Chapitre 5

Opérateur Dynamique Adaptatif

5.1 Motivation

Nous nous sommes tournée vers d'autres concepts en évolution artificielle pour arriver à améliorer les résultats déjà prometteurs de nos opérateurs. Nous avons défini dans le chapitre précédent l'opérateur *arc-crossover* qui prend en compte le réseau de contraintes pour combiner les valeurs des parents d'une manière plus appropriée pour créer un fils. Il est basé sur une priorité statique des contraintes, qui est définie au début de l'algorithme suivant la connectivité. *Arc-crossover* nous semble être un bon opérateur car il nous a permis de modifier la vision d'une recherche évolutionniste aveugle par rapport au problème. tout en gardant un degré de généralité pour pouvoir être appliqué à différents types de CSP et ainsi de profiter de certaines caractéristiques du réseau de contraintes. Nous souhaitons améliorer sa performance en incorporant les idées de l'adaptation. Ce concept a été récemment repris et défini d'une manière plus formelle en [HME97]. Il donne à l'algorithme génétique plus de souplesse en lui permettant l'adaptation et le changement de sa configuration, suivant l'état courant de la recherche. Dans le contexte des CSP, ceci signifie qu'on pourrait éventuellement permettre à l'algorithme la non-satisfaction temporaire de certaines contraintes. L'algorithme pourrait commencer son évolution avec un sous ensemble de contraintes, composé par celles qui sont les plus difficiles à satisfaire. Une fois que l'algorithme trouve des valeurs pour les variables satisfaisant ces contraintes, il incorpore le reste

de contraintes et continue sa recherche pour trouver une solution pour le problème global. Une autre idée serait de pouvoir changer la priorité des contraintes dans l'analyse. Par exemple, en rendant comme prioritaire la contrainte qui a été historiquement (pendant la recherche de l'algorithme), la plus difficile à satisfaire [Eib96].

Nous commençons ce chapitre par un résumé des concepts qu'on utilise en adaptation. Ensuite, nous définissons un nouvel opérateur qui est né à partir d'*arc-crossover* mais qui a inclus certaines idées d'adaptation. A la fin du chapitre, nous comparons les résultats d'un algorithme évolutionniste pour le problème de 3-coloriage, avec ceux d'autres algorithmes qui utilisent d'autres opérateurs. Nous montrons aussi les résultats obtenus pour le 3-coloriage avec des graphes peu denses. Nous concluons alors ce chapitre par un ensemble de tests réalisés avec des CSP aléatoires.

5.2 Adaptation

L'adaptation de paramètres et d'opérateurs est un des sujets les plus prometteurs en évolution artificielle. L'idée est de faire une sorte de "syntonisation" de l'algorithme avec le problème pendant sa résolution. Hinterding et al. [HME97] proposent le classement suivant pour les types d'adaptation:

Définition 5.2.1 *(Adaptation Statique)*
On dit que l'algorithme réalise une adaptation statique *si ses paramètres stratégiques ont une valeur constante pendant son exécution.*

En conséquence, on a besoin d'un agent ou d'un mécanisme externe pour syntoniser les paramètres et sélectionner les valeurs les plus appropriées. Un cas typique est lorsqu'on fait tourner l'algorithme plusieurs fois, en essayant de trouver les valeurs des paramètres qui le rendent plus performant. Ce cas correspond justement à notre algorithme de sélection (voir 3.12), pour lequel nous avons trouvé les valeurs de "syntonisation" pour α et β.

Définition 5.2.2 *(Adaptation Dynamique)*
On dit que l'algorithme réalise une adaptation dynamique *s'il existe un mécanisme qui modifie un paramètre stratégique au cours de l'exécution, sans un contrôle externe.*

Dans notre algorithme proposé dans le chapitre 3, l'opérateur de permutation (voir 3.20) est activé selon les conditions actuelles de l'évolution. Dans cette catégorie, on peut encore faire le sous-classement suivant le mécanisme d'adaptation utilisé.

Définition 5.2.3 *(Adaptation Dynamique - Déterministe)*
Un algorithme réalise une adaptation dynamique déterministe *si la modification obéit à une règle pré-établie, indépendante du déroulement de l'algorithme.*

Ceci signifie que l'adaptation est faite sans utiliser aucune information provenant du déroulement de l'algorithme. Un exemple dans cette catégorie est l'altération de la probabilité de mutation conditionnée au nombre de générations, par exemple:

$$p_m = 0.5 - 0.3\frac{g}{G} \tag{5.1}$$

où g est le nombre actuel de générations et G le nombre maximum de générations. L'algorithme changera la probabilité de mutation, en comptant le nombre de générations, mais sans regarder si la recherche suit le bon chemin, sans détecter non plus un besoin réel du changement. D'autres exemples dans cette catégorie concernent le changement de la fonction d'évaluation. Dans le cas spécifique des COP (Constraint Optimization Problems), le changement affecte les pénalités pour les contraintes qui ne sont pas encore satisfaites. Pour les CSP, Eiben et al. ont proposé en [ERR96] une augmentation des pénalités pour les contraintes violées après chaque exécution complète de l'algorithme. L'idée est d'exécuter l'algorithme plusieurs fois. Après chaque exécution, la procédure d'adaptation modifie les poids de la fonction, suivant les contraintes qui n'ont pas pu être satisfaites pendant l'exécution actuelle. Ensuite, l'algorithme génétique utilisant la fonction adaptée est re-démarré. Leur mécanisme d'adaptation est appelé *Off-Line weight adaptation*, il est montré sur la figure 5.1.

Définition 5.2.4 *(Adaptation Dynamique - Adaptative)*
On dit qu'un algorithme réalise une adaptation dynamique adaptative *s'il existe une*

```
Procédure Off-Line weight adaptation
Début
Appliquer des poids initiaux à la fonction d'évaluation f
Pour x test de l'algorithme génétique faire
     exécuter l'algorithme génétique avec f
     redéfinir f après la fin de l'algorithme génétique
Fin /* procédure Off-Line weight adaption */
```

FIG. 5.1 – *Structure de la procédure* Off-Line weight adaptation

sorte de "feedback" de l'algorithme, qui est utilisé pour déterminer les sens et/ou grandeur du changement des paramètres stratégiques.

Dans cette catégorie, pour les CSP, Eiben et al. [EvdH97] utilisent aussi une stratégie pour le changement de pénalités dans la fonction d'évaluation. La procédure est appelée *On-Line weight adaptation*. Cette fois-ci, l'augmentation des pénalités est effectuée en suivant le paramètre T_p, qui indique le nombre maximum de générations à faire avec une fonction objectif. La procédure est montrée sur la figure 5.2

```
Procédure On-Line weight adaptation
Début
Appliquer des poids initiaux à la fonction d'évaluation f
tant que non fin faire
        Pour les T_p évaluations suivantes de la fonction faire
            utiliser f dans l'algorithme génétique
         Redéfinir f et ré-évaluer les individus
Fin /* procédure On-Line weight adaptation */
```

FIG. 5.2 – *Structure de la procédure* On-Line weight adaptation

Définition 5.2.5 *(Adaptation Dynamique - Auto-Adaptative)*
On dit qu'un algorithme réalise une adaptation dynamique auto-adaptative *si les paramètres qui sont adaptés se trouvent codés dans le chromosome et sont affectés par la mutation et la recombinaison.*

Ces paramètres codés n'affectent pas l'évaluation des individus, si ce n'est que les meilleures valeurs produiront de meilleurs individus qui auront plus de chances de survivre, de produire des enfants et de propager les meilleures valeurs des paramètres.

Il existe aussi différents niveaux d'adaptation:

1. Une adaptation peut-être au niveau de l'environnement, comme par exemple le changement de pénalités pour les contraintes violées.

2. Une adaptation est effectuée au niveau de la population quand le changement affecte la population entière, c'est par exemple ce que fait notre opérateur de permutation.

3. Une adaptation est au niveau de l'individu, quand le changement affecte seulement l'individu, par exemple une adaptation des points de croisement.

4. Le dernier niveau est pour une adaptation au niveau du composant, qui change le paramètre pour un composant ou un gène particulier d'un individu.

5.3 CDA: Crossover Dynamique Adaptatif

L'opérateur *Crossover Dynamique Adaptatif* (CDA) est basé sur l'idée d'*arc-crossover*, c'est-à-dire, avec deux parents produire un fils qui hérite la meilleure combinaison des valeurs de leurs variables. Mais, dans *CDA*, nous incorporons les idées d'adaptation, en lui permettant de changer la priorité d'analyse des contraintes dans le croisement, suivant les caractéristiques des parents. Cela veut dire que les positions de recombinaison ne sont pas fixées et l'ordre de l'analyse des contraintes non plus. La figure 5.3 montre la conséquence d'une analyse suivant une priorité des contraintes différente de celle qu'utilise *arc-crossover*. Dans l'exemple, les deux parents ne violent que la contrainte C_4. *Arc-crossover* définit une priorité $\mathbf{P}$ suivant la contribution de chaque contrainte à la fonction d'évaluation. Prenons une autre priorité $\mathbf{P_{ca}}$ qui considère comme la contrainte la plus prioritaire celle qui est violée (C_4). Ensuite, nous réalisons la procédure de *arc-crossover* séparément avec les deux priorités, nous voyons qu'avec $\mathbf{P_{ca}}$ nous trouvons une solution, alors qu'avec $\mathbf{P}$ le fils peut être une copie du $parent_1$.

Nous avons construit $\mathbf{P_{ca}}$ en sachant que la contrainte C_4 est violée par les deux parents, en prenant donc en compte l'information des parents. C'est cette idée que

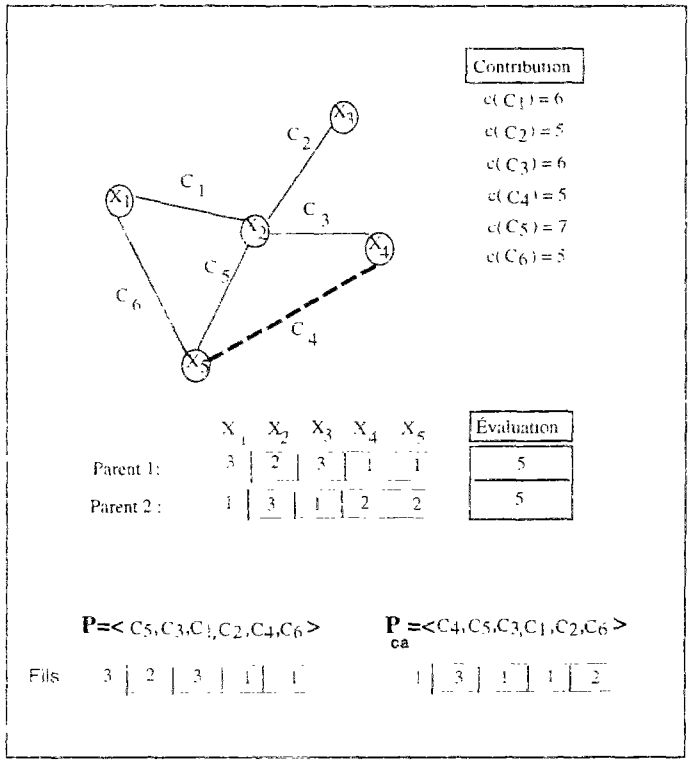

FIG. 5.3 – *Exemple: Différentes Priorités pour Crossover*

nous allons incorporer dans le nouvel opérateur. Pour ce faire, nous avons besoin des définitions suivantes:

Définition 5.3.1 *(Nombre de violations)*
Étant donné un CSP $P = (V, D, \zeta)$, *deux instanciations* $\mathbf{I_1}$ *et* $\mathbf{I_2}$, *et les fonctions* $\mathbf{e}(C_\alpha.\mathbf{I})$ *pour* $\mathbf{I}$= $\mathbf{I_1}$, $\mathbf{I_2}$ *pour toute contrainte* C_α. *On définit le* nombre de violations communes *pour la contrainte* C_α, $\mathbf{nv}(C_\alpha, \mathbf{I_1}, \mathbf{I_2})$ *comme:*

$$\mathbf{nv}(C_\alpha, \mathbf{I_1}, \mathbf{I_2}) = \begin{cases} 0 & \mathbf{e}(C_\alpha, \mathbf{I_1}) = \mathbf{e}(C_\alpha, \mathbf{I_2}) = 0 \\ 1 & \textit{soit } \mathbf{e}(C_\alpha, \mathbf{I_1}) \neq 0 \textit{ ou } \mathbf{e}(C_\alpha, \mathbf{I_2}) \neq 0 \\ 2 & \mathbf{e}(C_\alpha, \mathbf{I_1}) \neq 0, \textit{ et } \mathbf{e}(C_\alpha, \mathbf{I_2}) \neq 0 \end{cases}$$

$\mathbf{I_1}$ et $\mathbf{I_2}$ correspondent respectivement aux instanciations du $parent_1$ et du $parent_2$. Pour chaque contrainte C_α, le nombre de violations sera zéro si les deux parents satisfont la contrainte. Il vaut un dans le cas où un parmi eux satisfait la contrainte

et deux quand la contrainte est violée par les deux parents. En utilisant cette fonction, nous allons définir une nouvelle priorité qui prend en compte l'état actuel des parents.

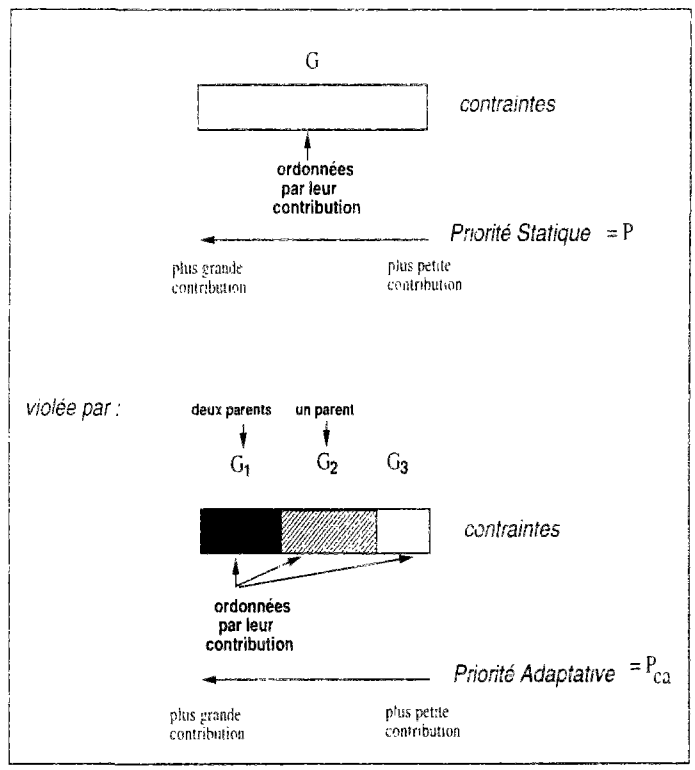

FIG. 5.4 - *Exemple: Priorité Statique et Adaptative pour Crossover*

Définition 5.3.2 *(Pré-ordre Dynamique des Contraintes)*
Étant donné un CSP binaire $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$, *deux instanciations* $\mathbf{I_1}$ *et* $\mathbf{I_2}$ *et la Contribution de* C_α *à la fonction d'évaluation* $\mathbf{c}(C_\alpha)$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$ *(def 3.2.3). On définit un Pré-ordre Dynamique des Contraintes qui utilise la règle suivante:*

$$C_{k_i} \preceq C_{k_j} \text{ si:}$$

- $\mathbf{nv}(C_{k_i}, I_1, I_2)$ $\mathbf{nv}(C_{k_j}, I_1, I_2)$ *ou*
- $\mathbf{nv}(C_{k_i}, I_1, I_2) = \mathbf{nv}(C_{k_j}, I_1, I_2)$ *et* $\mathbf{c}(C_{k_i}) \geq \mathbf{c}(C_{k_j})$

Nous ajoutons à la définition du pré-ordre des contraintes (voir def: 4.2.3), la condition du nombre de violations. Cela veut dire que nous répartissons les contraintes dans trois groupes suivant le *nombre de violations* et qu'une contrainte qui est violée par les deux parents sera la première dans l'analyse. Dans le cas où il y en a plusieurs qui ne sont pas satisfaites par les deux parents, elles seront classées en priorité suivant leur *contribution à la fonction d'évaluation*. Sur la figure 5.4, nous illustrons la différence entre la priorité statique et la priorité dynamique.

En résumé, nous aurons trois groupes des contraintes: G_1 (le groupe prioritaire) composé par les contraintes qui sont violées par les deux parents, le groupe G_2 composé par les contraintes qui ne sont violées que par un parent, et G_3 composé par les contraintes satisfaites par les deux parents. A l'intérieur de chaque groupe les contraintes seront ordonnées suivant leur contribution à la fonction d'évaluation. Avec la priorité statique, nous n'avons qu'un seul groupe G, qui est ordonné suivant la contribution de chaque contrainte à la fonction d'évaluation.

Définition 5.3.3 *(Priorité Adaptative de Contraintes)*
Étant donné un CSP binaire $P = (V, D, \zeta)$ avec une matrice de contraintes $\mathbf{R}$, *deux instanciations* $\mathbf{I_1}$ *et* $\mathbf{I_2}$ *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 Adaptative de Contraintes* $\mathbf{P_{ca}(I_1, I_2)}$ *comme une séquence sur un pré-ordre dynamique des contraintes telle que:*

$$\mathbf{P_{ca}(I_1, I_2)} = \langle C_{k_1}, \ldots, C_{k_\eta} \rangle \text{ avec } C_{k_i} \preceq C_{k_{i+1}}, \forall i = 1, \ldots, \eta - 1$$

$\mathbf{P_{ca}(I_1, I_2)}$ est un η-uplet ordonné de contraintes. Elle est fonction des instanciations I_1 et I_2. Intuitivement, nous ordonnons les contraintes suivant leur contribution à la fonction d'évaluation des parents.

Le nouvel opérateur se trouve dans le procédure de transformation de l'algorithme évolutionniste, comme cela est illustré sur la figure 5.5

La structure de la procédure de *Crossover Dynamique Adaptatif* est montrée sur la figure 5.6.

Procédure Crossover Dynamique Adaptatif $(Parent_1, Parent_2)$
Début
Pour chaque C_α suivant l'ordre donné par $\mathbf{P_{ca}}(\mathbf{Parent_1}, \mathbf{Parent_2})$
Analyser C_α (x_i, x_j) du $Parent_1$ et du $Parent_2$
si $((X_i \dashv \mathbf{I_p})^a$ et $(X_j \dashv \mathbf{I_p}))$ alors
  si $(\mathbf{nv}(C_\alpha, \mathbf{Parent_1}, \mathbf{Parent_2}) = 0)$ alors
    si $(\mathbf{Z}(Parent_2) > \mathbf{Z}(Parent_1))$ alors
      $\mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
    sinon
      si $(\mathbf{Z}(Parent_1) > \mathbf{Z}(Parent_2))$ alors
        $\mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
      sinon
        $\mathbf{I_p}(X_i, X_j) = random((x_{i_1}, x_{j_1}), (x_{i_2}, x_{j_2}))$
  sinon
    si $(\mathbf{nv}(C_\alpha, \mathbf{Parent_1}, \mathbf{Parent_2}) = 1)$ alors
      si $(C_\alpha \models Parent_1)$ alors
        $\mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
      sinon $\mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
    sinon
      $\mathbf{I_p}(X_i, X_j) = argmin_{s_1 \in S_1}(\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_1}, x_{j_2}))),$
        $\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_2}, x_{j_1}))))^b$
sinon
  si $((X_i \dashv \mathbf{I_p})$ ou $(X_j \dashv \mathbf{I_p}))$ alors
    si $(X_i \dashv \mathbf{I_p})$ alors k=i sinon k=j
  $\mathbf{I_p}(X_k) = argmin_{s_2 \in S_2}(\mathbf{mff_p}(X_k, (\mathbf{I_p} \cup x_{k_1})), \mathbf{mff_p}(X_k, (\mathbf{I_p} \cup x_{k_2})))^c$
Fin/* crossover dynamique adaptatif */

[a] X_i non-instanciée dans $\mathbf{I_p}$
[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. 5.6 – *Structure de la procédure Crossover Dynamique Adaptatif*

Algorithme	Opérateurs
A	UAX, mutation
B	*arc-crossover*, *arc-mutation*
C	(#,r,b), mutation
D	*CDA*, *arc-mutation*

TAB. 5.1 – *Quatre algorithmes qui diffèrent par leurs opérateurs*

chaque connectivité. Le nombre moyen de générations nécessaires pour trouver une solution pour l'*Algorithme D* est inférieur à celui pour les autres algorithmes. De plus, son comportement est plus uniforme. Pour l'*Algorithme C*, les problèmes avec la connectivité 4.7 sont assez difficiles, avec les arc-opérateurs et CDA nous avons trouvé de meilleurs résultats. Sur la figure 5.8, on montre le pourcentage de solutions trouvées par les quatre algorithmes. Les nouveaux opérateurs ont en moyenne de meilleurs résultats. L'*Algorithme A* a trouvé dans le pire des cas 15% de solutions, en revanche l'*Algorithme B* a trouvé pour le pire des cas 83% de solutions, l'*Algorithme C* 70% et finalement l'*Algorithme D* 97%. La plus importante amélioration en utilisant le nouvel opérateur se trouve pour des problèmes avec une connectivité entre [4.5, 5.3]. Il est connu que cela correspond à la région où nous espérons trouver les problèmes de 3-coloriage les plus difficiles, [CKT91].

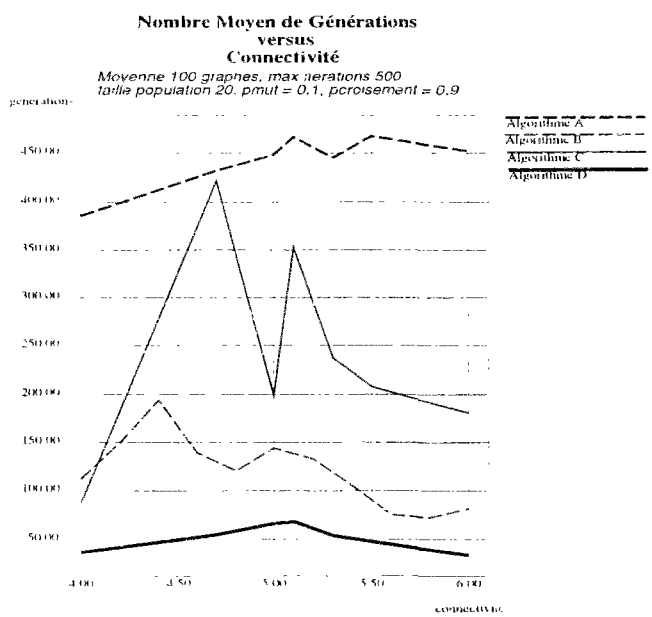

FIG. 5.7 – *Comparaison: Nombre moyen de générations par les Algorithmes A, B, C et D pour différentes connectivités*

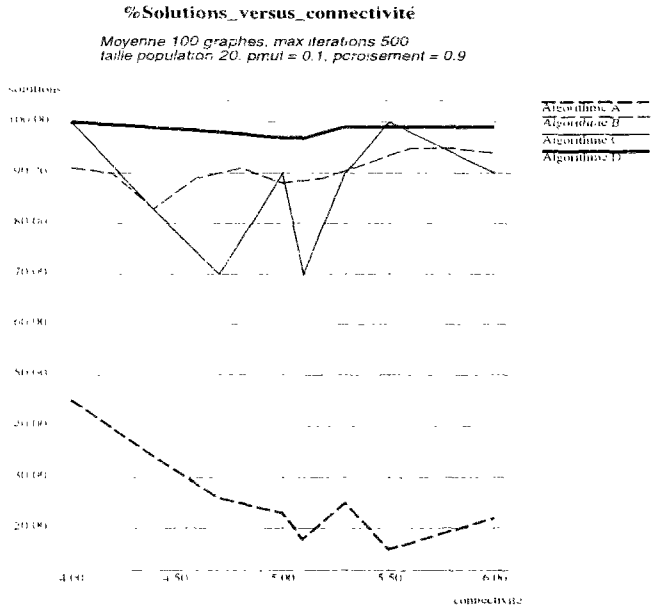

FIG. 5.8 – *Comparaison: Pourcentage de solutions trouvées par les Algorithmes A, B, C et D pour différentes connectivités*

5.4.2 Tests: graphes peu denses

Les graphes peu denses sont d'un intérêt particulier pour la communauté en contraintes. Cheeseman et al. [CKT91], en analysant la connectivité du graphe de contraintes par rapport à la difficulté de le résoudre, ont trouvé que quand la connectivité augmente il apparaît une "phase de transition" où les problèmes de 3-coloriage deviennent plus difficiles. En d'autres termes, cela veut dire qu'un problème sera facile s'il est sur restreint ou sous contraint. Les problèmes difficiles se trouvent donc dans la frontière entre sur et sous restreint et nos graphes peu denses se trouvent dans cette région. Notre but maintenant est d'analyser le comportement de notre algorithme évolutionniste avec le nouvel opérateur pour ce type de problèmes. Pour cela, nous avons comparé trois algorithmes. L'*Algorithme A* utilise l'opérateur spécifiquement adapté pour le problème de coloriage de graphe *knowledge-crossover* (voir sous-section 2.5.2). L'*Algorithme B* utilise *arc-crossover* et *arc-mutation* et l'*Algorithme C* utilise *CDA* et *arc-mutation*. Nous avons généré des graphes peu denses. Par rapport aux graphes testés dans la section précédente, les graphes peu denses que nous considérons dans

Algorithme	Opérateurs
A	*knowledge-crossover*, mutation
B	*arc-crossover*, *arc-mutation*
C	*CDA*, *arc-mutation*

TAB. 5.2 – *Trois algorithmes qui diffèrent par leurs opérateurs pour des graphes peu denses*

cette section ont tous une connectivité égale à 4, car ils suivent la relation $\eta = 2n$. Pour chaque η, nous avons 100 graphes différents. Nous avons fixé le nombre maximum d'itérations à 500. Les résultats sont montrés sur la figure 5.9 et la figure 5.10. Avec l'*Algorithme C*, nous avons trouvé les meilleurs résultats, l'algorithme a été capable de trouver la solution pour plus de 80% des graphes peu denses. L'*Algorithme B* dans le pire des cas a trouvé 68% de solutions, et l'*Algorithme A* pour le pire des cas a trouvé 53% de solutions. Pour tous, quand le nombre de contraintes augmente, le problème devient de plus en plus difficile à résoudre. Il est intéressant de remarquer que Minton et al. en utilisant un algorithme qui utilise leur heuristique *min-conflits* (voir définition 3.3.2) avec 100 differentes pré-solutions solutions générées avec l'heuristique de Brelaz et une maximum de 270 réparations ont obtenu une probabilité de trouver une solution pour les graphes peu denses avec 30 variables de 0.3. En revanche, avec l'*Algorithme A*, nous avons obtenu 65% de solutions, l'*Algorithme B* 93% et l'*Algorithme C* 99%.

5.5 Ensemble de problèmes pour CDA: CSP aléatoires avec solution

Notre intérêt pour la génération des graphes aléatoires est d'analyser le comportement de notre algorithme pour la résolution de graphes avec différentes connectivités et différent degrés de difficulté. Pour cela, nous avons utilisé la procédure décrite dans le Chapitre 4. Tous les graphes ont au moins une solution. Les graphes sont du même type que en [Smi95] pour n=8 variables et leur taille du domaine égal à 10 (m=10), Pour chaque p_2, nous avons généré 50 graphes et le nombre maximum d'itérations est

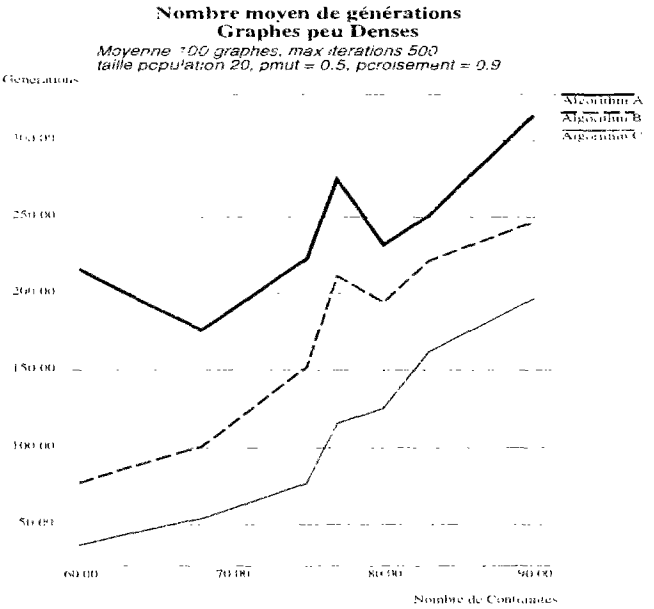

FIG. 5.9 - *Comparaison: Nombre moyen de générations par les Algorithmes A, B et C par nombre de contraintes*

égal à 500. La figure 5.12 montre le pourcentage de solutions trouvées par rapport à p_2 pour des valeurs de p_1 entre $\{0.5, 0.8, 0.85, 1.0\}$. Comme avec les arc-opérateurs dans le chapitre précédent, il y a des problèmes qui sont plus difficiles à résoudre pour notre algorithme. Plus précisément, le cas le plus difficile est avec des graphes qui sont complètement connectés et avec $p_2 = 0.5$. Ce cas correspond aux problèmes qui ont plus de choix locaux qui satisfont partiellement les contraintes. Le point minimal de chaque courbe sur la figure correspond au point où le nombre de solutions espérées est égal à un, par exemple pour $p_1 = 1$ et $p_2 = 0.5$, nous espérons n'avoir qu'une seule solution, et à partir de ce point les problèmes n'ont tous qu'une solution. Mais au fur et à mesure qu'on s'approche de $p_2 = 1$, le nombre de combinaisons de valeurs possibles des variables qui satisfont localement les contraintes diminue. En conséquence, en s'approchant de $p_2 = 1$, une solution satisfaisante localement le sera aussi globalement, donc le problème est facile à résoudre. Nous remarquons que l'algorithme est amélioré avec le nouvel opérateur, par rapport aux résultats présentés dans le chapitre antérieur. Alors, la question intéressante est: est-ce que tous les problèmes avec $p_2 = 0.5$ sont difficiles pour notre algorithme? La réponse est non,

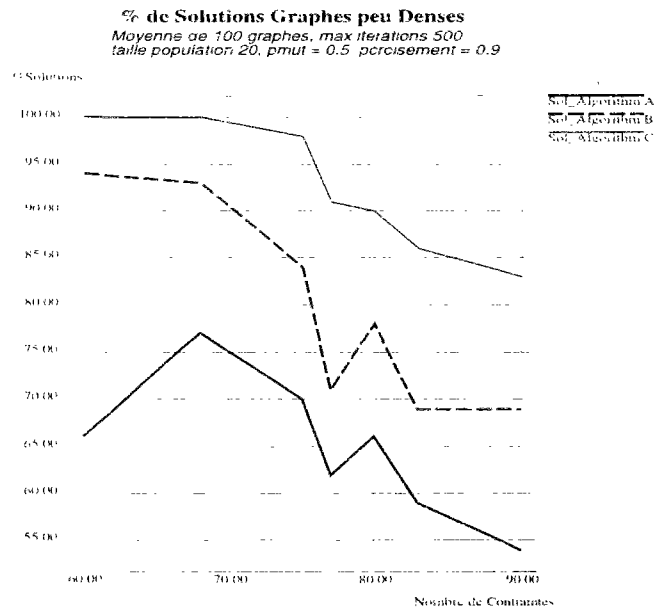

FIG. 5.10 – *Comparaison: Pourcentage de solutions trouvées par les Algorithmes A, B et C par nombre de contraintes*

comme cela est illustré sur la figure 5.12. Pour p_1 entre $\{0.4..0.7\}$, nous trouvons que les problèmes sont faciles à résoudre. Cela s'explique en analysant la figure 5.13. Elle montre le nombre de solutions espérées pour des graphes avec p_2=0.5 pour différentes valeurs de p_1. Pour $p_1 = 0.75$ le nombre espéré de solutions est presque 50. Ensuite, avec la descente exponentielle, pour $p_1 = 0.8$ le nombre de solutions espéré est de 16. Au fur et à mesure qu'on s'approche de $p_1 = 1$, l'algorithme a plus de possibilités de choisir des valeurs des variables qui satisfont localement une contrainte, mais qui ne font pas partie d'une solution globale du problème, pour un même nombre fixe de valeurs consistantes par contrainte (qui ne dépend que de p_2).

5.6 Relation entre l'adaptation et les opérateurs

Nous avons affirmé dans la première section de ce chapitre qu'il existe différents types et niveaux d'adaptation. Nous voulons classer des opérateurs selon leur type d'adaptation. Cela est montré sur le tableau 5.3. Notre opérateur de *Permutation* est classé comme étant dynamique adaptatif, car il reçoit l'information de l'algorithme

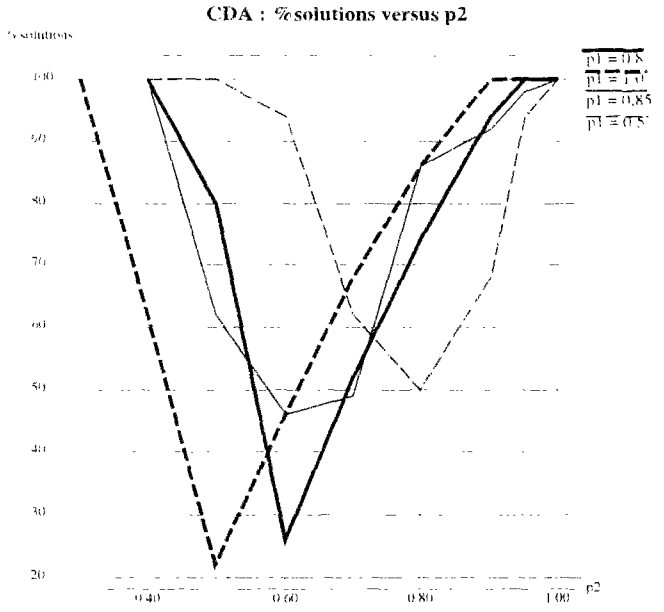

FIG. 5.11 - *CDA: %solutions trouvées en fonction de* p_2 *avec* $p_1 \in \{0.5, 0.8, 0.85, 1.0\}$

d'une détection de *stabilité* pendant la recherche, il est donc activé en conséquence. Son application concerne tous les chromosomes de la population. $(\#, r, b)$ est classé comme statique, car comme pour *croisement à un point*, *croisement à points multiples* et pour *mutation* standard ses paramètres sont fixés au début et ils se maintiennent constants pendant l'exécution. Son application affecte un individu. *Uniform Adaptive Crossover* est dynamique auto-adaptatif, chaque individu a une information additionnelle codée dans le chromosome qui va permettre à UAX de changer les points de croisement. *Arc-crossover* est dynamique déterministe, car la priorité des contraintes est fixée au départ de l'algorithme et il la garde pendant l'exécution, en conséquence les positions de croisement sont fixes. Par contre, les valeurs dans les points de croisement dépendent de la satisfaction ou violation des contraintes, du déroulement donc de l'algorithme. *Knowledge-crossover* et *CDA* se trouvent dans la catégorie de dynamique adaptatif, car les deux reçoivent l'information de la satisfaction des contraintes et avec cela ils déterminent les positions du croisement et leurs valeurs. Les deux génèrent un fils à partir de deux parents. La différence fondamentale entre ces deux opérateurs est que *Knowledge-crossover* réalise la construction d'un fils en analysant

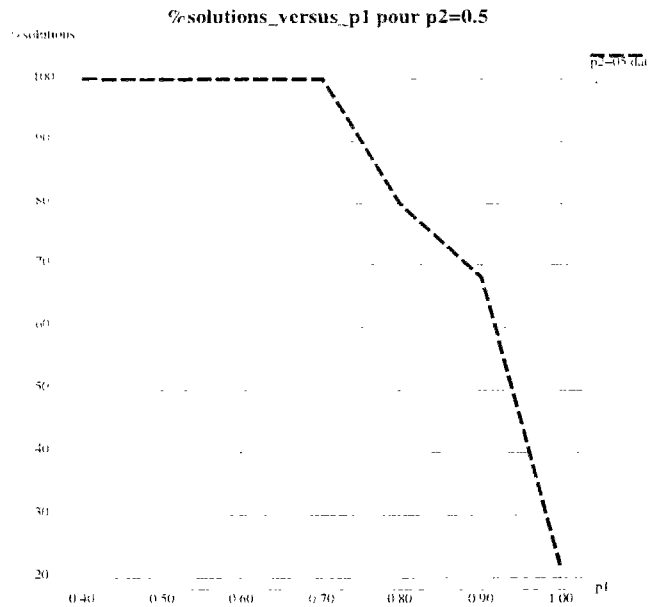

FIG. 5.12 – *CDA: %solutions trouvees en fonction de* p_1 *avec* $p_2 = 0.5$

Opérateur	Type d'adaptation	Niveau d'Adaptation
Permutation	Dynamique Adaptatif	Population
$(\#, r, b)$	Statique	Individu
UAX	Dynamique Auto-adaptatif	Individu
Knowledge-crossover	Dynamique Adaptatif	Individu
Arc-crossover	Dynamique Déterministe	Individu
CDA	Dynamique Adaptatif	Individu

TAB. 5.3 – *Comparaison du type d'adaptation pour différents opérateurs*

les variables (nœuds dans le graphe de contraintes), par contre *CDA* fait son analyse par contrainte (arêtes du graphe de contraintes). De plus, *Knowledge-crossover* est conçu spécifiquement pour le problème de 3-coloriage.

5.7 Conclusion du chapitre

Nous avons montré dans ce chapitre que le concept d'adaptation peut représenter une contribution importante dans la conception d'un algorithme évolutionniste, pour résoudre les problèmes de satisfaction de contraintes. Nous avons mentionné

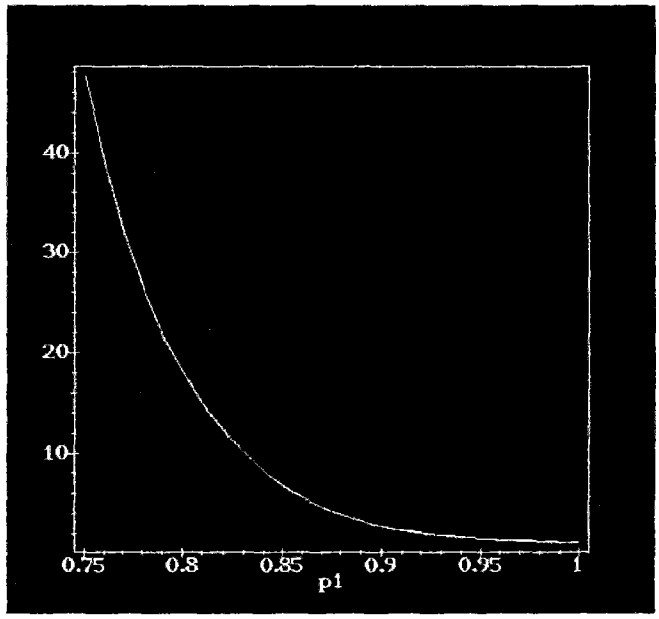

FIG. 5.13 – *Nombre de solutions espérées en fonction de* p_1 *avec* $p_2 = 0.5$

les différentes approches qui essayent d'utiliser le concept d'adaptation pour guider la recherche vers des parties de l'espace plus prometteuses pour la satisfaction des contraintes.

Nous avons introduit un nouvel opérateur pour le croisement qui détermine dynamiquement les positions de la recombinaison, en utilisant l'information des parents en termes de leur satisfaction de contraintes. Cette méthode est plus offensive que celle qui est implémentée dans *arc-crossover*, car ici nous forçons l'algorithme à faire en priorité un croisement par contrainte violée, en laissant pour la fin de la construction de l'instanciation, l'héritage direct des meilleures caractéristiques restantes. Avec cette stratégie, on incorpore un surcoût dû au recalcul de la priorité, par contre au niveau du nombre de tests de contraintes, l'algorithme réalise autant de vérifications de contraintes pour chaque application de *CAD* qu'*arc-crossover*.

La performance de notre algorithme dépend de la connectivité du graphe de contraintes, mais aussi du nombre de valeurs qui peuvent satisfaire localement une contrainte. Quand l'algorithme a plus de choix locaux (p_2 plus petit) et que le nombre espéré de solutions du graphe s'approche de 1, alors la recherche stochastique pour

notre algorithme devient de plus en plus difficile. En d'autre termes, cela veut dire que si nous n'avons qu'une solution pour le CSP, le problème deviendra plus difficile quand nous aurons plusieurs possibilités de valeurs qui satisfont chaque contrainte localement, mais qui ne font pas partie de la solution globale. Cette difficulté est partagée avec la plupart de méthodes incomplètes pour la résolution de CSP.

Les travaux exposés dans ce chapitre ont été présentés en [Rif97c], [Rif97b].