Un algorithme distribué pour la résolution des problèmes de contraintes en domaines finis

Nicolas Prcovic

Rapport de Recherche CERMICS 95.44

Novembre 95

Un algorithme distribué pour la résolution des problèmes de contraintes en domaines finis

Rapport de stage du DEA IARFA (Paris VI)

Nicolas Prcovic

Novembre 95

Projet SECOIA
CERMICS-INRIA Sophia Antipolis
2004, Route des Lucioles, B.P. 93
06902 Sophia Antipolis Cedex

nprcovic@sophia.inria.fr

Résumé

Nous présentons un algorithme distribué destiné à trouver toutes les solutions des Problèmes de Satisfaction de Contraintes. Il est basé sur l'algorithme Backtrack et il répartit les sous-arbres de l'arbre de recherche entre différents processus qui s'exécutent en parallèle. Nous montrons que l'accélération de la résolution devient asymptotiquement linéaire lorsque le nombre de variable du problème devient élevé.

De plus, nous étudions la greffe des techniques de réduction issues du Lookahead sur notre algorithme, ainsi que l'intégration du Nogood Recording. Enfin, nous donnons une piste pour son adaptation à la recherche d'une seule solution.

Mots-clés: Problème de satisfaction de contraintes, algorithme distribué, Nogood Recording.

Abstract

We present a distributed algorithm which finds all solutions of Constraint Satisfaction Problems. Based on the Backtrack algorithm, it spreads the subtrees of the search tree over processes running in parallel. We show that the speedup of the resolution is asympotically linear as the number of variables increases.

Furthermore, we study the addition of Lookahead pruning techniques and the inclusion of Nogood Recording. Finally, we suggest a way of adapting our algorithm for finding only one solution.

Keywords: Constraint satisfaction problem, distributed algorithm, Nogood Recording.

1 Introduction

De nombreux problèmes posés à l'IA peuvent être définis et représentés par des contraintes (planification, CAO,...). Ces problèmes de satisfaction de contraintes sur des domaines finis (CSP en anglais) ont été de plus en plus largement étudiés ces dernières années. Ils sont NP-complets et leur résolution reste en conséquence essentiellement basée sur l'exploration d'un arbre de recherche. De nombreuses techniques d'élagage de ces arbres ont été découvertes mais le coût de la recherche de solutions garde un caractère exponentiel en temps dans la plupart des cas, ce qui interdit la résolution de problèmes de grande taille dans la pratique. Depuis quelques années, on commence à explorer une autre façon d'accélérer la résolution en la distribuant sur plusieurs processus qui fonctionnent en parallèle [Hen89]. On espère ainsi réduire les temps de résolution à des limites acceptables en pratique grâce à une accélération qui dépend alors du nombre de processus que l'on veut bien engager.

Nous présenterons ici un algorithme de distribution qui trouve toutes les solutions (développement de l'arbre en entier), reposant sur la répartition sur plusieurs processus de différents sous-arbres qui composent l'arbre de recherche du CSP. Nous examinerons aussi la façon d'y intégrer les techniques d'élagage de l'arbre existantes.

2 Rappels théoriques

2.1 Définitions et conventions d'écriture

2.1.1 Définition d'un CSP

Un CSP est défini par un triplet (X, D, C) avec :

- $X = \{x_1, ..., x_n\}$ un ensemble de n variables.
- $D = \{D_1, ..., D_n\}$ où D_i est le domaine fini dans lequel x_i peut prendre ses valeurs.
- C l'ensemble des contraintes sur les variables, qui définit la compatibilité des affectations de variables entre elles.

2.1.2 Autres définitions

- Une *affectation de variable* est l'association d'une variable avec une valeur de son domaine.
- Une *affectation totale* est une liste d'affectations de toutes les variables du problème.
- Une *affectation partielle* est une liste d'affectations de plusieurs variables.
- Une affectation est *incohérente* si l'affectation de certaines de ses variables viole une contrainte de C.
- Une *solution* du CSP est une affectation totale qui n'est pas incohérente.
- La *signature* d'un processus ou d'un sous-arbre est l'affectation partielle qui définit son nœud-racine.

2.2 Les techniques d'élagage de l'arbre de recherche d'un CSP

Toutes ces méthodes générales sont fondées sur l'algorithme *Backtrack* [GB65] qui consiste à essayer de construire pas à pas une solution en revenant en arrière sur ses choix à chaque fois qu'il apparaît que l'affectation partielle courante est incohérente.

Il existe 2 grands types d'amélioration du *Backtrack*:

- *le Lookahead* (*Forward Checking*, *Maintien de la Consistance d'Arc*, ...) qui permet d'éliminer un certain nombre de valeurs que l'on prévoit comme étant incompatibles avec une solution.

- *l'apprentissage par l'échec* (*Conflict Directed Backjumping*, *Nogood Recording*) qui analyse les causes d'un échec afin d'éviter de retomber dans une configuration similaire.

2.2.1 Le Lookahead

On profite du supplément d'information que constitue la dernière affectation de variable de l'affectation partielle courante pour voir si on peut en déduire que certaines valeurs des domaines des variables restant à affecter peuvent être retirées. Ce filtrage des domaines est réalisé en testant la cohérence des valeurs possibles de variables non affectées grâce aux contraintes qui les lient avec une variable nouvellement affectée. Suivant l'intensité du filtrage que l'on veut appliquer, on peut tester la *consistance de nœud, d'arc ou de chemin* ou utiliser le simple *Forward Checking*[1] jusqu'au *Real Full Lookahead* [Tsa93].

2.2.2 L'apprentissage par l'échec

A chaque fois qu'une affectation partielle se révèle incohérente, on recherche les affectations de variables coupables de l'échec avant de revenir en arrière. Ceci permet de:

- revenir sur le choix de la valeur de la dernière variable fautive (au lieu de revenir inutilement sur le dernier choix). C'est le *Conflict Directed Backjumping*.

- mémoriser l'ensemble des affectations de variables qui rendent l'affectation partielle incohérente (qu'on appelle un *nogood*) afin de détecter plus tôt dans une autre branche de l'arbre l'incohérence d'une affectation partielle qui contient ce nogood. C'est le *Nogood Recording*[1] [SV94].

2.3 La distribution sur plusieurs machines

Plusieurs processus (un par machine) s'exécutent en parallèle et peuvent communiquer par envoi de messages. Le but de la distribution est d'accroître linéairement les performances, c'est-à-dire d'effectuer le même travail qu'un processus seul mais en divisant son temps de résolution par le nombre de processus engagés [Rou89]. On peut approcher cet objectif en minimisant le surcoût dû aux opérations supplémentaires qu'entraîne la distribution[2] [MR92]:

- nombre et taille des messages interprocessus.

- temps d'attente de réponse à un message. Le processus sollicité ne répond pas tout de suite s' il n'est pas en état d'être interrompu ou s'il répond déjà à un autre processus.

1. Voir l'exemple en annexe
2. Voir le schéma en annexe

– temps d'inactivité d'un processus. Le partage du travail est tel que le processus n'a plus rien à faire (ceci arrive surtout en fin de résolution). Il faut donc bien prendre garde d'allouer autant que faire se peut la même charge de travail à chacun des processus.

3 L'algorithme de distribution de l'arbre de recherche

Distribuer un arbre de recherche consiste à répartir des nœuds racines de sous-arbre entre différents processus afin que chacun d'eux développe son sous-arbre parallèlement aux autres. Un arbre a une structure qui ne lui permet généralement pas de se diviser en un nombre donné de morceaux de taille identique. De plus, on ne peut connaître a priori quelle sera sa forme. Il nous faut donc une granularité plus fine dans le découpage (chaque processus développera plusieurs sous-arbres) tout en sachant que plus il y aura de grains plus le nombre de communications sera élevé. Afin de trouver un bon compromis entre granularité et fréquence de demande de travail, il est possible de choisir une profondeur dans l'arbre à partir de laquelle on distribue les nœuds [Man92]. Les inconvénients de cette méthode sont analysés dans [Mau93]. Ils résident dans le fait que si nous voulons que la phase d'attente de terminaison (phase pendant laquelle un processus va rester inactif jusqu'à la fin de la recherche) soit raisonnablement courte, il faut sélectionner une profondeur adéquate qui varie pour chaque type de problème. Or il serait nettement plus pratique d'avoir une méthode qui ne dépende d'aucun paramètre.

La solution adoptée ici est de distribuer d'abord ce qu'on pense être les gros morceaux (les sous-arbres dont la racine est la moins profonde dans l'arbre) en prenant soin de garder une partie du travail qui va à nouveau pouvoir être divisée quand un processus aura terminé la tâche qui lui avait été assignée. De plus, nous supprimons la phase d'attente de terminaison en redistribuant le travail à chaque fois qu'un processus est inactif.

3.1 Mécanisme de base de l'algorithme

Tous les processus engagés ont une même fonction commune qui est de développer un sous-arbre en profondeur d'abord. Un processus a le statut spécial de maître (les autres sont de simples travailleurs). C'est lui qui répartit le travail. Lorsqu'un travailleur a terminé sa tâche, il demande au maître de lui envoyer un nœud. Le maître va alors lui céder un fils du nœud le moins profond qu'il possède, sauf si ce nœud géniteur est le dernier qu'il lui reste et si son fils est le dernier engendrable, c'est alors un fils de ce dernier qui sera donné. On aura donc une distribution telle qu'illustrée par la figure 1.

Au départ, le maître possède la racine de l'arbre et les travailleurs n'ont aucun nœud. La granularité de la répartition du travail s'affine en cours de résolution jusqu'à atteindre la taille d'un nœud. Chacun des processus s'arrête lorsqu'il reçoit un nœud vide (le maître n'a plus de nœud).

3.2 Les défauts de ce mécanisme

Ce mécanisme permet une répartition équitable lorsque l'arbre est équilibré. Mais il suffit qu'un des sous-arbres soit beaucoup plus grand que les autres pour qu'en fin de résolution des processus soient inactifs. Pire encore, si la branche gauche de l'arbre se révèle courte, seuls de grands sous-arbres sont distribués et la répartition peut devenir catastrophique. Nous allons donc y rajouter un mécanisme correctif.

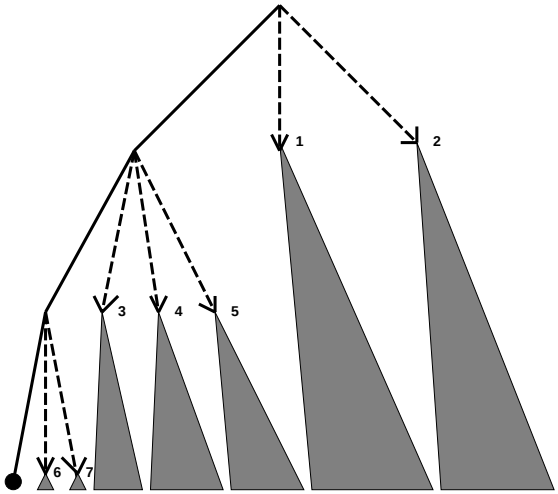

FIG. 1 – *Ordre de distribution des sous-arbres*

3.3 Le mécanisme correctif de l'algorithme

Lorsque le maître n'a plus de travail, il demande à tous les travailleurs la profondeur du nœud le moins profond qu'ils possèdent et il échange son statut de maître avec un de ceux qui ont le nœud le moins profond. Si aucun des travailleurs n'a plus de nœud alors le maître envoie un signal d'arrêt à tous les travailleurs et il s'arrête lui-même. De cette façon, chaque processus reste actif jusqu'à la fin et la répartition du travail est optimale. Par contre, en pratique, il faudra quand même fixer un seuil de profondeur au delà duquel on interdit les changements de maître et les demandes de nœud car à ce stade les communications coûtent plus que le reste de la résolution.

3.4 L'algorithme

3.4.1 Le protocole de communication

Chaque processus est capable d'atteindre tous les autres mais il n'y aura que des messages entre maître et travailleurs.

Nous n'utiliserons qu'une seule primitive de communication :

remote-call(*nom-groupe*, *message*)

qui permet de faire exécuter *message* par tous les éléments de *nom-groupe* (il existe deux groupes : celui des *Travailleurs* et celui qui ne contient qu'un seul élément, le *Maître*) et de renvoyer la liste des résultats.

L'exécution de **remote-call** provoque l'envoi d'un même message à tous les destinataires concernés et l'entrée dans une boucle d'attente d'événements. Un événement peut être :

- une réponse au message envoyé. Il est alors stocké dans une liste.
- un message envoyé par un processus. Le message est exécuté et sa réponse renvoyée au destinataire.

La boucle prend fin quand tous les processus sollicités ont renvoyé une réponse.

Un processus ne peut être interrompu par un message qu'à des moments précis :

- dans un **remote-call**.
- après chaque génération de nœud, il teste si un ou plusieurs messages sont en attente dans sa file d'événements et les exécute le cas échéant.

Tout ceci permet d'éviter le phénomène d'interblocage car un processus peut traiter un message d'un autre processus alors qu'il attend déjà une réponse de ce dernier.

3.4.2 Structures de données utilisées et fonctions associées

On utilise une pile à laquelle on peut accéder par le sommet (fonctions **empile**(nœud), **dépile()**, **sommet-pile()**) mais aussi par le bas (fonctions **empile-bas**(nœud), **dépile-bas()**, **bas-pile()**). La pile est la structure idéale pour un parcours de l'arbre en profondeur mais nous voulons pouvoir y accéder aussi par le bas car c'est à cet endroit que se trouve le nœud le moins profond (voir figure 2).

Le nœud contient (parmi d'autres informations) la liste des fils qu'il peut générer (voir figure 2). La fonction **générer-nœud()** génère un fils à partir du nœud au sommet de la pile (et modifie la liste des fils en conséquence) et le retourne. Elle retourne *nil* si le fils généré est incohérent. La fonction **générer-nœud-bas()** fait la même chose à partir du nœud du bas de la pile.

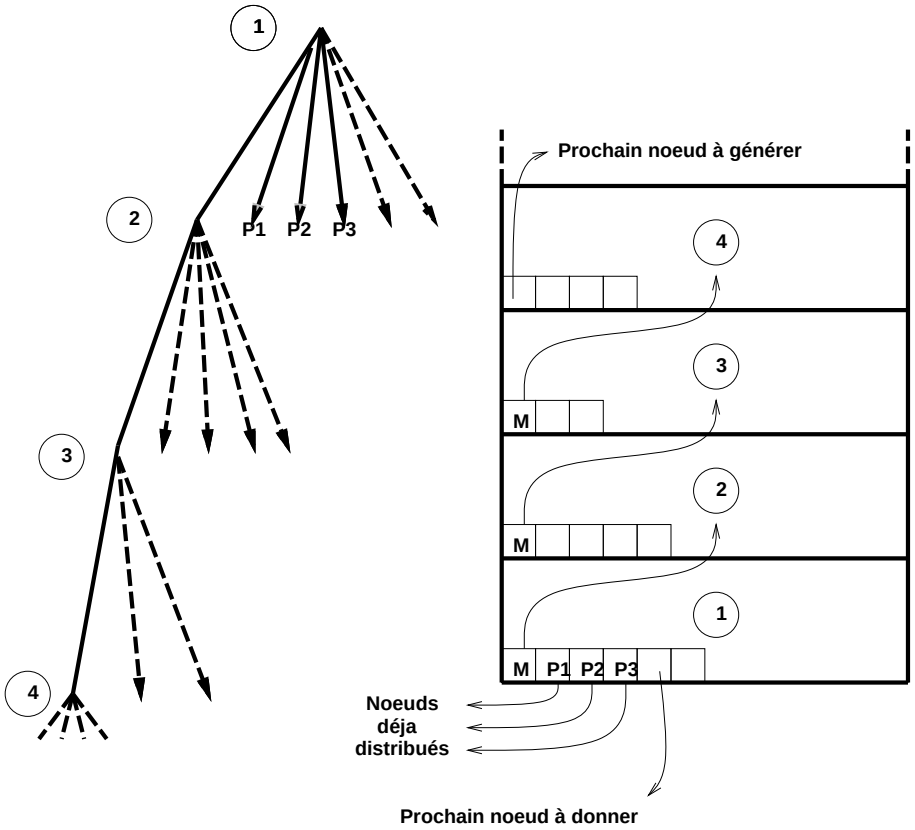

FIG. 2 – *L'arbre développé par le maître et sa pile*

3.4.3 Enoncé de l'algorithme

énumération-distribuée() :

```
Stop = FAUX
TANTQUE Stop == FAUX
    SI le processus est le Maître
    ALORS
        énumération-locale()
        change-maître()
    SINON
        nœud = remote-call(Maître, donne-nœud())
        SI nœud ≠ nil
        ALORS
            empile(nœud)
            énumération-locale()
        FINSI
    FINSI
FINTANTQUE
```

énumération-locale() :

```
TANTQUE Pile ≠ ∅
    TANTQUE Pile ≠ ∅ ET sommet-pile() ne peut engendrer de fils
        SI sommet-pile() est une feuille
        ALORS
            afficher sommet-pile() (c'est une solution)
        FINSI
        dépile()
    FINTANTQUE
    SI Pile ≠ ∅
        nœud = génère-nœud()
        SI nœud ≠ nil
        ALORS
            empile(nœud)
        FINSI
    FINSI
    TANTQUE message en attente
        traiter message
    FINTANTQUE
FINTANTQUE
```

donne-nœud() :

TANTQUE Pile $\neq \emptyset$
TANTQUE Pile $\neq \emptyset$ ET **bas-pile()** ne peut engendrer de fils
SI **bas-pile()** est une feuille
ALORS
afficher **bas-pile()** (c'est une solution)
FINSI
dépile-bas()
FINTANTQUE
TANTQUE **bas-pile()** est le dernier nœud et ne peut générer qu'un fils
fils = **génère-nœud-bas()**
dépile-bas()
SI fils $\neq$ *nil*
empile-bas(fils)
FINSI
FINTANTQUE
SI Pile $\neq \emptyset$
ALORS
retourne **génère-nœud-bas()**
SINON
retourne *nil*
FINSI

change-maître() :

liste = **remote-call**(*Travailleurs*, **bas-pile()**)
meilleur = **sélection**(liste)
SI meilleur $\neq$ nil
échange-statut(meilleur,*Maître*)
SINON
remote-call(*Travailleurs*,"Stop = VRAI")
Stop = VRAI
FINSI

Initialisation :
Une fois *Maître* et *Travailleurs* définis dans chaque processus, on lance l'éxécution au niveau du *Maître* par :
remote-call(*Travailleurs*, **énumération-distribuée()**)
énumération-distribuée()

3.5 Evaluation du nombre de messages

Il existe deux types de message qui surviennent plusieurs fois pendant la résolution : ceux concernant le changement de maître et ceux concernant la demande de travail au maître.

Nous utilisons les notations suivantes :

- n : nombre de variables.
- d : cardinal du plus grand domaine.

- p : nombre de processus.
- π_i : profondeur du nœud le moins profond d'un processus i.

3.5.1 Nombre maximal de changements de maître

On considère $\Pi = \sum_{i=1}^{p} \pi_i$.

Le maître est celui qui avait au départ le nœud le moins profond (profondeur : π_m). Lorsqu'il échange son statut, il devient travailleur et demande au nouveau maître un nœud de profondeur π. Or $\pi_m < \pi$ et comme les autres π_i n'ont pas pu décroître, on a Π qui croît strictement. Au départ $\Pi > 0$ et comme $\forall i \; \pi_i \leq n$ alors $\Pi \leq pn$. Le nombre de changement de maître est donc inférieur à pn .

3.5.2 Nombre maximal de nœuds demandés

Tant que le maître reste inchangé, il va distribuer tous les frères qui se situent à droite des nœuds de sa branche gauche, donc au maximum $\sum_{i=1}^{n}(|D_i| - 1)$ ce qui est inférieur à nd. Ceci sera fait pour chaque maître donc, en tout, moins de dpn^2 fois.

3.6 Evaluation théorique de l'efficacité de l'algorithme

- L'algorithme permet un partage optimum du travail entre chacun des processus : le travail est repartagé à chaque fois qu'un processus a terminé son sous-arbre et un processus n'est inactif que le temps qu'il reçoive une réponse à un message.
- Le nombre de messages est borné par une fonction polynomiale (dpn^2).
- Le nombre de nœuds générés est le même que celui du Backtracking normal. La différence est qu'il y a plusieurs piles et qu'un nœud créé à partir d'une pile peut être empilé sur une autre.

Nous pouvons donc espérer que le surcoût dû au mécanisme de distribution sera de plus en plus insignifiant par rapport au temps de recherche (qui est en $O(d^n)$) lorsque la taille d'un problème sera grande, et alors prévoir un gain de performance asymptotiquement linéaire quand n ou d devient grand.

4 Intégration des techniques d'élagage de l'arbre de recherche

4.1 Intégration du Lookahead

Le Lookahead agit sur les domaines des variables qui restent à affecter donc en aval du nœud qu'il examine et par conséquent uniquement au sein du processus qui contient le nœud. Les effets locaux des techniques issues du Lookahead leur permettent donc d'être greffées directement sur l'algorithme tel qu'il a été présenté. Seule la taille des messages change : il faut transmettre la liste (d'une longueur en $O(nd)$) des domaines filtrés au travailleur.

4.2 Intégration du Nogood Recording

Comme nous l'avons dit en 2.2.2, un nogood est une affectation partielle qu'on sait incohérente et qui peut servir à élaguer l'arbre de recherche en différents endroits. Son intérêt est donc global et la découverte d'un nogood peut engendrer de nouvelles communications interprocessus.

4.2.1 Circulation des nogoods

Nous évoquerons d'abord deux modèles simplistes de cette intégration qui amèneront à la présentation du modèle que nous avons mis en œuvre.

- 1^{er} modèle (égoïste) :
  Un processus qui trouve un nogood n'en informe pas les autres. Le surcoût en communications est nul mais l'arbre est bien moins élagué qu'en séquentiel.

- 2^{eme} modèle (trop bavard) :
  Un processus qui trouve un nogood en informe tous les autres. L'arbre est alors bien élagué mais certains processus sont dérangés inutilement comme nous allons le voir juste après et le surcoût en communications est excessif.

- 3^{eme} modèle (économique) :
  La découverte d'un nogood n'est transmis qu'aux processus qui peuvent en avoir besoin. En effet, chaque sous-arbre définit un sous-problème restreint par l'affectation d'un certain nombre de variables (sa signature). Le processus qui développe ce sous-arbre n'a que faire des nogoods qui contiennent des variables affectées différemment de celles de sa signature.

Nous allons donc utiliser 2 filtres (peu coûteux en temps) qui empêcheront les nogoods d'atteindre des processus où ils n'ont rien à faire :

- un nogood n'est rendu public que s'il ne contient pas la signature du processus qui l'a découvert. En effet, son utilité ne sera que locale au processus puisqu'il n'existera pas d'affectation partielle à l'extérieur du processus qui contiendra ce nogood.

- un nogood public ne parviendra jusqu'à un processus que si ce dernier n'a aucune affectation de variable dans sa signature qui soit incompatible avec une affectation de variable du nogood.

4.2.2 Le mécanisme de gestion des nogoods

Le maître possède la liste des signatures de tous les processus. Il servira de relais au processus découvreur du nogood et sélectionnera les destinataires (le maître lui-même pouvant en faire partie).

Il faut ainsi compléter le comportement des processus :

- Quand un processus demandera du travail, le maître lui fournira le nœud-racine ainsi que la partie de sa base de nogoods qui lui sera utile (et il mettra à jour sa liste de signatures).

- Quand un processus maître changera de statut, il transfèrera sa liste de signatures à son successeur.

La base de nogood sera donc répartie entre tous les processus (un nogood pouvant être commun à plusieurs processus).

4.2.3 Limitation du nombre de nogoods

Le nombre de nogoods étant potentiellement du même ordre que le nombre de nœuds de l'arbre, on ne les mémorise pas tous en pratique. De plus, dans le cas présent, cela induirait un nombre de messages équivalent. Il est d'usage de limiter la longueur (c'est-à-dire le nombre de variables) des nogoods afin de n'en avoir qu'un nombre polynomialement borné.

4.2.4 Evaluation théorique de l'efficacité de l'algorithme

La découverte d'un nogood utile entraîne au maximum l'envoi de p messages. Comme le nombre de nogoods pris en compte est polynomial, il en résulte que le surcoût dû aux messages devrait être négligeable. On ne peut toutefois pas conclure de ceci que l'aspect linéaire de l'accélération est maintenue. En effet, la parallélisation induit un comportement différent au niveau de l'élagage : en séquentiel, un nogood sert à élaguer des sous-arbres situés plus "à droite" que le nœud où il a été détecté tandis qu'en parallèle, la direction de son action est indéterminée. On a donc un arbre différent suivant ces deux cas.

L'étude théorique du comportement du Nogood-Recording sur cet algorithme semble complexe et trop longue pour être développée dans le cadre de ce rapport. Elle mérite surement d'être approfondie dans l'avenir.

5 Adaptation de l'algorithme dans le cadre de la recherche d'une seule solution

5.1 Inefficacité pour la recherche d'une seule solution

La linéarité de l'accélération de l'obtention d'une seule solution ne peut absolument pas être garantie. On peut par exemple n'avoir aucune accélération ou bien en avoir une supralinéaire. Ceci dépend de la nature du problème : sous-contraint, sur-contraint ou intermédiaire :

- Les problèmes sous-contraints ont beaucoup de solutions et si celles-ci sont équitablement réparties dans toutes les feuilles de l'arbre, il y a alors de très fortes chances que la 1^{ere} solution se trouve dans le sous-arbre le plus à gauche et qu'il n'y ait pas d'accélération.

- Les problèmes sur-contraints n'ont aucune solution. Leur arbre de recherche est donc complètement développé et l'accélération de la résolution est linéaire (dans l'idéal).

- Les problèmes intermédiaires ont peu de solutions (voir [Smi94]) et on ne peut savoir à l'avance dans quelles parties de l'arbre elles sont situées ce qui fait que l'accélération sera imprévisible : elle variera entre 1 et un nombre très supérieur à p.

Cet algorithme n'est donc pas bien adapté à la recherche d'une seule solution dans le cas général. Il semble par contre intéressant d'expérimenter son comportement sur des problèmes intermédiaires (les plus durs à résoudre [Pro94]). D'autre part, comme nous allons le voir ci-dessous, nous pouvons le modifier de façon à limiter l'amplitude de la variation de l'accélération.

5.2 Adaptation envisageable

Il faudrait concentrer d'abord la distribution sur la partie basse gauche de l'arbre pour passer prioritairement en revue les solutions potentielles dans le même ordre que dans la résolution avec un seul processus. Il suffirait donc de distribuer non pas les nœuds les plus élevés dans l'arbre mais au contraire les plus bas. Ceci aurait l'inconvénient majeur de rendre le nombre de sous-arbres à répartir (et donc le nombre de messages) très élevé. Or le temps pour traiter un message doit être court par rapport à la résolution du sous-arbre qu'il implique. Le meilleur compromis entre nombre de messages et granularité de la distribution devrait donc être fixé par la profondeur à laquelle on distribue les nœuds (comme cela a été évoqué en 3 [Man92]).

6 Résultats expérimentaux

L'algorithme a été implanté en tant que bibliothèque de fonctions destinée à faire partie de PROSE [Ber92], une boîte à outils fonctionnelle pour l'interprétation de contraintes, et en utilisant CHOOE [Leb93], un gestionnaire d'environnement distribué.

Les tests ont été effectués sur un réseau de machine Sun. Ils ont d'abord porté sur le problème classique des n dames à placer sur un échiquier nxn en utilisant le *Forward-Checking*.

n Dames	t_1	t_2	t_3	t_4	t_6	t_8
7	1.09	0.67	0.50	0.69	0.66	0.61
8	3.91	2.51	1.67	1.18	1.13	0.85
9	16.8	8.97	5.96	4.80	3.90	2.86
10	66.4	33.4	23.9	17.2	12.6	9.52
11	315	157	106	78.5	53.9	43.0
12	1660	831	557	411	278	210
13	8986	4665	3111	2311	1527	1141
14	53500	28819	18065	13460	9015	6689

Temps pour trouver toutes les solutions des n-Dames par p processus [3]

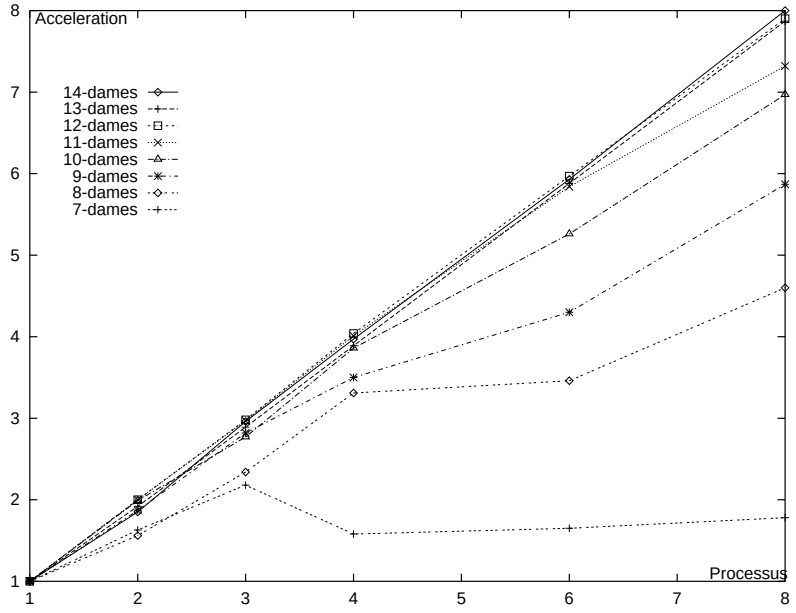

FIG. 3 – *Accélération : temps monoprocessus/temps avec p processus*

3. La précision des mesures est de l'ordre de quelques pour cent.

n	2	3	4	6	8
7	81%	73%	39%	27%	22%
8	78%	78%	83%	58%	57%
9	93%	94%	87%	72%	57%
10	99%	92%	96%	88%	87%
11	100%	99%	100%	97%	91%
12	100%	99%	100%	99%	99%
13	96%	96%	97%	98%	98%
14	92%	99%	99%	99%	100%

Efficacité de la distribution par rapport à l'idéal (accélération linéaire)

Les résultats obtenus confirment ce que l'évaluation théorique de l'efficacité de l'algorithme avait laissé présager :

- Plus p est élevé, plus le temps utilisé par les communications affecte l'accélération.
- On obtient une accélération sensiblement linéaire lorsque n devient grand (visible ici à partir de n=10).

Nous avons de plus fait des tests de recherche d'une seule solution sur des problèmes aléatoires définis par 21 variables dont les domaines ont pour cardinal 20, avec une connectivité du graphe des contraintes et une dureté des contraintes égales à 0.42, afin d'obtenir des problèmes qui n'ont que très peu de solutions.

Problème	t_1	t_2	Acc.	t_3	Acc.
0	24	5	*4.8*	5	*4.8*
1	52	28	*1.8*	4	*13*
2	155	80	*1.9*	62	*2.5*
3	112	60	*1.8*	46	*2.4*
4	35	13	*2.7*	4	*8.7*
5	128	60	*2.1*	30	*4.2*
6	24	24	*1.0*	12	*2.0*
7	273	143	*1.9*	87	*3.1*
8	79	39	*2.0*	25	*3.1*
9	255	129	*2.0*	73	*3.5*
10	110	68	*1.6*	36	*3.0*
11	85	26	*3.2*	26	*3.2*
12	29	12	*2.4*	12	*2.4*
13	44	25	*1.7*	10	*4.4*
14	19	19	*1.0*	9	*2.1*
15	79	50	*1.6*	6.5	*12*
16	6.0	6.0	*1.0*	6.0	*1.0*
17	14	11	*1.3*	11	*1.3*
18	193	66	*2.9*	37	*5.2*
19	25	11	*2.3*	11	*2.3*
moyenne			*2.0*		*4.2*
ecart-type			*0.9*		*3.2*

On voit que l'accélération est en moyenne satisfaisante. Elle est cependant trop variable pour garantir à l'algorithme de bonnes performances quel que soit le CSP sur lequel il opère.

7 Conclusion

Les tests de l'algorithme ont montré son efficacité dans la recherche exhaustive des solutions. Ils ont aussi établi son manque de fiabilité dans la recherche d'une seule solution même si en moyenne l'accélération est avantageuse.

Le mécanisme de distribution tel qu'il a été présenté est très général et peut s'appliquer à d'autres problèmes que la satisfaction de contraintes. Mais il semble surtout assez souple pour permettre l'intégration de techniques d'élagage de l'arbre de recherche. Pour celle qui était la plus difficle à intégrer, le Nogood-Recording, il reste à étudier ses performances tant de manière théorique que par l'expérimentation. L'adaptation envisagée de l'algorithme pour la recherche d'une seule solution a elle aussi besoin d'être testée voire améliorée.

Si les résultats dans ces deux cas s'avéraient satisfaisants, ils permettraient de conclure à la pertinence de l'approche de la distribution de la résolution d'un CSP que nous avons étudiée, celle de la répartition de ses sous-arbres entre plusieurs processus.

A Annexes

A.1 Un exemple de Lookahead

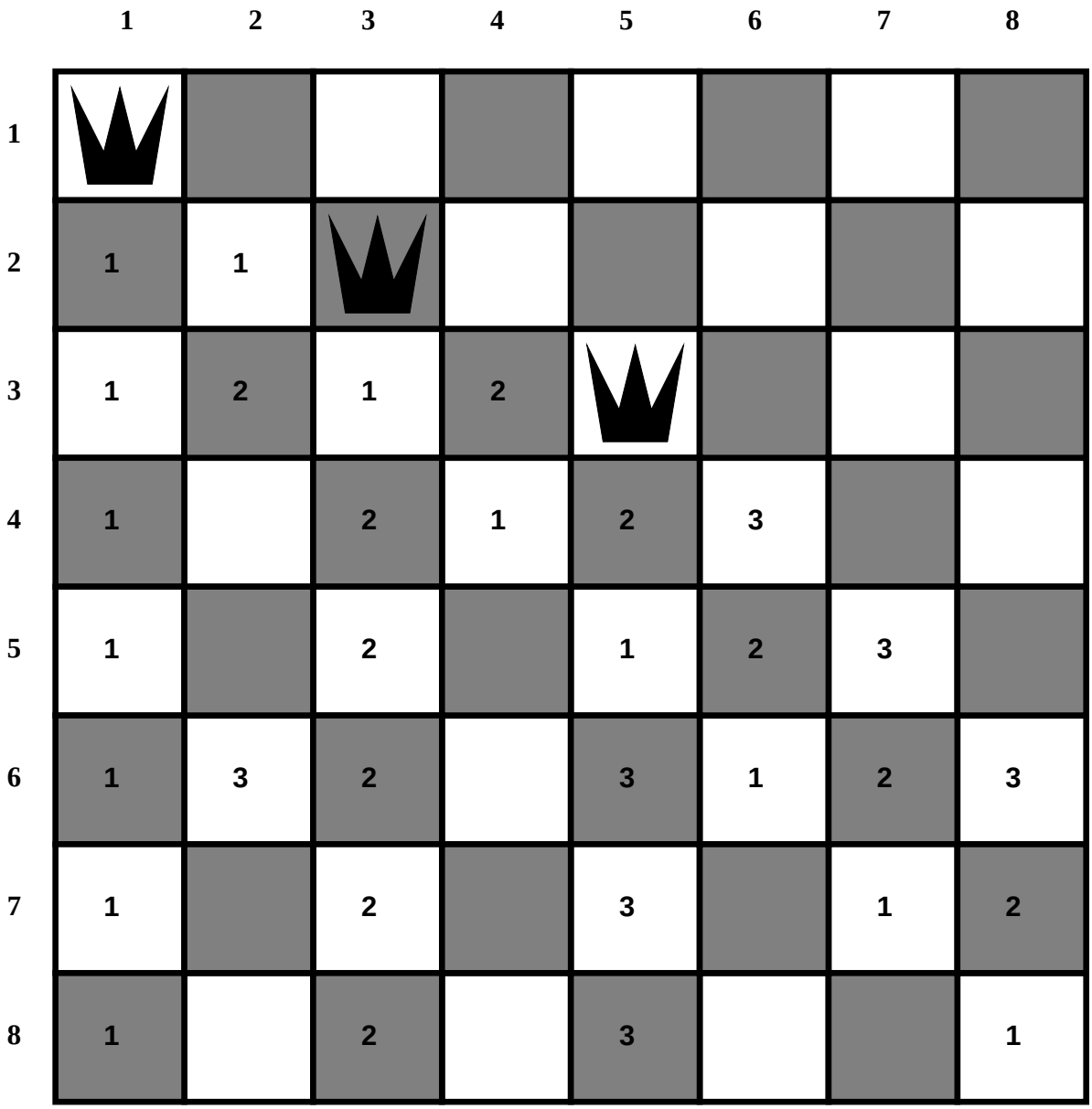

FIG. 4 – *Un exemple de filtrage des domaines grâce au Forward-Checking*

L'affectation de la 1^{ere} variable à 1 permet de déduire un ensemble d'affectations incompatibles (représenté par les cases marquées par un 1 sur l'échiquier de ce problème des 8 Dames). La 2^{eme} variable voit par exemple son domaine réduit à 6 valeurs. Après l'affectation de la 2^{eme} variable, la 3^{eme} n'a plus que 4 valeurs disponibles, etc...

L'affectation partielle illustrée par la figure permet de se rendre compte qu'il ne reste plus qu'une valeur possible pour la variable numéro 6.

A.2 Un exemple d'apprentissage par l'échec

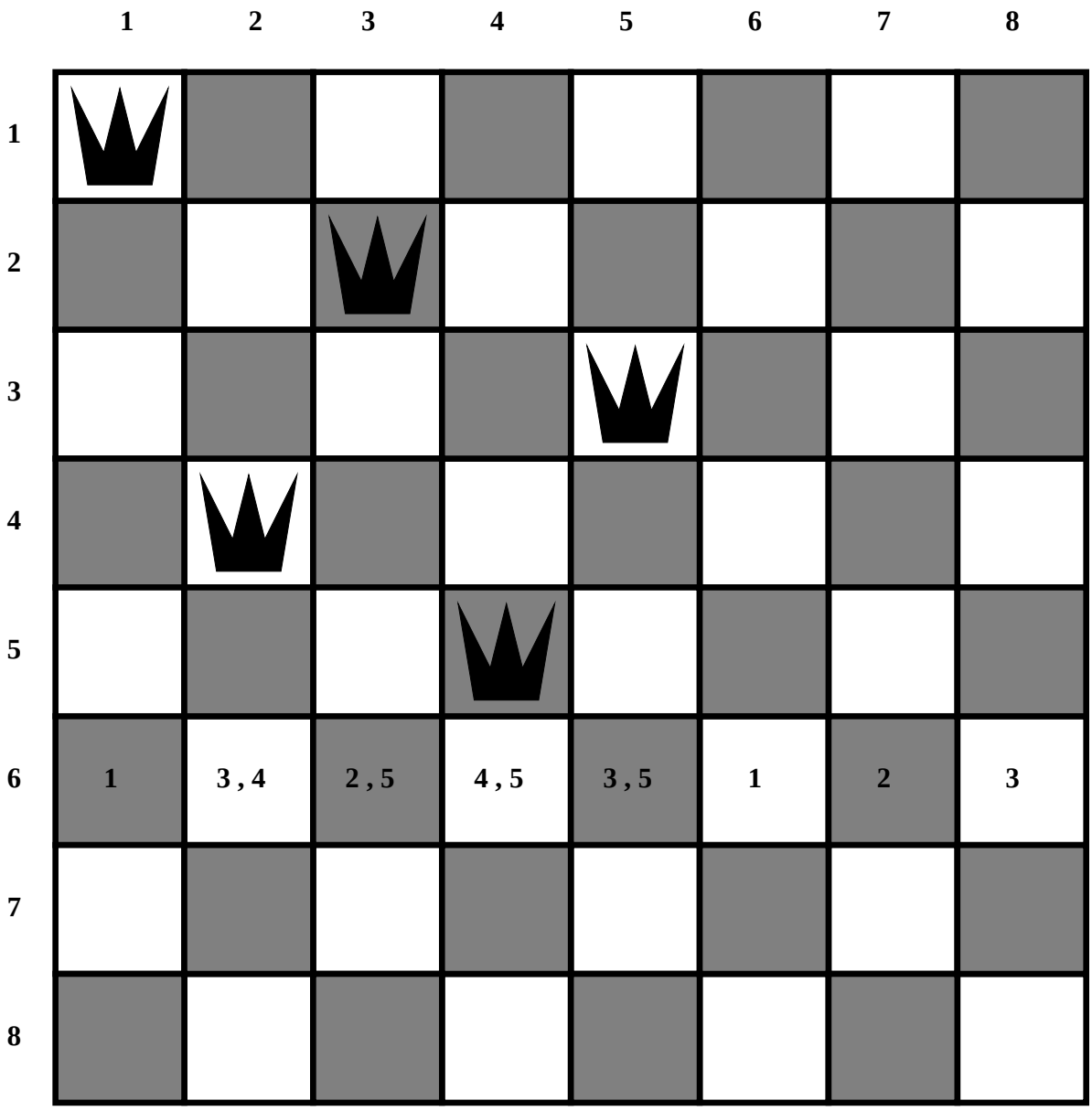

FIG. 5 – *Un exemple d'apprentissage par l'échec*

L'affectation partielle de ce problème des 8 Dames est la suivante: (1,1)(2,3)(3,5)(4,2)(5,4) Toute tentative d'affectation de la 6^{eme} variable mène à une incohérence (on a représenté en 6^{eme} ligne les numéros des variables incompatibles).

On peut déduire de ces informations que revenir sur le choix de la 5^{eme} variable ne résoudra pas les conflits, on peut donc backtracker directement sur la 4^{eme} variable. On peut d'autre part, mémoriser les nogoods [(1,1)(2,3)(3,5)(4,2)] et [(1,1)(2,3)(3,5)(5,4)] car maintenant on sait que si une affectation partielle contient un des ces deux ensembles d'affectations de variable alors l'affectation partielle mènera à une incohérence.

A.3 Le surcoût de la distribution

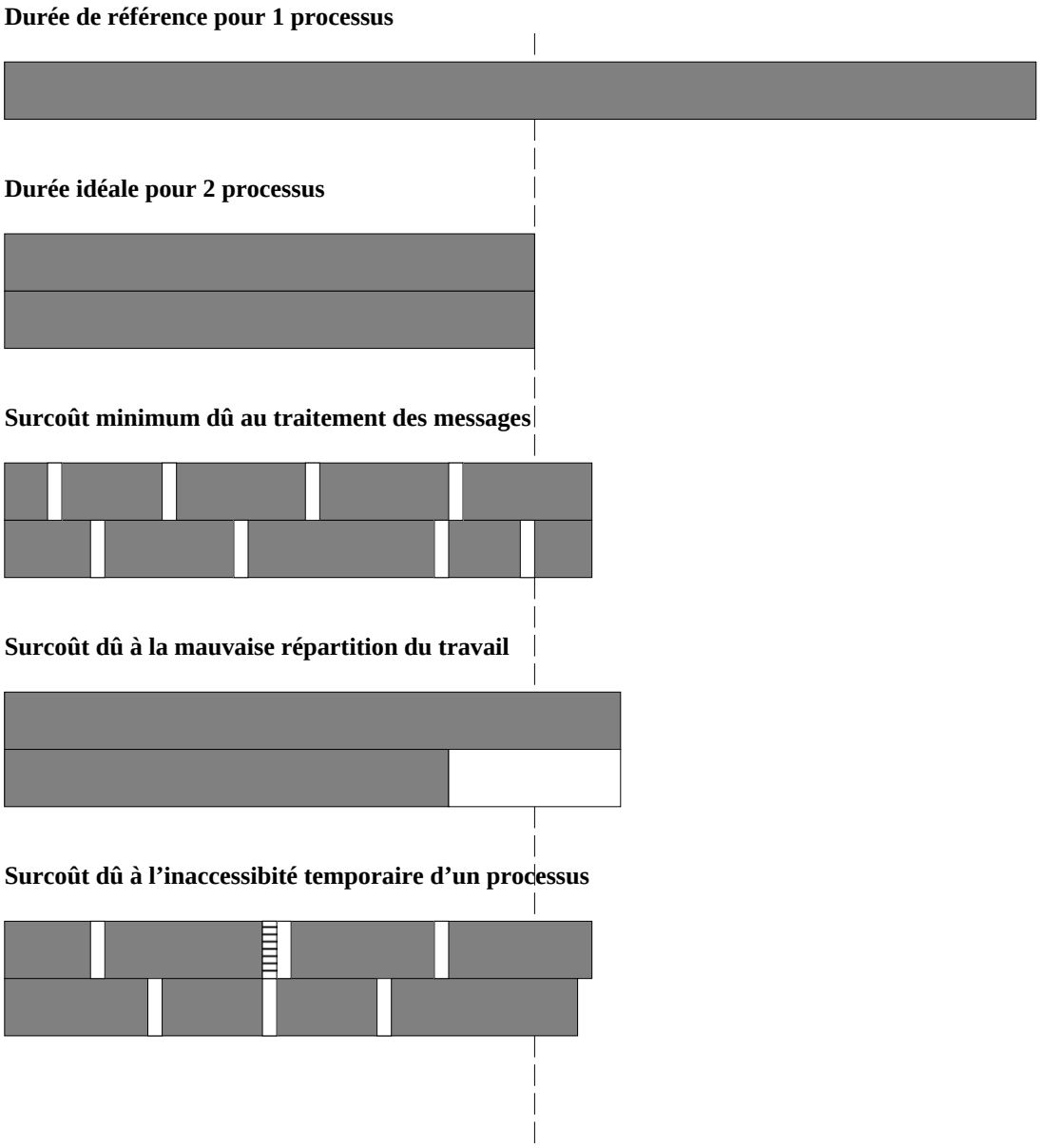

FIG. 6 – *Les différents types de surcoûts dus aux communications interprocessus*

A.4 Exemples de distribution sur un CSP

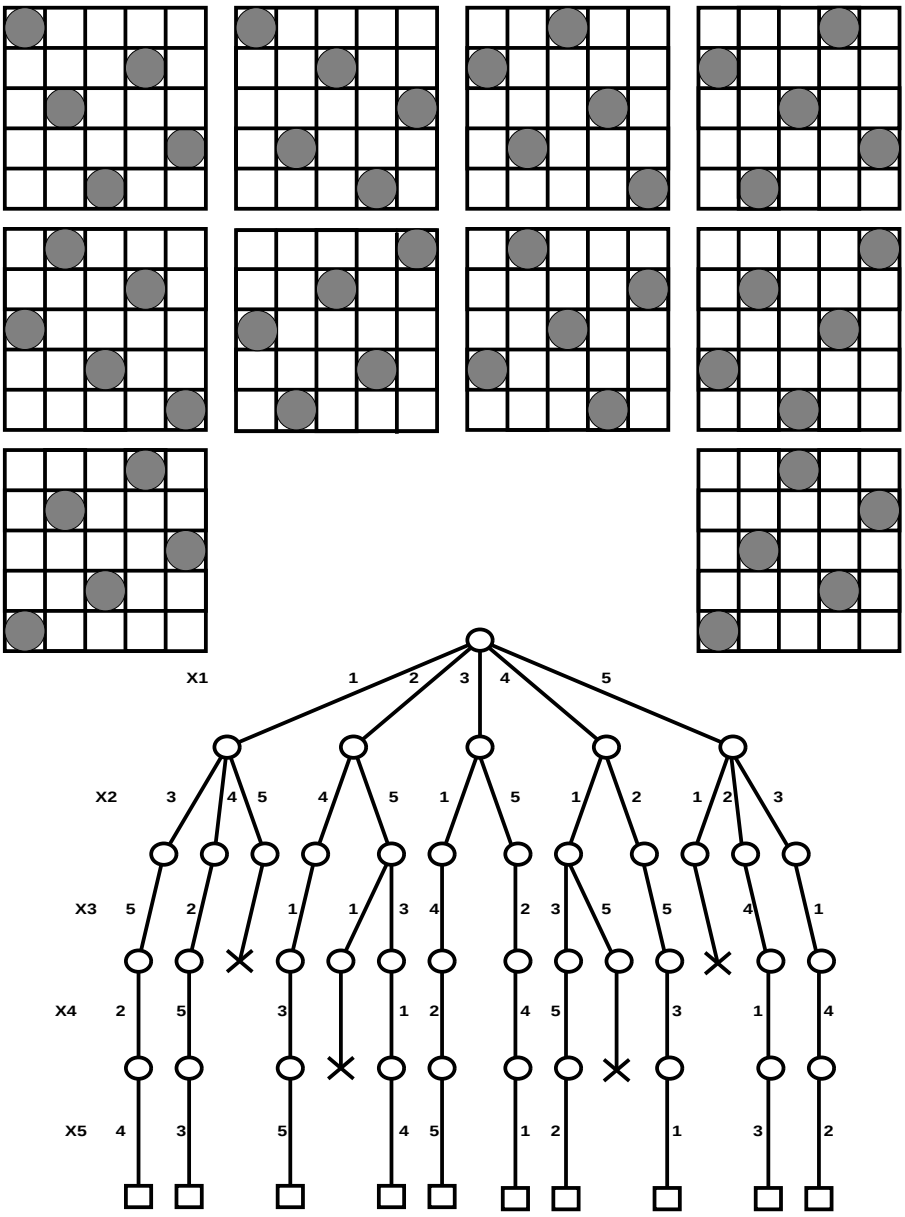

FIG. 7 – *Les 10 solutions du problème des 5 dames*

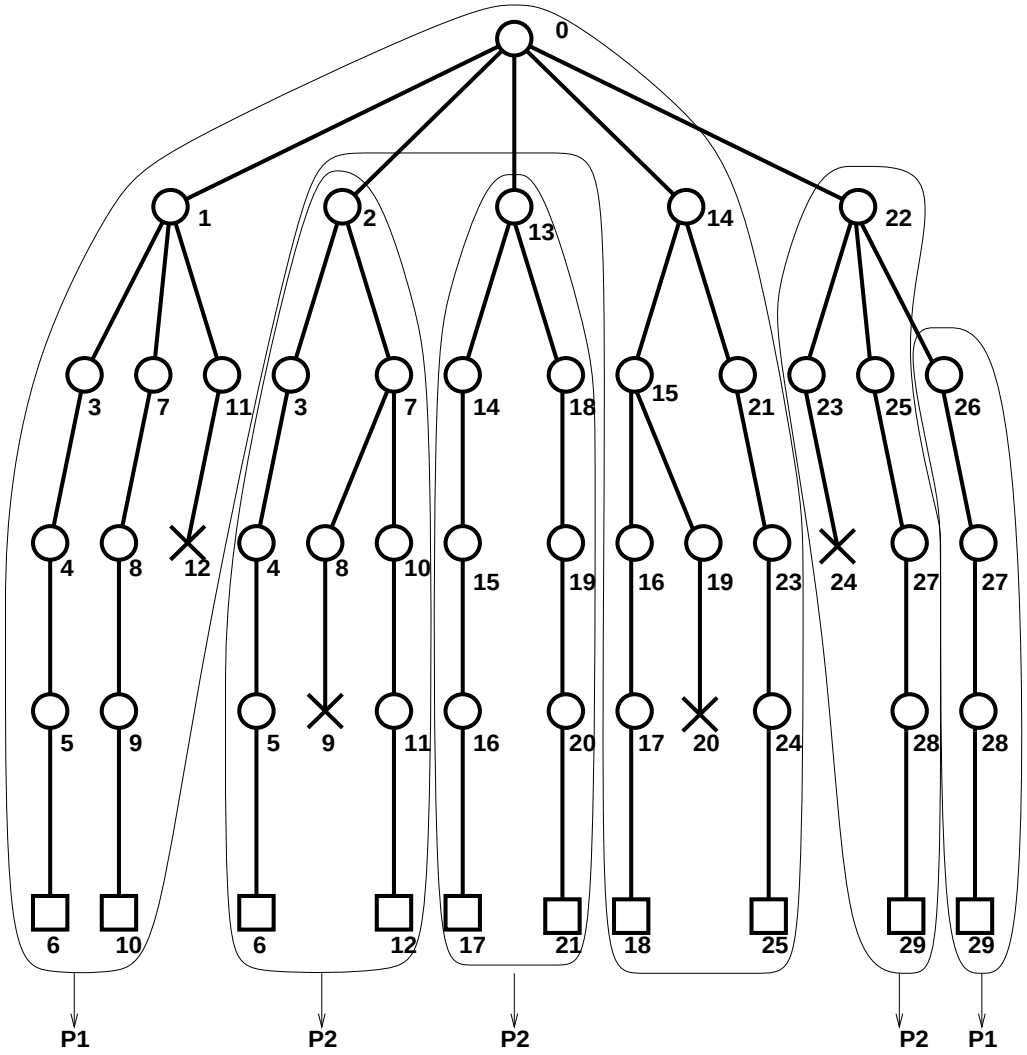

FIG. 8 – *La distribution sur 2 processus du problème des 5 dames*

Ceci est une simulation faite "à la main" où on prend comme hypothèse que les processus sont parfaitement synchrones.

Cycle	Opération
0	P1 possède la racine de l'arbre.
1	P1 génère son 1^{er} nœud, les autres processus vont demander un nœud.
2	P1 donne un nœud à P2.
13	P1 donne un nœud à P2.
22	P1 donne un nœud à P2.
26	P1 n'a plus de nœud. P2 devient maître et donne un nœud à P1.
29	Plus aucun processus n'a de nœud: fin.

Accélération: 1.86

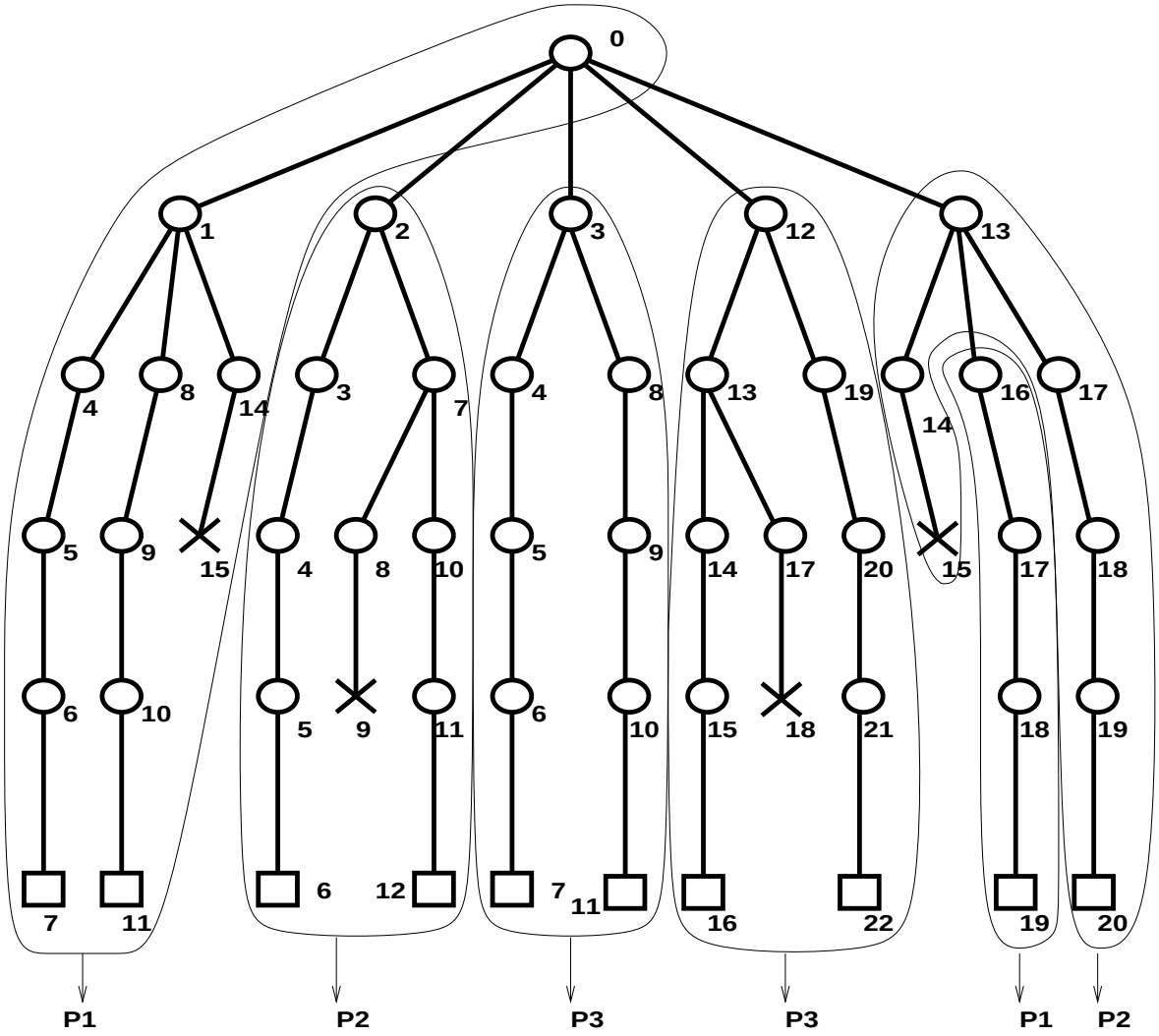

FIG. 9 – *La distribution sur 3 processus du problème des 5 dames*

Cycle	Opération
0	P1 possède la racine de l'arbre.
1	P1 génère son 1^{er} nœud, les autres processus vont demander un nœud.
2	P1 donne un nœud à P2.
3	P1 donne un nœud à P3.
12	P1 donne un nœud à P3.
13	P1 donne un nœud à P2.
16	P1 n'a plus de nœud. P2 devient maître et donne un nœud à P1.
20	P1 n'a plus de nœud mais P2 ne peut pas en donner.
21	P2 n'a plus de nœud. P3 devient maître mais ne peut donner de nœud.
22	Plus aucun processus n'a de nœud: fin.

Accélération: 2.45

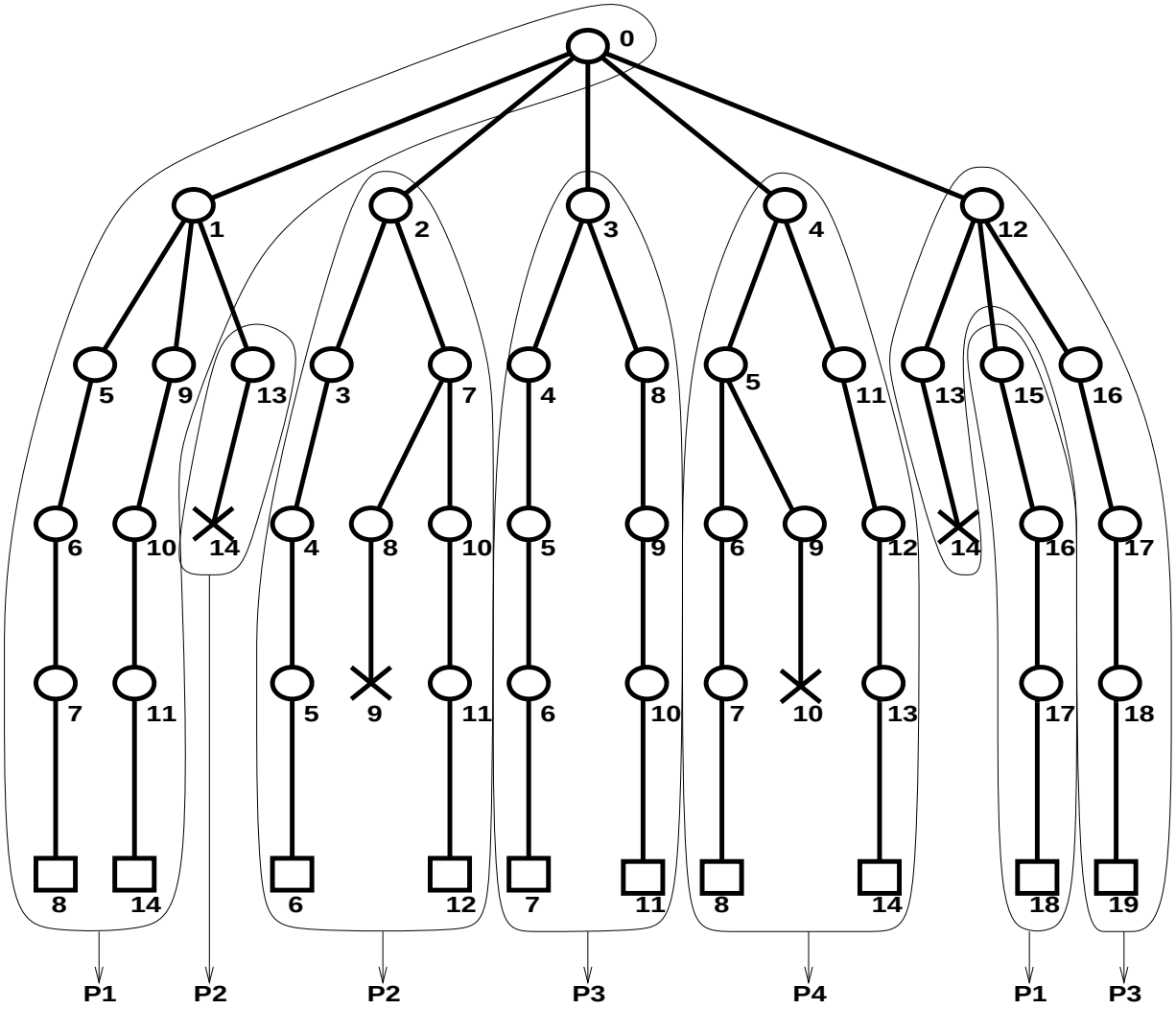

FIG. 10 – *La distribution sur 4 processus du problème des 5 dames*

Cycle	Opération
0	P1 possède la racine de l'arbre.
1	P1 génère son 1^{er} nœud, les autres processus vont demander un nœud.
2	P1 donne un nœud à P2.
3	P1 donne un nœud à P3.
4	P1 donne un nœud à P4.
12	P1 donne un nœud à P3.
15	P1 et P4 n'ont plus de nœud. P3 devient maître et donne un nœud à P1.
16	P4 n'a toujours pas de nœud mais P3 ne peut lui en donner.
19	Plus aucun processus n'a de nœud: fin.

Accélération: 2.84

A.5 Circulation des nogoods: un exemple

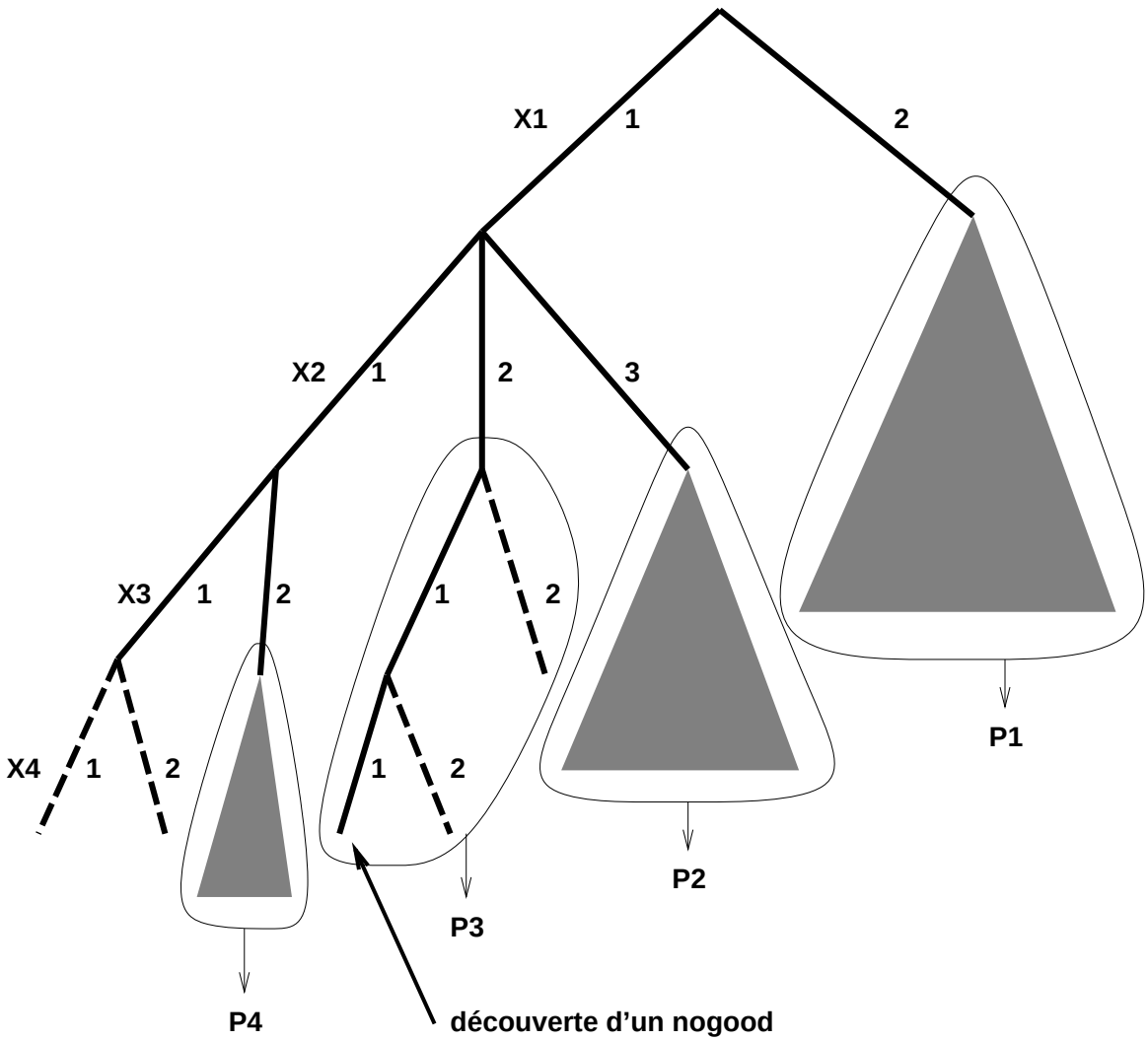

FIG. 11 – *Le nogood découvert dans P3 ne sera pas forcément utile aux autres processus*

Suivant le contenu du nogood découvert, on le transmettra à différents processus:

$\{(x_1,1),(x_2,2),(x_4,1)\} \longrightarrow$ aucun processus (il ne devient pas public).
$\{(x_1,1),(x_3,1)\} \longrightarrow$ P2 et le Maître.
$\{(x_2,2),(x_4,1)\} \longrightarrow$ P1 et le Maître.
$\{(x_3,1),(x_4,1)\} \longrightarrow$ P1, P2 et le Maître.
$\{(x_4,1)\} \longrightarrow$ tous les processus.

Table des matières

Références

[Ber92] P. Berlandier. PROSE: Une boîte à outils fonctionnelle pour l'interprétation de contraintes. Rapport technique 145, INRIA-Sophia Antipolis, 1992.

[GB65] S. Golumb and L. Baumert. Backtrack programming. In *Journal of the ACM*, pages 516–524, 1965.

[Hen89] P. Van Hentenryck. Parallel constraint satisfaction in logic programming : Preliminary results of CHIP within PEPSys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Leb93] F. Lebastard. CHOOE : Un gestionnaire d'environnement distribué. Rapport Technique 93-22, CERMICS-INRIA Sophia Antipolis, 1993.

[Man92] B. Mans. *Contribution à l'algorithmique non numérique parallèle : parallèlisations de méthodes de recherche arborescentes.* Thèse de doctorat, Université Paris VI, pages 169-182, 1992.

[Mau93] T. Mautor. *Contribution à la résolution des problèmes d'implantation : algorithmes séquentiels et parallèles pour l'affectation quadratique.* Thèse de doctorat, Université Paris VI, pages 117-132, 1993.

[MR92] B. Mans and C. Roucairol. Performances des algorithmes branch and bound parallèles à stratégie meilleur d'abord. Rapport de recherche 1716, INRIA Rocquencourt, 1992.

[Pro94] P. Prosser. Binary constraint satifaction problems : Some are harder than others. In *11th European Conference on Artificial Intelligence*, pages 95–99, 1994.

[Rou89] C. Roucairol. Parallel branch and bound algorithms, an overview. Rapport de recherche 962, INRIA Rocquencourt, 1989.

[Smi94] B. M. Smith. Phase transition and the mushy region in constraint satisfaction problems. *11th European Conference on Artificial Intelligence*, pages 100–104, 1994.

[SV94] T. Schiex and G. Verfaillie. Nogood Recording for static and dynamic CSP. In *5th IEEE International Conference on Tools with Artificial Intelligence*, pages 48–55, 1994.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*, pages 124–136. London, Academic Press Ltd, 1993.