Chapitre 4

Opérateurs Génétiques Spécialisés

Dans le chapitre précédent, nous avons présenté une nouvelle fonction d'évaluation qui nous semble plus appropriée pour résoudre les CSP. Dans le but d'améliorer la recherche de l'algorithme évolutionniste guidée par cette fonction, nous allons, dans ce chapitre, aborder la conception de deux opérateurs spécialisés pour les CSP. Ils prennent en compte la structure du graphe de contraintes en faisant de l'exploration et de l'exploitation pendant l'évolution. Le premier opérateur qui est conçu pour réaliser de l'exploration est appelé *arc-mutation*. Cet opérateur réalise des opérations de mutation en examinant les arêtes liées (contraintes) à la variable qu'on veut muter. Le deuxième que nous avons appelé *arc-crossover* réalise de l'exploitation. Il utilise une heuristique pour réaliser un croisement "intelligent" en examinant les arêtes du graphe de contraintes.

Ce chapitre commence avec une brève description des motivations et des difficultés de concevoir de bons opérateurs. Ensuite, nous donnons quelques définitions nécessaires avant d'aboutir à la définition des deux opérateurs. Après avoir défini les opérateurs, nous montrons les résultats obtenus par un algorithme évolutionniste qui les utilise dans un ensemble de tests sur des problèmes de 3-coloriage de graphe et des CSP aléatoires et nous le comparons avec un autre algorithme. Finalement, nous définissons les opérateurs *construint-crossover* et $arc_n - mutation$, qui sont une extension d'*arc-crossover* et d'*arc-mutation* pour les CSP n-aires.

4.1 Motivations et difficultés

Le principal problème des algorithmes évolutionnistes pour résoudre les CSP est leur haute probabilité de se trouver piégé dans un optimum local de la fonction d'évaluation. Notre principal problème à aborder est donc d'augmenter la probabilité de convergence de l'algorithme vers une solution. En conséquence, nous aurions diminué la probabilité de rester dans un optimum local.

Mais, pourquoi concevoir des nouveaux opérateurs? D'abord, parce que nous voulons améliorer la performance de l'algorithme évolutionniste en faisant une transformation plus adaptée au type de problème. Nous avons montré dans le chapitre précédent que le phénomène d'épistasie des CSP fait que l'algorithme avec un opérateur de *croisement à un point* est sensible à l'ordre dans lequel se trouvent les variables dans le chromosome. Nous souhaitons aussi avoir des opérateurs qui ne se sentent pas affectés par cet ordre. Notre but est de définir un algorithme évolutionniste pour résoudre les CSP en général. Pour ce faire, nous avons donc besoin de prendre en compte dans la conception des opérateurs, des caractéristiques propres aux CSP, mais qui soient assez générales pour nous permettre de les appliquer à différents types de problèmes. En général, inclure un opérateur spécialisé est plus coûteux en temps de calcul qu'utiliser un opérateur qui travaille d'une façon aveugle par rapport au problème. Dans le cas précis des CSP, nous souhaitons que l'incorporation des nouveaux opérateurs réalise un compromis entre le temps de calcul et l'augmentation de la probabilité de convergence vers une solution.

Une autre motivation est de donner une réponse à la communauté des contraintes, car plusieurs chercheurs en contraintes sont réticents envers ce type de méthodes stochastiques. Ils se posent les questions de pourquoi une méthode qui fait "sans justification" une recherche aléatoire en croisant ou en mutant des variables pourrait-elle arriver à trouver la solution pour un problème de satisfaction de contraintes? Comment pourrait-elle devenir compétitive par rapport aux méthodes traditionnelles complètes ou incomplètes, car au moins ces dernières utilisent une heuristique "intelligente" pour réparer les pré-solutions?

Ces questions nous ont motivée à définir nos opérateurs qui regardent le graphe de

contraintes pendant l'évolution. Ils essayent, en utilisant des heuristiques, de faire des mutations et des croisements plus informés qui permettent d'améliorer la recherche stochastique de l'algorithme évolutionniste. Dans la section suivante, nous allons montrer les idées qui sont derrière la conception des nouveaux opérateurs. Nous donnerons aussi quelques définitions préliminaires avant de définir ces opérateurs plus formellement.

4.2 Opérateurs

Les deux opérateurs que nous allons décrire sont dans le cadre des CSP binaires. À la fin de ce chapitre, nous présentons leur extension pour les CSP n-aires.
Nous rappelons au lecteur que les opérateurs sont classés dans l'étape *transformation* dans l'algorithme évolutionniste (voir figure 2.1).
Nous avons nommé le premier opérateur *arc-mutation*. Il fait de l'*Exploration* (il permet d'incorporer des nouvelles valeurs à partir du domaine des variables, qui ne sont pas présentes dans la population courante). Il utilise l'idée de base de la mutation, c'est à dire, de changer une valeur d'une variable dans le chromosome, mais son heuristique pour choisir la valeur prend en compte le graphe de contraintes.
Pour faire de l'*Exploitation* (utiliser des dernières pré-solutions pour construire une nouvelle), nous avons conçu un opérateur sexué nommé *arc-crossover*. Son objectif est de réaliser une recombinaison entre deux individus sélectionnés aléatoirement, les parents, pour générer un nouvel individu, leur fils, qui hérite le meilleur couple de leurs valeurs des variables par contrainte. Dans le contexte des CSP, cela signifie créer un fils qui aurait plus de chance de satisfaire plus de contraintes dans le réseau. Pour faire cela, nous avons ordonné les contraintes à satisfaire suivant leur contribution à la fonction d'évaluation (voir définition 3.2.3). *Arc-crossover* est un processus glouton qui construit un fils en utilisant cet ordre. Il va donc chercher à satisfaire en priorité les contraintes ayant une plus forte contribution à la fonction d'évaluation quand elles sont violées, quand il choisit les valeurs des variables parmi celles des parents.

La figure 4.1 montre la façon dont ces opérateurs sont utilisés dans l'algorithme évolutionniste, plus précisément dans l'étape de *transformation* (voir figure 3.15). Si

l'algorithme réalise *arc-crossover*, il génère un seul fils qui pourrait, éventuellement (suivant la probabilité de mutation) être muté par *arc-mutation*. Si *arc-crossover* n'a pas lieu, alors chacun des deux parents, qui avaient été choisis par la procédure de sélection, pourrait être muté dans *arc-mutation* suivant la probabilité de mutation. En résumé, la procédure de transformation pourrait soit générer un seul fils, dans le cas où les parents sont croisés par *arc-crossover*, soit deux fils dans le cas où *arc-crossover* n'aurait pas lieu.

Procédure Transformation($Parent_1, Parent_2$)
Début /* procédure transformation*/
Générer un nombre aléatoire r entre [0..1]
si r<probabilité de croisement **alors**
  $Fils$=arc-crossover($Parent_1, Parent_2$)
  $Fils$=arc-mutation($Fils$)
sinon
  $Fils_1$=arc-mutation($Parent_1$)
  $Fils_2$=arc-mutation($Parent_2$)
Fin /* procédure Transformation*/

FIG. 4.1 – *Structure de la procédure de Transformation*

Les deux opérateurs sont utilisés pour améliorer la recherche stochastique. Pour que les opérateurs utilisent pendant la recherche l'information du réseau de contraintes, nous avons défini trois fonctions d'évaluation particulières, une pour *arc-mutation* et deux pour *arc-crossover*. La première est nommée *fonction d'évaluation pour mutation* que nous utiliserons pour choisir la nouvelle valeur d'une variable. La deuxième est nommée *fonction d'évaluation partielle pour croisement* qui avec la *fonction d'évaluation partielle pour mutation* va nous permettre de guider la sélection d'une combinaison des valeurs des variables d'une contrainte.

4.2.1 Arc-mutation

L'opérateur *arc-mutation* réalise une exploration guidée par le graphe de contraintes. Il sélectionne d'abord aléatoirement la variable à muter. Le choix de la valeur pour cette variable est fait en utilisant la *fonction d'évaluation pour mutation*. Pour calculer

cette fonction, il identifie d'abord l'ensemble de contraintes qui ont comme variable pertinente la variable à muter, soit plus formellement:

Définition 4.2.1 *($\mathbf{M_j}$)*
Étant donné un CSP $P = (V, D, \zeta)$. On définit l'ensemble de contraintes $\mathbf{M_j} \subseteq \zeta$ pour une variable X_j par $C_\alpha \in \mathbf{M_j}$ ssi $X_j \rhd C_\alpha$.

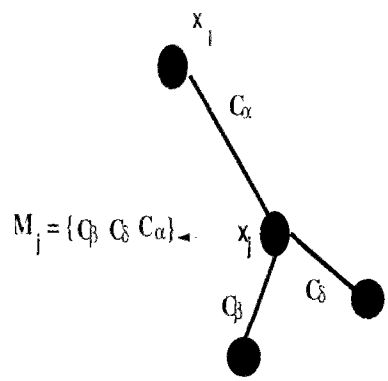

FIG. 4.2 - *Définition 4.2.1*

Définition 4.2.2 *(Fonction d'évaluation pour Mutation)*
Étant donné un CSP binaire $P = (V, D, \zeta)$, une instanciation $\mathbf{I}$, l'ensemble de contraintes $\mathbf{M_j}$ pour la variable X_j, et les fonctions $\mathbf{e}(C_\alpha, \mathbf{I})$ pour toute contrainte C_α, on définit la Fonction d'évaluation pour Mutation, **mff** *pour X_j comme:*

$$\mathbf{mff}(X_j, \mathbf{I}) = \sum_{C_\gamma \in \mathbf{M_j}} \mathbf{e}(C_\gamma, \mathbf{I}) \tag{4.1}$$

Remarque 4.2.1 *Cette fonction est calculée en prenant en compte seulement les arêtes impliquées (liées à X_j).*

Arc-mutation sélectionne pour une variable X_j la valeur x_j qui minimise la *fonction d'évaluation pour mutation*. La procédure est montrée sur la figure 4.3. Pour illustrer le mécanisme suivi par *arc-mutation*, considérons l'exemple avec quatre variables et quatre contraintes pour un problème de 3-coloriage. Le graphe de contraintes et le chromosome à muter sont montrés sur la figure 4.4. Les domaines de chaque variable

Procédure arc-mutation (chromosome)
Début
Pour chaque variable X_j
    Générer un nombre aléatoire r entre [0..1]
    si r <probabilité de mutation **alors**
      $\mathbf{I}(X_j) = argmin_{x_j \in D_j - \{x_{j_a}\}}\{\mathbf{mff}(X_j, ((\mathbf{I} - x_{j_a}{}^a) \cup x_j))\}^b$
Fin /* procédure arc-mutation */

[a] sa valeur courante
[b] $argmin_{l \in S}\{a_l\}$ donne la valeur l^* tel que $a_{l^*} \leq a_l, \forall l \in S$

FIG. 4.3 – *Structure de la procédure arc-mutation*

sont $D_i = \{1, 2, 3\}, \forall i = 1, ..., 4$. Supposons que nous voulons changer la valeur de la variable X_3 du chromosome, qui actuellement a la valeur 2. L'ensemble M_3 sera composé par $\{C_1, C_3, C_4\}$. Nous avons donc deux valeurs possibles pour X_3 soit 1 soit 3. Comme $\mathbf{mff}(X_3 = 1, \mathbf{I}) = 5$ et $\mathbf{mff}(X_3 = 3, \mathbf{I}) = 4$, *arc-mutation* choisira la valeur 3 pour X_3. En faisant cela, ce chromosome sera plus facile à réparer ensuite, en lui changeant, par exemple, la valeur de sa variable X_4 qui est moins contrainte.

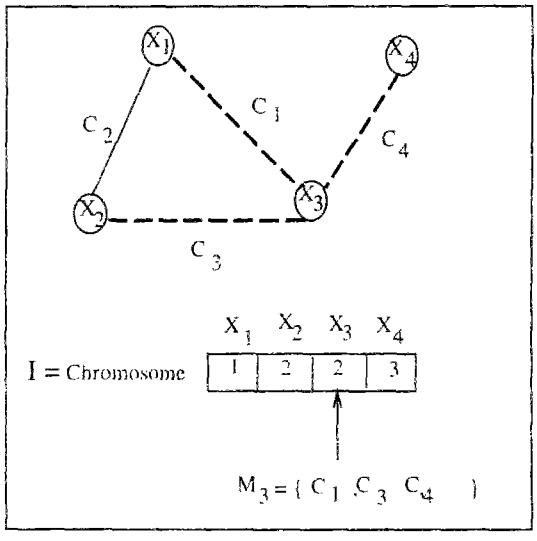

FIG. 4.4 – *Exemple: Arc-mutation*

Méthode	Ensemble de choix de la variable X_j	Heuristique pour sélection de la valeur	Domaine de choix de la valeur
min-conflits	$\mathbf{K(I)}$	$\min(\mathbf{mc}^{a}(X_j, \mathbf{I}))$	D_j
min-réseau-conflits	$\mathbf{K(I)}$	$\min(\mathbf{nc}^{b}(X_j, \mathbf{I}))$	D_j
arc-mutation	V	$\min(\mathbf{mff}(X_j, \mathbf{I}))$	D_j-(valeur courante)

TAB. 4.1 - *Comparaison entre min-conflits, min-réseau-conflits et arc-mutation*

[a] mc:définition 3.3.2
[b] nc:définition 3.3.3

Relation entre *arc-mutation*, *min-conflits* et *min-réseau-conflits*

Nous pouvons remarquer qu'il y a certaines similitudes entre ce que fait *arc-mutation* et ce que font les deux méthodes stochastiques *min-conflits* et *min-réseau-conflits* que nous avons décrites dans le chapitre précédent. Les trois choisissent une variable et elles changent sa valeur dans l'instanciation. Néanmoins, elles diffèrent par leur stratégie pour le choix de la variable et pour la sélection de la valeur de la variable.

En ce qui concerne le choix de la variable à muter, *min-conflits* et *min-réseau-conflits* prennent la variable à partir de l'ensemble des variables en conflits ($\mathbf{K(I)}$), c'est-à-dire, parmi les variables qui appartiennent à une contrainte qui est violée. En revanche, *arc-mutation* réalise la sélection de la variable d'une manière purement aléatoire, parmi l'ensemble V de variables du problème. Il parcourt toute l'instanciation, variable par variable, et suivant la probabilité de mutation l'opérateur décide si elle sera mutée.

Pour le choix de la valeur, *min-conflits* choisit la valeur qui minimise le nombre de contraintes violées dans le graphe de contraintes. Par contre, *min-réseau-conflits* sélectionne la valeur qui minimise la valeur de la *fonction de min-réseau-conflits* (équation 3.5), la valeur donc minimise le nombre de variables impliquées dans la violation des contraintes. *Arc-mutation* cherche la valeur qui apporte une amélioration globale au problème, une diminution donc de la valeur de la fonction d'évaluation $\mathbf{Z(I)}$ mesurée en utilisant **mff**. Il est évident que la valeur de la fonction **mff** correspond exactement à celle de la fonction **nc**. Les deux fonctions mesurent le nombre de conflits propagés sur le réseau de contraintes. La différence la plus remarquable entre *min-réseau-conflits* et *arc-mutation* est qu'une variable peut être choisie par *arc-mutation*

sans qu'elle soit liée à une contrainte non-satisfaite, que ce n'est pas le cas de *min-réseau-conflits*.

Il est important de remarquer que *min-conflits* ainsi que *min-réseau-conflits* gardent la valeur actuelle de la variable quand une amélioration n'est produite par aucun changement. Par contre, *arc-mutation* empêche l'affectation de la valeur courante à la variable. Avec *arc-mutation*, il est possible donc que la valeur de la fonction d'évaluation diminue, ce qui n'est pas le cas des heuristiques *min-conflits* et *min-réseau-conflits*. Le résumé de cette comparaison est montré sur le tableau 4.1

4.2.2 Arc-crossover

Les meilleurs résultats trouvés jusqu'à présent dans la littérature pour la résolution de CSP par des algorithmes évolutionnistes, ont été obtenus avec des algorithmes qui utilisent des opérateurs asexués, [ERR94], [DBB94]. Ces algorithmes sont plutôt orientés vers l'exploration que vers l'exploitation, à cause en partie de la caractéristique d'épistasie des CSP, en d'autres termes à cause de la haute corrélation entre les variables dans le chromosome. De plus, utiliser l'opérateur classique de croisement peut, éventuellement, produire une dégradation de la performance de l'algorithme comme nous l'avons remarqué dans le chapitre 2.

Nous proposons un nouvel opérateur sexué *arc-crossover* qui prend en compte la structure du CSP. Notre objectif premier ici est la réparation des contraintes. Il consiste donc à créer un fils à partir d'une "bonne" recombinaison (voir déf 2.2.4) des valeurs de deux parents. Pour illustrer l'idée qu'il y a derrière sa conception, voyons la figure 4.5.

Dans ce cas, nous avons sélectionné deux parents Cr_1 et Cr_2 et nous souhaitons générer un fils qui hérite de la meilleure combinaison de leurs valeurs des variables. Supposons que Cr_1 satisfait le $Graphe_1$ et que Cr_2 satisfait le $Graphe_2$. De plus, supposons que la valeur de la variable X_4 dans Cr_2 et la valeur de la variable X_5 dans Cr_1 satisfont la contrainte C_1, et que la valeur de la variable X_3 dans Cr_1 et la valeur de la variable X_6 dans Cr_2 satisfont la contrainte C_2. Si nous faisons un bon croisement entre ces parents, nous pourrons obtenir un fils qui satisfait toutes

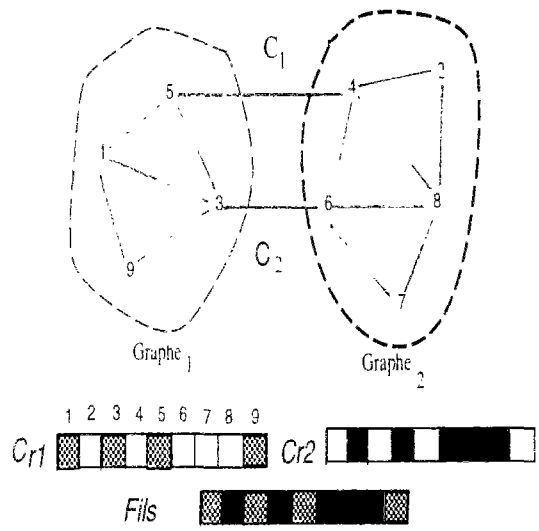

FIG. 4.5 - *Exemple: Bon croisement*

les contraintes du graphe. Il est évident que c'est une situation spéciale, mais nous montrerons que le fait d'inclure un opérateur qui prenne en compte l'idée de partition en sous-graphes nous permettra d'obtenir une amélioration de la performance de l'algorithme évolutionniste. *Arc-crossover* travaille sur la base d'une séquence ordonnée de contraintes.

Définition 4.2.3 *(Pré-ordre des Contraintes)*
Soit un CSP binaire $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$*, une instanciation* $\mathbf{I}$ *et la Contribution de* C_α *à la fonction d'évaluation* $\mathbf{c}(C_\alpha)$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$*. On définit un Pré-ordre des Contraintes (noté "*$\preceq$*"), par la règle suivante:*

$$C_{k_i} \preceq C_{k_j} \text{ ssi } \mathbf{c}(C_{k_i}) \geq \mathbf{c}(C_{k_j})$$

Définition 4.2.4 *(Priorité de Contraintes)*
Soit un CSP binaire $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$*, une instanciation* $\mathbf{I}$ *et la Contribution de* C_α *à la fonction d'évaluation* $\mathbf{c}(C_\alpha)$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$*. On définit la Priorité de Contraintes* $\mathbf{P}$ *comme une séquence sur le pré-ordre* $\preceq$ *des contraintes telle que:*

$$\mathbf{P} = \langle C_{k_1}, \ldots, C_{k_n} \rangle \text{ avec } C_{k_i} \preceq C_{k_{i+1}}, \forall i = 1, \ldots, \eta - 1$$

$\mathbf{P}$ est un η-uplet ordonné de contraintes. Intuitivement, nous ordonnons les contraintes suivant leur contribution à la fonction d'évaluation $\mathbf{Z(I)}$, suivant donc le nombre de variables impliquées dans la violation d'une contrainte.

Au début de la procédure, le fils n'a aucune variable instanciée. L'algorithme commence son analyse à partir de la contrainte la plus prioritaire selon $\mathbf{P}$. Le fils instanciera alors les deux variables de cette contrainte, cela sera la première instanciation partielle $\mathbf{I_p}$. Ensuite, l'algorithme continue avec la contrainte suivante selon la priorité. Le fils est construit en instanciant les variables au fur et à mesure que l'algorithme analyse les contraintes du réseau. La sélection des valeurs est faite en utilisant la *fonction d'évaluation partielle pour croisement*. Pour ce faire, l'algorithme a besoin d'abord d'identifier l'ensemble de contraintes qui sont concernées par ce choix. Cela s'exprime plus formellement avec les définitions suivantes:

Définition 4.2.5 *($\mathbf{S}_\alpha$)*
Étant donné un CSP $P = (V, D, \zeta)$ et une instanciation partielle $\mathbf{I_p}$ (voir définition 1.1.7). On définit l'ensemble $\mathbf{S}_\alpha \subseteq \zeta$ pour une contrainte C_α complètement instanciée (c'est à dire toutes les variables pertinentes pour C_α sont instanciées) par $C_i \in \mathbf{S}_\alpha$ ssi

- $\exists X_i : X_i \rhd C_\alpha$ *et* $X_i \rhd C_\beta$ *($C_\alpha$etC_β ont une variable en commun)*
- $\forall X_j$ *pertinente à* C_β, X_j *est instanciée*

Cela veut dire que le changement de la valeur d'une variable de la contrainte C_α, pourrait altérer la satisfaction des contraintes qui appartiennent à l'ensemble $\mathbf{S}_\alpha$. Cette définition est utilisée par l'algorithme au fur et à mesure que les variables sont instanciées.

Remarque 4.2.2 *Il est évident que* $C_\alpha \in \mathbf{S}_\alpha$

Définition 4.2.6 *(Fonction d'évaluation Partielle pour Croisement)*
Étant donné un CSP binaire $P = (V, D, \zeta)$, une instanciation partielle $\mathbf{I_p}$, les ensembles $\mathbf{S}_\alpha$, et les fonctions $\mathbf{e}(C_\alpha, \mathbf{I_p})$ pour toute contrainte C_α complètement instanciée, on définit la Fonction d'évaluation Partielle pour Croisement, **cff** *pour* C_α

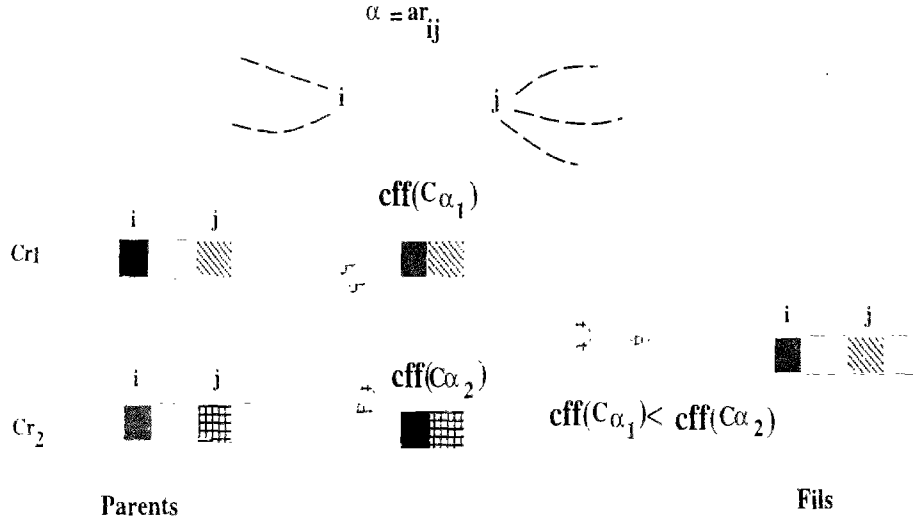

FIG. 4.7 – *Arc-crossover: Croisement pour une arête violée par les deux parents*

Une fois que la plupart des arêtes ont été analysées, il reste des contraintes qui ne sont pas encore traitées qui pourraient avoir déjà une de leurs variables instanciée

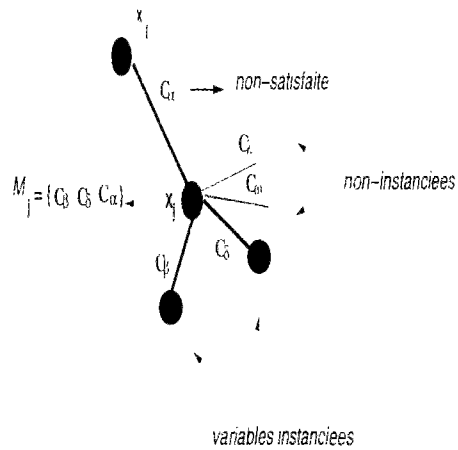

FIG. 4.8 – *Définition 4.2.7*

Définition 4.2.8 *(Fonction d'évaluation Partielle pour Mutation)*
Étant donné un CSP binaire $P = (V, D, \zeta)$, *une instanciation partielle* $\mathbf{I_p}$, *les ensembles* $\mathcal{M}_\mathbf{j}(\mathbf{I_p})$ *pour toute variable instanciée sous* $\mathbf{I_p}$, *et les fonctions* $\mathbf{e}(C_\alpha, \mathbf{I_p})$ *pour toute contrainte* C_α *complètement instanciée, on définit la* Fonction d'évaluation Partielle pour Mutation, $\mathbf{mff_p}$ *pour* X_j *comme:*

$$\mathbf{mff_p}(X_j, \mathbf{I_p}) = \sum_{C_\gamma \in \mathcal{M}_\mathbf{j}(\mathbf{I_p})} \mathbf{e}(C_\gamma, \mathbf{I_p}) \tag{4.3}$$

Remarque 4.2.4 *Cette fonction est calculée en prenant en compte seulement les arêtes impliquées (liées à* X_j*) et dont les variables sont instanciées sous* $\mathbf{I_p}$.

Quand il ne reste qu'une variable de l'arête à instancier, *arc-crossover* réalise la sélection de sa valeur, en utilisant la *fonction d'évaluation partielle de mutation*, quand aucune des deux valeurs des parents ne permet de satisfaire la contrainte analysée. Ce cas est montré sur la figure 4.9. La procédure d'*arc-crossover* est montrée sur la figure 4.10.

Pour illustrer comment procède arc-crossover, considérons l'exemple montré sur la figure 4.11. Nous avons les deux chromosomes (parents). Prenons la priorité suivante $\mathbf{P} = \langle C_1, C_3, C_2, C_4 \rangle$ pour les contraintes du graphe, suivant leur contribution à la fonction d'évaluation. La procédure réalise ce qui suit:

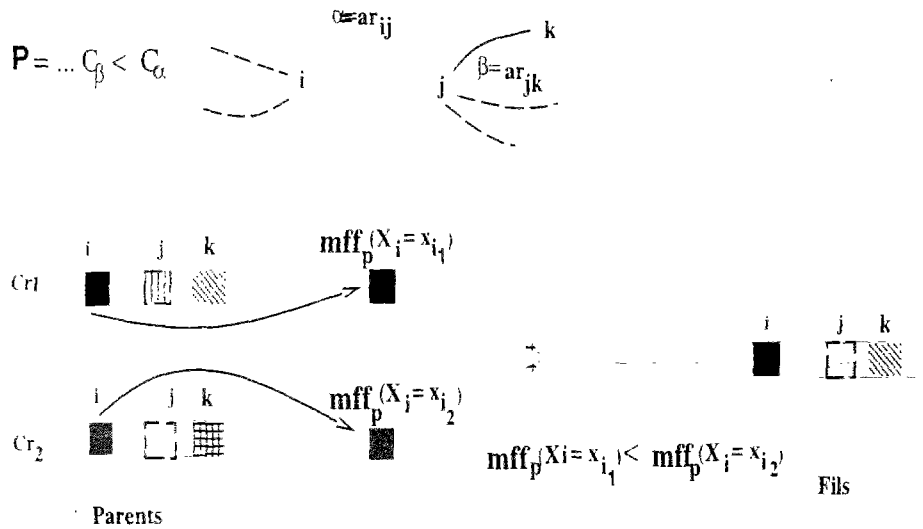

FIG. 4.9 – *Arc-crossover: Héritage de la valeur d'une seule variable qui viole la contrainte* C_α

Analyse de C_1. Variables: X_1, X_3

Comme aucun des parents ne satisfait cette contrainte, alors l'algorithme choisit parmi les paires des valeurs (1,2) et (2,1) pour les variables X_1 et X_3 respectivement. Les deux paires ont la même valeur de **cff**, supposons donc qu'elle choisisse la paire (1,2). Nous obtenons donc l'instanciation partielle suivante pour leur fils:

1		2	

- Analyse de C_3. Variables: X_2, X_3

  X_3 est déjà instanciée, il ne nous reste donc qu'à choisir la valeur pour X_2 parmi les valeurs de ses parents, donc entre 1 et 3. Avec $X_2 = 1$, le fils violera la contrainte C_3, par contre avec $X_2 = 3$, elle sera satisfaite. Par conséquent, la nouvelle instanciation partielle pour leur fils sera:

1	3	2	

- Analyse de C_2. Variables: X_1, X_2

Procédure arc-crossover $(Parent_1, Parent_2)$
Début
Pour chaque C_α suivant l'ordre donné par **P**
Analyser C_α (x_i, x_j) du $Parent_1$ et du $Parent_2$
selon
$((X_i \dashv \mathbf{I_p})^a$ et $(X_j \dashv \mathbf{I_p}))$:
selon
$((C_\alpha \models Parent_1)$ et$(C_\alpha \models Parent_2))$:
selon
$\mathbf{Z}(Parent_2) > \mathbf{Z}(Parent_1) : \mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
$\mathbf{Z}(Parent_1) > \mathbf{Z}(Parent_2) : \mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
autre: $\mathbf{I_p}(X_i, X_j) = random((x_{i_1}, x_{j_1}), (x_{i_2}, x_{j_2}))$
$((C_\alpha \models Parent_1)$ ou$(C_\alpha \models Parent_2))$:
si $(C_\alpha \models Parent_1)$ **alors**
$\mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
sinon $\mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
autre: $\mathbf{I_p}(X_i, X_j) = argmin_{s_1 \in S_1}(\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_1}, x_{j_2}))),$
$\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_2}, x_{j_1})))^b$
$((X_i \dashv \mathbf{I_p})$ ou $(X_j \dashv \mathbf{I_p}))$:
$(X_i \dashv \mathbf{I_p})$ **alors** k=i **sinon** k=j
selon
$(C_\alpha \models (\mathbf{I_p} \cup x_{k_1})$ et$(C_\alpha \models (\mathbf{I_p} \cup x_{k_2}) :$
$\mathbf{I_p}(X_k) = random(x_{k_1}, x_{k_2})$
$(C_\alpha \models (\mathbf{I_p} \cup x_{k_1})$ et$(C_\alpha \not\models (\mathbf{I_p} \cup x_{k_2}) :$
$\mathbf{I_p}(X_k) = x_{k_1}$
$(C_\alpha \not\models (\mathbf{I_p} \cup x_{k_1})$ et$(C_\alpha \models (\mathbf{I_p} \cup x_{k_2}) :$
$\mathbf{I_p}(X_k) = x_{k_2}$
autre :
$\mathbf{I_p}(X_k) = argmin_{s_2 \in S_2}(\mathbf{mff_p}(X_k, (\mathbf{I_p} \cup x_{k_1})),$
$\mathbf{mff_p}(X_k, (\mathbf{I_p} \cup x_{k_2})))^c$
Fin/* arc-crossover */

---

[a] X_i non-instanciée en $\mathbf{I_p}$
[b] $argmin_{s \in S}\{a_s\}$ donne l'ensemble s^* tel que $a_{s^*} \leq a_s, \forall s \in S$.
$S_1 = \{(x_{i_1}, x_{j_2}), (x_{i_2}, x_{j_1})\}$
[c] $S_2 = \{x_{k_1}, x_{k_2}\}$

FIG. 4.10 – *Structure de la procédure arc-crossover*

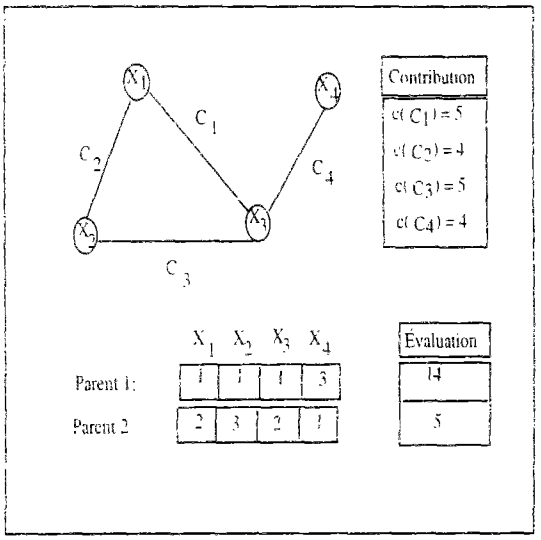

FIG. 4.11 – *Exemple: Arc-crossover*

Les deux variables sont déjà instanciées donc l'algorithme continue avec la dernière contrainte

Analyse de C_4. Variables: X_3, X_4
Il faut seulement instancier la variable X_4. Les deux valeurs possibles sont également bonnes. Avec les deux, nous obtenons un fils qui est une solution au problème.

1	3	2	1

1	3	2	3

Nous nous apercevons qu'un individu assez mal qualifié (Parent 1) peut apporter quelques valeurs qui peuvent nous aider en tant que structures intermédiaires vers la solution.

4.3 Ensemble de problèmes: 3-Coloriage

Nous avons testé l'algorithme avec des problèmes de 3-coloriage avec solution, générés aléatoirement. Nous avons généré des graphes aléatoires avec différentes topologies avec un degré de connectivité entre [4.6] pour 30 variables. Nous avons le même ensemble de problèmes que dans le chapitre 3, mais cette fois-ci nous comparons deux algorithmes qui utilisent la même fonction d'évaluation Z(**I**) et notre algorithme de sélection. Il ne diffèrent que par leurs opérateurs. L'*Algorithme A* utilise *arc-crossover* et *arc-mutation* dans la procédure *transformation* montrée sur la figure 4.1. L'*Algorithme B* utilise les opérateurs *mutation* et (#,r,b), [ERR96] qui est jusqu'à présent un des meilleurs algorithmes génétiques pour le problème de 3-coloriage. Pour chaque connectivité, nous avons généré 100 graphes différents. La figure 4.12 montre le pourcentage de solutions trouvées par les deux algorithmes. Les nouveaux opérateurs ont en moyenne de meilleurs résultats. L'*Algorithme A* a trouvé dans le pire des cas 83% de solutions, en revanche l'*Algorithme B* a trouvé pour le pire des cas 70% de solutions. La figure 4.13 montre le nombre moyen de générations pour chaque connectivité. Ce nombre moyen pour l'*Algorithme B* est supérieur à celui de l'*Algorithme A*. Nous pouvons donc conclure que les nouveaux opérateurs dans l'algorithme évolutionniste permettent de mieux guider sa recherche.

4.3.1 Opérateurs: nombre de vérifications de contraintes

Un des principaux objectifs des algorithmes de résolution de CSP systématiques est de réaliser le moindre nombre de vérifications de contraintes. Dans cette section, nous voulons estimer le nombre de vérifications de contraintes qu'effectuent nos opérateurs: *arc-mutation* et *arc-crossover*. Afin d'obtenir cette estimation, nous devrons faire d'abord quelques suppositions, cela à cause de la nature probabiliste de notre algorithme. Nous présentons d'abord le modèle utilisé pour faire les estimations, ensuite à l'aide du modèle nous analysons les opérateurs *arc-crossover*, *arc-mutation* et $(\#, r, b)$.

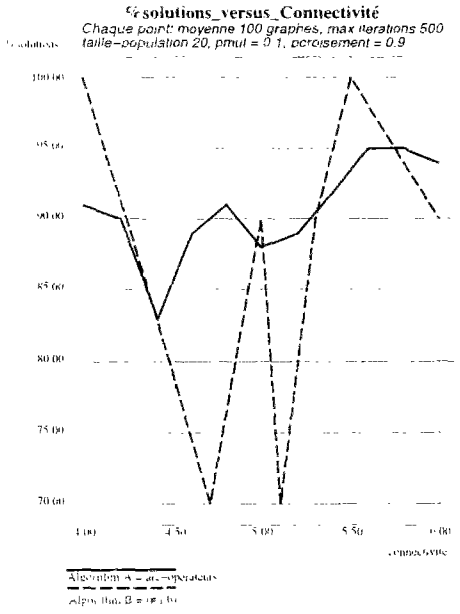

FIG. 4.12 – *Pourcentage de solutions trouvées par Algorithme A et Algorithme B pour différentes connectivités*

Le modèle

Paramètres du modèle

p_c	probabilité de croisement
p_m	probabilité de mutation
t_p	taille de la population
n	nombre de variables
p_1	probabilité de connectivité
m	taille des domaines

Dans ce modèle, la probabilité de connectivité correspond à la probabilité qu'il y ait une contrainte entre une paire de variables.

Conséquences du modèle

$\eta =$	$\frac{n(n-1)p_1}{2}$	nombre moyen de contraintes
$C =$	$p_1(n-1)$	connectivité moyenne par variable

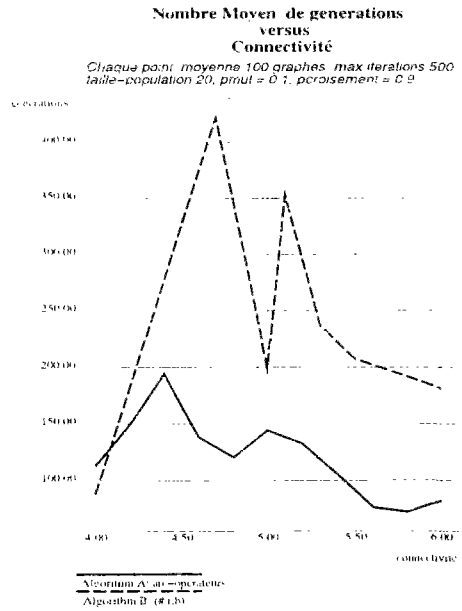

FIG. 4.13 – *Comparaison: Nombre moyen de générations pour Algorithme A et Algorithme B pour différentes connectivités*

Arc-opérateurs

Nous analysons tout d'abord le nombre de vérifications de contraintes réalisé par *arc-crossover*, ensuite celui fait par *arc-mutation* et finalement le nombre de tests de contraintes de l'algorithme complet.

Arc-crossover On étudie l'algorithme d'*arc-crossover* montré sur la figure 4.10. On fera l'analyse pour le pire des cas pour *arc-crossover*, celui quand la contrainte analysée C_α est violée par les deux parents et quand elle n'est pas instanciée.

- Quand *arc-crossover* traite une contrainte qui n'est pas instanciée, il teste cette contrainte pour chaque parent. Cela correspond alors à deux tests de contraintes.

- Quand la contrainte analysée C_α est violée par les deux parents et les deux variables ne sont pas instanciées, on dira qu'*arc-crossover* réalise un *bi-crossover*. Il devra choisir entre les deux paires de valeurs formées à partir des parents, en évaluant deux fois la fonction **cff**. Pour une évaluation de **cff**, *arc-crossover*

vérifie les contraintes qui sont liées à chacune des variables pertinentes de C_α et qui ont une variable instanciée. Nous dénotons par $N_{vb}(\alpha)$ le nombre de vérifications de contraintes réalisé par une application de **cff**.

Dans le cas où il ne reste qu'une variable de la contrainte C_β à instancier, *arc-crossover* utilise la fonction d'*arc-mutation partielle* $\mathbf{mff_p}$ deux fois (une pour chaque valeur des parents). Nous appelons $N_{va}(\beta)$ le nombre de vérifications de contraintes réalisées par une application de $\mathbf{mff_p}$.

On appelle $\mathcal{B}$ l'ensemble des contraintes qui ont réalisé un *bi-crossover* et $\mathcal{A}$ l'ensemble de contraintes qui ont été instanciées par une *arc-mutation-partielle*. Donc, en tout pour *arc-crossover* nous avons:

$$N_{vc} = \sum_{i=1}^{\eta} 2 + 2 \sum_{\alpha \in \mathcal{B}} N_{vb}(\alpha) + 2 \sum_{\beta \in \mathcal{A}} N_{va}(\beta) \tag{4.4}$$

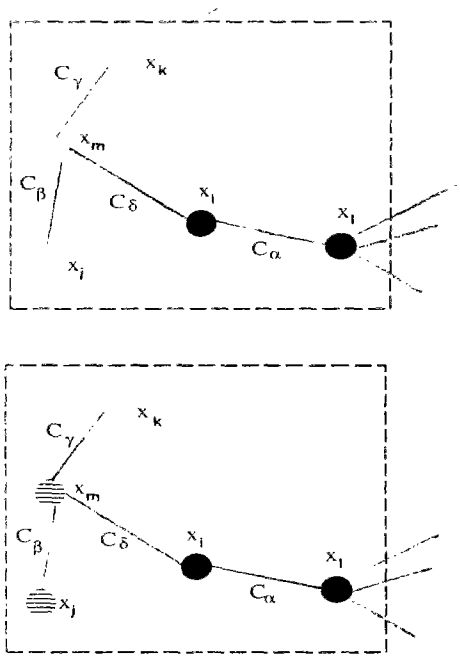

FIG. 4.14 – *Ex: vérifications de contraintes par arc-crossover*

Remarque 4.3.1 *Chaque contrainte n'est vérifiée que deux fois (une fois pour chaque paire de valeurs) en instanciant un fils avec arc-crossover.*

Pour montrer cela, regardons la figure 4.14. La figure montre une partie d'un graphe de contraintes avec 4 contraintes. Supposons la priorité d'analyse suivante: $\mathbf{P} :< C_\alpha, C_\beta, C_\delta, C_\gamma >$. On commence donc par instancier les variables x_i et x_l, dans ce cas on ne testera que C_α car les autres contraintes ne sont pas instanciées. Ensuite, *arc-crossover* analyse C_β et il teste C_β et C_δ, car x_l, l'autre variable pertinente de C_δ, est instanciée. La contrainte suivante à analyser est C_δ, mais comme elle est déjà complètement instanciée, l'algorithme ne réalise cette fois aucun test et il continue avec C_γ, qui sera instanciée en réalisant une *arc-mutation-partielle*, car le fils a déjà une valeur pour x_m. Une fois qu'une contrainte a ses deux variables instanciées, elle ne sera donc plus testée. Cela peut s'exprimer plus formellement par:

$$\sum_{\alpha \in \mathcal{B}} N_{vb}(\alpha) + \sum_{\beta \in \mathcal{A}} N_{va}(\beta) \leq \eta \tag{4.5}$$

En conséquence, le nombre de vérifications de contraintes N_{vc} (voir équation 4.4) réalisés par un *arc-crossover* aura une borne supérieure égale à:

$$N_{vc} \leq 4\eta \tag{4.6}$$

Arc-mutation Chaque fois qu'on réalise *arc-mutation*, l'algorithme change la valeur d'une variable suivant la probabilité de mutation. Pour le choix de la valeur, il teste avec **mff** toutes les valeurs du domaine de la variable sauf sa valeur actuelle. Cela s'exprime par:

$$N_{vm} = (m-1)p_1(n-1)np_m = 2\eta(m-1)p_m \tag{4.7}$$

où N_{vm} est le nombre de vérifications de contraintes réalisé par *arc-mutation*.

Algorithme qui utilise les Arc-opérateurs Le nombre de vérifications de contraintes réalisé par l'algorithme dépend du nombre de fois où chaque opérateur est utilisé, et de la taille de la population (t_p). On peut estimer sa valeur par:

$$N_{v_{algo}} = \frac{t_p p_c}{2 - p_c} N_{vc} + t_p N_{vm} \leq \left(\frac{4\eta p_c}{2 - p_c} + 2\eta(m-1)p_m \right) t_p. \tag{4.8}$$

Le facteur $\frac{p_c}{2-p_c}$ provient du fait que pour chaque appel à *arc-crossover* on peut générer soit un fils, dans le cas où l'algorithme réalise le croisement (avec probabilité p_c), soit deux fils (avec probabilité $1 - p_c$).

(#,r,b)

(#,r,b) est un opérateur asexué qui choisit $\frac{n}{4}$ variables qui vont subir une mutation. La sélection des variables est aléatoire, par contre la sélection de la valeur est faite en minimisant le nombre de conflits dans lesquels intervient la variable en question. Le nombre de vérifications de contraintes réalisé par cet opérateur sera:

$$N'_{vc} = mn\frac{p_1(n-1)}{4} = \frac{\eta m}{2} \tag{4.9}$$

Cette valeur dépend des nombre de contraintes, mais aussi de la taille du domaine des variables. Le nombre de vérifications de contraintes réalisé par l'algorithme qui utilise (#,r,b) et mutation sera:

$$N'_{valgo} = N'_{vc}t_p p_c = \frac{t_p p_c \eta m}{2} \tag{4.10}$$

car l'opérateur standard de *mutation* est purement aléatoire et ne teste aucune contrainte. N'_{vc} montre que l'utilisation de l'opérateur (#,r,b) est moins coûteuse pour chaque application qu'*arc-crossover*, en termes du nombre de vérifications de contraintes pour le problème de 3-coloriage (m=3). Il est évident que si on considère seulement le nombre de vérifications de contraintes par chaque application de l'opérateur, alors pour les problèmes avec une taille de domaine supérieure à 7, (#,r,b) devient moins performant qu'*arc-crossover*. Néanmoins, l'efficacité d'un opérateur pour résoudre les CSP dépend de deux facteurs [ERR95]:

le pourcentage de cas pour lequel l'algorithme trouve une solution,

le nombre de générations nécessaires pour trouver une solution.

En effet, la véritable efficacité d'un opérateur doit considérer le nombre de fois qu'il est appliqué jusqu'à trouver une solution. Cela est lié au nombre de générations effectuées par l'algorithme.

Comparaison expérimentale

Pour tester les deux opérateurs, nous avons généré des graphes pour le 3-coloriage avec solution, de façon aléatoire avec différents pourcentages de connectivité entre [10..90], en mesurant le nombre de générations et le nombre de vérifications de contraintes réalisées par chaque opérateur. Nous avons mesuré aussi le temps CPU utilisé par la procédure *génération*, qui réalise la *transformation* (application des opérateurs) et l'évaluation des nouveaux individus. Les deux algorithmes évolutionnistes utilisent la fonction $Z(\mathbf{I})$ comme fonction d'évaluation, notre algorithme de sélection, une probabilité de croisement de 0.9 et une probabilité de mutation de 0.5. Pour les arc-opérateurs, nous avons fixé le nombre de générations à 500, en revanche pour l'algorithme qui utilise (#,r,b) et mutation, le nombre maximum de générations est de 1500. Nous avons en cela considéré la capacité de ce dernier de réaliser moins de vérifications de contraintes par génération. Nous avons généré 100 graphes pour un nombre de contraintes donné, les problèmes ayant 30 variables et une taille de population égal à 20. Les résultats sont montrés sur le tableau 4.2 pour les arc-opérateurs et sur le tableau 4.3 pour l'opérateur (#,r,b) plus mutation. Les résultats pour les arc-opérateurs sont montrés sur les figures 4.15 et 4.16 respectivement pour le temps CPU de la procédure *génération* et le nombre de vérifications de contraintes. La figure 4.17 montre le temps CPU de la procédure *génération* et la figure 4.18 le nombre de vérifications de contraintes, les deux pour l'opérateur (#,r,b).

Nous pouvons conclure qu'en moyenne nos opérateurs sont plus efficaces que l'opérateur (#,r,b) plus mutation, car ils permettent à l'algorithme de converger plus rapidement vers une solution, ainsi que de résoudre plus de problèmes.

4.4 Ensemble de problèmes: CSP aléatoires

Nous avons conçu une série de tests avec des CSP générés aléatoirement qui ont une solution. Chaque ensemble de problèmes est caractérisé par 4 paramètres: n, le nombre de variables: m, le nombre de valeurs dans chaque domaine; p_1 la probabilité qu'il y ait une contrainte entre une paire de variables; p_2 la probabilité conditionnelle

Contraintes	Générations	$N_{c_{algo}}$	CPU[sec] [a]
30	2.48	7609	18
45	11.24	44967	110
60	83.44	422426	1053
90	63.43	465359	917
120	25.96	250505	632
150	18.34	219715	452
180	15.43	219847	580
210	12.18	201619	419
240	11.70	220700	419
270	11.88	252820	673

TAB. 4.2 – *Performances* arc-opérateurs*: Nombre moyen de générations et de* N_{vc} *par graphe*

[a] les données de temps CPU correspondent aux temps utilisés par la procédure génération pour résoudre les 100 graphes

Contraintes	Générations	$N'_{c_{algo}}$	CPU[sec] [a]
30	24.16	11141	67
45	294.51	185671	1144
60	1020.47	843709	4891
90	1137.93	1401437	6372
120	798.21	1311153	5908
150	516.38	1060255	5938
180	433.47	1070010	5961
210	356.12	1022748	5693
240	285.36	936505	5296
270	312.00	1153107	5087

TAB. 4.3 – *Performances (#,r,b): Nombre moyen de générations et de* N'_{vc} *par graphe*

[a] les données de temps CPU correspondent aux temps de la procédure génération pour résoudre les 100 graphes

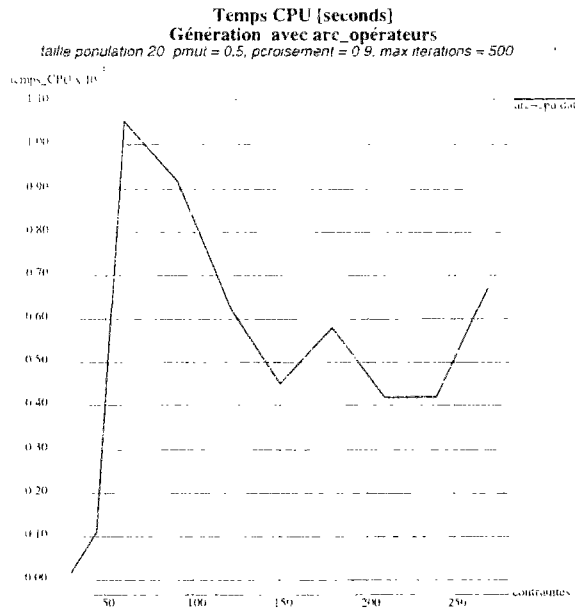

FIG. 4.15 – *Arc-opérateurs: Temps CPU pour résoudre 100 graphes en fonction du nombre de contraintes*

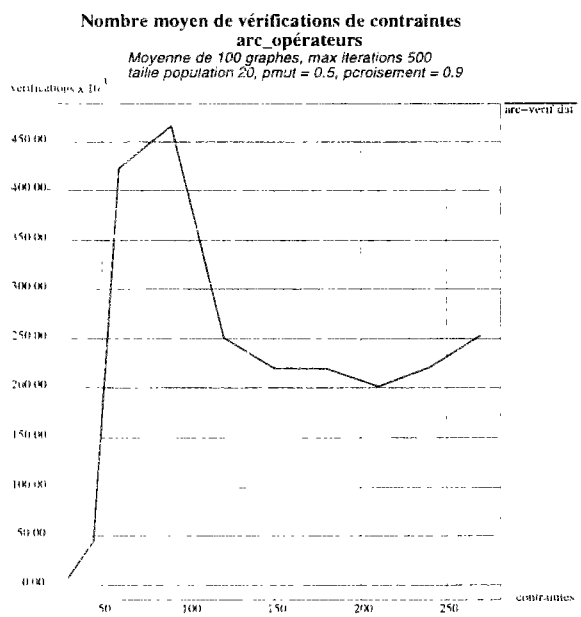

FIG. 4.16 – *Arc-Opérateurs: Nombre moyen de vérifications de contraintes*

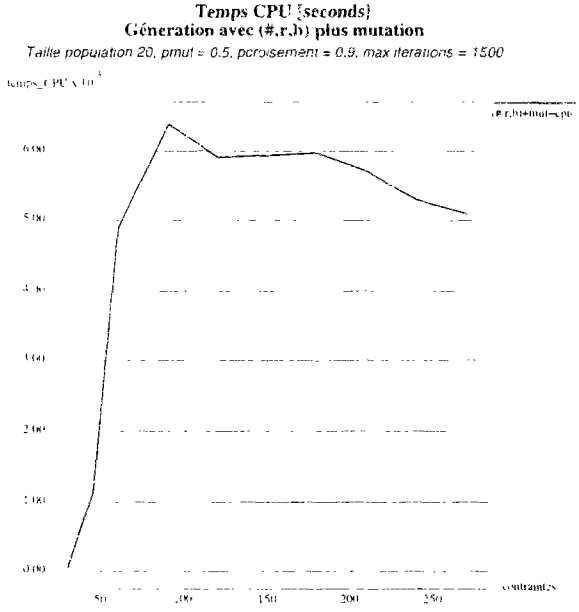

FIG. 4.17 – *(≠,r.b): Temps CPU pour résoudre 100 graphes en fonction du nombre de contraintes*

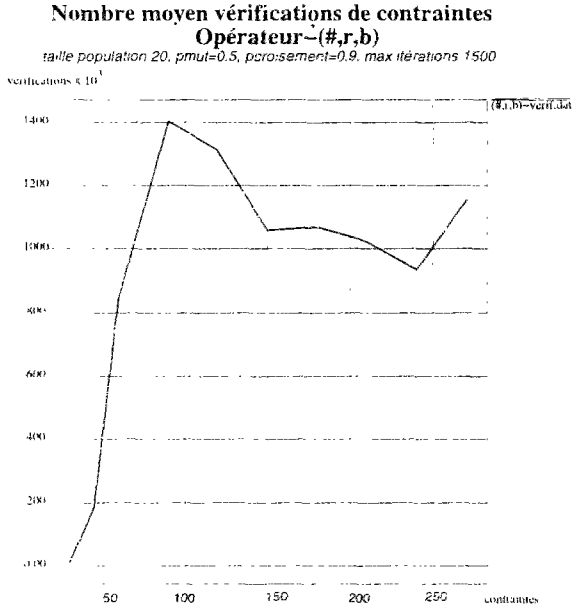

FIG. 4.18 – *(≠,r.b): Nombre moyen de vérifications de contraintes*

qu'une paire de valeurs soit inconsistante pour une paire de variables, sachant qu'il y a une contrainte entre elles. La procédure de génération de CSP aléatoires crée d'abord la structure du réseau de contraintes suivant la valeur de p_1. Une fois que les contraintes sont créées, la procédure génère aléatoirement une solution au problème. Nous rappelons que nous souhaitons créer des CSP qui ont, avec sûreté, une solution. Nous tirons aléatoirement une instanciation solution. Nous connaissons (marquons) donc une paire de valeurs compatibles pour chaque contrainte. Ensuite, pour chaque contrainte, en suivant la probabilité p_2, l'algorithme définit aléatoirement quelles seront les paires de valeurs incompatibles entre les variables. L'algorithme est montré sur la figure 4.19.

La valeur initiale de p_1 sera modifiée dans le cas où il y a des variables qui n'appartiennent à aucune contrainte. On veut un graphe sans variables isolées, on ajoute donc des contraintes pour l'obtenir (voir Annexe B). Pour chaque contrainte, le nombre de paires de valeurs inconsistantes est $m^2 p_2$.

Les paramètres utilisés sont n=8, m=10, pour différentes valeurs de p_1 et p_2. Nous avons généré 50 graphes aléatoires pour chaque p_2 et nous avons fixé le nombre d'itérations à 500. Tous les graphes ont au moins une solution. La figure 4.20 montre le pourcentage de solutions trouvées par rapport à p_2 pour $p_1 \in \{0.5, 0.8, 0.85, 1.0\}$. Nous pouvons observer que pour chaque p_1 il y a une valeur de p_2 pour laquelle trouver une solution pour notre algorithme est très difficile. Cependant, pour des grandes et des petites valeurs de p_2 il est facile de trouver une solution. Nous pouvons estimer les valeurs de p_2 qui sont critiques pour notre algorithme en fonction de p_1, ces valeurs sont liées au nombre de solutions espérées (voir Annexe B). Pour notre algorithme, le problème sera plus difficile quand il y aura un grand nombre de combinaisons de valeurs qui satisfont localement une contrainte, mais qui ne font pas partie d'une solution globale.

Pour les grandes valeurs de p_2, le nombre de choix de valeurs consistantes par contrainte diminue, ainsi quand p_2 est près de 1, le problème n'a qu'une solution et les valeurs consistantes par arête correspondent donc exactement aux valeurs qui font partie de la solution unique. En d'autres termes, quand notre algorithme a trouvé une paire consistante, il a trouvé les valeurs de la solution pour cette contrainte.

Procédure Génération CSP aléatoires (p_1, p_2, n, m)
Début
nombre-de-contraintes = $\frac{n*(n-1)*p_1}{2}$
contraintes-créées = 0
$\zeta = \emptyset$
tant que (contraintes-créées <nombre-de-contraintes) **faire**
    Générer aléatoirement nœud$_1 \in [1..n]$
    Générer aléatoirement nœud$_2 \in [1..n]$, tel que nœud$_1 \neq$ nœud$_2$
    si ($\nexists C_\alpha$ entre nœud$_1$ et nœud$_2$) **alors**
        créer C_α entre nœud$_1$ et nœud$_2$
        $\zeta = \zeta \cup \{C_\alpha\}$
        contraintes-créées++
tant que ($\exists$nœud$_i$/ $\nexists C_\alpha$nœud$_i \rhd C_\alpha$) **faire**
    Générer aléatoirement nœud$_j \in [1..n]$, tel que $\exists C_\beta$nœud$_j \rhd C_\beta$
    créer C_α entre nœud$_i$ et nœud$_j$
    $\zeta = \zeta \cup \{C_\alpha\}$
    nombre-de-contraintes++
Générer une solution **I**= $(x_{1_{k_1}}, ..., x_{n_{k_n}})$ du graphe
Pour chaque C_α entre nœud$_i$ et nœud$_j$ **faire**
    domaine-marqués$_\alpha = 0$
    Marquer $(x_{i_{k_i}}, x_{j_{k_j}})$ comme solution pour C_α [a]
    domaine-marqués$_\alpha$++
    tant que ((domaine-marqués$_\alpha < m^2 * p_2$) et (domaine-marqués$_\alpha \neq (m^2 - 1)$)) **faire**
        Générer aléatoirement $x_i \in [1..m]$
        Générer aléatoirement $x_j \in [1..m]$
        si $(x_i, x_j) \neq (x_{i_{k_i}}, x_{j_{k_j}})$ **alors**
            si (x_i, x_j) ne sont pas marqués incompatibles **alors**
                Marquer (x_i, x_j) comme incompatibles
                domaine-marqués$_\alpha$++
Fin /* procédure Génération CSP aléatoires*/

[a] $k_l \in [1..m], \forall l \in [1..n]$

FIG. 4.19 – *Structure de la procédure Génération CSP aléatoires*

La figure 4.21 montre les pourcentages de solutions trouvées par des graphes avec $p_2 = 0.5$, pour 50 graphes aléatoires pour chaque p_1, le nombre maximum d'itérations étant fixé à 500. Tous les graphes ont au moins une solution. Le nombre de paires de valeurs inconsistantes pour chaque contrainte est de 50 (nous avons 100 paires de valeurs possibles) pour différentes connectivités. Pour notre algorithme, les problèmes sont faciles jusqu'à une connectivité de 0.7. Les problèmes deviennent de plus en plus difficiles au fur et à mesure qu'ils sont plus connectés.

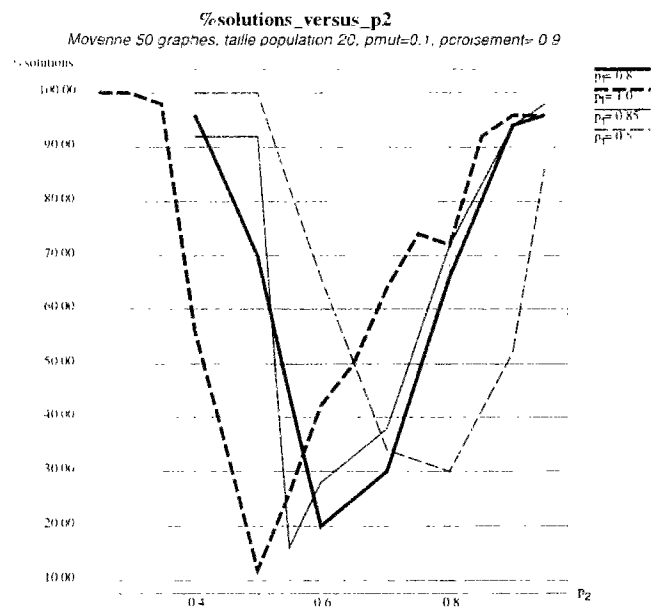

FIG. 4.20 – *%solutions trouvées en fonction de* p_2 *avec* $p_1 \in \{0.5, 0.8, 0.85, 1.0\}$

4.5 Opérateurs pour CSP n-aire

Pour faire l'extension des opérateurs présentés dans les sections précédentes aux CSP n-aires, il faut prendre en compte la réflexion suivante: *Arc-crossover* cherche à trouver la meilleure combinaison des valeurs pour chaque contrainte binaire traitée à partir des deux parents qui ont été sélectionnés aléatoirement. Nous utiliserons donc la même idée pour créer un opérateur pour CSP n-aire que nous appelons *constraint-crossover*. Cet opérateur cherchera à trouver la meilleure combinaison de valeurs

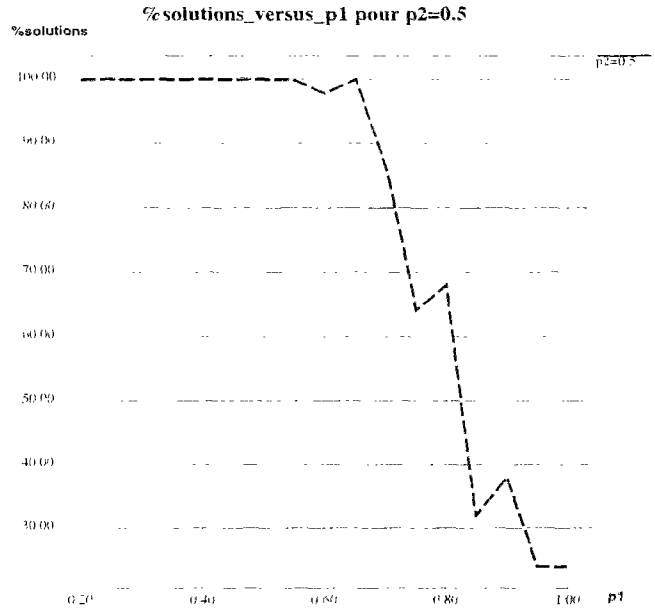

FIG. 4.21 – *%solutions trouvées en fonction de* p_1 *avec* $p_2 = 0.5$

pour les variables pertinentes de chaque contrainte en prenant chaque valeur parmi celles des deux parents. D'un autre côté, *arc-crossover* utilise aussi une priorité, qui pour le cas n-aire sera donnée par la contribution n-aire à la fonction d'évaluation (voir définition 3.5.2). En ce qui concerne *arc-mutation*, l'extension est plus simple et directe, car de la même façon nous allons chercher avec $arc_n - mutation$ à changer la valeur d'une variable qui nous permettra d'avoir plus de chance d'arriver à satisfaire toutes les contraintes du CSP.

4.5.1 $Arc_n - mutation$

Pour faire l'extension de *arc-mutation*, il suffit tout simplement d'étendre le concept de **mff** qui est calculé par arête, en faisant le même analyse, mais cette fois-ci en utilisant la fonction d'évaluation n-aire.

Définition 4.5.1 *(Fonction d'évaluation n-aire pour Mutation)*
Étant donné un CSP n-aire $P = (V, D, \zeta)$, *une instanciation* $\mathbf{I}$, *les ensembles* $\mathbf{M_j}$ *pour toute variable* $X_j \in V$, *et les fonctions* $\mathbf{e_n}(C_\alpha, \mathbf{I})$ *pour toute contrainte* C_α, *on*

définit la Fonction d'évaluation n-aire pour Mutation, $\mathbf{mff_n}$ *pour* X_j *comme:*

$$\mathbf{mff_n}(X_j, \mathbf{I}) = \sum_{C_\gamma \in \mathbf{M_j}} \mathbf{e_n}(C_\gamma, \mathbf{I}) \tag{4.11}$$

Remarque 4.5.1 *La définition de* $\mathbf{M_j}$ *(voir définition 4.2.1) n'a pas besoin d'être modifiée et elle est valable pour les CSP binaires et n-aires.*

Remarque 4.5.2 *L'algorithme* $Arc_n-mutation$ *sera le même que pour* arc-mutation *sauf pour le choix de la valeur. Ce dernier appelle la fonction d'évaluation n-aire pour mutation.*

4.5.2 Constraint-crossover

Pour l'extension d'*arc-crossover*, nous avons besoin de redéfinir le concept de priorité et les fonctions d'évaluation partielle de croisement et de mutation. De plus, l'algorithme doit être modifié, car le nombre de combinaisons possibles est plus grand que pour les CSP binaires.

Définition 4.5.2 *(Pré-ordre des Contraintes n-aires)*
Étant donné un CSP $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$, *une instanciation* $\mathbf{I}$ *et la Contribution n-aire de* C_α *à la fonction d'évaluation n-aire* $\mathbf{c_n}(C_\alpha)$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$. *On définit un Pré-ordre des Contraintes n-aires qui utilise la règle suivante:*

$$C_{k_i} \preceq C_{k_j} \text{ ssi } \mathbf{c_n}(C_{k_i}) \geq \mathbf{c_n}(C_{k_j})$$

Définition 4.5.3 *(Priorité de Contraintes n-aires)*
Étant donné un CSP $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$, *une instanciation* $\mathbf{I}$ *et la Contribution n-aire de* C_α *à la fonction d'évaluation n-aire* $\mathbf{c_n}(C_\alpha)$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$. *On définit la Priorité de Contraintes n-aires* $\mathbf{P_n}$ *comme une séquence sur un* pré-ordre des contraintes n-aires, *telle que:*

$$\mathbf{P_n} = \langle C_{k_1}, \ldots, C_{k_\eta} \rangle \text{ avec } C_{k_i} \preceq C_{k_{i+1}}, \forall i = 1, \ldots, \eta - 1$$

$\mathbf{P_n}$ est un η-uplet ordonné de contraintes. Intuitivement, nous ordonnons les contraintes suivant leur contribution à la fonction d'évaluation $\mathbf{Z_n(I)}$, suivant donc le nombre de variables impliquées dans la violation d'une contrainte.

Définition 4.5.4 *(Fonction d'évaluation Partielle n-aire pour Croisement)*
Étant donné un CSP $P = (V, D, \zeta)$, *une instanciation partielle* $\mathbf{I_p}$, *les ensembles* $\mathbf{S}_\alpha$, *et les fonctions* $\mathbf{e_n}(C_\alpha, \mathbf{I_p})$ *pour toute contrainte* C_α *complètement instanciée, on définit la* Fonction d'évaluation Partielle n-aire pour Croisement, $\mathbf{cff_n}$ *pour* C_α *comme:*

$$\mathbf{cff_n}(C_\alpha, \mathbf{I_p}) = \sum_{C_\gamma \in \mathbf{S}_\alpha} \mathbf{e_n}(C_\gamma, \mathbf{I_p}) \tag{4.12}$$

Remarque 4.5.3 *La définition de l'ensemble* S_α *(voir définition 4.2.5) n'a pas besoin de modification et elle reste valable pour les CSP n-aires.*

Remarque 4.5.4 *De la même manière que pour la définition des fonctions pour les CSP binaires nous étendons le domaine d'application des fonctions* $\mathbf{e_n}(C_\gamma, \mathbf{I})$ *à* $\mathbf{e_n}(C_\gamma, \mathbf{I_p})$

Définition 4.5.5 *(Fonction d'évaluation Partielle n-aire pour Mutation)*
Étant donné un CSP n-aire $P = (V, D, \zeta)$, *une instanciation partielle* $\mathbf{I_p}$, *les ensembles* $\mathcal{M}_\mathbf{j}(\mathbf{I_p})$ *pour toute variable instanciée sous* $\mathbf{I_p}$, *et les fonctions* $\mathbf{e_n}(C_\alpha, \mathbf{I_p})$ *pour toute contrainte* C_α *complètement instanciée, on définit la* Fonction d'évaluation Partielle n-aire pour Mutation, $\mathbf{mff_{pn}}$ *pour* X_j *comme:*

$$\mathbf{mff_{pn}}(X_j, \mathbf{I_p}) = \sum_{C_\gamma \in \mathcal{M}_\mathbf{j}(\mathbf{I_p})} \mathbf{e_n}(C_\gamma, \mathbf{I_p}) \tag{4.13}$$

Remarque 4.5.5 *La définition de* $\mathcal{M}_\mathbf{j}(\mathbf{I_p})$ *(voir définition 4.2.7) n'a pas besoin d'être modifiée et elle est valable pour les CSP binaires et n-aires.*

La procédure de *constraint-crossover* est montrée sur la figure 4.22.

Elle utilise la même idée qu'*arc-crossover*. Avec les deux parents choisis aléatoirement, elle analyse les contraintes suivant l'ordre donné par $\mathbf{P_n}$.

Si aucune variable n'est instanciée, le fils hérite soit:

- Les valeurs d'un des parents au cas où la contrainte est satisfaite par au moins un parmi eux.

Procédure constraint-crossover $(Parent_1, Parent_2)$
Début
Pour chaque C_α suivant l'ordre donné par $\mathbf{P_n}$
Analyser C_α du $Parent_1$ et du $Parent_2$
si $((\exists X_{\alpha_i} \rhd C_\alpha \text{et} X_{\alpha_i} \dashv {}^{a}I_p)$ et $(nI_\alpha{}^{b} \geq 2))$
  si $((C_\alpha \models (Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) \cup \mathbf{I_p}))$ et
    $(C_\alpha \models (Parent_2(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) \cup \mathbf{I_p})))$ **alors**
    si $(\mathbf{Z}(Parent_2) > \mathbf{Z}(Parent_1))$ **alors**
      $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}})$
    sinon
      si $(\mathbf{Z}(Parent_1) > \mathbf{Z}(Parent_2))$ **alors**
        $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = Parent_2(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}})$
      sinon
        $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = random(Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}),$
        $Parent_2(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}))$
  sinon
    si $((C_\alpha \models (Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) \cup \mathbf{I_p}))$ ou
      $(C_\alpha \models (Parent_2(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) \cup \mathbf{I_p})))$ **alors**
      si $(C_\alpha \models (Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) \cup \mathbf{I_p}))$ **alors**
        $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = Parent_1(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}})$
      sinon
        $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = Parent_2(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}})$
    sinon[c]
    $\mathbf{I_p}(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = argmin_{s_1 \in S_1}(\mathbf{cff_n}(C_\alpha, (\mathbf{I_p} \cup comb_i, \forall i = 1, \ldots, 2^{nI_\alpha - 2})))$
sinon
  si $(X_{\alpha_i} \dashv \mathbf{I_p})$ **alors**
    $\mathbf{I_p}(X_{\alpha_i}) = argmin_{s_2 \in S_2}(\mathbf{mff_{pn}}(X_{\alpha_i}, (\mathbf{I_p} \cup x_{\alpha_{i_1}})), \mathbf{mff_{pn}}(X_{\alpha_i}, (\mathbf{I_p} \cup x_{\alpha_{i_2}})))$
Fin/* constraint-crossover */

---

[a] non-instanciée
[b] nI_α = nombre de variables de C_α non-instanciées
[c] $comb_i(X_{\alpha_1}, \ldots, X_{\alpha_{nI_\alpha}}) = (x_{\alpha_{1_{k_1}}}, \ldots, x_{\alpha_{i_{k_j}}}, \ldots), k_j \in [1,2] et \nexists k_1 = k_2 \ldots = k_{nI_\alpha}$.
$argmin_{s \in S}\{a_s\}$ donne l'ensemble s^* tel que $a_{s^*} \leq a_s, \forall s \in S$.
$S_1 = \{comb_1, \ldots, comb_{2^{nI_\alpha - 2}}\}, S_2 = \{x_{\alpha_{i_1}}, x_{\alpha_{i_2}}\}$

FIG. 4.22 – *Structure de la procédure constraint-crossover*

- Une combinaison de valeurs choisie parmi les $2^{a_c} - 2$ combinaisons possibles selon la valeur de $\mathbf{cff}_n$. a_c est l'arité de C_c, c'est donc le nombre de variables à instancier. Nous avons deux choix pour chaque variable (un à partir de chaque parent) et on élimine les deux combinaisons actuelles du $parent_1$ et du $parent_2$.

Dans le cas où il manque au moins deux variables de la contrainte à instancier, le fils hérite soit:

- Les valeurs d'un des parents au cas où la contrainte est satisfaite en prenant les valeurs qui manquent à partir d'un parmi eux.
- Une combinaison de valeurs choisie parmi les $2^{nI_c} - 2$ combinaisons possibles. nI_c est le nombre de variables pertinentes de C_c qui ne sont pas encore instanciées dans le fils. Nous avons deux choix pour chaque variable manquante (un à partir de chaque parent) et on élimine les deux combinaisons actuelles du $parent_1$ et du $parent_2$.

Dans le cas où une seule variable n'est pas instanciée, le choix est fait avec $arc_n - mutation$ en considérant les valeurs des deux parents, quand aucune de leurs valeurs ne permet de satisfaire la contrainte.

4.6 Conclusion du chapitre

Il y a des sujets qui sont d'un intérêt constant pour la communauté des contraintes, et qui ont été étudiés pour la résolution des CSP. Nous avons voulu incorporer certains d'entre eux dans notre approche. Le premier est le concept de *structure*: il a été considéré dans la définition des fonctions d'évaluation et dans les définitions des fonctions d'évaluation partielles.
Un autre concept important est la *décomposition* en sous-graphes ou sous-problèmes. L'idée de base est de partitionner un graphe de contraintes en sous-graphes et ensuite résoudre chaque sous-problème séparément. Une fois que chacun d'eux a été résolu, on cherche à construire avec les sous-solutions une solution globale, [Dec90]. Nous avons

utilisé cette idée dans la conception d'*arc-crossover* et de *constraint-crossover*. Si un parent satisfait un sous-graphe, et l'autre parent satisfait les autres contraintes dans le graphe, nous souhaitons, en utilisant *arc-crossover*, construire un fils qui satisfasse toutes les contraintes du graphe, qui soit donc une solution globale au problème.
La minimisation du *nombre de vérification de contraintes* est le but de tout algorithme qui résout un CSP d'une façon systématique. Nous avons utilisé ce concept dans le calcul de la *fonction d'évaluation pour mutation* et des fonctions d'évaluation partielles. Si *arc-mutation* change la valeur d'une variable, la *fonction d'évaluation pour mutation* est calculée en considérant seulement les variables liées à cette variable par une contrainte. De la même manière avec *arc-crossover*, la *fonction d'évaluation partielle pour croisement* est calculée en analysant seulement les arêtes qui partagent une variable avec la contrainte courante et la *fonction d'évaluation partielle pour mutation* est évaluée en considérant seulement les contraintes liées à la variable et qui sont complètement instanciées.

Dans le chapitre suivant, nous allons introduire le concept d'adaptation qui a été traité par Schwefel en [Sch81] et qui a été récemment repris comme sujet d'intérêt dans la communauté des algorithmes génétiques. Nous l'incorporons dans la conception d'un opérateur de croisement qui utilise comme base l'idée développée dans ce chapitre pour *arc-crossover*.

Les travaux exposés dans ce chapitre ont été présentés en [Rif97a].