Vers une unification des algorithmes de résolution de CSP

Narendra Jussien
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
email: `jussien@emn.fr`

Olivier Lhomme
ILOG
1681, route des Dolines
F-06560 Valbonne
email: `olhomme@ilog.fr`

Résumé

Ginsberg et McAllester [GM94] ont montré que les algorithmes de recherche de solution systématiques et non systématiques pouvaient être très proches. Dans cet article, nous souhaitons aller plus loin vers la compréhension des relations entre les approches systématiques et non-systématiques en introduisant le modèle **PLM**. Nous présentons un algorithme générique de recherche pour résoudre des CSP et nous montrons que de nombreux algorithmes de résolution peuvent se décomposer en utilisant un ensemble restreint de primitives. La connaissance de ces primitives permet non seulement de comparer plus aisément les algorithmes de recherche mais aussi de définir de nouveaux algorithmes.

1 Introduction

Ces dernières années ont vu l'apparition de nombreux algorithmes de résolution de *Problèmes de Satisfaction de Contraintes* (CSP – *Constraint Satisfaction Problems*). Il s'agit aussi bien d'améliorations du backtrack chronologique, d'algorithmes de recherche locale, d'algorithmes génétiques que d'approches hybrides combinant différentes méthodes.

Cette recrudescence d'algorithmes nouveaux amènent plusieurs questions. Pourquoi tant d'algorithmes ? Quels sont leurs points communs ? Leurs différences ? Est-il possible de les inscrire dans un cadre unique ? Étant donnée une méthode nouvelle, quelles en sont les aspects réellement nouveaux ?

Pour être capable de comparer différents algorithmes ou pour en définir de nouveaux, il est indispensable de connaître les principes et les propriétés des différents algorithmes. Ces propriétés semblent faciles à étudier : un cadre général pour les algorithmes de recherche en IA a été utilisé depuis des années. Néanmoins, si chaque algorithme de résolution est une instance de ce cadre général, il est légitime de se demander pourquoi tant d'algorithmes existent.

Le *Backjumping*, l'*apprentissage* [SV94, VS94, Pro93], les techniques de *filtrage*, des heuristiques sur l'*ordre des variables ou des valeurs*, les techniques de *réparation*, les méthodes *gloutonnes*, la *Recherche Taboue* (TS) [GL93], GSAT [SLM92]), les recherches basées sur les *conflits*, la revision, les techniques visant à briser les symétries, *Dynamic Backtracking* (DBT) [Gin93], MAC-DBT [JDB00], *Dynamic Domain Splitting* (DynDS) [JL98], PBS [Jun00], LDS [HG95], IDFS [Mes97], ... sont toutes des idées très intéressantes. Pourtant elles ne sont pas faciles à comparer ou à combiner ; certaines semblent incompatibles entre elles, mais comment le prouver ? Le cadre général des algorithmes de recherche en IA ne semble guère être utile pour répondre à nos questions. En effet, étant basé sur un espace d'états et des opérateurs, il semble trop général pour capturer les propriétés spécifiques que nous cherchons. Différentes tentatives d'unification de plusieurs algorithmes de recherche ont aussi été proposées (par exemple [Prc01]). Mais là encore, elles ne semblent pas suffisantes pour répondre à nos questions.

Dans cet article, nous montrons que la plupart des algorithmes de résolution peuvent se décomposer en trois composantes **P**, **L** et **M** où :

- **P** concerne la propagation,
- **L** concerne l'apprentissage (*learning*)
- **M** concerne le déplacement dans l'espace de recherche (*move*).

Les deux premières composantes, **P** et **L**, réduisent dynamiquement l'espace de recherche de manière à éviter des portions de celui-ci qui conduisent à une impasse. Elles exploitent, respectivement, de l'information sur le futur de la recherche et de l'information sur son passé. **M**, la troisième composante est responsable des déplacements dans l'espace de recherche. Elle choisit une direction dans l'espace de recherche non encore explorée et réalise un déplacement contraint par les composantes **P** et **L** qui permettent en effet de préciser les mouvements autorisés. Ces trois composantes se décomposent elles-mêmes en une série d'opérateurs basiques qui permettent de souligner les similitudes et les différences entre différents algorithmes de recherche.

Nous présentons dans cet article un algorithme générique de résolution, **PLM**, qui propage les contraintes du problèmes, apprend et se déplace dans l'espace de recherche. Il permet de fournir un moule commun simple pour de nombreux algorithmes existants. Il permet aussi de définir de nouveaux algorithmes combinaisons originales de composantes **P**, **L** ou **M** existantes. Cet algorithme générique a ainsi déjà produit un nouvel algorithme qui combine recherche locale, algorithmes de filtrage et utilisation de conflits et qui s'est montré performant sur des problèmes d'open shop : *decision-repair* (DR) [JL02] (précédemment appelé *path-repair* [JL00]).

En section 2, on trouvera une description des trois composantes communes aux algorithmes de résolution. La section 3 présente notre algorithme générique. Enfin, la section 4 montre comment divers algorithmes bien connus s'inscrivent dans ce cadre, introduisant ainsi une véritable taxonomie des algorithmes de résolution.

2 Trois composantes : P, L et M

Nous pensons que la plupart des algorithmes de résolution peuvent se décomposer en une composante d'apprentissage (*learning*), une composante de propagation et une composante de déplacement dans l'espace de recherche (*move*). Cette section décrit ces trois composantes.

Nous traitons ici des CSP représentés par un couple (V, C). C est un ensemble de contraintes et V un ensemble de variables. Le but d'un algorithme de résolution est de trouver une *solution*[1] vérifiant toutes les contraintes de C. Nous supposerons dans la suite que les variables des problèmes considérés peuvent avoir des valeurs aussi bien discrètes que continues et que les domaines de ces variables sont représentés par des contraintes *unaires*.

De plus, nous considérerons ici la phase d'énumération classiquement utilisée pour résoudre des CSP comme une séquence d'ajouts et de retraits de contraintes de *décision*. En effet, nous souhaitons ne pas restreindre cette phase d'énumération aux seules contraintes d'affectation. On recense en effet différents types de contraintes de décision : des contraintes de séquencement entre tâches pour l'ordonnancement, des contraintes de *splitting* pour les CSP numériques, ...

L'état courant d'un algorithme de résolution est représenté dans la suite de cet article par un 5-uplet $P = \{V, C, C_P, C_L, C_D\}$ où :

- V est l'ensemble des variables du problème,
- C est l'ensemble des contraintes initiales du problème,
- C_P est un ensemble de contraintes redondantes introduites par la composante de propagation de l'algorithme. Par exemple, lorsque l'on cherche à maintenir la consistance d'arc [Mac77] pendant la résolution, les réductions de domaines réalisées peuvent être considérés comme des ajouts de contraintes unaires.
- C_D est l'ensemble des *contraintes de décision* actives à ce point de la recherche.
- C_L est un ensemble de contraintes qui ont été *apprises* pendant la recherche (des états inconsistants, ...). Notons que de manière analogue à une mémoire *humaine* une partie de cette information peut être *oubliée* au cours de la recherche.

Un état pour lequel l'ensemble des solutions du CSP $(V, C \cup C_D)$ est non vide est appelé un état *consistant*.

Par la suite, nous manipulerons les ensembles de contraintes aussi bien sous forme d'ensemble que de formules logique. L'ensemble vide sera ainsi représenté par **true**.

La section 3 décrit en détail l'algorithme générique dont nous donnons ici juste un aperçu général. Il s'agit d'une boucle en trois étapes :

1. *Utiliser la composante **P*** : propager l'ensemble de contraintes courant $C \cup C_D$ amenant ainsi des modifications à C_P.

2. Si cette première étape de propagation conduit à un état inconsistant, *utiliser la composante **L*** pour mémoriser l'impasse courante (induisant ainsi des modifications sur C_L).

3. *Utiliser la composante **M*** : elle permet de se déplacer dans l'espace de recherche en modifiant l'ensemble C_D. Un mouvement peut être : un mouvement *en avant* comme les extensions classiques de solutions partielles ; un mouvement *en arrière* (retour arrière classique, *back-jumping*, ...) ; ou encore, d'autres types de mouvements.

Les trois sections suivantes décrivent précisément la sémantique opérationnelle des composantes **P**, **L** et **M** pour les modifications de $P = \{V, C, C_P, C_L, C_D\}$, l'état courant de l'algorithme de résolution.

[1]Cette notion de *solution* dépend évidemment du type de problèmes que l'on cherche à résoudre : pour un CSP *classique*, une solution est trouvée lorsque les domaines des variables sont tous réduits à un singleton ; pour un CSP numérique, une solution est trouvée lorsque les intervalles représentant les domaines des variables sont de taille assez réduite ; pour des problèmes d'ordonnancement, une solution est trouvée lorsque toutes les disjonctions ont été arbitrées ; ...

2.1 La composante P

Le but de la composante **P** est de propager l'information à travers le réseau de contraintes dès qu'une décision est prise pendant la recherche. Cette composante est constituée d'un opérateur (`filter`) et d'un analyseur d'état (`check`).

L'opérateur `filter` filtre l'espace de recherche pour en éviter certaines parties qui mènent à une impasse. Ce filtrage est réalisé en pratique par l'ajout de contraintes (redondantes) à l'état courant en prenant en considération l'ensemble C_D actuel de contraintes de décision. L'opérateur `filter` peut ne pas être complet. On a :

$$P_{\texttt{filter}}(\{V, C, C_P, C_L, C_D\}) = \{V, C, C'_P, C_L, C_D\}$$

où $C \wedge C_D \implies C'_P$. En pratique, ces contraintes ajoutées par la composante **P** sont des informations sur les réductions de domaine réalisables.

L'opérateur `check` est utilisée pour analyser l'état résultant de `filter`. C'est un opérateur d'approximation qui peut donner trois résultats différents :

- **solution trouvée**.
- **pas de solution** : aucune solution n'existe étant donné l'ensemble courant de contraintes de décision C_D (la plupart du temps, cette situation est identifiée lorsque le domaine d'une des variables du problème est vide). Lorsque cette réponse est donnée, il est indispensable que la composante **P** soit capable de fournir un *conflit* (*c.-à-d.* un ensemble $N \subset C_D$ de contraintes de décision tel que le CSP $(V, C \cup N)$ ne possède pas de solution). Notons que dans tous les cas, l'ensemble C_D lui-même est un conflit. Néanmoins, pour certains algorithmes, il possible et préférable de travailler sur des conflit plus petits et plus précis[2].
- **pas assez d'information (p.a.i.)** : L'état courant de la recherche ne permet pas de statuer sur l'existence ou non d'une solution au problème courant (déterminé par les décisions prises jusqu'à maintenant). Il faut faire d'autres choix.

Il nous semble important de remarquer que le coût de calcul de `check` doit rester relativement bas. Par exemple, lorsqu'on travaille sur les CSP classiques avec un filtrage par consistance d'arc, `check` consiste a considérer uniquement les contraintes unaires et peut s'écrire :

> *S'il existe un domaine vide dans l'état courant alors la réponse est **pas de solution**. Si tous les domaines sont réduits à des singletons alors la réponse est **solution trouvée**. Dans tous les autres cas, la réponse est **pas assez d'information**.*

2.2 La composante L

Le but de la composante **L** est de s'assurer que le mécanisme de recherche va éviter (autant que faire se peut) de retourner sur des états incohérents déjà rencontrés. L'apprentissage considéré ici, par analogie avec le cerveau humain, est un interaction entre deux processus : un processus d'enregistrement et un processus d'oubli. Dans notre modèle, la composante **L** est composées de trois opérateurs : un opérateur d'enregistrement `record`, un opérateur d'oubli `forget` et un opérateur de test `check_memory`.

[2]Le calcul de conflits est étudié depuis les travaux sur les TMS [Doy79] et les ATMS [dK86]. Des travaux plus récents (dont [Jun01] et [Jus01]) montrent que des techniques efficaces existent pour calculer des conflits très précis lorsque l'on utilise des techniques de filtrage.

check_memory est un prédicat qui signale si un état donné est connu pour être inconsistant ou non (il consulte la *mémoire* d'état constituée par l'ensemble C_L).

record enregistre une nouvelle information N. On a :

$$L_{\texttt{record}}(\{V, C, C_P, C_L, C_D\}, N) = \{V, C, C_P, C'_L, C_D\}$$

Les pré-conditions de l'opérateur record sont les suivantes : C_L est un ensemble de contraintes redondantes ($C \Longrightarrow C_L$), et C_D est un nouvelle impasse ($C \Longrightarrow \neg N$, $N \subset C_D$, $L_{\texttt{check\_memory}}(C_L, C_D)$ is true).

L'objectif de record est d'enregistrer le fait que la position courante conduit à une impasse (et accessoirement de s'assurer que **M** ne va pas revenir, au moins immédiatement et de préférence jamais, sur ce même état). Ainsi, on doit avoir $L_{\texttt{check\_memory}}(C'_L, C_D) =$ false. Une solution consiste à choisir C'_L de telle sorte que $C'_L \Longrightarrow \neg N$. On doit aussi maintenir que C'_L est un ensemble de contraintes redondantes ($C \Longrightarrow C'_L$), et que C'_L contient plus d'information que C_L ($C'_L \Longrightarrow C_L$).

L'opérateur forget quant à lui a pour rôle d'oublier une partie de l'information stockée jusqu'à présent. On a :

$$L_{\texttt{forget}}(\{V, C, C_P, C_L, C_D\}, N) = \{V, C, C_P, C'_L, C_D\}$$

Les pré-conditions pour un appel à $L_{\texttt{forget}}$ sont : $L_{\texttt{check\_memory}}(C_L, C_D) =$ true et $C \Longrightarrow C_L$. Puisque $L_{\texttt{forget}}$ est un processus d'*oubli*, il va se contenter d'assurer que $C_L \Longrightarrow C'_L$.

2.3 La composante M

La composante **M** contrairement aux autres, ne cherche pas à réduire l'espace de recherche mais plutôt à l'explorer. La position courante est décrite, comme nous l'avons dit plus haut, par l'ensemble C_D des décisions prises. Dans de nombreux algorithmes de résolution, le déplacement dans l'espace de recherche dépend de la consistance ou non du CSP associé à l'état courant. Par exemple, dans le cas du retour-arrière chronologique, lorsque le CSP courant peut être consistant, le déplacement dans l'espace de recherche consiste à donner une valeur à une variable non instanciée, alors que si l'algorithme se trouve dans une impasse, un état précédent est restauré. Ainsi, la composante **M** est constituée de deux opérateurs : extend sélectionne et effectue un mouvement lorsque le CSP associé à l'état courant est (partiellement) consistant et repair sélectionne et effectue un mouvement dans le cas contraire.

Les opérateurs extend et repair sont amenés à modifier la position courante dans l'espace de recherche *c.-à-d.* l'ensemble de contraintes de décisions. On a :

$$M_{\texttt{extend}}(\{V, C, C_P, C_L, C_D\}) = \{V, C, C_P, C_L, C'_D\} \quad (1)$$
$$M_{\texttt{repair}}(\{V, C, C_P, C_L, C_D\}) = \{V, C, C'_P, C_L, C''_D\} \quad (2)$$

Soit $S = \{V, C, C_P, C_L, C_D\}$ l'état courant. Les pré-conditions pour extend sont :
- $P_{\texttt{check}}(S) =$ pas assez d'information (la position courante n'est pas contradictoire)
- $P_{\texttt{filter}}(S) = S$ (le point fixe a été atteint pour la propagation)
- $L_{\texttt{check\_memory}}(C_L, C_D) =$ true (la position courante n'est pas non plus contradictoire avec les informations conservées jusqu'à présent)

Les pré-conditions pour repair sont :

- $P_{\texttt{check}}(S) =$ pas de solution (la position courante est contradictoire)
- $L_{\texttt{check\_memory}}(C_L, C_D) = \texttt{false}$ (la position courante est contradictoire avec les informations conservées jusqu'à présent – en fait, on vient juste de l'apprendre comme on le verra dans l'algorithme général décrit section 3)

Dans le cas d'`extend`, l'ensemble des mouvements autorisés est restreint par l'ensemble C des contraintes originales, C_P le résultat de la propagation réalisée jusqu'à présent et C_L l'ensemble des impasses rencontrées par le passé (en tout cas, celles dont on se souvient). Plus précisément, dans l'équation 1, C'_D est tel que

$$P_{\texttt{check}}(\{V, C, C_P, C_L, C'_D\}) \neq \text{pas de solution}$$

c.-à-d. la nouvelle position est compatible avec l'ancienne ; et $L_{\texttt{check\_memory}}(C_L, C'_D) = \texttt{true}$ *c.-à-d.* la nouvelle position n'est pas réputée être une impasse.

Pour `repair`, l'ensemble des mouvements autorisés est restreint par C and C_L comme pour `extend`. Par contre, l'ensemble C_P peut demander des modifications pour s'assurer que $C \cup C'_D \implies C'_P$. En effet, certaines inférences (contraintes redondantes) de C_P peuvent dépendre de décisions apparaissant dans C_D mais qui ne sont plus valides dans C'_D. Elles doivent donc disparaître de C'_P car elles ne sont plus redondantes.

Dans certains cas, cette opération peut être réalisée facilement. Par exemple, si l'on regarde le retour-arrière chronologique standard (*Standard Backtracking* – `BT`), l'opérateur `repair` consiste à remplacer la décision (positive) la plus récente de C_D par sa négation et à supprimer de C_P toutes les inférences réalisées depuis le dernier appel à `extend` (c'est en effet celui qui a ajouté la décision remise en cause). Il faut en plus retirer de C_D toutes les décisions négatives prises entre temps.

Bien sûr, d'une manière générale, la résolution de ce problème est liée au retrait dynamique de contraintes. Par exemple, [Bes91] a introduit le maintien dynamique de l'arc-consistance. Plus récemment, [JDB00] utilise la notion d'*explanation* (ensemble de contraintes permettant de *justifier* des réduction de domaines) pour réaliser de telle opérations. Notons, d'ailleurs que ces *explications* peuvent être utilisées pour calculer les conflits précis [Jus01] dont se sert la composante **P**.

en pratique, les deux opérateurs associés à une composante **M** peuvent partager de l'information sur le chemin suivi pour atteindre le noeud courant. Par exemple, dans le backtrack chronologique, une pile permet de retourner àq un noeud ancêtre. Par simplicité, nous ne ferons pas apparaitre explicitement ce type d'information.

3 PLM : un algorithme générique

La combinaison des trois composantes (**P**, **L** et **M**) permet d'obtenir un algorithme générique de résolution de CSP : l'algorithme **PLM** (**P**ropager, Apprendre – **L**earn – et se Déplacer –**M**ove).

3.1 Description de l'algorithme

L'algorithme générique **PLM** est donné figure 1. Cet algorithme permet de vérifier les pré- et post-conditions des opérateurs liés aux composantes **P**, **L** et **M**. Les principes de cet algorithmes sont les suivants :

procedure PLM(V,C,C_D)
(1) **début**
(2) $Pb \leftarrow \{V, C, \emptyset, \emptyset, C_D\}$
(3) **répéter**
(4) $Pb \leftarrow P_{\texttt{filter}}(Pb)$
(5) `switch` $P_{\texttt{check}}(Pb)$
(6) **cas pas de solution :**
(7) $Pb \leftarrow L_{\texttt{forget}}(M_{\texttt{repair}}(L_{\texttt{record}}(Pb, N)))$
(8) **cas solution :**
(9) return Pb
(10) **cas pas assez d'information :**
(11) $Pb \leftarrow M_{\texttt{extend}}(Pb)$
(12) `finswitch`
(13) `jusqu'à` conditions of termination
(14) `fin`

FIG. 1 – L'algorithme générique PLM

- La recherche part d'un ensemble initial de contraintes de décision. Celui peut être aussi bien l'ensemble vide (c'est le cas pour un algorithme basé sur le retour arrière) qu'une affectation complète des variables (pour une recherche locale par exemple).
- Des décisions sont prises (`extend`) et propagées (`filter`) jusqu'à ce qu'une contradiction apparaisse.
- Lorsque cela se présente (ligne 6), une information sur l'impasse (en fait, un conflit N) est *appris* (`record`), l'état courant est *réparé* (`repair`) et certaines informations sont *oubliées* (`forget`).
- La recherche se termine dès qu'une solution est trouvée (ligne 8) ou lorsque les conditions de terminaison sont remplies (ligne 13). Ces conditions concernent par exemple le nombre maximal d'itérations autorisées, la découverte d'une preuve de l'absence de solution, ...

3.2 Recherches complètes et recherche incomplètes

Différentes combinaison de composante **P**, **L** et **M** donnent différents algorithmes de résolution. Il est intéressant de savoir si une combinaison donnée permet d'obtenir une recherche complète ou non. En général, la complétude vient d'une combinaison relativement subtile du travail réalisé conjointement par les trois composantes. Il est ainsi assez ardu de caractériser de manière générique cette propriété. Néanmoins, nous pouvons montrer que si des opérateurs *plus forts* de propagation et d'apprentissage sont utilisés, alors une recherche complète reste complète.

Une composante de propagation P est **plus forte** qu'une composante de de propagation Q ($P \gg Q$) ssi :

$$\begin{array}{rcl}
P_{\texttt{check}}(S) = \text{p. a. i.} & \Longrightarrow & Q_{\texttt{check}}(S) = \text{p. a. i.} \\
Q_{\texttt{check}}(S) \neq \text{p. a. i.} & \Longrightarrow & P_{\texttt{check}}(S) = Q_{\texttt{check}}(S) \\
\forall S, P_{\texttt{filter}}(S) = S_P, & & \\
Q_{\texttt{filter}}(S) = S_Q & \Longrightarrow & (C_P \Rightarrow C_Q)
\end{array}$$

où $S_P = \{V, C, C_P, C_L, C_D\}$, $S_Q = \{V, C, C_Q, C_L, C_D\}$ et p. a. i. = pas assez d'information.

Une composante d'apprentissage L est **plus forte** qu'une composante d'apprentissage K ($L \gg K$) ssi :

$$\begin{array}{rcl} L_{\texttt{check\_memory}}(C_\ell, C_d) = \texttt{true} & \Longrightarrow & K_{\texttt{check\_memory}}(C_\ell, C_d) = \texttt{true} \\ \forall S, L_{\texttt{op}}(S) = S_L, & & \\ K_{\texttt{op}}(S) = S_K & \Longrightarrow & (C_L \Rightarrow C_K) \end{array}$$

où $S_L = \{V, C, C_P, C_L, C_D\}$, $S_K = \{V, C, C_P, C_K, C_D\}$ et `op` est soit `record` soit `forget`.

Proposition 1 *Si* ***P/L/M*** *est une recherche systématique, alors* ***P'/L'/M*** *tel que* ***P'*** $\gg$ ***P*** *et* ***L'*** $\gg$ ***L*** *définit aussi une recherche systématique.*

> **Preuve (idée) :** *S'il n'y a pas de solution pour C, alors toutes les positions correspondant aux mouvements autorisés pour* ***P/L/M*** *seront ou bien montrées ou bien inferées comme étant des impasses. Comme* ***P'*** $\gg$ ***P*** *et* ***L'*** $\gg$ ***L P'*** *et* ***L'*** *infèrent au moins les même impases sans avoir à les essayer. Ainsi,* ***M*** *devrait aussi explorer dans* ***P'/L'/M*** *tous les mouvements autorisés.*

4 Instancier le modèle PLM

Dans ce qui suit, nous exprimons plusieurs algorithmes bien connus (et quelques nouveaux) comme des instances de **PLM**. Nous présentons aussi bien des approches complètes que des approches incomplètes. Nous présentons ensuite une taxonomie des algorithmes de recherche en partant des différentes valeurs possibles de **P**, **L** et **M**.

4.1 Algorithmes complets

BT et MAC Le retour-arrière chronologique standard (*Standard Backtracking*) ne réalise aucune propagation et se contente de tester la satisfiabilité de contraintes complètement instanciées. Lorsque une contradiction apparaît, le conflit est tout simplement défini comme étant l'ensemble C_D des décisions prises jusqu'à présent. Le déplacement suite à une propagation réussie revient à prendre une nouvelle décision. Se déplacer à la suite de la découverte d'une impasse revient à reconsidérer le dernier choix fait comme expliqué dans la section 2.3. La composante **L** est anecdotique puisque l'information *apprise* n'est utilisée qu'une seule fois. On a les équations suivantes :

$$\begin{array}{rcl} P_{\texttt{filter}}(S) & = & S \\ L_{\texttt{record}}(\{\ldots, C_L, \ldots\}, N) & = & \{\ldots, \{\neg N\}, \ldots\} \\ L_{\texttt{forget}}(\{\ldots, C_L, \ldots\}) & = & \{\ldots, \emptyset, \ldots\} \\ M_{\texttt{extend}}(\{\ldots, C_D\}) & = & \{\ldots, C_D \cup \{c\}\} \\ M_{\texttt{repair}}(\{\ldots, C_P, \ldots, C_D\}) & = & \{\ldots, C'_P, \ldots, C'_D \cup \{\neg c'\}\} \end{array}$$

où c est une contrainte de décision qui permet d'aller à un noeud fils de C_D, C'_D correspond à l'ancêtre (dans l'arbre de recherche) de C_D pour qui il reste au moins un fils à explorer et c' est la contrainte de décision qui représente le fils qui vient juste d'être rejeté.

L'introduction d'une heuristique de sélection de variable est facilement réalisable dans BT en spécifiant le choix de la contrainte de décision c utilisée dans $M_{\mathtt{extend}}$.

Entrelacer énumération et maintien d'arc-consistance dans BT consiste à simplement modifier l'opérateur de filtrage de la composante $\mathbf{P}$:

$$P_{\mathtt{filter}}(\{V, C, C_P, C_L, C_D\}) = \{V, C, C'_P, C_L, C_D\}$$

où C'_P est la clôture arc-consistance de $C \cup C_D$. Notons qu'on obtient alors l'algorithme MAC [SF94].

Dynamic Backtracking DBT [Gin93] remplace BT par une modification plus localisée de la solution courante. La complétude et l'efficacité de cette approche sont assurées par un mécanisme d'enregistrement spécifique. L'opérateur de filtrage de la composante $\mathbf{P}$ est capable de fournir des conflits beaucoup plus précis permettant ainsi d'améliorer le comportement de l'algorithme vis-à-vis des points de retour. De plus, la composante $\mathbf{L}$ est améliorée pour enregistrer les conflits passés, et faire la résolution de conflits.

$$\begin{aligned} L_{\mathtt{record}}(\{\ldots, C_L, \ldots\}, N) &= \{\ldots, C_L \cup \{\neg N\}, \ldots\} \\ L_{\mathtt{forget}}(\{\ldots, C_L, \ldots\}) &= \{\ldots, C'_L, \ldots\} \end{aligned}$$

Étant donné que les contraintes de C_L sont de la forme $\neg(x_{j_1} = v_{j_1} \wedge \ldots \wedge x_{j_k} = v_{j_k})$, C'_L est construit à partir de C_L est supprimant toutes les contraintes de ce dernier ensemble qui ont au moins deux variables x_i $(i \in \{j_1 \ldots j_k\})$ telles que la valeur courante de x_i n'est pas v_i. Notons que si on généralise ce processus en retirant les contraintes ayant non pas au moins deux telles variables mais plutôt $k+1$ telles variables, on obtient une généralisation bien connue de DBT : *k-relevance bounded learning* [BM96].

L'opérateur `repair` de l'opérateur **M**consiste tout simplement ici à supprimer la plus récente contrainte de décision.

$$M_{\mathtt{repair}}(\{\ldots, C_P, \ldots, C_D\}) = \{\ldots, C'_P, \ldots, C'_D\}$$

où C'_D est $C_D \setminus \{c\}$ avec c la plus récente contrainte de décision tel que C'_D soit compatible avec C_L (par construction, on sait qu'il existe au moins une telle contrainte de décisions, elle appartient au dernier conflit enregistré). N'oublions pas qu'on doit alors retirer de C_P les inférences qui dépendent de c donnant C'_P.

Conflict-directed Backjumping *Conflict-directed BackJumping* (CBJ) [Pro93] est une autre amélioration de BT. Les opérateurs sont identiques à ceux de DBT excepté :

$$M_{\mathtt{repair}}(\{\ldots, C_P, \ldots, C_D\}) = \{\ldots, C'_P, \ldots, C'_D\}$$

où C'_D correspond a un bactrack jusqu'a l'état dans lequel la décision c a été prise, c étant la plus récente contrainte de décision tel que C'_D soit compatible avec C_L.

MAC-CBJ [Pro95] intégre le maintien de l'arc-consistance dans CBJ. La seule différence est la définition de la composante de filtrage de la composante $\mathbf{P}$ qui calcule la clôture par arc-consistance de $C \cup C_D$ comme c'ets le cas pour MAC.

MAC-DBT MAC-DBT [JDB00] et DynDS [JL98] ajoutent, respectivement, le maintien de l'arc-consistance et de la consistance de borne à DBT.

Leur description dans le modèle **PLM** se déduit de celle de DBT en modifiant la composante **P** de la même façon que cela a été fait pour MAC ou MAC-CBJ ci-dessus.

4.2 Algorithmes incomplets

Le modèle **PLM** est aussi capable de prendre en compte les approches incomplètes.

Limited discrepancy search LDS [HG95] est un algorithme incomplet lorsque l'on borne le nombre d'ecarts autorisés par rapport a une instanciation donnée par une heuristique. Les opérateurs sont identiques à ceux du backtrack chronologique, excepté $M_{\mathtt{extend}}$ qui n'enregistre les alternatives que si le nombre d'ecarts autorisés est atteint.

Recherche taboue Dans l'algorithme TS [GL93], une position dans l'espace de recherche C_D est une affectation complète des variables. Une *recherche taboue* est caractérisée par la longueur K de sa liste *taboue*. Cette liste définit l'ensemble des positions passées qui sont déclarées *taboues* et vers lesquelles l'algorithme ne peut retourner. On a :

$$\begin{array}{rcl} P_{\mathtt{filter}}(S) & = & S \\ L_{\mathtt{record}}(\{\ldots, C_L, \ldots\}, N) & = & \{\ldots, C_L \cup \{\neg N\}, \ldots\} \\ L_{\mathtt{forget}}(C_L) & = & C'_L \\ M_{\mathtt{extend}} & & \text{not used} \\ M_{\mathtt{repair}}(\{\ldots, C_D\}) & = & \{\ldots, C'_D\} \end{array}$$

où C'_L est l'ensemble C_L si $|C_L| < K$ et $C_L \setminus \{c\}$ sinon (c étant la contrainte la plus ancienne de C_L) et où C'_D est une position voisine[3] de C_D qui reste compatible avec C_L. Enfin, le conflit fournit par l'algorithme est, comme pour BT, l'ensemble C_D lui-même.

GSAT GSAT [SLM92] est un algorithme de résolution de CSP booléens. Comme dans TS, la position courante C_D de l'algorithme dans l'espace de recherche est représentée par une affectation complète des variables du problème. Cet algorithme est assez proche de TS excepté le fait qu'aucune information n'est enregistrée et que les déplacement dans le voisinage sont plus contrôlés : GSAT est caractérisé par les deux quantités `max-tries` and `max-split`. Si le nombre d'itérations effectuées jusqu'à présent est un multiple de `max-split`, l'opérateur `repair` de la composante **M** choisit une nouvelle position au hasard. Dans le cas contraire, ce même opérateur *inverse* (*flit*) la variable qui permet de diminuer le plus possible le nombre de contraintes insatisfaites de C.

Decision-repair (*alias* path-repair DR [JL02] permet de profiter de la propagation de contraintes apportée par des algorithmes tels que (comme MAC-DBT) et de la liberté (dans l'exploration de l'espace de recherche) apportée par les algorithmes de recherche locale comme TS. On a :

[3] La définition d'un tel voisinage dépend du problème résolu.

$$\begin{aligned}
P_{\mathtt{filter}}(\{V, C, C_P, C_L, C_D\}) &= \{V, C, C'_P, C_L, C_D\} \\
L_{\mathtt{record}}(\{\ldots, C_L, \ldots\}, N) &= \{\ldots, C'_L \cup \{\neg N\}, \ldots\} \\
L_{\mathtt{forget}}(S) &= S \\
M_{\mathtt{extend}}(\{\ldots, C_D\}) &= \{\ldots, C_D \cup \{c\}\} \\
M_{\mathtt{repair}}(\{\ldots, C_P, \ldots, C_D\}) &= \{\ldots, C'_P, \ldots, C'_D\}
\end{aligned}$$

où C'_P est la clôture par arc-consistance de $C \cup C_D$; C'_L est C_L si $|C_L|$ est inférieur a la taille max de la liste taboue et $C_L \setminus \{\text{plus ancienne contrainte}\}$ sinon ; c est une contrainte de décision qui permet d'aller à un noeud fils de C_D ; C'_D est C_D minus c' une contrainte de décision telle que C'_D est compatible avec C_L (si une telle contrainte c' n'existe pas, une condition de terminaison de la recherche est atteinte). Notons que C_P doit être *nettoyé* des inférences qui dépendent de c' permettant d'obtenir l'ensemble C'_P.

`DR` est un exemple des nouveaux algorithmes pouvant être obtenus en utilisant le modèle **PLM**.

4.3 Une taxonomie des algorithmes de recherche

À partir des exemples précédents d'instantiation de notre modèle, on peut définir une liste (non exhaustive) des valeurs possibles pour les opérateurs introduits jusqu'à présent. À chacune des ces valeurs, nous attribuerons un code nous permettant d'introduire une nouvelle notation pour caractériser n'importe quel algorithme de recherche : **P/L/M**.

La composante de Propagation Cette composante est caractérisée par :

- son opérateur de filtrage : un simple test de consistence (`o`) comme dans `BT`, un algorithme classique de maintien de consistance locale (*par ex.* l'arc-consistence – `ac`, la consistance de bornes – `bc`), ...
- son mécanisme de calcul de conflit : aucun (`X-o`) ou un mécanisme *pragmatique* comme celui de [Jus01] (`X-e`) pour un opérateur de filtrage donnée `X`.

La composante d'apprentissage (Learning) La composante d'apprentissage est caractérisée par la durée de vie des informations enregistrées : nulle (*i.e.* information non utilisée) (`o`), usage unique (`su`), usage limité dans le temps (`tb`), usage limité par la taille (`sb`) comme dans [SV94], usage limité par la pertinence *relevance* use (`rb`), ...

La composante de déplacement (Moving) Nos exemples montrent deux types de mouvements : des déplacements de type *retour-arrière* (`bt` : *backtrack*) et des déplacements de type *saut* (dans l'arbre de recherche – déplacements dans le voisinage d'une position – `jp`).

Une taxonomie L'utilisation des codes proposés ci-dessus permet une description concise des exemples présentés. Ces description permet d'en saisir rapidement les similitudes et les différences. Le tableau 1 donne une description au format **PLM** des différents algorithmes.

Cette description permet une représentation en trois dimensions de l'espace des algorithmes de résolution. Cet espace présente de nombreux points non encore explorés.

Algorithme	Modèle **PLM**
BT	o-o/su/bt
MAC	ac-o/su/bt
CBJ	o-e/su/bt
MAC-CBJ	ac-e/su/bt
DBT	o-e/rb/jp
MAC-DBT	ac-e/rb/jp
DDS	bc-e/rb/jp
TS	o-o/tb/jp
GSAT	o-o/su/jp
PR	ac-e/tb/jp

TAB. 1 – Une description au format **PLM** de certains algorithmes classiques

Ainsi, notre modèle générique permet non seulement de les identifier mais aussi un cadre unique pour exprimer leur comportement[4].

5 Conclusion

La contribution principale de cet article est de considérer les algorithmes de résolution comme une combinaison de trois composantes : **P**, **L** et **M**. Il s'agit d'une étape vers la caractérisation de l'espace des algorithmes de résolution. Cette approche permet de mieux comprendre les relation entre les différents algorithmes. De plus, de nouvelles combinaisons des composantes **P**, **L** et **M** conduisent à de nouveaux algorithmes potentiellement intéressants. Cela a été le cas pour, par exemple, `MAC-DBT` [JDB00] et `DR` [JL02]. Il existe enfin une plate-forme de d'expérimentations : le système `PaLM` [Jus01].

Notre schéma peut être compléter dans différentes directions. Par exemple, nous avons vu que certaines astuces d'implémentation ne pouvaient être exprimées simplement dans notre modèle (les connections cachées entre les deux opérateurs de la composante **M**). D'autre part, nous avons supposé que l'information à conserver ne provenait que de l'analyse d'un échec de l'opérateur de filtrage. Ce n'est pas la seule possibilité : on peut aussi modifier l'algorithme général pour qu'il appelle les opérateurs de la composante **L** aussi en cas de succès de la propagation. Une troisième voie pour compléter notre schéma serait d'intégrer des connaissances ne pouvant s'exprimer sous forme de contraintes à la composante d'apprentissage ; ces nouvelles informations pouvant alors être utilisées dans les opérateurs de la composante **M**.

Références

[Bes91] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

[BM96] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[4]Il existe une implémentation de cet algorithme général : le système `PaLM` [JB00]. Cette implémentation fournit aussi des routines de calcul de conflit telles que présentées dans [Jus01].

[dK86] Johan de Kleer. An assumption-based tms. *Artificial Intelligence*, 28 :127–162, 1986.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12 :231–272, 1979.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[GL93] F. Glover and M. Laguna. *Modern heuristic Techniques for Combinatorial Problems, chapter Tabu Search, C. Reeves.* Blackwell Scientific Publishing, 1993.

[GM94] Matthew Ginsberg and David McAllester. GSAT and dynamic backtracking. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94 : Second International Workshop, Orcas Island, Seattle, USA).

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings IJCAI-95 ; Vol. 1*, pages 607–615, Montréal, Québec, Canada, 1995.

[JB00] Narendra Jussien and Vincent Barichard. The palm system : explanation-based constraint programming. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[JDB00] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[JL98] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.

[JL00] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.

[JL02] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 2002. to appear.

[Jun00] Ulrich Junker. Preference-based search for scheduling. In *Proceedings AAAI/IAAI 2000*, pages 904–909, 2000.

[Jun01] Ulrich Junker. QUICKXPLAIN : Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.

[Jus01] Narendra Jussien. e-constraints : explanation-based constraint programming. In *CP01 Workshop on User-Interaction in Constraint Satisfaction*, Paphos, Cyprus, 1 December 2001.

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

[Mes97] Pedro Meseguer. Interleaved depth-first search. In *Proceedings IJCAI*, pages 1382–1387, 1997.

[Prc01] Nicolas Prcovic. Unification de la recherche arborescente avec la recherche locale. In *7iemes Journees nationales sur la resolution pratique de problemes NP-complets (JNPC'01)*, Toulouse, France, 2001.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[Pro95] Patrick Prosser. MAC-CBJ : maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde, 1995.

[SF94] Daniel Sabin and Eugene Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94 : Second International Workshop, Orcas Island, Seattle, USA).

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI 92, Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[VS94] Gérard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI'94*, pages 307–312, Seattle, WA, 1994.