Annexe A

Outils `choco` : développement et expérimentation

Nous décrivons des outils génériques de développement et d'expérimentation disponibles dans le solveur de contraintes `choco`*. Ces modules permettent de reproduire facilement toutes les expérimentations réalisées au cours de cette thèse.*

Sommaire

A.1 Procédure générique de traitement d'une instance 119
A.2 Assistant à la création d'un programme en ligne de commande 121
A.3 Intégration dans une base de données 121

Nous introduisons des outils génériques pour le développement et l'évaluation de modèle(s) `choco`. Ces outils sont basés uniquement sur des logiciels libres et sont donc livrés avec `choco`. Ils ont été initialement implémentés pour la compétition de solveurs (*CSP solver competition*) évoquée en section 9.5. Ils ont aussi servi aux évaluations réalisées aux chapitres 7, 8 (les évaluations du chapitre 6 ont été réalisées sur des versions préliminaires de ces outils). L'idée directrice est de proposer un ensemble cohérent de services pour la résolution d'une instance d'un CSP (section A.1), la création d'un programme en ligne de commande pour lancer et configurer la résolution d'une ou plusieurs instances (section A.2), et stocker les résultats obtenus dans plusieurs formats facilitant leur analyse numérique (section A.3).
Nous apportons régulièrement des améliorations à ces outils et espérons voir leur utilisation se généraliser. Un exemple d'utilisation sur un problème de *bin packing* est présenté en annexe B.3.

A.1 Procédure générique de traitement d'une instance

La classe abstraite `AbstractInstanceModel` propose une procédure générique de traitement d'une instance de CSP. La classe `AbstractMinimizeModel` spécialise cette dernière pour les problèmes de minimisation. Un tel objet est réutilisable (on peut résoudre différentes instances de CSP sans le recréer) et déterministe (on peut reproduire une exécution en spécifiant une graine pour le générateur de nombres aléatoires). Nous nous appuierons sur le listing 2 pour détailler ses principales fonctionnalités.

Listing 2 – Méthode principale de la classe `AbstractInstanceModel`

```
1  public final void solveFile(File file) {
2  initialize();
3  try {
4    LOGGER.log(Level.CONFIG, INSTANCE_MSG, file.getName());
5    boolean isLoaded = false;
6    time[0] = System.currentTimeMillis();
7    try {
```

```
8       load(file);
9       isLoaded = true;
10    } catch (UnsupportedConstraintException e) {
11      Arrays.fill(time, 1, time.length, time[0]);
12      logOnError(UNSUPPORTED, e);
13    }
14    if( isLoaded) {
15      LOGGER.config("loading...[OK]");
16      time[1] = System.currentTimeMillis();
17      isFeasible = preprocess();
18      status = postAnalyzePP();
19      initialObjective = objective;
20      logOnPP();
21      time[2] = System.currentTimeMillis();
22      if( applyCP() ) {
23        //try to solve the problem using CP.
24        model = buildModel();
25        logOnModel();
26        time[3] = System.currentTimeMillis();
27        solver = buildSolver();
28        logOnSolver();
29        time[4] = System.currentTimeMillis();
30        //isFeasible is either null or TRUE;
31        if( isFeasible == Boolean.TRUE) solve();
32        else isFeasible = solve();
33        time[5] = System.currentTimeMillis();
34        status = postAnalyzeCP();
35      }else {
36        //preprocess is enough to determine the instance status
37        Arrays.fill(time, 3, time.length, time[2]);
38      }
39      //check the solution, if any
40      if( isFeasible == Boolean.TRUE) {
41        checkSolution();
42        LOGGER.config("checker...[OK]");
43      }
44      //reporting
45      makeReports();
46    }
47
48  } catch (Exception e) {
49    logOnError(ERROR, e);
50  }
51  ChocoLogging.flushLogs();
52  }
```

Après l'initialisation, une instance d'un CSP est généralement chargée (ligne 8) par la lecture d'un fichier donné en paramètre. Cependant, on peut aussi définir un générateur aléatoire d'instance. En cas de succès du chargement, on procède au prétraitement des données (ligne 17) nécessaire à la construction du modèle (tris, calculs intermédiaires, bornes inférieures, bornes supérieures). Le prétraitement peut éventuellement renvoyer une solution dont l'analyse détermine le statut de la résolution. Si nécessaire, on procède alors à la construction du modèle (ligne 24) qui définit les variables et les contraintes puis à celle du solveur (ligne 27) qui définit l'algorithme de recherche, la stratégie de branchement, et les algorithmes de filtrage. On lance ensuite la résolution (ligne 32) dont le résultat est analysé. En présence d'une solution, `choco` applique un processus de vérification automatique des solutions indépendant des algorithmes de filtrage (ligne 41). Notez qu'il est maintenant possible de vérifier toutes les solutions trouvées par le solveur en activant les assertions Java (option `-ea`). On peut aussi utiliser des outils externes comme le vérificateur de la CSC'2009 ou un fichier de `Properties` contenant les meilleurs résultats connus sur les instances.

Finalement, on rédige différents comptes-rendus sur la résolution (ligne 45). L'utilisateur peut définir ses propres rapports, par exemple un affichage ou une analyse de la solution, en surchargeant cette méthode. La méthode `solveFile(File)` intercepte toutes les exceptions Java pour éviter une interruption brutale de la résolution qui disqualifie des compétitions de solveurs.

La présence d'instructions de la classe `java.util.Logging` témoigne du fait que la classe `AbstractInstanceModel` est intégrée au système de logs de `choco` (la sortie par défaut est la console). La syntaxe des messages respecte le formalisme de la *CSP solver competition*. À cette occasion, nous avons remanié le système de logs qui est devenu non intrusif ce qui a entraîné une amélioration significative des performances de `choco`. L'architecture de ce système et la syntaxe des messages (selon la verbosité) sont décrites précisément dans la documentation officielle.

A.2 Assistant à la création d'un programme en ligne de commande

La classe abstraite `AbstractBenchmarkCmd` facilite la création de programmes en ligne de commande pour lancer et configurer la résolution d'une ou plusieurs instances par un objet héritant de `AbstractInstanceModel`. Cette classe est particulièrement utile lorsque les évaluations sont réalisées sur une grille de calcul puisqu'il est alors nécessaire de disposer d'une ligne de commande. Cette classe s'appuie sur la librairie `args4j` qui propose la lecture des arguments d'une ligne de commandes à partir d'annotations Java.
Notre classe définit quelques options pour la configuration d'un objet `AbstractInstanceModel` pour : (a) gérer les entrées (fichiers d'instance) et les sorties (comptes-rendus), (b) sélectionner un fichier `.properties` contenant la `Configuration` du solveur, (c) assurer le déterminisme de la commande (générateur de nombres aléatoires), et (d) créer une connexion avec la base de données. Si l'argument de l'option obligatoire `-f` est un chemin de fichier, alors on résout cet unique fichier d'instance. Si la valeur est un chemin vers un répertoire, alors on explore récursivement ce répertoire en traitant certains fichiers. Un fichier est traité s'il correspond à un des motifs (*pattern*) définis dans un nombre quelconque d'arguments supplémentaires de la commande. Les jokers (*wildcards*) sont autorisés dans les motifs.
Un écran d'aide de la commande est généré automatiquement à partir de ces annotations :

```
the available options are:
Example of command with required arguments:
java [-jar myCmd.jar| -cp myCmd.jar MyCmd.class]  -f (--file, -file) FILE
Example of command with all arguments:
java [-jar myCmd.jar| -cp myCmd.jar MyCmd.class]  -e (--export) FILE -f (--file, -file) FILE
  -o (--output) FILE -p (--properties) FILE -s (--seed, -seed) N -tl (--timeLimit, -time) N
  -u (--url) VAL -v (--verbosity) [OFF|SILENT|QUIET|DEFAULT|VERBOSE|SOLUTION|SEARCH|FINEST]
Options:
 -e (--export) FILE                    : activate embedded database and export it to odb file
 -f (--file, -file) FILE               : Instance File or directory with option
                                         al wildcard pattern arguments.
 -o (--output) FILE                    : specify output directory (logs, solutions, ...)
 -p (--properties) FILE                : user properties file
 -s (--seed, -seed) N                  : global seed
 -tl (--timeLimit, -time) N            : time limit in seconds
 -u (--url) VAL                        : connect to remote database at URL.
 -v (--verbosity) [OFF | SILENT | QUIET : set the verbosity level
  | DEFAULT | VERBOSE | SOLUTION | SEAR :
 CH | FINEST]                          :
```

A.3 Intégration dans une base de données

Pour finir, nous avons défini une base de données relationnelle en Java basée sur `hsqldb` qui permet de stocker les résultats de la résolution d'un objet `Solver` ou `AbstractInstanceModel`. Le solveur communique avec la base de données grâce à l'interface Java Database Connectivity (JDBC). Le format de fichier du logiciel oobase de la suite Open Office est une archive contenant une base de données `hsqldb`

et divers autres fichiers propres `oobase`. Nous avons développé un modèle `oobase` contenant une base de données vide et plusieurs rapports et formulaires prédéfinis. Une commande (section A.2) peut générer un fichier `oobase` basé sur ce modèle contenant les résultats de son exécution. On peut préférer communiquer les résultats à un serveur `hsqldb` pour agréger les résultats de plusieurs commandes, par exemple obtenus dans une grille de calcul. Il incombe alors à l'utilisateur de lancer le serveur (droit administrateur) et y extraire la base de données du modèle `oobase`.

Annexe B

Tutoriel `choco` : ordonnancement et placement

Ce tutoriel introduit les bases pour l'utilisation des modules de `choco` *décrits au chapitre 9. Les deux premiers cas d'utilisation couvrent un large spectre du module d'ordonnancement, et le dernier se concentre sur le module de développement et d'expérimentation à travers la résolution d'un problème de placement en une dimension. La difficulté des cas d'utilisation suit une progression croissante.*

Sommaire

B.1 Gestion de projet : construction d'une maison . . . 123
  B.1.1 Approche déterministe . . . 124
  B.1.2 Approche probabiliste . . . 126
B.2 Ordonnancement cumulatif . . . 129
B.3 Placement à une dimension . . . 132
  B.3.1 Traitement d'une instance . . . 132
  B.3.2 Création d'une commande . . . 134
  B.3.3 Cas d'utilisation . . . 135

Ce chapitre présente trois tutoriels sur les fonctionnalités de `choco` présentées en chapitre 9. Les sections B.1 et B.2 décrivent l'utilisation du module d'ordonnancement sous contraintes. Ces deux tutoriels sont basés sur l'interface `samples.tutorials.Example` qui définit un schéma de traitement simple des exemples. La section B.3 décrit l'utilisation des outils de développement et d'expérimentation pour la résolution d'un problème de placement. Ce dernier tutoriel est plus avancé puisqu'il aborde d'autres sujets que la modélisation.

B.1 Gestion de projet : construction d'une maison

Ce tutoriel décrit la gestion d'un projet [181] de construction d'une maison avec `choco`. Un projet est défini [ISO10006, 1997] comme :

> « un processus unique, qui consiste en un ensemble d'activités coordonnées et maîtrisées comportant des dates de début et de fin, entreprises dans le but d'atteindre un objectif conforme à des exigences spécifiques telles que des contraintes de délais, de coûts et de ressources. »

Les grandes phases d'un projet sont :

Avant-projet passer de l'idée initiale au projet formalisé elle aboutit à la décision du maître d'ouvrage de démarrer ou non le projet.

Définition mise en place du projet, de l'organisation du projet et des outils de gestion associés.

Réalisation superviser l'exécution des différentes tâches nécessaires à la réalisation du projet et gérer les modifications qui apparaissent au fur et à mesure de la réalisation.

Négliger la définition d'un projet est une erreur car sa réalisation va forcément prendre du retard s'il n'est pas correctement défini au départ. En effet, La caractéristique d'irréversibilité forte d'un projet signifie que l'on dispose d'une capacité d'action très forte au début, mais au fur et à mesure de l'avancement du projet et des prises de décision, la capacité d'action diminue, car les choix passés limitent les possibilités d'action en fin de projet.
La gestion d'un projet est souvent évaluée en fonction du dépassement de coûts et de délais ainsi que sur la qualité du produit fini. On considère souvent une part d'incertitude sur la réalisation des tâches ou due à l'environnement extérieur.
Les statistiques réunies par le *Standish Group* concernant les États-Unis en 1998 révèlent que : (a) seulement 44% des projets sont achevés dans les temps, (b) les projets dépassent en moyenne leur budget de 189%, (c) 70% des projets ne se réalisent pas comme prévu, (d) les projets sont en moyenne 222% plus long que prévu. Nous nous intéressons à deux méthodes de gestion de projet très répandues qui améliorent ces performances :

Programme Evaluation Review Technique (PERT) introduit par l'armée américaine (navy) en 1958 pour contrôler les coûts et l'ordonnancement de la construction des sous-marins Polaris.

Critical Path Method (CPM) introduit par l'industrie américaine en in 1958 (DuPont Corporation and Remington-Rand) pour contrôler les coûts et l'ordonnancement de production.

Les méthodes PERT/CPM ignorent la plupart des dépendances, c'est-à-dire qu'elles ne considèrent que les contraintes de précédences. On ignore donc les contraintes de coûts, partage de ressources ou d'éventuelles préférences. Par contre, elles permettent de répondre aux questions suivantes.

- Quelle sera le délai total du projet ? Quels sont les risques ?
- Quelles sont les activités critiques qui perturberont l'ensemble du projet si elles ne sont pas achevées dans les délais ?
- Quel est le statut du projet ? En avance ? Dans les temps ? En retard ?
- Si la fin du projet doit être avancée, quelle est la meilleure manière de procéder ?

Les méthodes PERT/CPM sont disponibles dans de nombreux logiciels de gestion de projet généralement par le biais d'algorithmes dédiés. Nous montrons d'abord qu'il est plus facile d'appliquer ces méthodes grâce à un solveur de contraintes. Un autre intérêt de cette approche est la prise en compte de contraintes additionnelles pendant la propagation. On bénéficie aussi de tous les services offerts par le solveur pour le diagnostic, la simulation, la détection des inconsistances, l'explication des conflits, la modélisation d'une partie des coûts et bien sûr une recherche complète ou incomplète pour la satisfaction ou l'optimisation.
Ce tutoriel vous aidera à :

- créer un modèle d'ordonnancement de projet contenant des tâches et des contraintes temporelles (`precedence` et `precedenceDisjoint`),
- construire et visualiser le graphe disjonctif,
- appliquer la méthode PERT/CPM (par propagation) pour la gestion d'un projet.
- donner une intuition de la gestion de contraintes additionnelles en considérant deux gammes opératoires différentes pour la réalisation du projet.

Le code est disponible dans la classe `samples.tutorials.scheduling.PertCPM` qui implémente l'interface `samples.tutorials.Example`.

B.1.1 Approche déterministe

L'approche déterministe ne considère aucune incertitude sur la réalisation des activités. Nous décrivons les étapes classiques de la phase de définition d'un projet et d'un modèle en ordonnancement sous contraintes. L'étape 1 décrit certains champs de la classe. Les étapes 2, 3 et 4 sont réalisées dans la méthode `buildModel()`. L'étape 5 est réalisée dans la méthode `buildSolver()`. Les étapes 6 et 7 sont réalisées dans la méthode `solve()`.
Étape 1 : spécifier l'ensemble des activités.

```
1  private TaskVariable masonry, carpentry, plumbing, ceiling,
    roofing, painting, windows,facade, garden, moving;
```

Étape 2 : spécifier les caractéristiques des activités.
Nous ne considérons aucune incertitude sur la durée des activités. L'horizon de planification est la date

d'achèvement du projet en cas d'exécution séquentielle des tâches (la somme de leurs durées).

```
masonry=makeTaskVar("masonry",horizon, 7);
carpentry=makeTaskVar("carpentry",horizon, 3);
plumbing=makeTaskVar("plumbing",horizon, 8);
ceiling=makeTaskVar("ceiling",horizon, 3);
roofing=makeTaskVar("roofing",horizon, 1);
painting=makeTaskVar("painting",horizon, 2);
windows=makeTaskVar("windows",horizon, 1);
facade=makeTaskVar("facade",horizon, 2);
garden=makeTaskVar("garden",horizon, 1);
moving=makeTaskVar("moving",horizon, 1);
```

Étape 3 : déterminer les contraintes de précédences.

```
model.addConstraints(
  startsAfterEnd(carpentry,masonry),
  startsAfterEnd(plumbing,masonry),
  startsAfterEnd(ceiling,masonry),
  startsAfterEnd(roofing,carpentry),
  startsAfterEnd(roofing,ceiling),
  startsAfterEnd(windows,roofing),
  startsAfterEnd(painting,windows),
  startsAfterEnd(facade,roofing),
  startsAfterEnd(garden,roofing),
  startsAfterEnd(garden,plumbing),
  startsAfterEnd(moving,facade),
  startsAfterEnd(moving,garden),
  startsAfterEnd(moving,painting)
);
```

Étape 4 : déterminer les contraintes additionnelles.
Nous considérons une unique contrainte additionnelle qui définit deux gammes opératoires pour la réalisation des tâches `facade` et `plumbing`. Ces gammes sont définies par une contrainte de disjonction avec temps d'attente `facade` $\sim$ `plumbing`. L'objectif est de déterminer le meilleur arbitrage, ce qui est équivalent, dans ce cas simple, à un arbitrage complet du graphe disjonctif généralisé. Notre approche reste valable lorsqu'il faut arbitrer un petit nombre de disjonctions pour obtenir un arbitrage complet.

```
direction = makeBooleanVar(StringUtils.dirName(facade.getName(), plumbing.getName()));
model.addConstraint(precedenceDisjoint(facade, plumbing, direction, 1, 3));
```

Étape 5 : construire et dessiner le graphe disjonctif généralisé.
Le prétraitement du modèle décrit en section 9.4 est optionnel. Cependant, il permet d'accéder à une interface de visualisation du graphe disjonctif dont les résultats sont donnés en figure B.1.

```
PreProcessCPSolver solver = new PreProcessCPSolver();
PreProcessConfiguration.keepSchedulingPreProcess(solver);
solver.createMakespan();
solver.read(model);

createAndShowGUI(s.getDisjMod());
//DisjunctiveSModel disjSMod = new DisjunctiveSModel(solver);
//createAndShowGUI(disjSMod);
```

Étape 6 : estimation initiale des fenêtres de temps des activités.
Cette première estimation présentée en figure B.1(b) est simplement obtenue par la propagation initiale sans prendre de décision d'arbitrage.

```
solver.propagate();
```

Étape 7 : identifier le ou les chemins critiques.
On propage la date de fin au plus tôt de chaque arbitrage. Il existe au moins un ordonnancement au plus tôt réalisable puisque qu'il n'y a ni contraintes de coût, ni contraintes de partage de ressource. On fixe au plus tôt la date d'achèvement du projet (`makespan`), puis on propage cette décision. Remarquez que l'on gère les structures backtrackables de `choco` avec des appels à `worldPush()` et `worldPop()`.
Un chemin critique est un plus long chemin dans le graphe de précédence. Nous ne détaillons pas le calcul des chemins critiques, par exemple grâce à l'algorithme dynamique de Bellman [37]. Le résultat de cette étape est identique à celui de la méthode PERT/CPM. Les figures B.1(d) et B.1(c) illustrent les résultats obtenus pour les deux arbitrages possibles. On constate que le meilleur arbitrage est `plumbing` $\prec$ `facade`, car le projet finit plus tôt (à la date 21 au lieu de 24) et il y a un unique chemin critique (au lieu de deux).

```
final IntDomainVar dir = solver.getVar(direction);
final IntDomainVar makespan = solver.getMakespan();

solver.worldPush();

dir.instantiate(1, null, true); //set forward
//dir.instantiate(0, null, true); //set backward
solver.propagate();
makespan.instantiate(makespan.getInf(), null, true); //instantiate makespan
solver.propagate(); //compute slack times
createAndShowGUI(disjSMod);

solver.worldPop();
```

Les activités appartenant à un chemin critique, dites *activités critiques*, déterminent à elles seules la date d'achèvement du projet. Ces activités critiques ne peuvent pas être retardées sans retarder l'ensemble du projet. Réciproquement, il est nécessaire de réduire la longueur des chemins critiques pour avancer la fin du projet. Ensuite, l'affectation des personnes et ressources peut être améliorée en se concentrant sur ces activités critiques. Les activités non critiques n'influencent pas la date d'achèvement du projet. On peut alors les réordonnancer et libérer les ressources qui leur sont allouées sans perturber le reste du projet (lissage des ressources).
L'analyse des chemins critiques est donc un aspect important de la gestion de projet. Par exemple, on peut identifier certaines tâches fondamentales (*milestone*) qui sont cruciales pour l'accomplissement du projet. Dans notre cas, la tâche `masonry` est fondamentale puisqu'elle appartient nécessairement à tous les chemins critiques. On doit la réaliser avec une attention approfondie et un contrôle rigoureux.

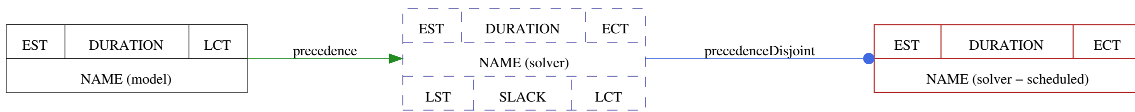
Une solution classique pour avancer un projet est de procéder à du *crashing* qui consiste à réquisitionner des ressources ou utiliser un prestataire pour réaliser plus rapidement une tâche. En général, le *crashing* a un coût et il faut l'utiliser avec précaution.

B.1.2 Approche probabiliste

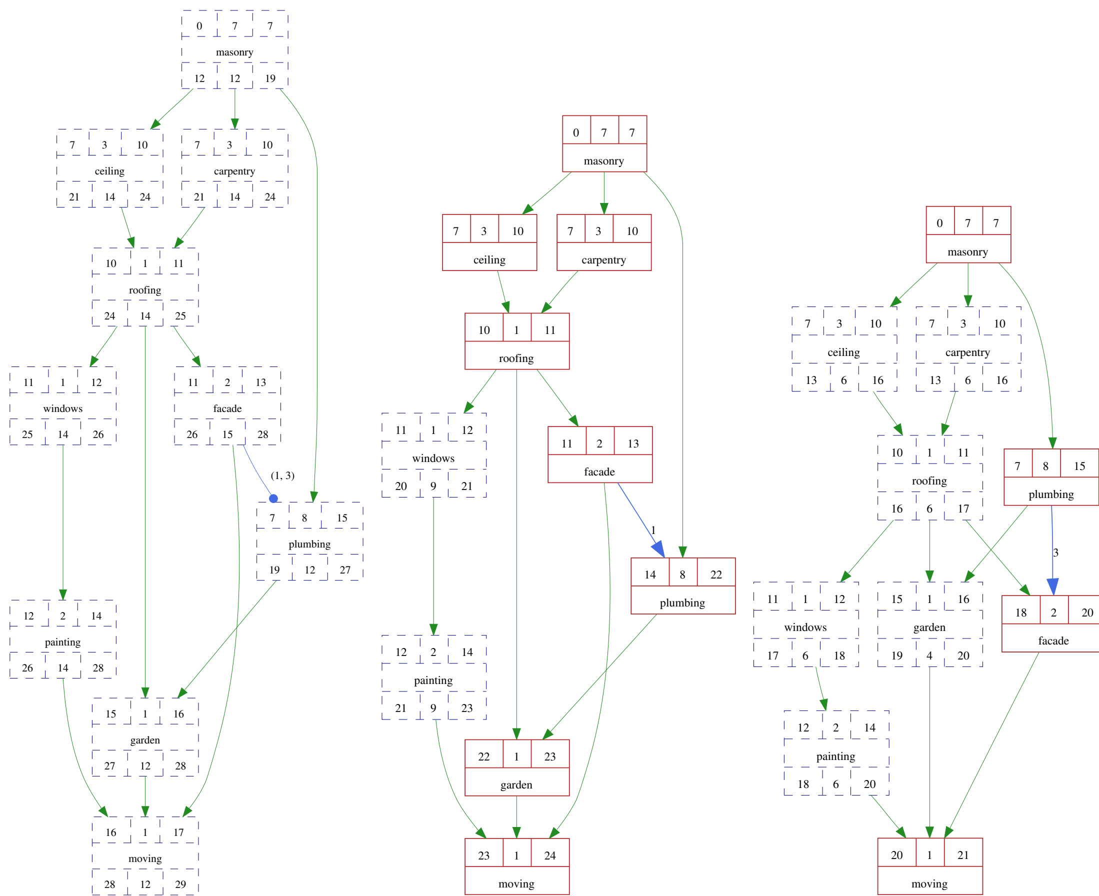
L'approche probabiliste permet de modéliser les incertitudes sur les durées d'exécution des activités. On considère que la durée d'une activité est une variable aléatoire, ce qui signifie qu'elle peut prendre plusieurs valeurs, selon une distribution de probabilité. À partir de ces valeurs, on pourra calculer la durée moyenne, la variance et l'écart type des tâches et des chemins critiques. Si la durée d'une tâche se trouve accrue, la durée du projet peut changer et le chemin critique se déplacer. En gestion de projet, une estimation est généralement (dans les livres et les logiciels) proposée par défaut, elle considère trois durées pour chaque tâche : la durée optimiste p^O, la durée probable (*likely*) p^L et la durée pessimiste p^P. On peut alors calculer l'espérance de la durée p^E, son écart-type σ et sa variance σ^2 grâce aux formules suivantes :

$$p^E = \frac{p^P + 4 \times p^L + p^O}{6} \qquad \sigma = \frac{p^O - p^P}{6}$$

On considère donc que la durée probable se réalise quatre fois plus souvent que les durées optimiste ou pessimiste. Ce ne sera généralement pas le cas, mais ces estimateurs sont sans biais et convergents,



(a) Légende du graphe disjonctif généralisé (modèle et solveur).



(b) Étape 6 (c) Ét. 7 : `facade` $\prec$ `plumbing` (d) Ét. 7 : `plumbing` $\prec$ `facade`

FIGURE B.1 – Visualisation des graphes disjonctifs (activée à l'étape 5) pendant les étapes 6 et 7.

c'est-à-dire que les écarts avec la réalité seront tantôt minorés, tantôt majorés, mais sur un grand nombre de tâches, l'écart sera faible. Le tableau B.1 donne les estimations des durées et variances des tâches. Par un heureux hasard, les durées espérées sont égales aux durées du cas déterministe et la figure B.1 illustre les résultats de la méthode PERT/CPM. On peut calculer directement les durées espérées dans le modèle en substituant des variables entières définies à l'aide de la formule précédente aux constantes dans l'étape 2. La longueur d'un chemin S est égale à la somme des durées des tâches du chemin. Dans

Tâche	p^O	p^L	p^P	p^E	σ^2
masonry	2	6	16	7	5.44
carpentry	1	2	9	3	1.77
plumbing	4	8	12	8	1.77
ceiling	1	3	5	3	0.44
roofing	0	1	2	1	0.11
painting	1	1	7	2	1
windows	0	1	2	1	0.11
facade	1	1	7	2	1
garden	0	1	2	1	0.11
moving	1	1	1	1	0

TABLE B.1 – Estimation probabiliste des durées des tâches.

le cas probabiliste, la somme des durées S est une variable aléatoire qui peut prendre plusieurs valeurs selon une distribution de probabilité. Si le nombre de tâches du chemin est grand ($n > 30$), la loi suivie par la somme tend vers une loi normale de moyenne S et de variance V où V est la somme des variances sur le chemin critique (théorème central limite).

Arbitrage	Chemin critique	S	V
`facade` $\prec$ `plumbing`	`masonry, facade, plumbing, moving`	21	8.21
`plumbing` $\prec$ `facade`	`masonry, ceiling, roofing, facade, plumbing, garden, moving`	24	8.87
`plumbing` $\prec$ `facade`	`masonry, carpentry, roofing, facade, plumbing, garden, moving`	24	10.2

TABLE B.2 – Longueurs et variances des chemins critiques.

Le tableau B.2 récapitule les longueurs et variances des chemins critiques pour les deux arbitrages possibles. Une erreur est introduite dans le calcul de la durée moyenne et de la variance du projet entier, car nous ne considérons qu'un seul chemin critique sans tenir compte d'aucun autre chemin. Ces erreurs varient en fonction de la configuration du réseau. La date d'achèvement du projet est influencée de manière optimiste, mais l'écart-type peut être influencé dans les deux sens. Si un graphe possède un unique chemin critique dont la longueur est nettement supérieure à celles des autres chemins, l'erreur sera acceptable. Au contraire, si un graphe possède de nombreux chemins critiques ou quasi critiques, l'erreur sera plus importante. Cependant, si les chemins en question possèdent un grand nombre d'activités communes, l'erreur introduite sera amoindrie.
La probabilité que le projet finisse au plus tard à la date D est calculée en se référant à la probabilité correspondant à Z dans une table standard de la distribution normale :

$$Z = \frac{D - S}{\sqrt{V}}$$

Les deux courbes de la figure B.2 représentent ces probabilités en considérant seulement un chemin critique dont la variance est maximale pour chaque arbitrage (lignes 1 et 3 du tableau B.2). On constate sans surprise que l'arbitrage `facade` $\prec$ `plumbing` reste le meilleur.

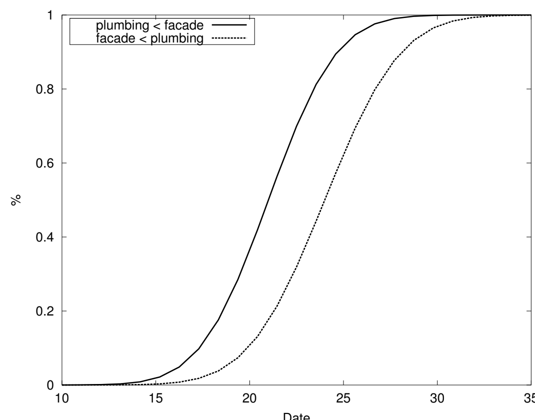


FIGURE B.2 – Probabilité que le projet finisse au plus tard à une date D.

B.2 Ordonnancement cumulatif

Nous présentons ici un problème d'ordonnancement cumulatif simple résolu à l'aide de la contrainte `cumulative`. Ce problème consiste à ordonnancer un nombre maximal de tâches sur une ressource cumulative alternative avec un horizon fixé. Une tâche optionnelle qui a été éliminée de la ressource n'appartient pas à l'ordonnancement final.
La figure B.3 décrit l'instance utilisée à titre d'exemple. À gauche, nous donnons le profil suivi par la capacité de la ressource sur notre horizon de planification. À droite, les tâches sont représentées par des rectangles dont la largeur est la durée d'exécution de la tâche, et la longueur est la consommation instantanée (hauteur) de la tâche sur la ressource. Ce tutoriel vous aidera à :

- poser deux modèles logiquement équivalents basés sur une contrainte globale `cumulative` régulière ou alternative,
- modéliser une capacité variable dans le temps alors que les constructeurs de la contrainte cumulative acceptent uniquement une capacité fixe,
- utiliser une fonction objectif dépendant de l'exécution ou non des tâches optionnelles,
- configurer une stratégie de recherche qui alloue les tâches à la ressource, puis les ordonnance.

Le code est disponible dans la classe `samples.tutorials.scheduling.CumulativeScheduling` qui implémente l'interface `samples.tutorials.Example`. L'étape 1 décrit certains champs de la classe. Les étapes 2, 3, 4 et 5 sont réalisées dans la méthode `buildModel()`. Les étapes 6 et 7 sont respectivement réalisées dans les méthodes `buildSolver()` et `solve()`. L'étape 8 est réalisée dans la méthode

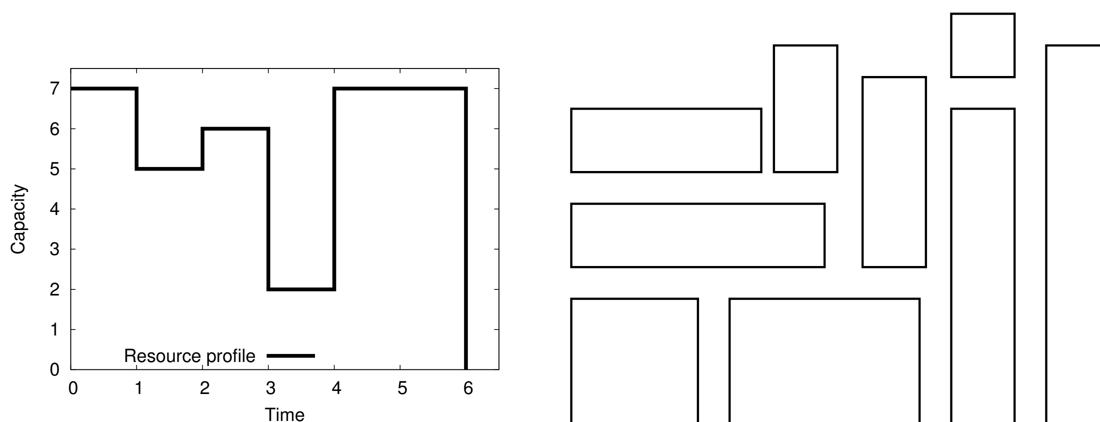


FIGURE B.3 – Une instance de notre problème d'ordonnancement cumulatif.

prettyOut().
Étape 1 : modéliser le profil de la ressource.
Il suffit de fixer la capacité de la ressource à sa valeur maximale sur l'horizon et d'ordonnancer des tâches fictives simulant la diminution temporaire de la capacité. Dans notre cas, nous avons besoin de trois (NF) tâches fictives de durées unitaires et de hauteurs égales à 2, 1 et 4. On considère N tâches dont NF tâches fictives et NT tâches réelles. Les durées et hauteurs des tâches sont contenues dans des objets int[] où les tâches fictives correspondent aux NF premiers éléments.

```
protected final static int NF = 3, NT = 11, N = NF + NT;

private final static int[] DURATIONS = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4, 2, 3, 1, 1};
private final static int HORIZON = 6;

private final static int[] HEIGHTS = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1, 3, 1, 1, 2};
private final static int CAPACITY = 7;
```

Étape 2 : définir les variables du modèle.
On définit N tâches avec des fenêtres de temps initiales identiques $[0, \texttt{HORIZON}]$. On impose que les domaines des variables entières représentant la tâche soient bornés (V_BOUND) car la gestion d'un domaine énumérée est plus coûteuse et de surcroît inutile puisque la contrainte cumulative ne raisonne que sur les bornes. On définit ensuite NT variables booléennes qui indiquent si les tâches sont *effectivement* exécutées sur la ressource, c'est-à-dire appartiennent à l'ordonnancement final. La variable usages$[i]$ est égale à 1 si la tâche tasks$[i + \texttt{NF}]$ est effective. Finalement, on définit la variable objectif (V_BOUND) avec un domaine borné. La définition des hauteurs n'apparaît pas encore, car elle varie en fonction de l'utilisation d'une ressource régulière ou alternative.

```
//the fake tasks to establish the profile capacity of the ressource are the NF firsts.
TaskVariable[] tasks = makeTaskVarArray("T", 0, HORIZON, DURATIONS, V_BOUND);
IntegerVariable[] usages = makeBooleanVarArray("U", NT);
IntegerVariable objective = makeIntVar("obj", 0, NT, V_BOUND, V_OBJECTIVE);
```

Étape 3 : établir le profil de la ressource.
Il suffit d'ordonnancer les tâches fictives pour simuler le profil de la ressource.

```
model.addConstraints(
startsAt(tasks[0], 1),
startsAt(tasks[1], 2),
startsAt(tasks[2], 3)
);
```

Étape 4 : modéliser la fonction objectif.
La valeur de l'objectif est simplement la somme des variables booléennes usages.

```
model.addConstraint(eq( sum(usages), objective));
```

Étape 5 : poser la contrainte de partage de ressource cumulative.
On distingue deux cas en fonction de la valeur du paramètre booléen useAlternativeResource.

vrai on pose une contrainte cumulative alternative en définissant simplement des hauteurs et une capacité constantes. La ressource alternative offre plusieurs avantages : faciliter la modélisation, proposer un filtrage dédié aux tâches optionelles, la gestion d'allocation multi-ressources (useResources).

faux on profite du fait que la contrainte cumulative accepte des hauteurs variables pour simuler l'exécution effective ou l'élimination des tâches. Les tâches fictives gardent une hauteur constante mais les autres variables de hauteur ont un domaine énuméré contenant deux valeurs : zéro et la hauteur. L'astuce consiste à poser les contraintes de liaison suivantes : usages$[i] = 1 \Leftrightarrow$ heights$[i - \texttt{NF}] =$ HEIGHTS$[i]$.

```
Constraint cumulative;
if(useAlternativeResource) {
```

```
heights = constantArray(HEIGHTS);
cumulative = cumulativeMax("alt-cumulative", tasks, heights, usages, constant(CAPACITY),
    NO_OPTION);
}else {
heights = new IntegerVariable[N];
//post the channeling to know if the task uses the resource or not.
for (int i = 0; i < NF; i++) {
  heights[i] = constant(HEIGHTS[i]);
}
for (int i = NF; i < N; i++) {
  heights[i] = makeIntVar("H_" + i, new int[]{0, HEIGHTS[i]});
  model.addConstraint(boolChanneling(usages[i- NF], heights[i], HEIGHTS[i]));
}
cumulative =cumulativeMax("cumulative", tasks, heights, constant(CAPACITY), NO_OPTION);
}
model.addConstraint(cumulative);
```

Étape 6 : configurer le solveur et la stratégie de branchement.
On lit le modèle puis configure le solveur pour une maximisation avant de définir la stratégie de branchement. Par sécurité, on commence par supprimer tout résidu de stratégie. On définit ensuite un premier objet de branchement binaire qui alloue une tâche (branche gauche) puis l'élimine (branche droite) en suivant l'ordre lexicographique des tâches (les indices). À la fin de ce branchement, une allocation des tâches sur la ressource est fixée. Le second branchement (n-aire) ordonnance les tâches en instanciant leurs dates de début avec les heuristiques de sélection dom-minVal. Cette stratégie de branchement simple devrait certainement être adaptée et améliorée pour résoudre un problème réel. Par exemple, on pourrait d'abord allouer les tâches pour lesquelles l'occupation de la ressource est minimale ou éliminer dynamiquement des ordonnancements dominés.

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

StrategyFactory.setNoStopAtFirstSolution(solver);
StrategyFactory.setDoOptimize(solver, true); //maximize

solver.clearGoals();
solver.addGoal(BranchingFactory.lexicographic(solver, solver.getVar(usages), new MaxVal()));
IntDomainVar[] starts = VariableUtils.getStartVars(solver.getVar(tasks));
solver.addGoal(BranchingFactory.minDomMinVal(solver, starts));
solver.generateSearchStrategy();
```

Étape 7 : lancer la résolution.
La variable objectif a été indiquée par le biais d'une option du modèle et la configuration du solveur réalisée à l'étape précédente. Les deux modèles obtiennent les mêmes résultats, car l'algorithme de filtrage par défaut raisonne aussi bien sur les hauteurs que les usages. Ceci n'est généralement pas le cas pour les raisonnements énergétiques. La découverte d'une première solution avec neuf tâches effectives, puis la preuve de son optimalité sont réalisées en moins d'une seconde en explorant 458 nœuds et en effectuant 998 backtracks. Notre approche est meilleure que la stratégie par défaut qui découvre 10 solutions en explorant 627 nœuds et effectuant 2114 backtracks. Une cause probable de cette amélioration est que la stratégie par défaut de `choco` emploie l'heuristique minVal. Par conséquent, elle élimine d'abord la tâche de la ressource avant d'essayer de l'allouer ce qui semble contradictoire avec notre objectif.

```
solver.launch();
```

Étape 8 : visualiser la solution.
On peut visualiser les contraintes de partage de ressource grâce à la librairie `jfreechart`. On propose un mode de visualisation interactive ou d'export dans différents formats de fichier (`.png`, `.pdf`). La figure B.4 montre la solution optimale de notre instance avec une palette de couleurs monochrome. Les tâches T_{10} et T_{13} n'appartiennent pas à l'ordonnancement final.

```
createAndShowGUI(title, createCumulativeChart(title, (CPSolver) solver, cumulative, true));
```

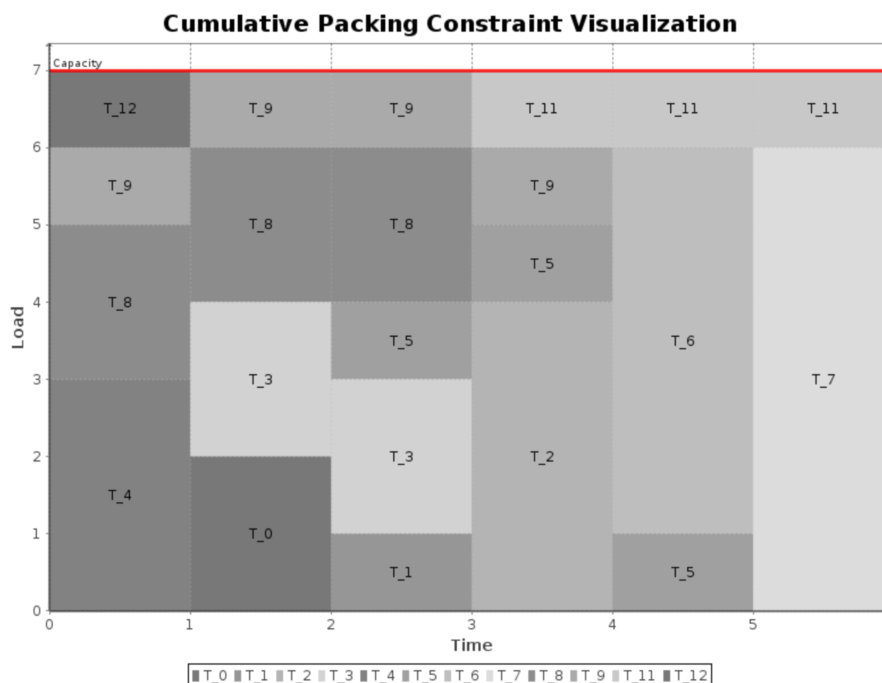


FIGURE B.4 – Une solution optimale de l'instance décrite en figure B.3.

B.3 Placement à une dimension

Ce tutoriel décrit la résolution d'un problème de placement à une dimension (*bin packing*) grâce aux outils de développement et d'expérimentation présentés en chapitre A. On cherche un rangement d'un ensemble d'articles I caractérisés par leur longueur entière positive s_i dans un nombre minimal de conteneurs caractérisés par leur capacité C ($s_i \leq C$). Ce tutoriel vous aidera à :

- créer un modèle pour le placement basé sur la contrainte `pack`,
- appliquer différentes stratégies de recherche,
- étendre la classe `AbstractInstanceModel` pour répondre à vos besoins,
- créer une ligne de commande pour lancer la résolution d'une ou plusieurs instances,
- se connecter à une base de données par le biais de la ligne de commande.

Le code est disponible dans le package `samples.tutorials.packing.parser`.

B.3.1 Traitement d'une instance

Dans cette section, nous décrivons la classe `BinPackingModel` qui hérite de la classe abstraite `AbstractMinimizeModel`. Cette dernière spécialise la classe abstraite `AbstractInstanceModel` (voir section A.1) pour les problèmes de minimisation. La valeur de la propriété booléenne `LIGHT_MODEL` de la classe `Configuration` déterminera la construction d'un modèle *Light* ou *Heavy*.
Étape 1 : définir un parseur pour les fichiers d'instances.
Cet objet implémenté l'interface `InstanceFileParser` qui définit les services requis par `AbstractInstanceModel`. Dans notre cas, une instance est encodée dans un fichier texte dont les première et seconde lignes contiennent respectivement le nombre d'articles et la capacité des conteneurs alors que les suivantes contiennent les longueurs des articles.
Étape 2 : définir le prétraitement de l'instance.
On récupère le parseur pour construire un objet appliquant les heuristiques *complete decreasing best fit*

et *complete decreasing first fit*. Lorsque l'heuristique renvoie une solution, on calcule une borne inférieure (utilisée dynamiquement par pack) afin de tester l'optimalité de la solution heuristique après le prétraitement. Les heuristiques et bornes inférieures utilisées sont disponibles dans choco.

```
1  @Override
   public Boolean preprocess() {
   final BinPackingFileParser pr = (BinPackingFileParser) parser;
   heuristics = new CompositeHeuristics1BP(pr);
5  Boolean b = super.preprocess();
   if(b != null && Boolean.valueOf(b)) {
     computedLowerBound = LowerBoundFactory.computeL_DFF_1BP(pr.sizes, pr.capacity, heuristics.
         getObjectiveValue().intValue());
     nbBins = heuristics.getObjectiveValue().intValue() - 1;
   }else {
10   computedLowerBound = 0;
     nbBins = pr.sizes.length;
   }
   return b;
   }
```

Étape 3 : construire le modèle.
Le modèle repose entièrement sur la contrainte pack et les services de l'objet PackModel. Le modèle *Heavy* pose des contraintes supplémentaires d'élimination des symétries par rapport au modèle *Light*. Finalement, nous précisons la variable objectif, c'est-à-dire le nombre de conteneurs non vide, définie automatiquement par PackModel.

```
1  @Override
   public Model buildModel() {
   CPModel m = new CPModel();
   final BinPackingFileParser pr = (BinPackingFileParser) parser;
5  modeler = new PackModel("", pr.sizes, nbBins, pr.capacity);
   pack = Choco.pack(modeler, C_PACK_AR, C_PACK_DLB, C_PACK_LBE);
   m.addConstraint(pack);
   if( ! defaultConf.readBoolean(LIGHT_MODEL) ) {
     pack.addOption(C_PACK_FB);
10   m.addConstraints(modeler.packLargeItems());
   }
   modeler.nbNonEmpty.addOption(V_OBJECTIVE);
   return m;
   }
```

Étape 4 : construire et configurer le solveur.
Les variables de décision sont les variables d'affectation d'un article dans un conteneur. Le modèle *Heavy* utilise une stratégie de branchement *complete decreasing best fit* avec élimination dynamique de symétries lors d'un retour arrière (voir section 8.3). Le modèle *Light* utilise une stratégie de branchement lexicographique équivalente à *complete decreasing first fit* sans élimination des symétries.

```
1  @Override
   public Solver buildSolver() {
   Solver s = super.buildSolver(); // create the solver
   s.read(model); //read the model
5  s.clearGoals();
   if(defaultConf.readBoolean(BasicSettings.LIGHT_MODEL) ) {
     s.addGoal(BranchingFactory.lexicographic(s, s.getVar(modeler.getBins())));
   }else {
     final PackSConstraint ct = (PackSConstraint) s.getCstr(pack);
10   s.addGoal(BranchingFactory.completeDecreasingBestFit(s, ct));
   }
   s.generateSearchStrategy();
```

```
return s;
}
```

Étape 5 : personnaliser les logs et la visualisation.
On utilise la contrainte `pack` pour afficher textuellement la solution dans les logs. De même, on crée un objet `JFreeChart` pour visualiser le rangement (interactif ou export). L'affichage et la visualisation sont désactivés si la solution optimale est obtenue de manière heuristique. Nous verrons comment régler les logs et la visualisation en section B.3.3.

```
1  @Override
   public String getValuesMessage() {
   if(solver != null && solver.existsSolution()) {
     return ( (PackSConstraint) solver.getCstr(pack) ).getSolutionMsg();
5  } else return "";
   }

   @Override
   public JFreeChart makeSolutionChart() {
10 return solver != null && solver.existsSolution() ?
       ChocoChartFactory.createPackChart(getInstanceName()+" : "+getStatus(), (PackSConstraint)
            solver.getCstr(pack)) : null;
   }
```

B.3.2 Création d'une commande

Dans cette section, nous décrivons la classe `BinPackingCmd` qui hérite de la classe abstraite `AbstractBenchmarkCmd` (voir section A.2).
Étape 6 : définir une nouvelle option pour la ligne de commande.
On associe une annotation Java à un champ de la classe.

```
1  @Option(name="-l",aliases={"--light"},usage="set the light model")
    protected boolean lightModel;
```

Étape 7 : vérifier les arguments de la ligne de commande.
On charge aussi un fichier `.properties` contenant tous les optimums de notre jeu d'instances.

```
1   @Override
   protected void checkData() throws CmdLineException {
   super.checkData();
   seeder = new Random(seed);
5  //check for Boolean, if null then keep default setting (property file)
   if(lightModel) settings.putTrue(BasicSettings.LIGHT_MODEL);
   //load status checkers
   SCheckFactory.load("/bin-packing-tut/bin-packing-tut.properties");
     }
```

Étape 8 : spécifier comment construire une instance réutilisable.

```
1 @Override
   protected AbstractInstanceModel createInstance() {
       return new BinPackingModel(settings);
   }
```

Étape 9 : définir le traitement d'une instance.

```
1   @Override
    public boolean execute(File file) {
        instance.solveFile(file);
        return instance.getStatus().isValidWithOptimize();
```

```
5   }
```

Étape 10 : définir la méthode `main(String[])`.
La méthode a toujours la même structure, mais on doit adapter la classe de `cmd`.

```
1   public static void main(String[] args) {
    final BinPackingCmd cmd = new BinPackingCmd();
    if (args.length == 0) {
          cmd.help();
5    } else {
          cmd.doMain(args);
     }
    }
```

B.3.3 Cas d'utilisation

Nous avons sélectionné un jeu de 36 instances avec 50 ou 100 articles issues de `http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm`.
Étape 11 : résoudre une instance et visualiser la solution.

```
java -ea -cp $CHOCO_JAR $CMD_CLASS -v VERBOSE --properties bp-chart.properties
     -f instances/N1C3W1_A.BPP;
```

`CHOCO_JAR` est un chemin vers une archive jar de `choco` et `CMD_CLASS` est la classe définissant la ligne de commande. L'activation des assertions java force la vérification automatique de toutes les solutions. La verbosité est augmentée pour suivre les différentes étapes. Ensuite, on spécifie un fichier `properties` qui active la visualisation de la meilleure solution de la manière suivante :

```
tools.solution.report=true
tools.solution.export=true
tools.output.directory=java.io.tmpdir
```

Finalement, on spécifie le chemin de l'unique fichier d'instance à résoudre. La commande génère un fichier `.pdf` affiché en figure B.5 contenant la meilleure solution découverte par le solveur. Les logs reproduits ci-dessous donnent un aperçu de la configuration et de la résolution de l'instance. Le système de logs est intégré dans `choco` et permet d'afficher des informations très précises sur la recherche. Au contraire, on peut diminuer ou supprimer la verbosité. Par exemple, la verbosité par défaut, c'est-à-dire les messages préfixés par les lettres `s`, `v`, `d` et `c`, respectent le format défini dans la CSP Solver Competion.

```
properties...[load-properties:bp-chart.properties]
cmd...[seed:0][db:null][output:null]
properties...[load-properties:/bin-packing-tut/bin-packing-tut.properties]
cmd...[OK]
========================================================
Treatment of: N1C3W1_A.BPP
loading...[OK]
preprocessing...[status:SAT][obj:17]
model...dim:[nbv:83][nbc:14][nbconstants:44]
solver...dim:[nbv:83][nbc:30][nbconstants:44]
** CHOCO : Constraint Programming Solver
** CHOCO v2.1.1 (April, 2010), Copyleft (c) 1999-2010
- Search completed
  Minimize: NbNE:16
  Solutions: 1
  Time (ms): 152
  Nodes: 25
  Backtracks: 9
  Restarts: 0
checker...[OK]
s OPTIMUM FOUND
v [100, 45, 3] [100, 45] [96, 54] [94, 51, 5] [90, 59] [88, 62] [87, 62, 1] [85, 65]
```

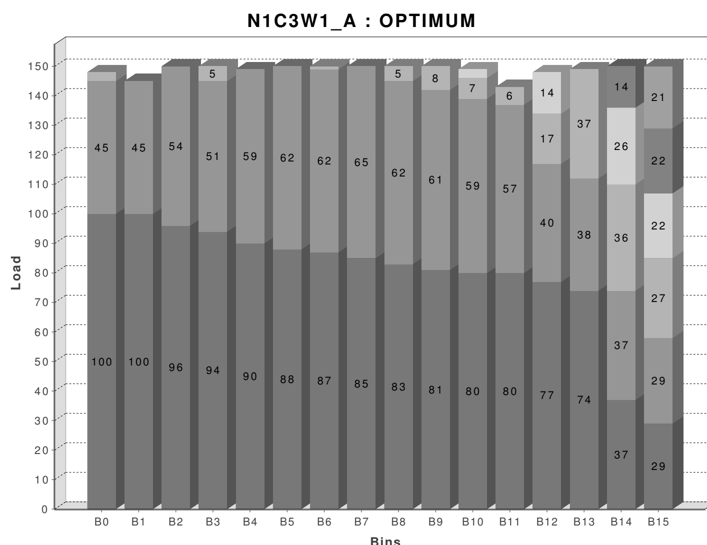


FIGURE B.5 – Une solution optimale de l'instance N1C3W1_A.

```
[83, 62, 5] [81, 61, 8] [80, 59, 7, 3] [80, 57, 6] [77, 40, 17, 14] [74, 38, 37]
[37, 37, 36, 26, 14] [29, 29, 27, 22, 22, 21]
d RUNTIME 0.42
d OBJECTIVE 16
d INITIAL_OBJECTIVE 17
d NBSOLS  1
d NODES 25
d NODES/s 59.38
d BACKTRACKS 9
d BACKTRACKS/s 21.38
d RESTARTS 0
d RESTARTS/s 0
d BEST_LOWER_BOUND 16
d BESTSOLTIME 108
d BESTSOLBACKTRACKS 9
d HEUR_TIME 0
d HEUR_ITERATION 2
c MINIMIZE    N1C3W1_A    0 SEED
c 0.421 TIME    17 PARSTIME    34 PREPROC    78 BUILDPB    140 CONFIG    152 RES

chart...[pdfExport:/tmp/N1C3W1_A.pdf][OK]
cmd...[END]
```

Étape 12 : réaliser des expérimentations.

```
java -cp $CHOCO_JAR $CMD_CLASS -time 60 --export bp-tut-heavy.odb -f instances/
java -cp $CHOCO_JAR $CMD_CLASS --light -time 60 --export bp-tut-light.odb -f instances/
java -cp $CHOCO_JAR $CMD_CLASS --properties bp-cancel-heuristics.properties
     --export bp-tut-heavy-noH.odb -f instances/
```

On lance plusieurs commandes pour comparer des variantes de notre méthode de résolution. Dans la première, on lance la méthode par défaut (modèle *Heavy*) où on (a) impose une limite de temps de 60 secondes, (b) active l'intégration de la base de données en indiquant le nom du fichier `oobase`, et (c) spécifie le répertoire contenant les instances. L'ordre de traitement des fichiers lors de l'exploration récursive du répertoire reste identique d'une exécution à l'autre. Dans la seconde, une option active le modèle *Light*. Dans la troisième, on spécifie un fichier `.properties` qui désactive les heuristiques *first fit* et *best fit* dans le modèle *Heavy* et impose une limite de temps :

```
tools.preprocessing.heuristics=false
tools.cp.model.light=false
```

```
cp.search.limit.type=TIME
cp.search.limit.value=60000
```

Nous résumons brièvement les principaux résultats. Tout d'abord, 24 des 36 instances étudiées sont résolues uniquement grâce à la combinaison des heuristiques et de la borne inférieure. Le modèle *Light* résout seulement 4 instances supplémentaires. Le modèle *Heavy* résout encore 5 autres instances portant le total à 33 instances résolues sur 36. On constate que les performances ne sont presque pas affectées par la suppression de l'heuristique contrairement à ce qui se passait dans le chapitre 6. L'explication est simple, notre stratégie de branchement est directement inspirée de l'heuristique *best fit*. En cas de suppression des heuristiques, la première solution découverte est identique à celle de *best fit*.
La figure B.6 donne un aperçu des rapports générés automatiquement dans le fichier de base de données.

SOLVERS SUMMARY (COMPACT)

Date: 11/02/2011

STATUS OPTIMUM FOUND

NAME	RUNTIME	NB_SOLS	OBJ	TIME	NODES	BACKT.	RESTARTS	VARS	CSTR
N1C1W1_A	0,07	0	25	0	0	0	0	0	0
N1C1W1_B	0,02	0	31	0	0	0	0	0	0
N1C1W2_B	0,01	0	30	0	0	0	0	0	0
N1C1W4_A	0,03	0	35	1	0	0	0	153	68
N1C1W4_B	0,01	0	40	0	0	0	0	0	0
N1C2W1_A	0,00	0	21	0	0	0	0	0	0
N1C2W1_B	0,01	0	26	0	0	0	0	0	0
N1C2W2_A	0,01	0	24	0	0	0	0	0	0
N1C2W2_B	0,01	0	27	0	0	0	0	0	0
N1C2W4_A	0,01	0	29	0	0	0	0	0	0
N1C2W4_B	0,01	0	32	0	0	0	0	0	0
N1C3W1_A	0,05	1	16	42	25	9	0	99	30
N1C3W1_B	0,01	0	16	0	0	0	0	0	0
N1C3W2_A	0,00	0	19	0	0	0	0	0	0
N1C3W2_B	0,05	1	20	15	20	0	0	111	39
N1C3W4_A	0,04	1	21	15	23	0	0	114	38
N1C3W4_B	0,02	1	22	6	19	0	0	117	39
N2C1W1_A	31,40	0	48	31375	29054	62557	0	242	90
N2C1W1_B	0,01	0	49	0	0	0	0	0	0
N2C1W2_A	2,29	0	64	2254	6930	9329	0	290	124
N2C1W2_B	0,01	0	61	0	0	0	0	0	0
N2C1W4_A	0,01	0	73	0	0	0	0	0	0
N2C1W4_B	0,03	0	71	6	5	12	0	311	138
N2C2W1_A	0,01	0	42	0	0	0	0	0	0

FIGURE B.6 – Aperçu d'un rapport généré par oobase.