Synthèse d'algorithmes d'ordonnancement à partir de modèles de haut niveau

Jean-Noël Monette, Yves Deville, Pascal van Hentenryck

AEON : Synthèse d'Algorithmes d'Ordonnancement à partir de Modèles de Haut Niveau*

Jean-Noël Monette[1], Yves Deville[1] et Pascal Van Hentenryck[2]

[1] INGI, UCLouvain, 1348 Louvain-la-Neuve, Belgium
[2] Brown University, Box 1910, Providence, RI 02912
{jean-noel.monette,yves.deville}@uclouvain.be, pvh@cs.brown.edu

Résumé

Ce papier décrit AEON, un système dédié à la synthèse d'algorithmes d'ordonnancement à partir de modèles de haut niveau. AEON, qui est entièrement écrit en COMET, prend en entrée le modèle de haut niveau d'un problème d'ordonnancement et l'analyse pour générer un algorithme dédié et qui exploite la structure du problème. AEON offre une variété de synthétiseurs pour générer des algorithmes complets ou heuristiques. En outre, ces synthétiseurs peuvent être composés, permettant de générer naturellement des algorithmes hybrides complexes. Les premiers résultats montrent que cette approche peut être compétitive avec l'état de l'art des algorithmes de recherche.

1 Introduction

Les problèmes d'ordonnancement sont omniprésents dans l'industrie et ont été l'objet d'une important recherche depuis plusieurs dizaines d'années. Il existe, à l'heure actuelle, des algorithmes efficaces pour différentes classes de problèmes et des systèmes pour modeler et résoudre des problèmes complexes sont disponibles. Cependant, une des difficultés avec les outils existants est que l'utilisateur ne peut pas se contenter de connaître son domaine d'application, il doit aussi être expert dans les aspects algorithmiques et combinatoires. En effet, deux applications qui peuvent sembler pratiquement identiques du point de vue du modèle peuvent nécessiter des approches totalement différentes pour obtenir des solutions de bonne qualité.

Le présent travail est une première étape pour supprimer ces limitations et combler le trou entre modélisation de haut niveau et résolution efficace de problèmes d'ordonnancement. Cet article présente AEON[1], un système qui permet de transformer des modèles de haut niveau en des algorithmes efficaces en exploitant la structure du modèle. Les modèles dans AEON sont écrits avec les abstractions habituelles et leur structure est analysée pour synthétiser des algorithmes *ad hoc* pour l'application décrite. L'utilisateur décrit son modèle et se contente de choisir un synthétiseur qui va générer et exécuter un algorithme d'un paradigme particulier (par exemple, la programmation par contraintes (ppc) ou la recherche locale). Les synthétiseurs d'AEON sont compositionnels, ce qui permet de déclarer des algorithmes hybrides de façon naturelle.

Le système présente différents avantages. Du point de vue de l'utilisateur, AEON permet de se concentrer sur la description de son application à un niveau élevé d'abstraction, le déchargeant de s'occuper des aspects algorithmiques. De plus, comme les modèles révèlent la structure des applications, les synthétiseurs d'AEON sont capables d'exploiter toute la richesse de la recherche en ordonnancement pour dériver des algorithmes efficaces. Finalement, grâce à la séparation nette entre le modèle et la résolution, AEON permet d'appliquer des paradigmes de recherche différents et de développer des hybridations, dont le potentiel a été démontré pour de nombreuses applications. Au niveau de l'implémentation, AEON présente aussi des innovations. Premièrement, l'analyse du modèle est extensible et permet d'ajouter des nouvelles classes de problèmes en les décrivant selon un format XML standard. Deuxièmement, de nouveaux synthétiseurs peuvent être ajoutés simplement et compositionnellement. Finalement, plusieurs abstractions simplifient l'écrit-

*Traduction française d'un article présenté à ICS09 [6]

[1] Aeon est un nom alternatif pour le dieu du temps Chronos. Cela signifie éternité.

ure de synthétiseurs. En particulier, AEON fournit des *vues du modèles*, qui permettent d'accéder à un modèle général et à sa solution à travers une interface spécialisée pour un problème particulier.

Ce papier étend et généralise la recherche commencée dans [11] qui montrait comment synthétiser des algorithmes de recherche locale à partir de modèles COMET. Les synthétiseurs proposés ici s'appliquent plus particulièrement à l'ordonnancement de tâches, considèrent différents paradigmes de recherche (algorithmes gloutons, recherche locale, ppc) et sont extensibles et compositionnels. Le style des modèles utilisé est similaire à ceux de ILOG Scheduler, OPL, COMET et d'autres systèmes d'ordonnancement basés sur les contraintes. ILOG Concert propose aussi une couche de modélisation qui peut être utilisée par des solveurs différents, mais il n'y a pas de tentative de synthétiser des algorithmes de recherche. Il est par ailleurs utile de contraster notre travail avec des travaux récents en ppc qui visent la conception de procédures de recherche par défaut. Parmi d'autres, [4] décrit une recherche par voisinage large auto-adaptive et [8] présente l'utilisation d'impacts pour guider la recherche. Leur but est de proposer une procédure de recherche robuste sur une large gamme de problèmes. Au contraire, notre objectif est d'exploiter la structure du modèle pour dériver des algorithmes spécifiques. Ces deux approches sont orthogonales car il faut aussi des procédures robustes pour différents types de problèmes. Cependant, révéler et exploiter la structure d'un modèle est une des principales contributions de la ppc et les algorithmes de recherche peuvent bénéficier énormément d'une analyse de la structure.

Le reste de cet article présente une vue globale des différentes parties du système. La Section 2 couvre l'utilisation d'AEON et les abstractions et synthétiseurs disponibles. Dans la Section 3, l'architecture est présentée et certaines caractéristiques du système sont mise en valeur. Ensuite, la Section 4 montre comment le système peut être étendu avec d'autres familles de problèmes ou d'algorithmes. La Section 5 présente et analyse des résultats expérimentaux.

2 Modélisation et Résolution de Problèmes d'Ordonnancement

La Figure 1 présente un modèle AEON pour la problème du Job-Shop (JSP) et un synthétiseur. L'initialisation des données aux lignes 1–6 n'est pas montré. Le modèle lui-même est donné aux lignes 8–16. D'abord un objet est créé pour le problème. Ensuite, les objets qui peuplent ce problème sont créés (lignes 9–11), activités, jobs et machines. Après les contraintes sont posées : demandes des machines (lignes 12-13), ordre

```
1 range jobs = 1..nbjobs;
2 range machines = 0..nbmachines-1;
3 range tasks = 1..nbjobs*nbmachines;
4 int proc[tasks];
5 int mach[tasks];
6 int job[jobs,machines];
7
8 Schedule<Mod> s();
9 Job<Mod> J[i in jobs](s,IntToString(i));
10 Machine<Mod> M[i in machines](s,IntToString(i));
11 Activity<Mod> A[i in tasks](s,proc[i],
   IntToString(i));
12 forall(i in tasks)
13    A[i].requires(M[mach[i]]);
14 forall(i in jobs)
15    J[i].containsInSequence(all(j in machines)A[
      job[i,j]]);
16 s.minimizeObj(makespanOf(s));
17
18 GreedyTabuSynthesizer synth();
19 Solution<Mod> sol = synth.solve(s);
20 sol.printSolution();
```

FIG. 1 – Un modèle pour le Job-Shop et un synthétiseur.

dans les jobs (lignes 14-15). Finalement, l'objectif à la ligne 16 minimise la fin des activités (makespan).

Les trois dernières lignes concernent la résolution. La ligne 18 définit le synthétiseur qui crée, dans ce cas, une recherche gloutonne suivie d'une recherche tabou. Il est facile de modifier le synthétiseur : il suffit de remplacer `GreedyTabuSynthesizer` par `CPSynthesizer` pour obtenir une recherche par ppc. La ligne 19 applique le synthétiseur au modèle. Cela entraîne l'analyse et la classification du modèle, la génération des variables contraintes et objectifs appropriés aux solvers et l'exécution d'un algorithme de recherche dédicacé au modèle. Le synthétiseur produit une solution qui peut être utilisée par la suite. Par exemple, la ligne 20 imprime la solution.

Il est clair sur la Figure 1 que la modélisation et la résolution sont clairement séparées. AEON offre un riche ensemble d'abstractions (classes, méthodes, fonctions) pour modéliser une large gamme de problèmes d'ordonnancement. Le reste de cette section passe en revue les abstractions disponibles à l'heure actuelle.

Les classes de modélisation se terminent par "<Mod>" pour dénoter qu'elles servent pour le modèle[2]. A l'intérieur du système, d'autres classes sont post-fixées avec "<CP>" ou "<LS>" et servent aux

[2]Malgré la syntaxe similaire au C++, il ne s'agit pas de classes template.

TAB. 1 – Résumé des classes disponibles pour la modélisation.

Description	Classes
Problème	`Schedule`
Activités	`Activity`
	`MultiModeActivity`
Jobs	`Job`
Ressources	`Resource`
	`Machine`
	`Reservoir`
	`StateResource`
Objectifs	`ScheduleObjective`
	`TaskObjective`
	`CompletionTime`
	`Lateness`
	`Tardiness`
	`Earliness`
	`UnitCost`
	`PiecewiseLinearFunction`
	`AbsenceCost`
	`AlternativeCost`
	`ModifObjective`
	`MultObjective`
	`ShiftObjective`
	`AgregObjective`
	`SumObjective`
	`MaxObjective`

algorithmes de recherche. Par exemple, le but de la classe `Activity<Mod>` est complètement différent de celui de la classe `Activity<CP>`. Bien qu'elles soient associées au même concept (une activité), la première propose des méthodes pour faire l'analyse du modèle, tandis que la seconde encapsule les variables de ppc qui représentent le moment de début et de fin d'une activité et une contrainte qui les relie. Pour simplifier la lecture, le postfixe "`<Mod>`" est omis dans cette section quand il est clair qu'on se réfère aux classes de modélisation.

La Table 1 présente les classes de modélisation disponibles actuellement dans AEON et dont voici les explications. La classe centrale est `Schedule` (i.e. `Schedule<Mod>`). Elle est passée en paramètre à la création de tous les autres objets et est responsable de la cohérence interne du modèle. Pour représenter les activités, il y a deux classes. `Activity` et `MultiModeActivity` représentent respectivement des activités avec un ou plusieurs modes. A sa création, une activité reçoit en entrée un schedule, une durée et un nom. La durée est soit fixée, soit définie par des bornes supérieure et inférieure. Une `MultiModeActivity` est initialisée avec un `Schedule`, le nombre de modes et un nom. La durée de chacun des modes est donnée séparément pour chaque mode. Les méthodes sur les activités permettent de spécifier l'appartenance à un `Job`, les besoins en ressources et les précédences entre activités. Les besoins dépendent des modes mais les autres contraintes sont communes à tous les modes. Les contraintes de précédence peuvent faire intervenir le début et la fin des activités et des jobs, ainsi que des délais. La classe `Job` représente des groupes d'activités. Les activités d'un job ne sont pas nécessairement ordonnées mais elles ne peuvent être exécutées en même temps. Les jobs partagent certaines caractéristiques des activités : Ils peuvent eux-mêmes être regroupés dans d'autres jobs, et leurs débuts et fins peuvent être contraints avec des précédences. Finalement, les activités et les jobs peuvent être définis comme optionnels, auquel cas ils ne doivent pas obligatoirement être exécutés.

Les ressources sont représentées par quatre classes, en fonction du type de ressource considéré. La classe `Machine` représente des ressources disjonctives. Deux activités qui ont besoin d'une même machine ne peuvent être exécutées en même temps. La classe `Resource` représente les ressources renouvelables. A tout moment, la somme des besoins des activités qui sont exécutées ne peut dépasser la capacité de la ressource. Au contraire, la classe `Reservoir` représente des ressources non-renouvelables et dont la capacité est diminuée à la fin de l'exécution de chaque activité. Une capacité minimum peut être définie pour les classes `Resource` et `Reservoir`. Pour ces deux classes et la classe `Machine`, il est aussi possible de définir des pauses (périodiques), i.e. des moments d'indisponibilité. Le dernier type de ressources est la classe `StateResource` qui représente un état du monde. Une telle ressource ne peut être que dans un état à la fois. Deux activités qui ont besoin d'états différents ne peuvent s'exécuter en même temps. Pour tous les types de ressources, il est possible de définir des temps et des coûts de setup qui dépendent de l'ordonnancement. L'ensemble des besoins d'une activité (ou d'une activité à plusieurs modes) a la forme d'un arbre dont les noeuds internes sont des conjonctions ou des disjonctions de besoins plus simples. Les noeuds externes sont les besoins de base : besoin d'une machine, une quantité de ressource demandée ou fournie, une quantité consommée ou produite dans un réservoir, un état particulier d'une ressource à états.

Les fonctions objectif sont des sous-classes de `ScheduleObjective`. Les sous-classes sont des fonctions simples ou composées. Les fonctions composées sont la somme, le minimum, le maximum d'autres fonctions et la multiplication par une constante. Les fonctions simples sont les classiques date de fin, retard, avance et de façon plus générale les fonctions linéaires

TAB. 2 – Résumé des classes disponibles pour la résolution.

Description	Classes
Synthétiseurs	`ScheduleSynthesizer`
	`CPSynthesizer`
	`TSSynthesizer`
	`SASynthesizer`
	`GreedySynthesizer`
	`SequenceSynthesizer`
	`ScheduleAnimator`
Solutions	`Solution`

par morceaux basées sur la date de fin des activités et des jobs. Les fonctions simples sont aussi le coût associé au mode des activités multimodales, à l'absence d'une activité optionnelle ou au setup des ressources. La fonction objectif globale est passée au `Schedule` par une méthode qui spécifie s'il s'agit d'une minimisation ou d'une maximisation. La fonction `makespanOf` à la Figure 1 est un raccourci pour le makespan, maximum des dates de fin, qui est un objectif commun et important.

Cet ensemble d'abstractions permet de modéliser des problèmes aussi variés que les classiques problèmes d'"atelier" (Job Shop, Open Shop, Flexible Shop, Flow Shop, Group Shop, Cumulative Shop, Just-In-Time Job Shop), des variations de l'ordonnancement de projet avec des contraintes de ressources (RCPSP, MRCPSP, RCPSP/max, MRCPSP/max) ([3]), le problème du trolley ([12]) and les classes NCOS et NCGS de MaScLib ([7]). Cela représente des problèmes avec différents types d'objectifs et différentes propriétés (disjonctif ou cumulatif, un ou plusieurs modes).

Bien que les abstractions de modélisation permettent de représenter un grand nombre de problèmes, l'ensemble des problèmes qui peut être résolu dépend de la recherche qui peut être synthétisée. La Table 2 présente les classes de synthèse qui sont disponibles dans AEON actuellement. Pour le moment, il y a trois solveurs sous-jacents : la ppc, la recherche locale (tabou et recuit simulé) et la recherche gloutonne.[3] Leurs possibilités définissent les possibilités de tout le système. Des solveurs plus complexes peuvent être synthétisés à partir de ceux de base. En particulier, il est possible de faire des solveurs hybrides et animés. Par exemple, un solveur animé emballe un solveur sous-jacent dans un environnement visuel qui montre la succession des solutions trouvées. Des solveurs hybrides peuvent être de simples séquences de solveurs ou peuvent suivre des schémas de décomposition plus compliqués. Les synthétiseur acceptent

[3]Dans le futur, nous allons aussi intégrer des solutions basées sur la programmation mathématique.

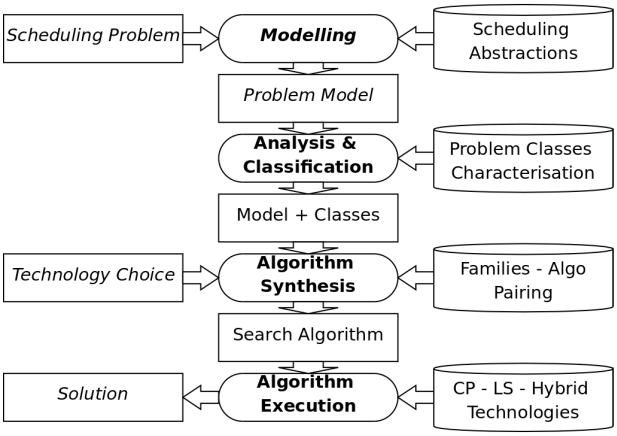

FIG. 2 – Vue d'ensemble d'AEON. Les rectangles arrondis représentent les actions pour résoudre le problème. Les rectangles et conteneurs représentent leurs entrées et sorties. Les conteneurs sur la droite sont fournis par le système, les rectangles sur la gauche sont les entrées de, et les sorties vers l'utilisateur et les rectangles au milieu sont des produits intermédiaires. Le texte en italique représente les endroits d'implication de l'utilisateur.

aussi des paramètres, comme par exemple une limite temporelle sur la recherche.

3 Architecture

La Figure 2 présente une vue d'ensemble d'AEON centrée sur la résolution d'un problème. Les rectangles arrondis sont les étapes successives vers une solution. Seule la première, la modélisation, fait participer l'utilisateur. Ensuite, le modèle est analysé et catégorisé dans des classes de problèmes. Un algorithme est alors synthétisé et exécuté pour produire une solution au problème. Les conteneurs sur la droite de la Figure 2 représentent ce qui est fourni par le système à chaque étape. Le reste de cette section explique plus en détails l'exécution de chaque étape. La Section 4 décrit comment il est possible d'agrandir les conteneurs.

3.1 Modélisation

La Section 2 a présenté la modélisation du point de vue de l'utilisateur. A l'intérieur, quand le modèle est exécuté, une représentation interne du problème est construite. La majorité de l'information est enregistrée dans les objets de modélisation qui ont été présentés. Par exemple, la classe `Activity` contient un attribut retenant si la préemption est autorisée ou pas. En outre, l'objet `Schedule` garde une référence vers tous les objets qui ont été créés. L'information est

enregistrée grâce à des structures de graphes : les relations de précédences dans un graphe dirigé, les fonctions objectif et les besoins en ressources dans des arbres plantés. Les contraintes de précédence sont des arcs étiquetés d'un graphe dont les noeuds représentent le début et la fin des activités, des jobs et du schedule. En plus des arcs ajoutés explicitement par l'utilisateur, il y a des arcs qui relient le début d'un job avec le début des activités contenues et la fin d'un job avec la fin des activités contenues. Il y a aussi des arcs qui assurent que toutes les activités et tous les jobs sont exécutés entre le début et la fin du schedule. Les arbres plantés représentant les fonctions objectifs et les besoins en ressources sont nécessaires pour représenter la combinaison de fonctions ou besoins simples. Les combinaisons sont des sommes, des produits, des minimums et des maximums pour les objectifs. Il s'agit de conjonctions et disjonctions pour les besoins.

3.2 Analyse et Classification

Le but de la seconde étape est de catégoriser le modèle dans une des classes prédéfinies. Cette classification est basée sur les caractéristiques du problème. Chaque classe de problèmes est définie par une combinaison de paires (caractéristique, valeur). La Table 3 présente un sous-ensemble des caractéristiques considérées. La dernière colonne spécifie les valeurs pour une classe bien connue. Un tiret signifie que la valeur peut être n'importe quoi. Cette table représente une version simplifiée de la définition des classes. En fait, il ne s'agit pas d'une simple conjonction de paires mais plutôt d'une formule booléenne, avec des négations, des disjonctions et des conjonctions de formules plus simples. Les atomes correspondent aux paires. Leur valeur de vérité est déterminée par analyse du modèle. Si la valeur renvoyée par l'analyse est égale à celle attendue, la formule atomique reçoit la valeur *Vrai*. Un modèle appartient à une classe de problèmes si l'évaluation de la formule qui définit la classe est *Vrai*.

De plus, des sous-formules récurrentes sont définies comme des caractéristiques de haut niveau, ou comme des modèles plus généraux dont les autres modèles peuvent hériter. Par exemple, le JSP avec Makespan est un cas spécial du JSP auquel on ajoute la caractéristique "avec Makespan". Le JSP est à son tour un cas de problème disjonctif. La hiérarchie des catégories forme un graphe dirigé acyclique (DAG). Cela signifie qu'un problème catégorisé dans une classe est aussi membre de toutes les classes dont elle hérite. Le résultat de la classification est donc une séquence de classes, plutôt qu'une seule classe. Cette séquence représente un ordre total des classes du problème compatible avec le DAG des classes. Cela signifie que, si une classe hérite d'une autre, elle doit apparaître avant son parent dans la séquence. Par contre l'ordre de classes qui ne sont pas apparentées est fixé arbitrairement.

Tab. 3 – Liste partielle des caractéristiques. La première colonne donne la caractéristique, la seconde définit le type de valeur. La troisième illustre les valeurs possibles pour le problème du Job-Shop (JSP).

Caractéristique	Type	JSP
Unit Processing Time	boolean	–
Fixed Processing Time	boolean	true
Preemption Allowed	enum	never
Common Release Dates	boolean	true
Common Deadlines	boolean	–
Deadlines Exist	boolean	false
Form of the Precedence Graph	enum	chains
Delay between Activities	boolean	false
No wait between Activities	boolean	false
Jobs inside Jobs	boolean	false
Number Of State Resources	integer	0
Maximum Capacity	integer	1
All Capacities are Equal	boolean	true
Reservoir Consumption	boolean	false
Reservoir Production	boolean	false
Setup Times	boolean	false
Disjunctive Requirements	boolean	false
All Activities in Jobs	boolean	true
Nb of Multi-Mode Activities	integer	0
Sum Of Requirements	integer	1
Objective Type	enum	minimize
Objective Form	enum	maximum
Objective Components	enum	end time
Objective Scope	enum	all activ.
All Due-Dates are equal	enum	–

L'analyse des caractéristiques en soi est accomplie par un ensemble de fonctions qui récupèrent l'information à partir de la représentation interne présentée plus haut. Avant l'analyse, une étape de normalisation est exécutée sur cette représentation. En particulier, le graphe de précédences est simplifié pour retirer les contraintes inutiles (réduction transitive). Les arbres pour les besoins et les objectifs sont aussi simplifiés. Par exemple, une somme de sommes est remplacée par une seule somme. Pour finir, les objets inutiles (machines inutilisées, jobs vides, par exemple) sont marqués comme tels.

Pour être utile, l'analyse doit être robuste aux variations de modélisation. Aeon compile les modèles dans une forme normalisée et l'analyse est accomplie sur la forme normalisée. Par exemple, le code de la Figure 3 montre une formulation alternative pour le JSP. Il y a plusieurs différences (activités multimodale, réservoirs, pas de jobs, fonction objectif explicite) par rapport au code de la Figure 1. Cependant, aeon le caté-

```
1 range machines;
2 range tasks;
3 int proc[tasks];
4 int mach[tasks];
5
6 Schedule<Mod> s();
7 Reservoir<Mod> M[i in machines](s,0,5,5,
  IntToString(i));
8 MultiModeActivity<Mod> A[i in tasks](s,1,
  IntToString(i));
9 forall(i in tasks) {
10   A[i].setProcTime(1,proc[i],proc[i]);
11   A[i].requires(1,M[mach[i]],3);
12 }
13 forall(i in tasks:i%nbmachines!=0)
14   A[i].precedes(A[i+1]);
15 s.minimizeObj(maxOf(all(i in tasks)
  completionTimeOf(A[i])));
16
17 GreedyTabuSynthesizer synth();
18 Solution<Mod> sol = synth.solve(s);
19 sol.printSolution();
```

FIG. 3 – Modèle Alternatif pour le Problème du Job-Shop

gorise correctement comme étant un JSP, ce qui est très désirable en pratique. En effet, c'est la sémantique du modèle qui est importante, pas la syntaxe.

3.3 Synthèse d'Algorithmes

Les classes responsables de la synthèse sont `ScheduleSynthesizer` et ses sous-classes (voir Table 2). Comme réfléchi sur la Figure 2, l'entrée de la synthèse est composée de trois parties : le modèle de l'utilisateur, sa classification et un choix fait par l'utilisateur pour un paradigme de recherche en particulier. La sous-classe de `ScheduleSynthesizer` choisie définit le paradigme de recherche (par exemple, la ppc pour `CP-Synthesizer`) et la méthode `solve` prend le modèle en argument.

A partir du résultat de la classification, le synthétiseur choisit la stratégie de résolution appropriée. Une stratégie est un algorithme de recherche spécifique à une classe de problèmes et qui va être instantié pour une instance particulière. Chaque synthétiseur associe une stratégie à chaque classe de problèmes. Par exemple, la classe `TSSynthesizer` associe la recherche Tabou de [2] à la classe Job-Shop avec Makespan. Chaque synthétiseur peut ne pas définir une stratégie pour toutes les classes de problèmes mais il est possible qu'il définisse une stratégie pour un problème plus général. Comme le résultat de la classification est un séquence de classes de problèmes, le synthétiseur cherche une stratégie pour la première classe. S'il n'y en a pas, il cherche pour la classe suivante. La séquence est visitée tant qu'il n'y a pas de stratégie qui correspond à une classe. Dans le pire des cas, le problème est reconnu comme un "problème d'ordonnancement général" pour lequel il y a une recherche par défaut de base.

Une fois la stratégie choisie, elle doit être instanciée pour le problème à résoudre. Le synthétiseur délègue ce travail à une sous-classe de la classe `Strategy`. Il y a à peu près une telle sous-classe pour chaque paire formée d'une classe de problème et d'un paradigme de recherche. Chaque sous-classe de `Strategy` est responsable de la mise en place et de l'exécution d'un algorithme pour le problème à résoudre. La difficulté est que, bien que la classe du problème soit connue, il peut être difficile de trouver l'information nécessaire à l'instanciation de la recherche. Pour faciliter cette étape, AEON fournit un ensemble de classes appelées vues. Les vues sont utilisées pour présenter le schedule et ses composants de manière unifiée, quelle que soit la façon dont ils ont été introduits. Différentes vues correspondent à différentes conceptions du problème. La vue la plus générale (`ScheduleView`) est une façon générique d'accéder à l'information, tandis que des vues spécifiques donnent un accès direct à un sous-ensemble de l'information utile pour certaines classes de problèmes. Par exemple, la vue `JobShopView` donne de l'information pour les JSPs. Elle fournit la même interface, quelle que soit la façon dont le problème a été modélisé par l'utilisateur (comme à la Figure 1 ou comme à la Figure 3).

3.4 Exécution de l'Algorithme

L'algorithme effectivement exécuté est différent pour chaque stratégie. Cependant, ils ont en commun qu'une solution est renvoyée. Les objets de la classes `Solution` assignent une valeur à chaque variable de décision du problème. Cette assignation est exprimée en fonction des objets du modèle. Par exemple, la méthode `getStartingTime( Activity<Mod> act)` renvoie le moment de départ d'une activité. En plus des temps de départ, les autre variables de décisions des activités sont la date de fin, l'ensemble des ressources effectivement utilisées, le mode (si il y en a plusieurs) et la présence ou l'absence (si elles sont optionnelles). La solution enregistre aussi la valeur de la fonction objectif associée. Le principal bénéfice des objets de solution est que le modèle reste indépendant. Il peut donc avoir plusieurs solutions qui peuvent être comparées. De plus les solutions peuvent servir de moyen de communication entre des stratégies qui coopèrent. Elles peuvent ainsi être utilisées

```
1 Solution<Mod> solve(Schedule<Mod> sched){
2   JobShopView view(sched);
3   range Activities = view.getActivities();
4   range Jobs = view.getJobs();
5   range Machines = view.getMachines();
6   int[] duration = all(i in Activities)
7                      view.getProcessingTime(i);
8   int[] machine = all(i in Activities)
9                            view.getMachine(i);
10  int[][] jobAct = all(j in Jobs)
11             view.getOrderedActivitiesOfJob(j);
12  JobshopAlgorithm ls(LocalSolver(),Activities,
13     Jobs, Machines, duration, machine, jobAct);
14  ls.solve();
15  SolutionView sol(view);
16  ls.saveSolution(sol);
17  return sol.getModelSolution();
18 }
```

FIG. 4 – Résoudre un Problème de Job-Shop

comme assignation de départ, pour fournir une borne sur l'objectif, ou pour guider des heuristiques.

Les solutions sont exprimées en fonction du modèle mais les stratégies travaillent sur des vues. Elles ont besoin d'une classe `SolutionView` pour exprimer la solution à partir de la vue. Un objet `SolutionView` est créé à partir d'une vue et les valeurs pour les variables de décision sont données dans les termes de la vue. La solution du modèle sous-jacente peut ensuite être récupérée à partir de sa vue. La Figure 4 présente le corps de la méthode `solve` de la class `DellAmico`. On y retrouve les classes `JobShopView` et `SolutionView`. Les lignes 2-11 montrent la création et l'utilisation de la vue pour les problèmes de Job-Shop. La recherche effective est déléguée à une autre classe appelée `JobshopAlgorithm` (lignes 12-14). La ligne 15 crée la vue pour la solution à partir de la vue du problème. Cette vue est ensuite remplie (ligne 16) et la solution est renvoyée à la ligne 17.

4 Ajouter des Classes de Problèmes et des Stratégies

Comme le principal objectif de ce travail est de simplifier l'utilisation d'algorithmes d'ordonnancement, il est aussi important de fournir des moyens simples d'étendre le système. En particulier, l'architecture d'AEON permet d'ajouter facilement des classes de problèmes, des synthétiseurs et des stratégies. L'extension des abstractions de modélisation n'est pas couvert car cela nécessite une plus grande modification du système. De nouvelles classes de problèmes peuvent être

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE Model SYSTEM "models.dtd">
3 <Model ID="JobShopWithMakespanWithTwoJobs">
4   <Constraints>
5     <IsA Name="Makespan"/>
6     <IsA Name="JobShop"/>
7     <Constraint Name="nbJobs" Value="2"/>
8   </Constraints>
9 </Model>
```

FIG. 5 – Définition XML du Job-Shop avec deux jobs.

définies dans des fichiers XML. Les synthétiseurs et les stratégies sont définis en étendant des classes existantes.

4.1 Ajout de Classes de Problèmes

Toutes les classes de problèmes et les caractéristiques de haut niveau sont définies dans des fichiers XML. Chaque classe est définie par un nom unique et une structure de contraintes que le problème doit respecter. Cette structure est récursivement composée des éléments suivants :

- SimpleConstraint : La caractéristique doit prendre une certaine valeur.
- And : Toutes les contraintes doivent être respectées.
- Or : Au moins une contrainte doit être respectée.
- Not : La contrainte ne peut pas être respectée.
- IsA : Les contraintes d'un autre modèle doivent aussi être respectées.

L'élément racine est appelé "Constraints" et il correspond à un "And". Pour ajouter un nouvelle classe, il faut écrire un fichier XML qui définit les contraintes à satisfaire. Il est facile de réutiliser les modèles existants grâce à l'héritage définit par l'élément "IsA". Par exemple, la Figure 5 montre un fichier pour le cas particulier du JSP avec exactement deux jobs qui peut être résolu en temps polynomial ([1]). Il s'agit d'une conjonction des contraintes que ce soit un Job-Shop, que l'objectif soit la minimisation du makespan et que le nombre de jobs soit deux.

4.2 Ajout de Stratégies

Une nouvelle stratégie est créée en étendant la classe `Strategy`, ce qui demande d'écrire deux méthodes : `solve(Schedule<Mod> s)` et `solve(Schedule <Mod> sched, Solution<Mod> initSol)`. La première méthode implémente la résolution d'un problème depuis le début et la seconde la résolution d'un problème en utilisant une solution existante.

```
1 class MySynthesizer extends TSSynthesizer{
2   MySynthesizer():TSSynthesizer(){
3     registerStrategy("JobShopWMakespanW2Jobs",
4                 new AkersAndFriedmanAlgorithm());
5   }
6 }
```

FIG. 6 – Ajouter une stratégie à TSSynthesizer

La solution initiale peut être laissée de côté, par exemple dans le cas d'une recherche gloutonne. Le corps de ces méthodes doit utiliser les vues. Cela est illustré sur la Figure 4 qui montre l'implémentation de la première méthode. La seconde est similaire. La seule modification est le remplacement de la ligne 14 par l'instruction `ls.solve(new SolutionView(view,initSol))` où une vue de la solution initiale est transmise à l'algorithme de recherche.

Une stratégie nouvellement créée doit être liée à une classe de problèmes au moyen d'un synthétiseur. Cet appariement se fait par la méthode `registerStrategy(string name, Strategy strategy)` définie dans la classe `Synthesizer`. Cette méthode associe une classe (définie par son nom) à une stratégie. Si une autre stratégie était déjà associée à la classe, la nouvelle remplace l'ancienne. Cette méthode est typiquement appelée dans le constructeur d'une nouvelle classe de synthétiseur. La Figure 6 montre un tel cas, ou un nouveau synthétiseur est définit comme une sous-classe de `TSSynthesizer`. Cela signifie qu'un JSP avec deux jobs va être résolu avec l'algorithme polynomial *ad hoc* et que les autres problèmes sont résolus avec une recherche Tabou.

Sur cet exemple, il est aussi clair que le choix de l'utilisateur pour un paradigme de recherche particulier (à la Figure 2) peut aussi être enlevé, permettant une recherche entièrement en boîte noire. Il suffit de créer un synthétiseur par défaut qui associe la meilleure stratégie à chaque classe de problèmes. Cependant, la meilleure stratégie n'est pas spécialement unique, même pour une sous-classe. Cela peut dépendre de contraintes temporelles, du besoin d'avoir des bornes supérieures et inférieures, du besoin d'optimalité et des caractéristiques individuelles de l'instance. Fournir plusieurs synthétiseurs augmente donc la flexibilité et l'efficacité du système.

4.3 Création de Nouvelles Stratégies de Façon Compositionnelle

De nouvelles stratégies peuvent aussi être construites à partir d'autres plus simples. L'architecture

```
1 class TS_CPJSP extends Strategy{
2   Strategy _s1;
3   Strategy _s2;
4   TS_CPJSP():Strategy(){
5     _s1 = new DellAmico();
6     _s2 = new CPJobShop();
7   }
8   Solution<Mod> solve(Schedule<Mod> s){
9     return _s2.solve(s,_s1.solve(s));
10  }
11  Solution<Mod> solve(Schedule<Mod> s,Solution<
    Mod> initSol){
12    return _s2.solve(s,_s1.solve(s, initSol));
13  }
14 }
15 class TS_CPSynthesizer extends
   ScheduleSynthesizer{
16  ScheduleSynthesizer _s1;
17  ScheduleSynthesizer _s2;
18  TS_CPSynthesizer():ScheduleSynthesizer(){
19    _s1 = new TSSynthesizer();
20    _s2 = new CPSynthesizer();
21  }
22  Solution<Mod> solve(Schedule<Mod> s){
23    string[] models = classify(s);
24    return _s2.solve(s,models,_s1.solve(s,
      models));
25  }
26 }
```

FIG. 7 – Deux implémentations pour une stratégie TS+CP

d'AEON permet de fabriquer des recherches composites par spécialisation ou par composition. La première possibilité est de créer une nouvelle stratégie pour une classe spécifique comme montré dans la sous-section précédente. A un niveau plus général, un synthétiseur peut créer systématiquement des stratégies composées à partir d'autres synthétiseurs. La Figure 7 présente les deux possibilités pour une composition simple : une recherche Tabou suivie d'une recherche en ppc. Les lignes 1-14 illustrent une stratégie composée pour le JSP et les lignes 15-26 montrent le code d'un synthétiseur enchaînant TS et CP. Les méthodes `classify` et `solve` avec plusieurs arguments sont définies dans la classe `ScheduleSynthesizer` et représentent les différentes étapes qui sont du ressort du synthétiseur : la classification et la résolution (avec ou sans solution initiale). Il est intéressant de voir comment le code du synthétiseur composé ressemble au code de la stratégie composée.

5 Expérimentations

Le but de cette section est de montrer que la généricité du système est compatible avec des résolutions effectives et efficaces de problèmes d'ordonnancement. Pour évaluer cela, nous avons choisi d'effectuer des expérimentations sur quelques jeux de test classiques, le problème du Job-Shop avec minimisation du makespan (JSP), le problème de l'Open-Shop avec minimisation du makespan (OSP) et le problème du Job-Shop avec minimisation du retard pondéré total (JSPWT). Pour chaque jeu de test, trois algorithmes synthétisés vont être considérés : une recherche locale (LS), une approche par ppc (CP) et un composé où une recherche Tabou donne une borne supérieure à la partie CP (LS+CP). Ces algorithmes vont être comparés avec l'implémentation en COMET [10] de respectivement la recherche Tabou de [2] pour le JSP, la recherche Tabou de [5] pour l'OSP et un algorithme de Metropolis [9] pour le JSPTW.

Les algorithmes LS sont les homologues des algorithmes originaux et ont les mêmes limites : 12.000 itérations pour le JSP et l'OSP et 600.000 pour le JSPTW. La recherche CP est limitée en temps à $max(300, 3 * \#\text{activités})$ secondes, c'est-à-dire 25 minutes pour les plus grandes instances.

Pour les algorithmes de recherche locale, vingts exécutions sont faites pour chaque instance. Ceux avec CP n'ont été exécutés qu'une fois car ils sont bien moins variables. Toutes les exécutions ont été réalisées sur un Intel Core 2 Duo, 1.66Ghz avec 1 Go de RAM.

La Table 4 présente un résumé des résultats pour le jeu de test. Des résultats plus détaillés se trouvent en ligne[4]. Pour chaque algorithme, l'erreur relative moyenne (MRE) est donnée. Le MRE est égal à $100 * (UB - LB)/LB$ où UB est la valeur moyenne de la solution trouvée par l'algorithme et LB est la meilleure borne inférieure connue pour chaque instance (et récupérée dans [14, 13, 10, 4]).

Pour illustrer une autre recherche hybride, nous avons aussi généré un recherche par voisinage large (LNS) pour l'OSP. Cette recherche est particulièrement efficace et a résolu toutes les instances sauf une en moins de deux minutes.

Cette table montre qu'il n'y a pas de différence significative entre une recherche générée par AEON et une recherche écrite à part. Bien sûr, l'approche CP n'est pas toujours utilisable pour les plus grands problèmes mais ce n'est pas une particularité d'AEON. Au contraire, l'utilisation de CP en conjonction avec une recherche locale permet de prouver l'optumalité de solutions trouvées heuristiquement. Concernant les temps d'exécution, l'approche CP n'est compétitive pour les problèmes plus grands mais la recherche locale est comparable à l'implémentation en COMET des algorithmes de référence.

[4] http ://becool.info.ucl.ac.be/aeon

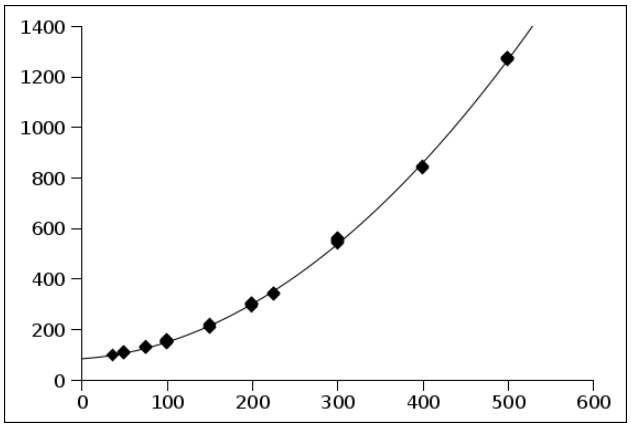

FIG. 8 – Temps (en millisecondes) pour analyser un problème et générer la recherche, en fonction du nombre d'activités.

Le coût induit par l'utilisation d'AEON est illustré à la Figure 8. Le temps utilisé pour la mise en place d'AEON et les opérations d'analyse, de classification et de génération est affiché en fonction du nombre d'activités du problème. La courbe suit une lente progression quadratique. Le temps d'analyse est de moins de 1,5 secondes même pour des problèmes à 500 activités.

6 Conclusion

Ce papier présente AEON, un système pour modéliser et résoudre des problèmes d'ordonnancement. Étant donné un modèle décrit selon un langage de modélisation de haut niveau, AEON reconnaît et classe sa structure et synthétise un algorithme de recherche approprié. L'algorithme synthétisé appartient à un paradigme particulier, tel que la recherche locale ou la ppc. L'approche permet d'exploiter la structure des modèles pour dériver des algorithmes dédicacés à des classes de problèmes.

AEON a certains traits fondamentaux : en premier, la classification du modèle ne dépend pas de la syntaxe ou des choix de modélisation. Les modèles sont transformés dans une forme normalisée sur laquelle l'analyse est effectuée, ce qui permet d'augmenter la robustesse de la modélisation. Deuxièmement, AEON est ouvert et extensible : de nouvelles classes de problèmes sont spécifiée en XML et des stratégies de recherche peuvent être ajoutées pour toutes les classes de problèmes. En outre, de nouveaux synthétiseurs peuvent être fabriqués à partir d'autres existants de façon simple.

TAB. 4 – Erreur relative moyenne (MRE) et temps d'exécution (en secondes) pour quatre algorithmes. Ref. désigne les algorithmes de référence, LS pour la recherche locale dans AEON, CP pour la ppc dans AEON et LS+CP pour un composé de LS et CP. Pour CP et LS+CP, le nombre entre parenthèses est le nombre d'instances pour lesquelles la recherche s'est finie et pour lesquelles le temps est compté. Pour l'OSP, la colonne CP présente deux valeurs. La deuxième est une recherche par voisinage large (LNS) qui est aussi générée par AEON.

Problème	#Inst.	MRE moyen				Temps moyen pour la meilleure solution			
		Ref.	LS	CP/LNS	LS+CP	Ref.	LS	CP/LNS	LS+CP
JSP	78	2.08	2.09	54.40	2.03	2.6	3.1	4.4(30)	3.4(52)
OSP	80	1.68	1.70	1.58/0.01	0.85	24.1	25.0	8.0(49)/ <120	30.2(50)
JSPTW	22	4.28	3.87	97.88	4.14	24.4	24.3	-(0)	-(0)

Les résultats expérimentaux montrent la faisabilité de l'approche. Le sur-coût d'AEON par rapport à des algorithmes dédiés est faible et le temps d'analyse est très acceptable et croit de manière quadratique avec la taille du problème.

Actuellement, nous travaillons à l'ajout d'une grande variété d'algorithmes pour de nombreuses classes de problèmes d'ordonnancement. A plus long terme, notre recherche va se concentrer dans deux directions. D'abord, la linéarisation automatique de modèles pour pouvoir utiliser des solveurs MIP ou pour obtenir des bornes inférieures par relaxation linéaire. Ensuite, l'intégration de recherches par défaut robustes pour des classes de problèmes générales (par exemple, l'ordonnancement disjonctif) où on trouve des contraintes hétéroclites.

Remerciements

Merci aux relecteurs pour leurs commentaires constructifs. Ce travail est partiellement soutenu par la région Wallonne, projet Transmaze (516207) et par le programme Pôles d'Attraction Interuniversitaire (politique scientifique fédérale belge).

Références

[1] S.B. Akers and J. Friedman. A non-numerical approach to production scheduling problems. *Operations Research*, 3 :429–442, 1955.

[2] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41 :231–252, 1993.

[3] R. Kolisch and A. Sprecher. Psplib — a project scheduling problem library. *European Journal of Operational Research*, 96 :205–216, 1997.

[4] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search : Application to single-mode scheduling problems. *In Proceedings MISTA-07, Paris*, 2007.

[5] Ching-Fang Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26 :109–126, 1999.

[6] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon : Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure*, pages 43–59, 2009.

[7] Wim Nuijten, T. Bousonville, Filippo Focacci, Daniel Godard, and Claude Le Pape. Towards an industrial manufacturing scheduling problem and test bed. *PMS*, 2004.

[8] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP 2004, Toronto (Canada)*, pages 557–571, 2004.

[9] Pascal Van Hentenryck and Laurent Michel. Scheduling abstractions for local search. In *CP-AI-OR'04, Nice*, pages 319–334, 2004.

[10] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[11] Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. *AAAI'07, Vancouver, British Columbia*, 2007.

[12] Pascal Van Hentenryck, Laurent Michel, Philippe Laborie, Wim Nuijten, and Jerome Rogerie. Combinatorial optimization in OPL studio. In *Portuguese Conference on Artificial Intelligence*, pages 1–15, 1999.

[13] Chao Yong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers & Operations Research*, 34 :3229–3242, 2007.

[14] Chao Yong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. A very fast ts/sa algorithm for the job shop scheduling problem. *Computers & Operations Research*, 35 :282–294, 2008.