrainte globale appliquant la règle des intervalles interdits proposée par Guéret et Prins [120] (voir section 6.1.3).

9.3.3 useResources (en développement)

Signature `useResources(TaskVariable task, int k, Constraint... resources)`
`task` est la tâche à laquelle on alloue des ressources.
`k` est le nombre de ressources nécessaire à la tâche (obligatoire > 0).
`resources` sont les ressources à allouer.

Description La contrainte `useResources` représente l'allocation d'exactement ou au moins k ressources de l'ensemble `resources` à la tâche `task`. La représentation des tâches optionnelles dans les ressources alternatives `cumulative` et `disjunctive` par des variables booléennes `usages` permet de gérer l'allocation de ressources via des contraintes booléennes. Cependant, l'inconvénient des contraintes d'allocation booléennes est que les fenêtres de temps ne sont pas réduites avant une allocation effective. En effet, supposons qu'une tâche requiert deux ressources sur trois ressources disponibles mais qu'une seule ressource soit en fait disponible à sa date de début au plus tôt (les deux autres ressources sont occupées à cet instant), alors ni la contrainte booléenne d'allocation, ni les contraintes de ressource ne peuvent supprimer la date de début au plus tôt du domaine qui est pourtant inconsistante quelle que soit l'allocation.
Pour pallier cet inconvénient, nous introduisons une nouvelle contrainte *transversale*, c'est-à-dire que son scope contient aussi des contraintes et non uniquement des variables. Nous donnons plus de détails sur la définition de `useResources` et ses règles de filtrage en annexe D. Cette représentation offre une grande flexibilité par rapport aux contraintes multi-ressources, car on peut spécifier différentes demandes d'allocation sur un ensemble de ressources (une tâche requiert deux ressources et une autre requiert seulement une ressource parmi quatre) ou pour une tâche (une tâche nécessite deux machines, deux opérateurs et quatre travailleurs).
Notez que certains aspects de cette contrainte transversale sont difficiles à implémenter proprement dans choco. C'est pourquoi, pour l'instant, `useResources` est disponible dans `choco.ChocoContrib` mais pas incluse dans la livraison principale de choco.

Algorithme de filtrage Chaque tâche optionnelle d'une ressource alternative raisonne sur une fenêtre de temps *hypothétique*, c'est-à-dire en supposant qu'elle est effective sur la ressource (sans se soucier des autres contraintes). Cette fenêtre de temps est cohérente avec la fenêtre de temps principale de la tâche définie par les domaines de ses variables entières. Si la fenêtre de temps hypothétique devient vide, alors la tâche ne peut pas être exécutée sur la ressource. La contrainte `useResources` propage d'abord la contrainte booléenne d'allocation avant de réduire la fenêtre de temps principale en raisonnant sur les fenêtres de temps hypothétiques et l'allocation courante.

Notes sur l'API Remarquez que les contraintes `useResources` utilisent par défaut l'option `Options.C_POST_PONED`, car leur construction nécessite que toutes les contraintes de partage de ressource aient été préalablement créées dans le solveur.

9.3.4 precedence

Signature `precedence(TaskVariable t1, TaskVariable t2, int delta)`
`t1, t2` sont les tâches de la précédence.
`delta` est le temps d'attente entre les tâches.

Description La contrainte `precedence` représente une contrainte de précédence avec temps d'attente $\texttt{t1} \prec \texttt{t2}$ entre deux tâches non préemptives introduite en section 3.2 :

$$\texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} \qquad (\texttt{t1} \prec \texttt{t2})$$

Algorithme de filtrage On délègue au `Solver` le soin de poser la contrainte la mieux adaptée pour cette relation arithmétique. Ce choix dépend d'autres options globales du `Solver` comme `Options.E_DECOMP`. Cependant, les observations montrent qu'on applique généralement la consistance de bornes.

Exemple Un exemple d'utilisation de la contrainte `precedence` est présenté en section B.1.

9.3.5 precedenceDisjoint

Signature `precedenceDisjoint(TaskVariable t1, TaskVariable t2, IntegerVariable direction, int fs, int bs)`
`t1, t2` sont deux tâches en disjonction.
`dir` indique l'ordre relatif des deux tâches (booléen).
`fs, bs` sont les temps d'attente pour chaque arbitrage de la disjonction.

Description La contrainte `precedenceDisjoint` représente une contrainte réifiée de disjonction avec temps d'attente $\texttt{t1} \sim \texttt{t2}$ entre deux tâches non préemptives introduite au chapitre 7 :

$$\begin{aligned} \texttt{dir} = 1 \quad &\Rightarrow \quad \texttt{t1.end} + \texttt{fs} \leq \texttt{t2.start} \qquad (\texttt{t1} \prec \texttt{t2}) \\ \texttt{dir} = 0 \quad &\Rightarrow \quad \texttt{t2.end} + \texttt{bs} \leq \texttt{t1.start} \qquad (\texttt{t2} \prec \texttt{t1}) \end{aligned}$$

Algorithme de filtrage Dans un souci d'efficacité, le `Solver` utilise différentes implémentations selon la nature des durées des tâches (fixes ou non) et de l'existence de temps d'attente. Le filtrage applique les règles précédence interdite (PI), précédence obligatoire (PO) ou la consistance de borne sur un arbitrage. L'algorithme de filtrage réagit aux évènements en temps constant :

1. Si une modification d'une fenêtre de temps déclenche une des règles (PI) ou (PO), alors la variable `dir` est instanciée selon l'arbitrage déduit.
2. Après l'instanciation de la variable `dir`, on réduit les fenêtres de temps par consistance de borne jusqu'à ce que la précédence soit satisfaite.

Notez qu'une contradiction est déclenchée lorsqu'aucun séquencement n'est possible.

Exemple Un exemple d'utilisation de la contrainte `precedenceDisjoint` est présenté en section B.1.

9.3.6 precedenceReified

Signature `precedenceReified(TaskVariable t1, int delta, TaskVariable t2, IntegerVariable b)`
`t1, t2` sont les deux tâches de la précédence réifiée.
`delta` est temps d'attente entre les tâches.
`b` indique si la précédence ou son opposé au sens arithmétique est actif (booléen).

Description La contrainte `precedenceReified` la réification de la précédence $\texttt{t1} \prec \texttt{t2}$ en fonction de la valeur de `b`.

$$\begin{array}{lcll} \texttt{b} = 1 & \Rightarrow & \texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} & (\texttt{t1} \prec \texttt{t2}) \\ \texttt{b} = 0 & \Rightarrow & \texttt{t1.end} + \texttt{delta} > \texttt{t2.start} & (\texttt{t1} \not\prec \texttt{t2}) \end{array}$$

Algorithme de filtrage On applique une variante de l'algorithme de filtrage de `precedenceDisjoint` où la détection et la propagation du cas $\texttt{b} = 0$ est modifiée.

9.3.7 precedenceImplied

Signature `precedenceImplied(TaskVariable t1, int delta, TaskVariable t2, IntegerVariable b)`
`t1, t2` sont les deux tâches de la précédence potentielle.
`delta` est temps d'attente entre les tâches.
`b` indique si la précédence est active ou non (booléen).

Description La contrainte `precedenceImplied` représente l'activation ou non de la précédence $\texttt{t1} \prec \texttt{t2}$ en fonction de la valeur de `b`.

$$\begin{array}{lcll} \texttt{b} = 1 & \Rightarrow & \texttt{t1.end} + \texttt{delta} \leq \texttt{t2.start} & (\texttt{t1} \prec \texttt{t2}) \\ \texttt{b} = 0 & \Rightarrow & \texttt{TRUE} & \end{array}$$

Algorithme de filtrage On applique une relaxation de l'algorithme de filtrage de `precedenceDisjoint`.

9.3.8 disjoint (décomposition)

Signature `disjoint(TaskVariable[] tasks1, TaskVariable[] tasks2)`
`tasks1, tasks2` sont deux ensembles de tâches en disjonction.

Description La contrainte `disjoint` impose que les paires de tâches composées d'une tâche de chaque ensemble soient en disjonction : $(\texttt{tasks1}[i] \simeq \texttt{tasks2}[j])$, c'est-à-dire qu'elles ne se chevauchent pas dans le temps. La méthode renvoie un objet de la classe `Constraint[]` qui modélise une décomposition de la contrainte globale `disjoint` en contraintes `precedenceDisjoint`. En effet, il n'existe aucune implémentation de cette contrainte globale, mais la cumulative colorée (indisponible dans `choco`) permet de reformuler efficacement cette contrainte en affectant une couleur différente à chaque ensemble de tâches et en imposant une capacité unitaire à la ressource. La cumulative colorée impose que le nombre de couleurs distinctes appartenant aux tâches s'exécutant à l'instant t reste inférieure à la capacité de la ressource.

9.3.9 pack

Signature `pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, Integer-ConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)`
`items` sont les articles à ranger dans les conteneurs.
`loads` sont les charges des conteneurs.
`bins` sont les conteneurs dans lesquels sont rangés les articles (≥ 0).
`sizes` sont les tailles (longueurs) des articles (constantes).
`nbNE` est le nombre de conteneurs non vide du rangement.

Description La contrainte `pack(items, loads, bins, sizes, nbNE)` impose qu'un ensemble d'articles soient rangés dans des conteneurs de manière à ce que la somme des tailles des articles dans un conteneur soit cohérente avec sa charge (9.5). La contrainte assure le respect des contraintes de liaison

entre les variables représentant le rangement (9.6), mais aussi la cohérence entre le nombre de conteneurs non vides et l'affectation des articles 9.7.

$$\texttt{loads}[b] = \sum_{i \in \texttt{items}[b]} \texttt{sizes}[i] \qquad (\forall \text{ conteneur } b) \tag{9.5}$$

$$\texttt{bins}[i] = b \iff i \in \texttt{items}[b] \qquad (\forall \text{ article } i)(\forall \text{ conteneur } b) \tag{9.6}$$

$$\text{card}\{b \mid \texttt{items}[b] \neq \emptyset\} = \texttt{nbNE} \tag{9.7}$$

Les deux restrictions principales sont le tri obligatoire des articles par taille décroissante et les tailles constantes des articles. En effet, l'implémentation n'est pas robuste à un tri des variables `bins` et `sizes` puisque les ensembles `items` référencent les articles par leur index dans ces tableaux. Un tri intempestif peut entraîner des effets de bords sur d'autres contraintes portant sur les variables ensemblistes. À terme, l'intégration de tailles variables ne devrait pas poser de problèmes mais elle peut compliquer voire empêcher l'application de certaines règles de filtrage, notamment `C_PACK_AR`. Cette contrainte générique permet de modéliser facilement des conteneurs de tailles hétérogènes ou des incompatibilités entre article/article ou article/conteneur.

Algorithme de filtrage Notre implémentation s'inspire librement de Shaw [70] puisque l'ajout des variables `items` et `nbNE` modifie la signature de la contrainte. Tout d'abord, la contrainte applique la propagation associée au *Typical model*, c'est-à-dire les contraintes (8.7) and (8.8) de la formulation mathématique présentée en section 8.4.3 page 88. Par ailleurs, Le filtrage applique des règles inspirées de raisonnements sur les problèmes de sac à dos (*additional propagation rules*) ainsi qu'une borne inférieure dynamique sur le nombre minimal de conteneurs non vides (*using a lower bound*).
Les raisonnements de type sac à dos consistent à traiter le problème plus simple du rangement des articles dans un seul conteneur : existe-t'il un rangement d'un sous-ensemble d'articles dans un conteneur b pour lequel la charge du conteneur appartient à un intervalle donné ? S'il n'existe aucun rangement d'un sous-ensemble d'articles dans le conteneur b pour lequel la charge du conteneur appartient au domaine de `loads`$[b]$, alors on peut déclencher un échec. Par ailleurs, si un article apparaît dans tous ces rangements, alors cet article est immédiatement rangé dans le conteneur. Au contraire, si un un article n'apparaît dans aucun rangement, alors cet article est éliminé des candidats pour ce conteneur. Ainsi, si aucun rangement réalisable dans un conteneur n'atteint une charge donnée, alors cette charge n'est pas réalisable. Shaw [70] a proposé des algorithmes efficaces ne dépendant pas des tailles et capacités, mais qui n'atteignent pas la consistance d'arc généralisée sur le problème *subset sum* à l'image de ceux proposés par Trick [73]. Ensuite, il adapte les bornes inférieures classiques pour ce problème pour les affectations partielles. Le principe consiste à transformer une affectation partielle en une nouvelle instance du problème de *bin packing*, puis d'appliquer une borne inférieure sur cette nouvelle instance.
Nous considérons aussi deux règles d'élimination dynamique des symétries qui sont fréquemment utilisées pour le problème de *bin packing*. D'autres règles d'élimination statique des symétries sont proposées dans la classe `PackModel` décrite ci-dessous.

options Les options sont des champs (`String`) de la classe `choco.Options`.
`NO_OPTION` filtrage par défaut : propagation du *Typical model* [70].
`C_PACK_AR` voir section *Additional propagation rules* [70].
`C_PACK_DLB` voir section *Using a lower bound* [70]. Nous remplaçons la borne inférieure L^{1D}_{MV} par L^{1D}_{CCM} [82] qui est dominante.
`C_PACK_FB` élimination des symétries : on range immédiatement un article dans un conteneur si l'article remplit complètement le conteneur.
`C_PACK_LBE` élimination des symétries : les conteneurs vides sont à la fin.

Notes sur l'API Un objet de la classe `PackModel` facilite la modélisation par le biais de ses nombreux constructeurs et de méthodes postant des contraintes d'élimination des symétries dont certaines sont décrites dans le tableau 9.4. Nous attirons l'attention du lecteur sur le fait que la validité de ces dernières dépend du problème traité et qu'on ne peut jamais les appliquer toutes simultanément.

Méthode *PackModel*	Description
`PackModel(int[], int , int)`	constructeur basique pour le problème de *bin packing*.
`packLargeItems()`	range les k plus grands articles dans les k premiers conteneurs. La valeur de k dépend des tailles et de de la capacité.
`allDiffLargeItems()`	contrainte redondante `allDifferent` sur les k plus grands articles.
`orderEqualSizedItems(int)`	ordonne les articles de taille identique dans les conteneurs.
`decreasingLoads(int)`	ordonne les conteneurs par charge décroissante.
`decreasingCardinalities(int)`	ordonne les conteneurs par nombre d'articles décroissant.

TABLE 9.4 – Principales méthodes de la classe `PackModel`.

Exemple Un exemple d'utilisation de la contrainte `pack` est présenté en section B.3.

9.3.10 Reformulation de contraintes temporelles

Le tableau 9.5 récapitule les méthodes de la classe `Choco` qui aident à la création et à la lisibilité de modèles utilisant des contraintes temporelles. Ces méthodes proposent une interface de reformulation de contraintes temporelles vers des contraintes arithmétiques. Le paramètre `delta` n'est pas obligatoire, c'est-à-dire il existe toujours une méthode du même nom dont la signature ne contient pas ce paramètre (la valeur est alors considérée nulle).

Méthode Choco.*	Reformulation
Contraintes sur les fenêtres de temps	
`endsAt(TaskVariable t, int time)`	$\texttt{t.end} = \texttt{time}$
`endsAfter(TaskVariable t, int ect)`	$\texttt{t.end} \geq \texttt{ect}$
`endsBefore(TaskVariable t, int lct)`	$\texttt{t.end} \leq \texttt{lct}$
`endsBetween(TaskVariable t, int ect, int lct)`	$\texttt{ect} \leq \texttt{t.end} \leq \texttt{lct}$
`startsAt(TaskVariable t, int time)`	$\texttt{t.start} = \texttt{time}$
`startsAfter(TaskVariable t, int est)`	$\texttt{t.start} \geq \texttt{est}$
`startsBefore(TaskVariable t, int lst)`	$\texttt{t.start} \leq \texttt{lst}$
`startsBetween(TaskVariable t, int est, int lst)`	$\texttt{est} \leq \texttt{t.start} \leq \texttt{lst}$
Contraintes d'enchaînement	
`startsBeforeBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} + \texttt{delta} \leq \texttt{t2.start}$
`startsAfterBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} \geq \texttt{t2.start} + \texttt{delta}$
`startsBeforeEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.start} + \texttt{delta} \leq \texttt{t2.end}$
`endsAfterBegin(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} \geq \texttt{t2.start} + \texttt{delta}$
`endsBeforeEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} + \texttt{delta} \leq \texttt{t2.end}$
`endsAfterEnd(TaskVariable t1, TaskVariable t2, int delta)`	$\texttt{t1.end} \geq \texttt{t2.end} + \texttt{delta}$
`endsBeforeBegin(TaskVariable t1, TaskVariable t2, int delta)`	`precedence(t1, t2, delta)`
`startsAfterEnd(TaskVariable t1, TaskVariable t2, int delta)`	`precedence(t2, t1, delta)`

TABLE 9.5 – Reformulation des contraintes temporelles.

9.4 Construction du modèle disjonctif

Nous décrivons une méthode générale pour la construction d'un modèle disjonctif généralisé $\mathcal{G}$ défini en section 3.4 à partir d'un modèle de base défini par l'utilisateur. Elle gère des contraintes hétérogènes provenant de sources différentes et potentiellement redondantes. Son intérêt est plus restreint dans le

cadre de problèmes académiques très structurés, car on peut alors appliquer les règles de reformulation et de déduction de manière *ad hoc*. Cependant, cette méthode a permis un gain de temps significatif lors de l'implémentation des modèles *Light* et *Heavy* du chapitre 7.
La construction du graphe disjonctif est activée par la valeur de la propriété booléenne `DISJUNCTIVE_MODEL_DETECTION` de la classe `PreProcessConfiguration`. La construction modulaire (on peut supprimer certaines étapes) du graphe disjonctif généralisé dépend d'autres propriétés de cette classe. La procédure assure l'unicité d'une contrainte temporelle entre deux tâches.

On distingue les opérations sur le graphe disjonctif de celles sur le modèle. Les opérations sur le graphe disjonctif ont la syntaxe suivante : `opération` élément(s) $\multimap$ ensemble(s). Les opérations sur le modèle (leur nom commence avec une majuscule) ont la syntaxe suivante : `Opération` élément(s).

9.4.1 Traitement des contraintes temporelles

La procédure buildDisjModT détecte les précédences issues des fenêtres, puis construit le graphe disjonctif en se basant sur les contraintes `precedence` et `precedenceDisjoint` du modèle. Après l'initialisation du graphe $\mathcal{G}$, on calcule un ensemble d'arcs fictifs $\mathcal{P}_0$ qui représente la relation d'ordre entre les fenêtres de temps des tâches : $(i,j) \in \mathcal{P}_0 \Leftrightarrow est_j - lct_i >= 0$. Cette étape modifie la valuation $\mathcal{L}$ sans modifier les arcs de $\mathcal{P}$. Ces arcs fictifs permettent d'éliminer les précédences satisfaites et les disjonctions arbitrées par les fenêtres de temps des tâches.

On parcourt d'abord les contraintes de précédence du modèle. Une contrainte `precedence` est supprimée du modèle lorsqu'elle est dominée par la valuation courante de $\mathcal{G}$. Dans le cas contraire, si l'arc est fictif, alors on ajoute simplement la contrainte dans $\mathcal{G}$. Si l'arc est déjà associé à une contrainte, on fusionne les contraintes pour mettre à jour la valuation et la contrainte dans $\mathcal{G}$. À la fin de cette boucle, le graphe de précédence est construit et nous calculons sa fermeture transitive qui va être utile dans le traitement des disjonctions. Pendant ce calcul, une sous-fonction lève une contradiction lorsqu'un cycle est détecté dans le graphe de précédence. Par convention, les arcs transitifs ont un temps d'attente nul.

On parcourt ensuite les contraintes de disjonction du modèle. Si T_j appartient à la fermeture transitive de T_i, la disjonction est déjà arbitrée. Si la précédence associée à cet arbitrage a un temps d'attente supérieur à celui existant dans $\mathcal{G}$, alors on distingue deux cas selon l'existence d'un arc dans $\mathcal{G}$. Si l'arc est fictif, on ajoute cette nouvelle précédence dans le modèle et dans $\mathcal{G}$. Sinon, on fusionne la nouvelle précédence avec celle de $\mathcal{G}$. La disjonction est alors supprimée du modèle et sa variable booléenne est instanciée en fonction de l'arbitrage déduit. L'élimination d'une disjonction transforme $\mathcal{G}$ sans modifier sa fermeture transitive.
Lorsque la disjonction n'est pas encore arbitrée, on distingue deux cas selon l'existence d'une arête dans $\mathcal{G}$. Si une arête existe, on fusionne les deux contraintes de disjonction dans $\mathcal{G}$, puis on remplace la variable booléenne b_{ij} associée à la disjonction par celle de $\mathcal{G}$. Dans le cas contraire, on ajoute la disjonction à $\mathcal{G}$.

Finalement, on calcule une approximation de la réduction transitive du graphe de précédence où une sous-fonction élimine les arcs de valuation nulle sont éliminés. En effet, l'élimination d'un arc de valuation strictement positive requiert l'évaluation des longueurs des chemins entre les deux tâches. De notre point de vue, un tel algorithme relève de la propagation puisqu'il entraîne des modifications des fenêtres de temps des tâches qui permet de déduire de nouvelles précédences jusqu'à atteindre un point fixe. De plus, ce raisonnement est renforcé par la propagation des autres contraintes du modèle. Malgré quelques considérations numériques simples, notre méthode est structurelle puisqu'elle repose sur l'analyse du réseau de contraintes et non des domaines. Actuellement, le calcul de la fermeture et de la réduction transitive n'est pas encore implémenté, car il n'était pas nécessaire pour les problèmes d'atelier.

9.4.2 Traitement des contraintes de partage de ressource

La procédure buildDisjModR analyse l'occupation des ressources (`cumulative` et `disjunctive`) pour détecter de nouvelles disjonctions. Pour chaque ressource, on détermine si un couple de tâches obligatoires (T_i, T_j) définit une nouvelle disjonction. Pour ce faire, on vérifie qu'il n'existe aucun chemin du graphe de précédence ou disjonction entre ces deux tâches et que leur exécution simultanée sur la ressource provoque un dépassement de sa capacité. Dans ce cas, on ajoute une contrainte de disjonction sans temps d'attente entre les deux tâches dans le modèle et dans $\mathcal{G}$.

Procédure buildDisjModT : détection des précédences.

initialiser $\mathcal{G} = (\mathcal{T}, \mathcal{P}, \mathcal{D}, \mathcal{A})$; // $\mathcal{T} \subset \mathcal{X}$, $\mathcal{P} = \mathcal{D} = \emptyset$, $l(i,j) = -\infty$
si estActif(DMD_USE_TIME_WINDOWS) **alors** calculer $\mathcal{P}_0 \multimap \mathcal{L}$;
pour chaque $T_i \prec T_j$ **faire**
  /* contraintes precedence */
  si $d_{ij} > l(i,j)$ **alors**
    si $(T_i, T_j) \in \mathcal{P}$ **alors**
      fusionner $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
      Retracter $T_i \prec T_j$;
    sinon ajouter $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
  sinon Retracter $T_i \prec T_j$;
calculer la fermeture transitive ($\rightsquigarrow$) du graphe partiel engendré par les arcs $\mathcal{P}_0 \cup \mathcal{P}$;
  sous-fonction lever une contradiction lorsqu'un cycle est détecté;
  pour chaque $T_i \sim T_j$ **faire**
    /* contraintes precedenceDisjoint */
    si $T_i \rightsquigarrow T_j$ **alors**
      /* $T_i \prec T_j$ est la contrainte dérivée de $T_i \sim T_j$ */
      si $d_{ij} > l(i,j)$ **alors**
        si $(T_i, T_j) \in \mathcal{P}$ **alors** fusionner $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
        sinon
          Poster $T_i \prec T_j$;
          ajouter $T_i \prec T_j \multimap \mathcal{P}, \mathcal{L}$;
      Retracter $T_i \sim T_j$;
      Poster $b_{ij} = 1$;
    sinon si $T_j \rightsquigarrow T_i$ **alors** ... ; // condition réciproque
    si $(T_i, T_j) \in \mathcal{D}$ **alors**
      fusionner $T_i \sim T_j \multimap \mathcal{D}, \mathcal{L}$;
      Remplacer b_{ij} par la variable booléenne associée à (T_i, T_j) ;
    sinon ajouter $T_i \sim T_j \multimap \mathcal{D}, \mathcal{L}$;
  calculer la réduction transitive du graphe partiel engendré par les arcs $\mathcal{P}_0 \cup \mathcal{P}$;
    sous-fonction supprimer uniquement les arcs de valuation nulle (sans temps d'attente);

Procédure buildDisjModR : traitement des ressources.

pour chaque $r \in \mathcal{R}$ **faire**
  pour chaque $(T_i, T_j) \in r$ **such that** $i < j$ **et** $u_i(r) = 1$ **et** $u_j(r) = 1$ **faire**
    /* T_i et T_j sont des tâches obligatoires */
    si $T_i \not\rightsquigarrow T_j$ **et** $T_j \not\rightsquigarrow T_i$ **et** $(T_i, T_j) \notin \mathcal{D}$ **et** $\min(h_i^r) + \min(h_i^r) > \max(C_r)$ **alors**
      Poster $T_i \simeq T_j$; // $d_{ij} = d_{ji} = 0$
      ajouter $T_i \simeq T_j \multimap \mathcal{D}, \mathcal{L}$;
  si $C_r = 1$ **et** estActif(DMD_REMOVE_DISJUNCTIVE) **alors** Retracter r;
  si $C_r > 1$ **et** estActif(DISJUNCTIVE_FROM_CUMULATIVE_DETECTION) **alors**
    $T \leftarrow \{T_i \in r \mid 2 \times \min(h_i^r) > \max(C_r)\}$;
    $U \leftarrow \{u_i(r) \mid T_i \in T\}$;
    si $|T| > 3$ **alors** Poster disjunctive (T, U) ;

Après cette étape, une option peut déclencher la suppression de la contrainte `disjunctive` puisqu'elle a été décomposée en contraintes `precedenceDisjoint`. Cette suppression entraîne une réduction du filtrage puisque les règles *not first/not last*, *detectable precedence* et *edge finding* ne sont alors plus appliquées.
Au contraire, une autre option déclenche le renforcement de la propagation de la contrainte `cumulative`. En effet, on peut poster une contrainte `disjunctive` sur la plus grande une clique disjonctive de la contrainte `cumulative`. Notez que la clique peut contenir des tâches obligatoires et optionnelles.

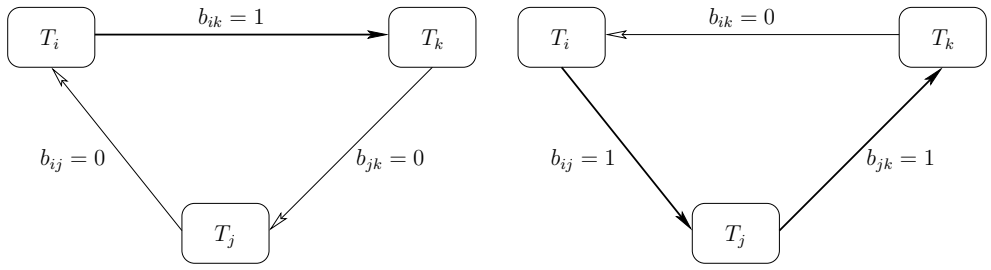
9.4.3 Déduction de contraintes « coupe-cycle »

Supposons que notre modèle contienne une clique disjonctive contenant trois tâches T_i, T_j et T_k. Si une affectation partielle contient les arbitrages $T_i \prec T_j$ et $T_j \prec T_k$, alors l'arbitrage $T_i \prec T_k$ peut être inféré par transitivité (inégalité triangulaire), car l'existence d'un cycle engendre une inconsistance. Les contraintes `precedenceDisjoint` sont propagées individuellement et les arbitrages par transitivité ne sont pas détectés. Cette remarque est valable pour des cliques de taille quelconque.
Dans les problèmes de tournées de véhicules, une problématique similaire est résolue en posant un nombre exponentiel de contraintes d'élimination des sous-tours [177]. La procédure postCoupeCycle pose un nombre quadratique de clauses booléennes similaires aux contraintes d'élimination des sous-tours de longueur 3. Dans le cas général, ces contraintes ne détectent pas forcément tous les arbitrages transitifs, par exemple, lorsqu'il existe un cycle de longueur 4 constitué de tâches ne formant pas une clique disjonctive. Par contre, ces contraintes sont suffisantes pour la décomposition des contraintes `disjunctive`. Ces contraintes peuvent réduire le nombre d'arbitrages visités pendant la résolution.

Notons b_{ij} la variable booléenne réifiant la contrainte de disjonction $T_i \sim T_j$. Pour chaque couple d'arêtes adjacentes d'extrémités T_i et T_k, on simplifie la contrainte d'élimination des sous-tours en fonction de l'existence d'un arbitrage d'une des deux disjonctions. La figure 9.1 illustre le cycle du graphe disjonctif éliminé par chaque clause ou sa simplification. Par souci de clarté, nous ignorons les éventuelles opérations de substitution sur les variables booléennes ($b_{ij} = \neg b_{ji}$) relatives à l'« orientation » des contraintes du modèle ($T_i \sim T_j$ ou $T_j \sim T_i$).

Procédure postCoupeCycle : ajout des contraintes d'élimination des sous-tours de longueur 3.

pour chaque $T_i \sim T_j,\ T_j \sim T_k \in \mathcal{D}$ **faire**
  si $T_i \rightsquigarrow T_k$ **alors**
    `Poster clause` $(b_{ij} \vee b_{jk})$;
  sinon si $T_k \rightsquigarrow T_i$ **alors**
    `Poster clause` $(\neg b_{ij} \vee \neg b_{jk})$;
  sinon si $(i,k) \in \mathcal{D}$ **alors**
    `Poster clause` $(\neg b_{ik} \vee b_{ij} \vee b_{jk})$;
    `Poster clause` $(b_{ik} \vee \neg b_{ij} \vee \neg b_{jk})$;



(a) $\neg b_{ik} \vee b_{ij} \vee b_{jk}$ (b) $b_{ik} \vee \neg b_{ij} \vee \neg b_{jk}$

FIGURE 9.1 – Élimination des arbitrages créant un cycle de longueur 3 dans le graphe disjonctif.

9.5 CSP Solver Competition 2009

Le but de la *CSP solver competition* (CSC) est d'améliorer notre connaissance sur l'efficacité des différents mécanismes disponibles dans les solveurs de contraintes tels que les algorithmes de recherche, stratégies de branchement et les algorithmes de filtrage. On considère la résolution de CSP, Max-CSP (on autorise la violation des contraintes et la qualité de la solution est estimée en fonction du nombre de contraintes satisfaites) et Weighted-CSP (un poids est alloué à la violation d'une contrainte et on cherche une solution dont le poids des violations est minimal). Notez qu'on ne considère pas les problèmes d'optimisation. On considère des contraintes binaires ou non définies en extension ou en intension ainsi que certaines contraintes globales définies sur des variables entières. Un solveur peut participer à la compétition même s'il ne peut résoudre toutes les catégories de problèmes.
Nous présentons dans cette section les résultats de la CSC'2009 sur les problèmes d'atelier (catégorie 2-ARY-INT) et d'ordonnancement cumulatif (catégorie Alldiff+Cumul+Elt+WSum). Neuf équipes ont participé à cette édition et proposé quatorze solveurs tous basés sur des logiciels libres (aucun solveur commercial ne participe à cette compétition).
La résolution d'une instance par un solveur est interrompue après 1800 secondes. Le *virtual best solver* (VBS) est une construction théorique qui renvoie la meilleure réponse fournie par un des solveurs participants. On peut le voir comme un méta-solveur qui utilise un oracle parfait déterminant le meilleur solveur sur une instance donnée ou comme un solveur exécutant en parallèle tous les solveurs participants.

Les deux solveurs basés sur `choco` participent à toutes les catégories de problèmes. Ils utilisent notamment le module décrit dans l'annexe A. Un processus de configuration automatique est nécessaire pour obtenir des performances raisonnables sur les différentes catégories. Tout d'abord, plusieurs méthodes structurelles d'analyse et de reformulation sont appliquées sur le modèle encodé dans un fichier d'instance. Ensuite, la configuration de l'algorithme de recherche est basée sur le résultat d'une étape de *shaving* pendant la propagation initiale. Cette étape est éventuellement interrompue brutalement par le déclenchement d'une limite de temps. L'intérêt du *shaving* est renforcé par la présence de nombreuses instances insatisfiables dans toutes les catégories, car celui-ci détecter une inconsistance avant la recherche. Après cette étape, on configure la stratégie de branchement et celle de redémarrage en exploitant les informations sur les domaines et le réseau de contraintes. On sélectionne une des heuristiques de sélection de variable (voir section 2.3) : *lex*, *random*, *dom/deg*, *dom/wdeg*, *impact*. L'heuristique de sélection de valeur est aléatoire. On lance ensuite un algorithme de recherche en profondeur d'abord avec retour arrière. À notre connaissance, il n'y a pas de différence majeure entre les deux solveurs sur les catégories discutées présentement.

9.5.1 Catégorie 2-ARY-INT

Nous nous intéressons d'abord à la résolution des instances d'open-shop et de job-shop (catégorie 2-ARY-INT). Ces instances sont dérivées des jeux d'instances pour l'optimisation de Taillard et Guéret-Prins par une transformation vers des problèmes de satisfaction en fixant un horizon de planification (satisfiable ou non). Les durées des opérations subissent éventuellement un processus de perturbation ou de normalisation. Le modèle encodé est basé sur des contraintes de disjonction (non réifiées) entre paires de tâches.
Nous appliquons en temps polynomial plusieurs méthodes de reformulation pour construire un modèle similaire au modèle *Heavy* (voir section 7.1) :

1. Identification des tâches appartenant aux disjonctions du modèle encodé.
2. Réification des contraintes de disjonctions par transformation en contraintes `precedenceDisjoint`.
3. Ajout de contraintes redondantes `disjunctive` par identification des cliques de disjonction.

Par contre, nous n'appliquons pas les stratégies de recherche proposées aux chapitres 6 et 7. Contrairement à la reformulation, elles nous apparaissent très dépendantes des problèmes d'atelier et ne correspondent donc pas à l'esprit du concours.

Nous récapitulons le nombre de réponses données par chaque solveur après la résolution de toutes les instances d'open-shop en figure 9.2 ou de job-shop en figure 9.3 : satisfiables (SAT), insatisfiables (UNSAT) ou non résolues (UNKNOWN). Notez que les solveurs peuvent donner le même nombre de réponses sans toutefois que ce soit sur les mêmes instances. `choco` résout le plus grand nombre d'instances d'open-shop et ses résultats sont très proches de ceux du VBS pour les instances insatisfiables. Les

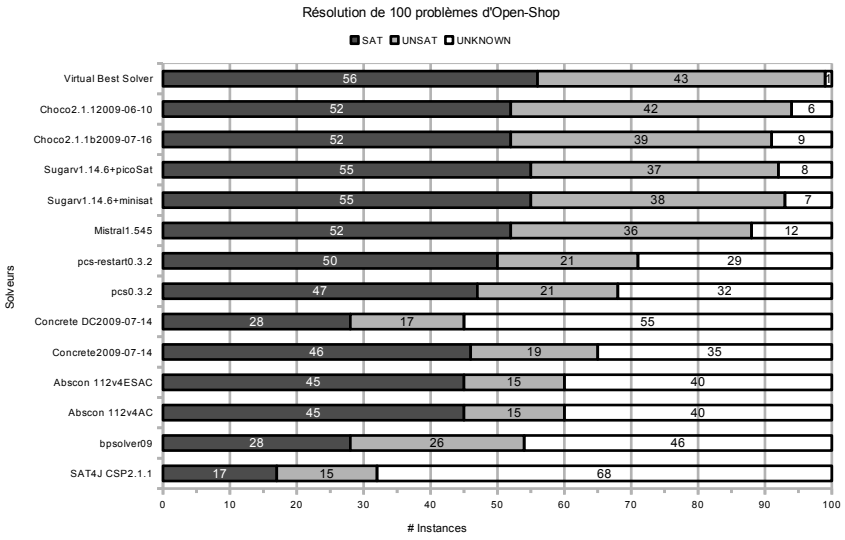


FIGURE 9.2 – Résolution de 100 problèmes d'open-shop (CSC'2009).

résultats de trois autres solveurs (`sugar` +`minisat`, `sugar` +`picosat`, `mistral`) rivalisent avec ceux de `choco`. Les solveurs `sugar` trouvent plus de solutions (SAT), mais moins d'inconsistances (UNSAT) que `choco`. Les résultats de `choco` sur les instances de job-shop restent parmi les meilleurs. Cependant, les

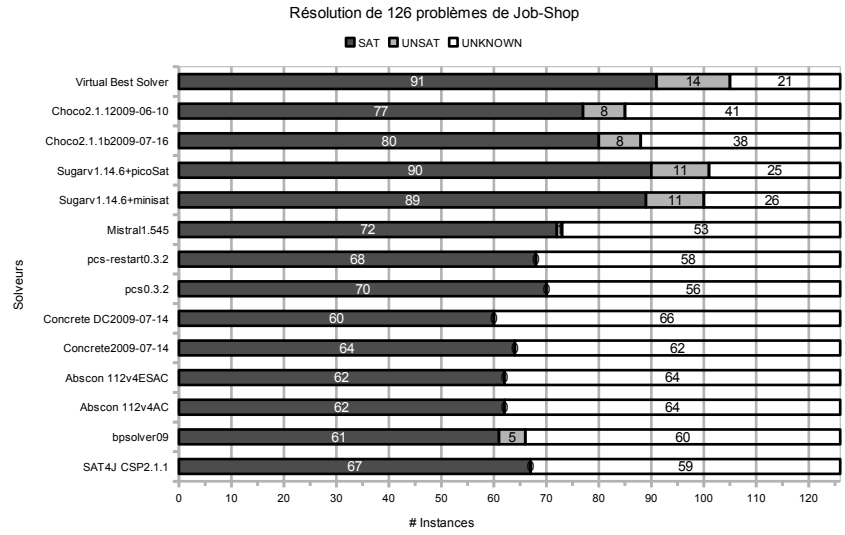


FIGURE 9.3 – Résolution de 126 problèmes de job-shop (CSC'2009)

solveurs `sugar` s'approchent plus du VBS que ce soit pour les instances satisfiables ou insatisfiables. Par contre, les résultats de `mistral` se dégradent légèrement. D'une manière générale, les différences entre les solveurs s'accentuent, notamment sur les instances insatisfiables détectées uniquement par quatre solveurs (`choco`, `sugar`, `mistral`, `bpsolver09`). En considérant toutes les instances, `choco` et `sugar` donnent respectivement le plus grand nombre de réponses insatisfiables et satisfiables.
les graphes de la figure 9.4 représentent le nombre d'instances qu'un solveur est capable de résoudre dans une limite de temps. Les instances d'open-shop et de job-shop sont groupées selon les réponses définitives de chaque solveur : SAT+UNSAT (en haut), SAT (en bas à gauche), UNSAT (en bas à

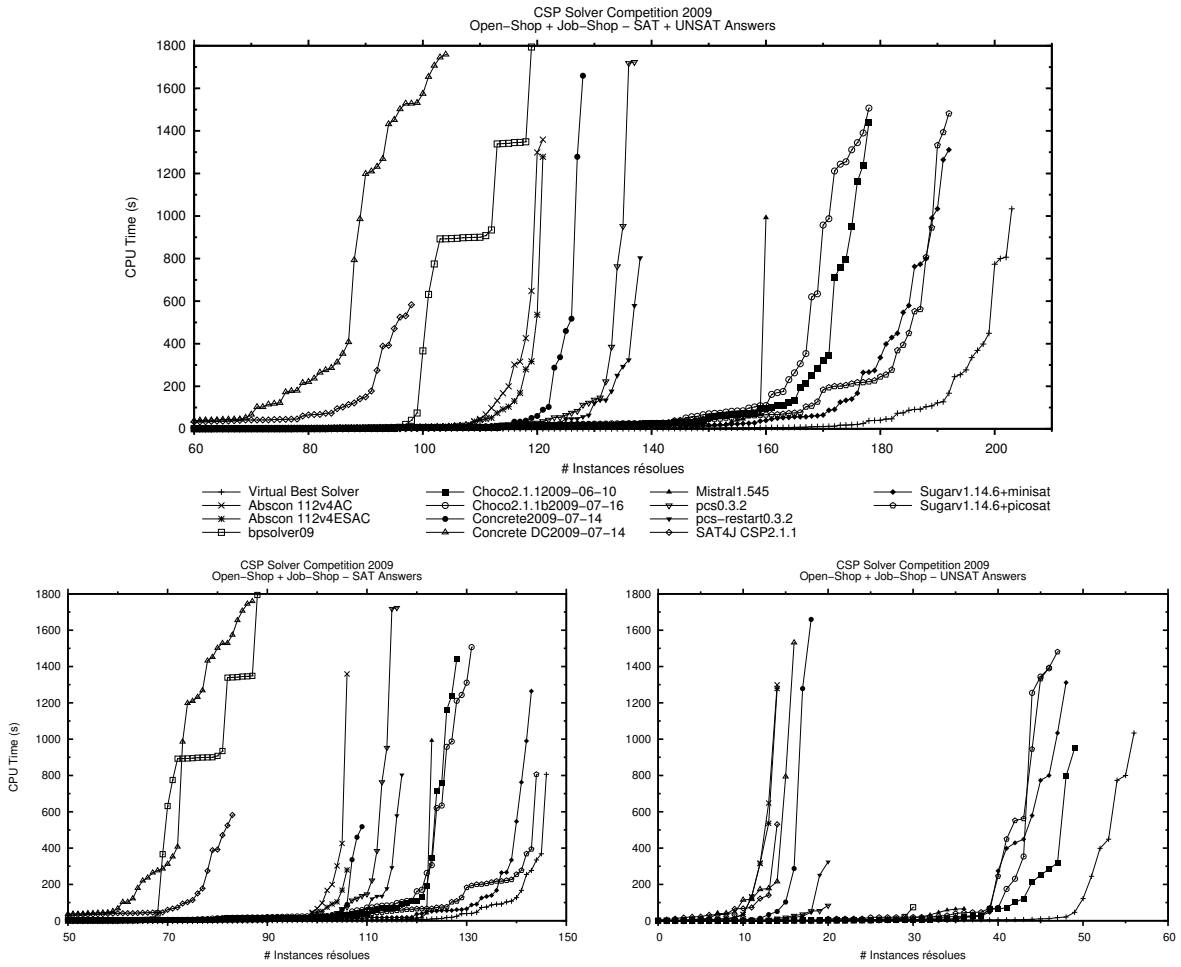


FIGURE 9.4 – Nombre d'instances résolues dans une limite de temps (2-ARY-INT).

droite). La coordonnée x est le nombre d'instances résolues par un solveur et la coordonnée y est la limite de temps en secondes pour la résolution de chaque instance. La transition de phase d'un solveur est caractérisée par sa position et sa forme. Cette transition a lieu quand les temps de résolution augmentent rapidement. Lorsque la transition a presque la forme d'un escalier, la réponse définitive du solveur est presque immédiate et celui-ci n'est pas capable de profiter du temps alloué supplémentaire. Lorsque la transition a une forme étendue, les réponses définitives du solveur sont dispersées sur une partie de l'intervalle de temps alloué.

Pour les réponses SAT+UNSAT (en haut), la courbe du VBS (à droite) est éloignée des autres ce qui laisse supposer que les temps de résolution d'une même instance varient beaucoup d'un solveur à l'autre. En d'autres termes, pour chaque instance résolue, il existe souvent un solveur pour lequel cette instance est résolue rapidement. Ensuite, ce graphe confirme que les solveurs les plus efficaces sont aussi les plus rapides (`choco`, `sugar`, `mistral`). La transition de phase de `mistral` a une forme en escalier contrairement à celles de `choco` et `sugar` qui sont plus étendues. D'ailleurs, la transition de phase plus étendue de `sugar` capture plus de réponses définitives. Les réponses SAT (graphe en bas à gauche) sont plus défavorables à `choco` puisque la courbe de `mistral` croise celles de `choco` et que les courbes de `sugar` se rapprochent de celle du VBS. La situation change pour les réponses UNSAT (graphe en bas à droite) où `choco` est le plus efficace et le plus rapide. Le solveur `mistral` reste rapide mais sa transition de phase est plus précoce.

Les solveurs `sugar` appliquent une technique d'encodage d'un CSP vers un problème de satisfiabilité booléenne similaire à celle de SAT-Ta discutée au chapitre 6. Dans le cadre de la CSC'2009, la comparaison

des résultats de `choco` et `sugar` sur les instances satisfiables confirment les conclusions des chapitres 6 et 7 à propos de l'importance des autres composantes de notre approche (solution initiale, stratégie de branchement). Ce phénomène est encore plus flagrant pour le solveur `mistral` dont les performances sont significativement dégradées et qui devient moins rapide que `choco`. À notre connaissance, `choco` est le seul solveur qui utilise une contrainte globale `disjunctive`. Ainsi, le succès de `choco` sur les instances insatisfiables montre que ces inférences supplémentaires permettent effectivement de prouver l'inconsistance du modèle plus rapidement, notamment pendant l'étape de *shaving*. En conclusion, nous soulignons que les résultats obtenus pour résolution des problèmes d'atelier figurent parmi les meilleures obtenues par `choco` au cours de cette compétition.

9.5.2 Catégorie Alldiff+Cumul+Elt+WSum

Nous nous intéressons maintenant à la résolution des instances pour les problèmes d'ordonnancement cumulatif (catégorie Alldiff+Cumul+Elt+WSum) comme le *resource-constrained project scheduling problem* [159]. Nous évaluons ainsi l'implémentation de la contrainte `cumulative`. Les graphes de la figure 9.5 représentent le nombre d'instances qu'un solveur est capable de résoudre dans une limite de temps pour les réponses définitives : SAT (à gauche), UNSAT (à droite). Quatre solveurs seulement participent à cette catégorie : `choco`, `sugar`, `mistral` et `bpsolver09`. Pour les réponses SAT, `choco` ne se place que troisième devant `bpsolver09` et légèrement derrière `mistral` alors que `sugar` obtient sans conteste les meilleurs résultats. Pour les réponses UNSAT, `choco` obtient les meilleurs résultats en dominant légèrement `sugar` en terme de nombre réponses grâce à une transition de phase plus étendue. Les performances de `mistral` et surtout de `bpsolver09` se situent un peu en retrait pour les réponses UNSAT.

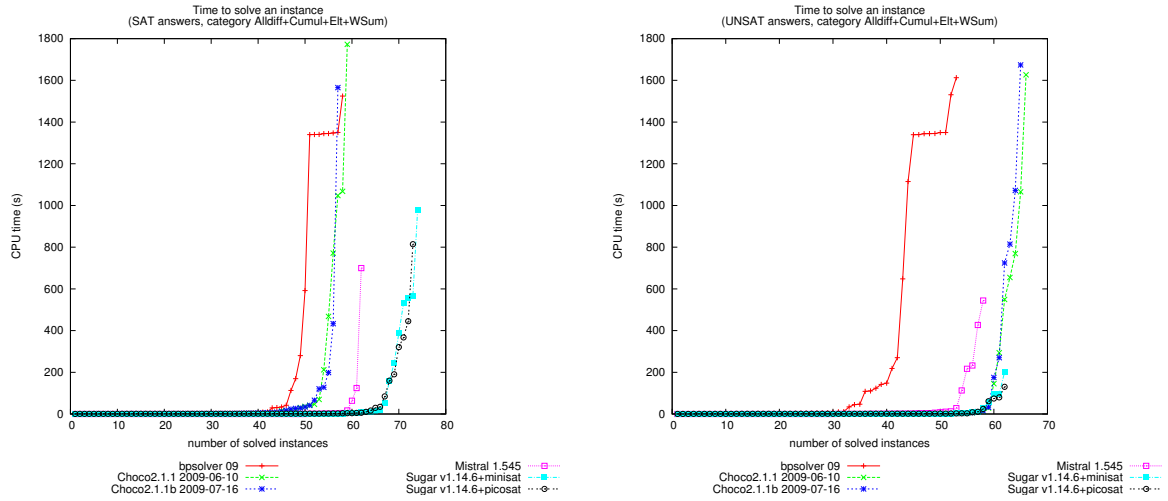


FIGURE 9.5 – Nombre d'instances résolues dans une limite de temps (Alldiff+Cumul+Elt+WSum).

9.6 Conclusion

Nous avons présenté nos principales contributions au solveur de contraintes `choco`. Elles concernent principalement l'ordonnancement, le placement et les redémarrages. Ce travail a aussi engendré une amélioration de certains mécanismes internes du solveur. Les résultats de la CSC'2009 confirment les résultats obtenus dans cette thèse quant aux bonnes performances de ces modules. Ces modules sont suffisamment stables pour que d'autres contributeurs puissent participer à leur développement. Ainsi, l'*edge finding* pour la contrainte `disjunctive` alternative a déjà été implémenté [6]. Une contrainte *soft* `cumulative` [178] et l'*edge finding* cumulatif sont actuellement en développement. Ces modules ont aussi permis de résoudre des problèmes de reconfiguration [179] et de placement [180].