Résolution de problèmes de satisfaction de contraintes avec des algorithmes évolutionnistes

Maria-Cristina Riff-Rojas

THÈSE

présentée à

L'École Nationale des Ponts et Chaussées

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité

MATHÉMATIQUES ET INFORMATIQUE

par

María-Cristina RIFF-ROJAS

Sujet de la thèse

RÉSOLUTION DE PROBLÈMES DE SATISFACTION DE CONTRAINTES AVEC DES ALGORITHMES ÉVOLUTIONNISTES

Soutenue le 8 Décembre 1997 devant le jury composé de:

M. Norbert Cot	Président
Mme. Catherine Roucairol	Rapporteur
M. Jean-Jacques Chabrier	Rapporteur
M. Marc Schoenauer	Rapporteur
M. Bertrand Neveu	Directeur
M. Philippe Collard	Examinateur

A mi abuelita María

A mis padres y Ximena

Résumé

Dans les disciplines de l'intelligence artificielle et de la recherche opérationelle, on rencontre de nombreux problèmes comme l'allocation de ressources, l'ordonnancement, la conception, le diagnostic automatisé. Ces problèmes se formulent aisément comme des problèmes de satisfaction de contraintes (CSP). Un CSP est défini comme étant un ensemble de *contraintes* impliquant un certain nombre de *variables*. L'objectif consiste simplement à trouver un ensemble de valeurs à affecter aux variables, de sorte que toutes les contraintes soient satisfaites. Dans le cas le plus général, les problèmes de satisfaction de contraintes ont un aspect fortement combinatoire qui leur confère une grande complexité.

Nous nous intéressons dans le cadre de cette thèse aux problèmes de satisfaction de contraintes binaires en domaines finis.

Les méthodes auxquelles nous nous intéressons pour résoudre un CSP sont les méthodes dites incomplètes: elles font une réparation d'une configuration en parcourant de manière non systématique l'espace des configurations. Dans cette catégorie de méthodes, notre intérêt s'est plus particulièrement tourné vers les Algorithmes Évolutionnistes. Ce sont des méthodes générales d'optimisation combinatoire qui sont inspirées de la théorie de l'évolution.

Dans un CSP classique, on recherche une solution, sans avoir à optimiser de fonction. Pour entrer dans le cadre des Algorithmes Évolutionnistes, on se doit de définir une *fonction d'évaluation* pour les CSP qui prend ses valeurs minimales sur les solutions du problème. Cette fonction pourrait être utilisée par toutes méthodes incomplètes, telles que les techniques *min-conflits*, *GSAT* et leurs variantes. Nous montrons

dans cette thèse l'application de notre *fonction d'évaluation* pour la méthode *min-conflits* ainsi que pour un algorithme évolutionniste.

D'un autre côté, dans le contexte plus spécifique des algorithmes génétiques, nous souhaitons guider l'évolution (i.e. recherche d'une solution), en faisant des transformations sur la population plus orientées vers le problème de satisfaction de contraintes. Nous définissons ainsi des opérateurs de mutation et de croisement spécialisés pour les CSP, qui sont basés sur la structure du graphe de contraintes. Ensuite, nous incorporons le concept d'adaptation dans l'opérateur de croisement, afin d'améliorer la recherche de l'algorithme.

Dans ce mémoire, nous décrivons et justifions les algorithmes mis en œuvre, en illustrant les techniques implémentées par la résolution de problèmes de coloriage de graphe avec trois couleurs, et de CSP générés aléatoirement.

Plan de lecture.

Dans le premier chapitre de cette thèse, une brève description des méthodes de résolution des problèmes de satisfaction de contraintes (CSP) et une comparaison entre les algorithmes systématiques et stochastiques sont exposées.

Dans le deuxième chapitre, nous présentons un rapide survol des Algorithme Génétiques et des Algorithmes Évolutionnistes.

Le troisième chapitre présente une nouvelle fonction d'évaluation pour les CSP binaires, qui est utilisée dans un algorithme stochastique avec l'heuristique *min-conflits*. Cette fonction est également incorporée dans un algorithme évolutionniste. Un ensemble de tests pour résoudre le CSP bien connu de coloriage du graphe avec trois couleurs est présenté. Finalement, une extension de cette fonction pour des CSP n-aires est proposée.

Dans le quatrième chapitre, des opérateurs sont définis spécialement pour résoudre les CSP binaires avec un algorithme évolutionniste. Une comparaison avec d'autres algorithmes est effectuée pour le problème de coloriage de graphe, ainsi que pour des CSP générés aléatoirement. À la fin du chapitre, une extension de ces opérateurs pour la résolution de CSP n-aires est présentée.

Le cinquième chapitre traite le concept d'adaptation, qui est utilisé pour définir un nouvel opérateur. L'algorithme utilisant cet opérateur est évalué sur un ensemble de problèmes pour le coloriage de graphe et des CSP aléatoires, puis comparé avec d'autres algorithmes.

Mots-clés. Problème de Satisfaction de Contraintes(CSP), Méthodes de Résolution Stochastiques, Algorithmes Génétiques(AG), Algorithmes Évolutionnistes, Fonction d'Évaluation, Opérateurs Spécialisés, Adaptation, Coloriage de Graphe, CSP aléatoires.

Remerciements

Je tiens à remercier ici

Ma famille: Juan, Cristina y Ximena qui m'ont soutenue pendant toute la période durant laquelle je suis restée loin d'eux pour réaliser cette thèse.

Kamal pour son grand support moral et sa compagnie pendant nos trois ans d'amitié.

Xavier qui m'a motivée pour venir faire ma Thèse en France et qui m'a aidée pendant mon séjour ici. Qui en plus a eu la patience de lire cette thèse. Pour son amitié entre 1992-1994 par Internet et après ici.

Bertrand Neveu qui m'accueillie dans son équipe et m'a toujours laissé une grande liberté pour le choix et l'organisation de mon travail. Qui a eu la patience de lire et d'essayer de comprendre ce que je voulais exprimer dans cette thèse, qui m'a donné l'opportunité de participer à différentes conférences scientifiques, et qui m'a guidée dans ma recherche.

Madame le Professeur Catherine Roucairol, Monsieur le Professeur Jean-Jacques Chabrier, Monsieur Marc Schoenauer pour avoir bien voulu être les rapporteurs de ma thése ainsi que Monsieur le Professeur Norbert Cot et Monsieur Philippe Collard pour avoir accepté de faire partie de mon jury.

Alain qui a lu cette Thèse, pour ses importants conseils de chercheur ainsi que pour les moments d'amitié partagés.

Eitan pour ses importantes remarques scientifiques

Jose qui m'a fait d'importantes remarques scientifiques.

Hany pour avoir fait que le début de mon séjour ici soit plus facile

Jérôme pour les bons moments de danse partagés

Eduardo, Lupe, Andrés, Sorabelia, Mabel, Alain J-M, Suzanne, Robert, Marta, Lazlo, Olivier, Tania, Eitan, Kamal, et à tous ceux qui ont partagé avec nous un pas de danse, pour avoir contribué à mon agréable séjour ici.

Gilles pour nos vols amusants.

Mes amis: Mónica, José, Roxana, Rodrigo, Roberto, Horst, Ana María, Carlos, María-Isabel, Sergio, Sandra qui m'ont donné leur amitié à distance avec lettres et mails.

Personnel technique et administratif de l'INRIA et du CERMICS.

L'equipe de contraintes et l'equipe de base de données du CERMICS.

Table des matières

Liste des tableaux

Table des figures

Introduction

Les problèmes de satisfaction de contraintes (CSP) sont au cœur de nombreuses applications en intelligence artificielle et recherche opérationnelle, telles que l'allocation de ressources, la planification, l'ordonnancement. Un CSP est défini comme étant un ensemble de *contraintes* impliquant un certain nombre de *variables*. L'objectif consiste simplement à trouver un ensemble de valeurs à affecter aux variables, de sorte que toutes les contraintes soient satisfaites. La recherche dans le domaine des CSP est motivée par le besoin de concevoir des méthodes efficaces pour les résoudre. Dans un CSP, le but est de trouver une solution qui satisfasse toutes les contraintes. Mais la plupart des CSP appartiennent à une classe de problèmes appelés NP-complets [GJ79] pour lesquels tous les algorithmes connus nécessitent, dans le pire des cas, un temps exponentiel. En essayant de diminuer le coût de la résolution d'un CSP, plusieurs méthodes ont été proposées par des chercheurs. Ces méthodes peuvent être classées en deux catégories, méthodes complètes ou systématiques et incomplètes ou stochastiques. Les méthodes complètes font une recherche arborescente en construisant une solution. Ces types de méthodes systématiques, à cause de la taille de l'espace de recherche, peuvent prendre beaucoup de temps pour arriver à trouver une solution, elles ne sont donc utiles en pratique que lorsque la taille des problèmes est relativement petite [GJ79]. Les méthodes incomplètes font une réparation d'une configuration (ou pré-solution) en parcourant de manière non systématique (aléatoire) l'espace de recherche. La recherche n'étant pas systématique, nous ne sommes pas sûrs de trouver une solution, et on ne sait pas alors décider si le problème admet une solution ou pas. Parmi les approches les plus répandues, on compte l'heuristique de réparation *Minimum-conflits* proposée par Minton [MJPL92] et GSAT proposée par

Selman [SLM92] pour résoudre le problème de satisfiabilité d'une formule booléene (SAT). Ces deux méthodes utilisent une stratégie fondée sur des optimisations locales nommée *escalade*[1]. Cette stratégie répare la configuration courante et en construit une nouvelle, en cherchant à satisfaire plus de contraintes que la pré-solution précédente. Son avantage principal est la simplicité de calcul, mais pour guider la recherche locale vers des espaces de recherches prometteurs pour trouver plus rapidement une solution, nous avons besoin d'une fonction d'évaluation judicieuse [Pea90]. Les fonctions d'évaluation à améliorations trompeuses peuvent conduire la recherche vers des zones qui ne contiennent en effet aucune solution. L'algorithme peut se retrouver dans un optimum local où il n'y a plus d'amélioration possible par des perturbations locales et le processus se termine sans avoir trouvé de solution. Une issue possible consiste à recommencer à nouveau à partir d'une autre configuration initiale. Mais alors nous pourrions explorer plusieurs fois le même endroit, car on n'est évidemment pas garanti de ne pas ré-explorer un même sous-espace. Une autre amélioration possible est de stocker les derniers mouvements pour éviter de rester sur les mêmes configurations, ce que fait la méthode de la *liste tabou* [Glo86]. Une autre alternative est d'accepter sous certaines conditions une configuration qui détériore la fonction d'évaluation comme le fait la méthode du *recuit simulé* [KGV83]. Nous pouvons aussi améliorer la recherche en prenant une population de configurations courantes en utilisant, par exemple un *algorithme génétique* [Gol89]. L'escalade, comme toute méthode incomplète est une stratégie surtout utile pour les problèmes de grande taille. Quand nous possédons une fonction de guidage hautement judicieuse, elle nous permet de nous éloigner des optima locaux et des plateaux pour nous mener rapidement vers l'optimum global. L'escalade recherche une solution en effectuant une *exploitation* du voisinage de la configuration courante pour une possible amélioration, mais elle ne réalise pas une véritable *exploration* de l'espace de recherche. Cette stratégie est dite *irrévocable* parce qu'elle n'a pas de mémoire des dernières pré-solutions trouvées et le processus ne peut pas revenir volontairement sur une pré-solution antérieure, même si elle pouvait offrir plus de promesses que la configuration courante [Pea81].

D'un autre côté, la recherche purement aléatoire est un exemple typique d'une

1. Hill-climbing en anglais

stratégie qui explore l'espace de recherche mais qui refuse d'exploiter les régions prometteuses de l'espace. Les algorithmes évolutionnistes constituent une autre famille de méthodes stochastiques, leurs bases théoriques ont été proposées par Holland [Hol75] et De Jong [DJ75] la même année a présenté une implémentation et son application pour résoudre des problèmes d'optimisation de fonctions sans contraintes. Ils relèvent à la fois de l'exploration et de l'exploitation. Ils sont basés sur une analogie avec le principe de l'évolution et de la sélection naturelle. Nous soutenons que ces types d'algorithmes peuvent conduire à une voie de recherche prometteuse pour la résolution de CSP, car ils possèdent en partie les avantages des méthodes avec escalade. Mais ils n'appartiennent pas à la catégorie des méthodes à solutions *irrévocables*, car ils gardent dans la population une mémoire implicite des solutions déjà rencontrées.

Il sera nécessaire d'adapter la structure des algorithmes évolutionnistes pour les appliquer à la résolution des CSP. Normalement, un problème de satisfaction de contraintes n'est pas en lui même un problème d'optimisation; il faudra donc définir une fonction d'évaluation à optimiser pour parcourir l'espace de recherche. Il sera intéressant de concevoir une méthode efficace pour réaliser l'exploration et l'exploitation à partir des pré-solutions et ainsi trouver plus rapidement une solution. Pour ce faire, nous regarderons du côté des méthodes systématiques en profitant de leur avantages et en incorporant certaines idées dans la conception de notre algorithme [Dec90], [Fre95].

Cette thèse est organisée comme suit. Le chapitre suivant constitue un état de l'art non-exhaustif des différentes approches pour résoudre un CSP. Le chapitre 2 présente un bref survol sur les algorithmes évolutionnistes. Le chapitre 3 présente une nouvelle fonction d'évaluation basée sur la structure du graphe de contraintes. Elle nous permettra de proposer un nouvel algorithme de type escalade pour améliorer la recherche de la méthode *minimum-conflits*. Cette fonction sera aussi le guide pour notre algorithme évolutionniste. Nous comparons ensuite avec d'autres méthodes qui utilisent d'autres types de fonctions de guidage. Le chapitre 4 dresse la conception d'opérateurs adaptés pour les CSP, et qui prennent aussi en compte la structure du

graphe de contraintes. Nous incorporons ensuite dans le chapitre 5 le concept d'adaptabilité dans la conception d'un opérateur afin d'améliorer les résultats déjà prometteurs obtenus avec nos premiers opérateurs. Nous comparons nos méthodes à celles utilisant d'autres opérateurs proposées dans la littérature. Enfin, nous présentons les conclusions et perspectives de nos travaux.

Chapitre 1

Problèmes de Satisfaction de Contraintes (CSP)

Une grande partie des problèmes de l'Intelligence Artificielle et d'autres domaines de l'informatique peuvent être considérés comme des cas particuliers des problèmes de satisfaction de contraintes (CSP), [Nad90]. Cette thèse concerne les problèmes de satisfaction de contraintes qui peuvent être définis de la manière suivante. Les données du problème sont un ensemble fini de variables, un domaine fini pour chaque variable et un ensemble de contraintes. Chaque contrainte est définie sur un sous-ensemble de l'ensemble de variables et elle limite les combinaisons de valeurs que les variables qui appartiennent à ce sous-ensemble peuvent avoir. Le but est de trouver des valeurs dans les domaines des variables qui satisfassent toutes les contraintes. Plus formellement, nous définissons dans les sections suivantes les concepts utilisés par les CSP.

1.1 Concepts et notations

Définition 1.1.1 *(Problème de Satisfaction de Contraintes)*
Un problème de satisfaction de contraintes ou CSP, $P = (V, D, \zeta)$ est défini par:

- *un ensemble $V = \{X_1, \ldots, X_n\}$ de n variables*
- *un ensemble $D = \{D_1, \ldots, D_n\}$ de n domaines finis pour les variables de V. Le*

domaine D_i est associé à la variable X_i,

un ensemble $\zeta = \{C_1, \ldots, C_\eta\}$ de η contraintes. Chaque contrainte C_i est définie par un couple (v_i, r_i):

- *v_i est un ensemble de variables $\{X_{i_1}, \ldots, X_{i_{m_i}}\} \subset V$ sur lesquelles porte la contrainte C_i. On appelle arité de C_i la longueur de la séquence v_i, donc le cardinal de v_i:*
- *r_i est une relation, définie par un sous-ensemble du produit cartésien $D_{i_1} \times \ldots \times D_{i_{m_i}}$ des domaines associés aux variables de v_i. Il représente les n-uplets de valeurs autorisés pour ces variables.*

Définition 1.1.2 *(Être pertinente pour)*
Chaque variable X_j **est pertinente pour** *C_k (noté par $X_j \rhd C_k$), si C_k porte sur X_j. $\forall k \in [1..\eta]$.*

Définition 1.1.3 *(Arité de C_α)*
Étant donné une contrainte C_α et ses variables pertinentes, on définit l'arité de C_α a_α comme le nombre de variables pertinentes pour C_α.

Définition 1.1.4 *(CSP binaire)*
Un CSP binaire est un CSP $P = (V, D, \zeta)$ dont toutes les contraintes $C_i \in \zeta$ ont une arité égale à 2, c'est à dire, chaque contrainte a exactement 2 variables pertinentes.

Un CSP binaire peut être représenté par un graphe de contraintes dans lequel chaque nœud représente une variable, et chaque arc correspond à une contrainte entre 2 variables. Rossi et al. [RPD89] ont montré qu'il est possible de convertir un CSP avec des contraintes n-aires en un CSP binaire équivalent. C'est pour cela que la plupart des recherches sur les méthodes pour la résolution de CSP avec domaines finis traitent, pour commencer, les CSP binaires, [Fre82]. Dans cette thèse, nous abordons

la résolution de CSP binaires en utilisant plus particulièrement leur représentation matricielle.

Définition 1.1.5 *(Matrice de Contraintes)*
Une Matrice de Contraintes $\mathbf{R}$ *est un tableau rectangulaire* $\eta \times n$, *tel que*

$$\mathbf{R}_{\alpha j} = \mathbf{R}[\alpha, j] = \begin{cases} 1 & \textit{Si variable } X_j \rhd C_\alpha \\ 0 & \textit{sinon} \end{cases}$$

S'il s'agit d'un CSP binaire, dans $\mathbf{R}$ il y a juste deux entrées non-nulles pour une contrainte C_α, comme cela est montré sur la figure 1.1. Dans l'exemple, les variables X_2 et X_n sont les deux variables pertinentes pour C_α. Cela est représenté dans la matrice de contraintes avec le numéro 1 à l'intersection entre la colonne de chaque variable et la ligne correspondant à C_α. Dans le graphe de contraintes, cela est indiqué par l'arête α qui lie la variable X_2 avec la variable X_n.

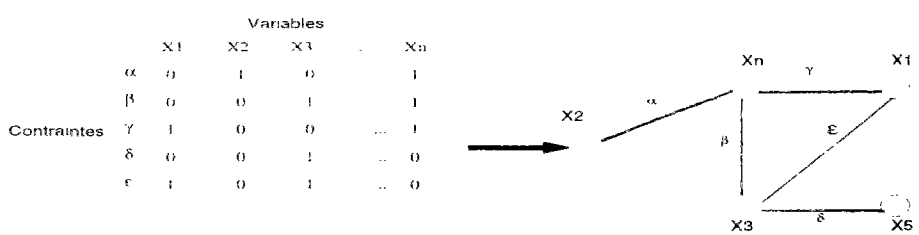


FIG. 1.1 – *Matrice de Contraintes et son Graphe de Contraintes*

Définition 1.1.6 *(Instanciation)*
Étant donné un CSP $P = (V, D, \zeta)$, *on appelle instanciation* $\mathbf{I}$ *une application qui associe à chaque variable* $X_i \in V$ *une valeur* $\mathbf{I}(X_i) \in D_i$

Définition 1.1.7 *(Instanciation Partielle)*
Étant donné un CSP $P = (V, D, \zeta)$, *on appelle instanciation partielle* $\mathbf{I_p}$ *de* $V_p = \{X_{p_1}, \ldots, X_{p_j}\} \subseteq V$ *une application qui associe à chaque variable* $X_{p_i} \in V_p$ *une valeur* $\mathbf{I_p}(X_{p_i}) \in D_{p_i}$ *(le domaine de* X_{p_i}*)*

Remarque: Dans le cas où $V_p = V$ alors $\mathbf{I_p}$ est une instanciation complète.

Définition 1.1.8 *(Satisfaction de contrainte)*
Étant donné un CSP $P = (V, D, \zeta)$, une instanciation partielle $\mathbf{I_p}$ satisfait *la contrainte $C_i = (v_i, r_i)$ de ζ (noté $\mathbf{I_p} \models C_i$) ssi $v_i \subset V_p$ et $\mathbf{I_p}$ de $v_i \in r_i$. À l'opposé, on dira qu'une instanciation $\mathbf{I_p}$ viole C_i ssi $v_i \subset V_p$ et $\mathbf{I_p}(v_i) \notin r_i$*

Définition 1.1.9 *(Instanciation consistante)*
Étant donné un CSP $P = (V, D, \zeta)$, une instanciation partielle $\mathbf{I_p}$ des variables est dite consistante *ssi:*

$$\forall C_i = (v_i, r_i) \in \zeta, \text{ telle que } v_i \subset V_p, \mathbf{I_p} \models C_i$$

Définition 1.1.10 *(Solution(s) d'un CSP)*
Une solution $\mathbf{S}$ *de $P = (V, D, \zeta)$ est une instanciation consistante de toutes les variables ($V_p = V$). On dit alors que l'instanciation* $\mathbf{S}$ *satisfait P (noté $\mathbf{S} \models \mathbf{P}$). L'ensemble des solutions de P sera noté* $\mathbf{S_P}$.

1.2 Exemples de CSP

Les exemples suivants sont d'un intérêt particulier, car ils sont des prototypes pour une grande classe de problèmes pratiques.

Le coloriage de graphe

Le problème de coloriage de graphe avec 3 couleurs consiste à colorier un graphe non-orienté comprenant n nœuds. Chaque nœud doit être colorié avec une des trois couleurs disponibles de telle sorte que deux nœuds voisins n'aient pas la même couleur. Ce problème est utilisé pour modéliser certains types de problèmes d'ordonnancement et de gestion de ressources. La figure 1.2 montre le problème de coloriage d'une carte avec 3 couleurs (noir, blanc, gris). Les contraintes sont toutes du même type: ne pas colorier avec la même couleur les pays voisins.

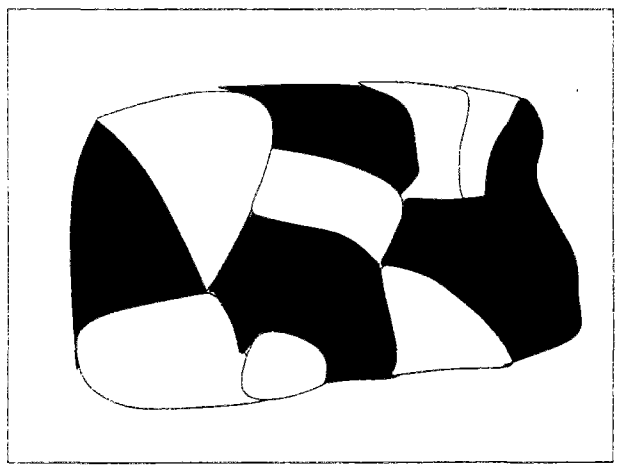


FIG. 1.2 – *Exemple: Coloriage avec 3 couleurs*

Les N-reines

Placer sur un échiquier de $N \times N$ cases, N reines qui ne s'attaquent pas (suivant les règles classiques des échecs). Il s'agit d'un problème équivalent à la recherche d'un ensemble intérieurement stable (qui sera maximum) dans un graphe de $N \times N$ nœuds (un nœud pour chaque case), chaque arête correspondant à une diagonale, une verticale ou une horizontale. Il peut s'envisager pour d'autres types de pièces.

Le problème du Zèbre

Attribué à Lewis Carroll, pasteur logicien et écrivain anglais auteur de nombreux autres puzzles. On considère cinq maisons, toutes de couleurs différentes (rouge, bleu, jaune, blanc, vert), dans lesquelles logent cinq personnes de profession différente (peintre, sculpteur, diplomate, docteur et violoniste) de nationalité différente (anglaise, espagnole, japonaise, norvégienne et italienne) ayant chacune une boisson favorite (thé, jus de fruits, café, lait et vin) et des animaux favoris (chien, escargots, renard, cheval et zèbre). On dispose des faits suivants: l'Anglais habite la maison rouge, l'Espagnol possède un chien, le Japonais est peintre, l'Italien boit du thé, le Norvégien habite la première maison à gauche, le propriétaire de la maison verte boit du café, la maison verte est à droite de la blanche, le sculpteur élève des escargots, le

diplomate habite la maison jaune, on boit du lait dans la maison du milieu, le Norvégien habite à côté de la maison bleue, le violoniste boit du jus de fruit, le renard est dans la maison voisine du médecin, le cheval est à côté de la maison du diplomate.

Il s'agit de trouver le possesseur du zèbre et le buveur de vin. En fait le problème n'admet qu'une seule solution et il s'agit de la trouver.

CSP aléatoires

Une pratique qui devient de plus en plus courante est de tester un nouvel algorithme sur un jeu important de CSP binaires générés entièrement aléatoirement, [Smi95]. Chaque jeu de problèmes est décrit par quatre paramètres: n, le nombre de variables, m le nombre de valeurs de chaque domaine, p_1 la probabilité qu'il y ait une contrainte entre deux variables, et p_2 la probabilité conditionnelle qu'une paire de valeurs soit inconsistante pour un couple de variables, étant donné qu'il y a une contrainte entre les deux variables. Ces tests sont généralement utilisés pour juger très grossièrement les performances relatives de différents algorithmes, et pour déterminer pour quels problèmes (avec des graphes denses, contraintes difficiles) ils sont efficaces.

Dans cette thèse, nous avons principalement testé nos algorithmes sur le problème de 3-coloriage de graphe et sur les CSP binaires générés aléatoirement.

1.3 Méthodes de résolution des CSP

Il existe différentes approches pour résoudre les problèmes de satisfaction de contraintes. Dans la figure 1.3, nous montrons un résumé non-exhaustif des différentes approches existantes pour aborder un CSP, [Fre95]. Celles qui sont prises en compte dans cette Thèse sont: Algorithmes Génétiques, Décomposition, Réparation par Escalade, Représentation par Réseaux de Contraintes.

Nous pouvons aussi classifier en deux classes les méthodes de résolution: Méthodes Complètes et Incomplètes. Les méthodes complètes font une recherche arborescente en instanciant les variables une par une et en effectuant un retour en arrière en cas d'échec. Les méthodes incomplètes font une réparation d'une configuration en parcourant de manière non systématique (aléatoire) l'espace de recherche.

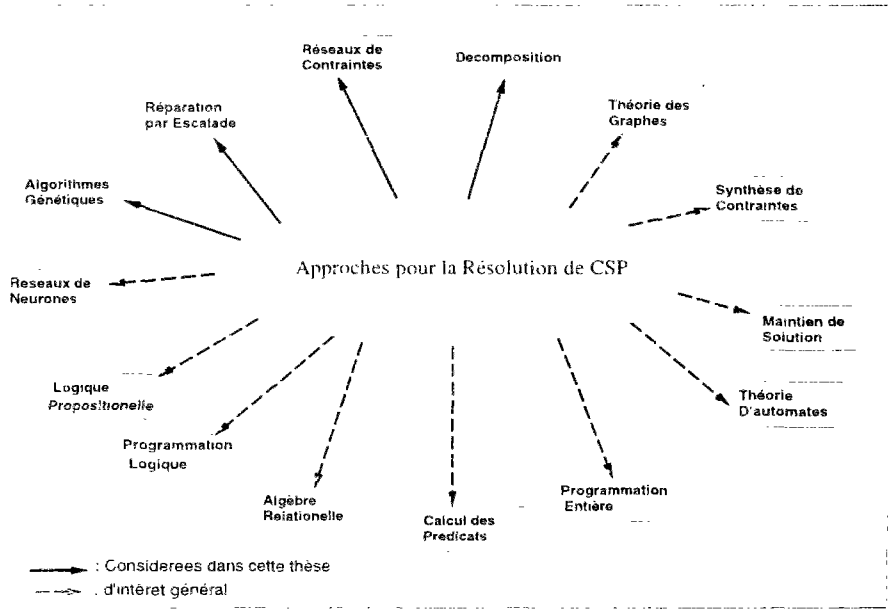


FIG. 1.3 – *Différentes Approches pour CSP*

1.3.1 Méthodes complètes

Selon Dechter, [Dec90] nous pouvons classer les techniques de résolution de CSP du point de vue de l'exploration en trois catégories:

- Exploration systématique: L'algorithme le plus courant dans cette catégorie est le retour-en-arrière[1] lequel traverse l'espace de recherche en profondeur d'abord, [Kum92]. Il démarre avec un ordre pré-établi des variables du CSP et des valeurs de chaque variable dans son domaine. L'algorithme affecte des valeurs aux variables, une après l'autre, et teste chaque fois si l'instanciation partielle courante est localement consistante. Lors de son exécution, il y aura retour-en-arrière quand une instanciation partielle choisie durant la recherche et localement consistante doit finalement être écartée parce qu'il n'existe pas de prolongement de cette instanciation partielle qui soit solution.

- Algorithmes prospectifs[2]: Ces algorithmes après chaque affectation de valeur à une variable, éliminent des domaines des variables non encore instanciées

1. en anglais: backtrack
2. en anglais: look-ahead

les valeurs qui conduisent d'évidence à un échec. La particularité de chaque algorithme sera dans le nombre de tests de contraintes et le nombre de valeurs éliminées, parmi eux on peut citer Forward-checking, Partial-lookahead, Full-lookahead, Real-full-lookahead [HE80], MAC [BFR95].

Algorithmes rétrospectifs[3]: Ces algorithmes sont incorporés dans le processus de recherche lui-même pour réaliser un retour-en-arrière "intelligent". Ils essaient de tirer parti des informations implicites contenues dans les situations d'échec pour économiser plus tard des essais de valeurs ou pour sélectionner un meilleur point de retour. Parmi eux, on trouve Back-jumping [Dec90], Conflict-Back-jumping [Pro91], Back-checking [HE80], Back-marking [Gas77].

Dans la même catégorie des méthodes complètes, on trouve d'autres types d'approches intéressants, comme par exemple:

- Algorithmes de filtrage: ils sont utilisés d'abord dans une phase de pré-traitement pour améliorer la performance de l'algorithme de recherche. Une procédure de filtrage augmente la cohérence locale d'un CSP. Elle transforme un CSP en un autre CSP dont l'espace de recherche est réduit par rapport à celui du CSP d'origine mais dont l'ensemble de solutions est le même. Il ne perd pas d'information dans la transformation, mais le nouveau CSP contient explicitement certaines contraintes qui étaient implicites dans le CSP origine. Cette explicitation de contraintes évitera à une procédure de recherche de parcourir certaines zones de l'espace de recherche qui mèneraient sûrement à un échec. Les algorithmes de filtrage ou de cohérence locale ne résolvent pas complètement un CSP mais éliminent les inconsistances locales, qui autrement auraient pu être découvertes plusieurs fois par les procédures de recherche de solution. Parmi eux, les niveaux de filtrage les plus répandus sont la cohérence d'arc [MH86], [Bes94], la cohérence de chemin [Mon74], la k-cohérence [Fre78]. Les méthodes prospectives s'appuient sur les algorithmes de cohérence d'arc. Dans [HE80] sont présentées les procédures qui fonctionnent avec un filtrage pendant

3. en anglais: look-back

la recherche comme Forward-checking, Partial-lookahead. Full-lookahead, Real-full-lookahead. Elles se distinguent par des niveaux de filtrages de plus en plus forts. Le niveau de filtrage le plus simple (Forward-checking) consiste en une analyse du voisinage immédiat de la variable dernièrement instanciée, le plus fort (Real-full-lookahead) établit de la consistance d'arc complète après chaque instanciation.

- Algorithmes basés sur la structure du graphe: Leur recherche est guidée par les caractéristiques du réseau de contraintes. L'idée est de décomposer un CSP de façon qu'il soit possible de construire les solutions du problème à partir des solutions de sous-problèmes. On divise l'ensemble de nœuds du graphe en k sous-ensembles définissant chacun un sous-CSP, de telle façon que toute contrainte appartienne à au moins un des sous-CSP. On pourra former les solutions du problème en résolvant un CSP à k variables (une pour chaque sous-CSP) dont les domaines sont formés par les ensembles de solutions du sous-CSP correspondant. Une contrainte existe entre des variables qui correspondent à des sous-problèmes ayant au moins une variable en commun, sa relation associée exprime simplement que la projection des solutions des sous-CSP coïncide sur ces variables communes [AS94]. Ces algorithmes peuvent servir d'appui aux algorithmes de cohérence ainsi que pour les algorithmes d'exploration systématique, [Dec92], [Jeg90].

Dans cette classification des méthodes complètes, nous nous sommes intéressée particulièrement aux algorithmes qui utilisent les caractéristiques topologiques du réseau de contraintes pour guider la recherche d'une solution, par exemple la méthode proposée par Dechter en [Dec92] pour réduire la taille du graphe de contraintes, la condition trouvée par Freuder pour faire une recherche sans retour-en-arrière en [Fre82], et d'autres méthodes qui cherchent à augmenter la performance de la recherche en utilisant la connaissance de la structure du graphe [DP88].

Procédure Min-conflits (V, D, ζ)
Début
I instanciation initiale des variables
Répéter
$K(\mathbf{I})$=variables en conflit dans I
Choisir une paire variable-valeur (X_i, d_{i_l}) dans **I** tel que $X_i \in K(\mathbf{I})$
Choisir la valeur d_{i_j} pour X_i tel que
$d_{i_j} \in D_i$ minimise le nombre de conflits
$\mathbf{I} = (\mathbf{I} - \{(X_i, d_{i_l})\}) \cup \{(X_i, d_{i_j})\}$
jusqu'à ce que toutes les contraintes soient satisfaites
ou un nombre maximum d'itérations
Fin /* procédure min-conflits */

FIG. 1.4 – *Structure de la procédure min-conflits*

1.3.2 Méthodes incomplètes

Les méthodes incomplètes réparent une configuration courante en parcourant de manière non systématique (aléatoire) l'espace de recherche. Dans cette catégorie, la méthode la plus répandue est proposée sous le nom de *min-conflits* dans [MJPL92] autour d'une heuristique de *réparation locale* avec *minimisation de conflits*. Cette heuristique consiste à choisir une variable intervenant dans une contrainte non-satisfaite, et à choisir pour cette variable, une valeur qui minimise le nombre de contraintes non-satisfaites.

Définition 1.3.1 *(Min-conflits)*
Étant donné: un CSP binaire et une instanciation **I**. *Deux variables seront en* conflit *si leurs valeurs violent une contrainte.*
Réparation: Sélectionner une variable qui est en conflit *et choisir pour elle une valeur qui minimise le nombre total de conflits.*

La procédure *min-conflits* est montrée sur la figure 1.4. Cette méthode est extraordinairement simple et efficace surtout pour les problèmes de grande taille d'ordonnancement et de gestion de ressources, d'après Minton et al. en [MJPL92]. Elle a tout même l'inconvénient de ne pas assurer de trouver une solution, et de pouvoir rester figée dans un minimum local. Plusieurs méthodes ont été définies pour l'aider à sortir

```
Procédure GSAT (CSP, MAX_TRIES, MAX_FLIPS)
Début
Pour i=1 jusqu'à MAX_TRIES
     I instanciation initiale des variables
    Pour j=1 jusqu'à MAX_FLIPS
         si I satisfait CSP alors solution
         sinon soit F l'ensemble des paires variable-valeur qui
                  quand elles sont changées améliorent le
                  nombre de contraintes satisfaites
              prendre aléatoirement un f ∈ F
         I = I avec f changé
Fin /* procédure GSAT */
```

FIG. 1.5 – *Structure de la procédure GSAT*

de ce type de situation, par exemple de recommencer avec une nouvelle instanciation initiale une fois que l'algorithme s'aperçoit qu'il n'arrive pas à trouver une meilleure solution. [Mor93]. Une approche similaire pour les problèmes de satisfiabilité (SAT) est proposée par Selman [SLM92] sous le nom de GSAT. L'algorithme commence par générer une instanciation initiale et il l'améliore ensuite de façon incrémentale en changeant la valeur d'une variable, de telle sorte que la nouvelle valeur représente la plus grande augmentation du nombre de contraintes satisfaites. Cela est fait jusqu'à ce que toutes les contraintes soient satisfaites ou jusqu'à avoir fait un nombre maximum pré-determiné de changements (MAX-FLIPS). La procédure standard de GSAT est montrée dans la figure 1.5. GSAT peut rester lui aussi figé dans un minimum local, donc plusieurs heuristiques ont déjà été proposées pour l'aider dans sa recherche aléatoire, comme celle du *chemin aléatoire*[4] présenté par Selman, Kautz et Cohen en [SKC94].

GSAT et *min-conflits* utilisent une stratégie fondée sur des optimisations locales nomée l'escalade. Cette stratégie répare la configuration courante et en construit une nouvelle, en cherchant à satisfaire plus de contraintes que la pré-solution précédente. Le succès des algorithmes de recherche locale peut être attribué à leur efficacité, car ils peuvent être plus performants que les méthodes de recherche systématiques, telles

4. en anglais:Random walk

que le retour-en-arrière, le backjumping [Pro91] pour un grand nombre de problèmes [BLS8], [SKC94], [BC97].

1.4 La complexité et les problèmes NP-Complets

La problématique pour trouver une solution pour un CSP est de concevoir un algorithme efficace en temps de calcul et en espace, c'est-à-dire qui soit capable de finir dans un temps raisonnable et d'utiliser une quantité de mémoire de l'ordinateur elle aussi raisonnable. Pour comprendre les difficultés des problèmes impliqués, nous devons introduire d'abord la théorie de la complexité. Si nous avons une vision claire des limitations que nous pourrions rencontrer dans la conception d'algorithmes, cela nous aidera à être capables de développer des bonnes techniques pour aborder ces problèmes difficiles. Pour continuer, nous présentons un résumé de la théorie de la complexité utile pour la résolution des CSP.

Les algorithmes en *temps polynomial* sont ceux dont le temps d'exécution dans le pire des cas est $O(n^k)$ pour un k constant donné et pour une entrée de taille n (c'est à dire, algorithmes de complexité $O(n^3), O(nlogn)$, etc). On dira qu'un algorithme est *en temps exponentiel* si sa complexité en temps n'est pas d'ordre polynomial (par exemple $O(n!), O(2^n)$, etc). Puisqu'une fonction exponentielle augmente beaucoup plus vite qu'une fonction polynomiale quand n croît, il est raisonnable de penser que les "algorithmes efficaces" sont des algorithmes en temps polynomial, c'est à dire, qu'ils ont besoin d'un nombre d'opérations qui n'augmente que de façon polynomiale avec la taille de l'entrée.

Définition 1.4.1 *(P)*
On dira qu'un problème est dans la classe de complexité P s'il peut être résolu par un algorithme connu en temps polynomial.

De cette définition, on peut penser intuitivement que P est une classe de problèmes faciles, car il existe des algorithmes efficaces (temps polynomial) pour résoudre ce type de problèmes. Il faut remarquer que le terme "problème" utilisé dans la théorie de la complexité est par rapport aux problèmes de décision (c'est à dire des problèmes

qui n'ont que deux solutions possibles: la réponse Oui ou la réponse Non) plutôt que des problèmes d'optimisation.

Définition 1.4.2 *(NP)*
Un problème appartient à la classe NP (polynomial non-déterministe) si une solution donnée x qui est une instance "oui" du problème de décision peut être vérifiée par un algorithme en temps polynomial

Selon cette définition, pour classer un problème comme étant NP, nous n'avons pas besoin de montrer qu'il y a un algorithme polynomial pour le résoudre. Nous avons seulement besoin que dans le cas où x est une instance "OUI" du problème, il existe un algorithme qui soit capable de vérifier sa validité en un temps polynomial. D'après les deux définitions, on peut conclure qu'un problème de décision qui est résolu par un algorithme déterministe en temps polynomial est aussi résolu pour un algorithme non-déterministe en temps polynomial, donc P $\subseteq$ NP. Le fait que P soit différent de NP reste un problème ouvert. Nous avons donc besoin de définir la classe de problèmes "difficiles" en NP (remarquons que P est la classe de problèmes "faciles" dans NP).

Définition 1.4.3 *(Réduction Polynomiale)*
On dira qu'un problème Q_1 est réductible en temps polynomial à Q_2 ssi il existe un algorithme en temps polynomial qui transforme toute instance de Q_1 en une instance de Q_2 de telle sorte que l'instance de Q_1 et l'instance transformée aient toujours les mêmes solutions.

Cette définition implique que si un problème Q_1 peut être réduit à un autre problème Q_2, alors Q_1 n'est "pas plus dur à résoudre" que Q_2. On peut définir maintenant la classe des problèmes NP-complets

Définition 1.4.4 *(NP-Complet)*
Un problème de décision Q est NP-complet si:

- *$Q \in NP$, et*
- *tous les autres problèmes en NP peuvent être réduits polynomialement à Q.*

De plus, si un problème Q satisfait la deuxième condition, mais pas nécessairement la première, nous disons que le problème est NP-difficile. La conséquence de cette définition est qu'une fois qu'on a trouvé un problème NP-complet il est relativement facile d'en trouver d'autres en utilisant la règle de réduction polynomiale, donc les problèmes NP-complets sont dans un sens "les plus importants" problèmes de NP. La signification pratique de la théorie de la NP-complétude est que pour résoudre les problèmes dans la classe NP-complet, on aura besoin d'un temps de calcul qui grandit exponentiellement avec la taille du problème (sauf si on montre que P=NP). En termes calculatoires, ces problèmes sont appelés des problèmes intraitables. Pour une description plus en détail de la théorie de la complexité, nous recommandons [CLR90]. Pour une démonstration que les problèmes de satisfiabilité 3-SAT, du voyageur de commerce et du sac à dos sont des problèmes dans la classe NP-complet, le lecteur peut consulter [AS94].

1.5 Algorithmes systématiques versus algorithmes stochastiques

Le fait que plusieurs CSP soient NP-complets suggère fortement qu'il n'existe pas un algorithme systématique efficace pour résoudre ce type de problème, et que les algorithmes complets peuvent être utilisés seulement pour résoudre des problèmes de petite ou moyenne taille. Cela peut apparaître comme décourageant si on cherche à trouver une solution pour un problème NP-complet. Mais, sans rejeter l'utilisation des méthodes complètes pour la résolution des problèmes NP-difficiles, nous énumérons quelques situations où les approches complètes peuvent encore être efficaces pour certains problèmes NP-difficiles de taille raisonnable.

- La NP-complétude est essentiellement un phénomène dans le *pire des cas*. En d'autres termes, nous pouvons seulement conclure que si un problème est NP-difficile, dans le pire des cas, il ne sera pas résolu en temps polynomial. Mais le comportement d'un algorithme exponentiel dans le pire des cas peut être polynomial en moyenne. Par exemple, le fameux algorithme du Simplexe [CLR90],

résout d'une manière assez efficace la plupart des problèmes de programmation linéaire, bien qu'il soit exponentiel dans le pire des cas.

- Il existe la possibilité que le problème en question soit un cas particulier d'un problème NP-complet et que ce cas particulier soit dans la classe P et donc puisse être résolu efficacement. Par exemple, le problème général 0-1 du sac à dos est NP-difficile, mais si tous les coûts sont égaux à un, le problème est P. Par exemple, SAT est NP-complet, mais 2-SAT est P.

Ces observations à propos des problèmes NP-complets sont la motivation pour les chercheurs qui essaient de résoudre ces problèmes d'une manière complète d'utiliser des méthodes d'énumération "intelligentes" avec heuristiques par exemple pour le choix de la valeur d'une variable, où en utilisant l'information pendant la recherche pour faire un retour-en-arrière plus performant. En pratique, nous avons besoin d'algorithmes efficaces pour résoudre des problèmes difficiles de grande taille, lesquels restent hors de portée des méthodes complètes. Dans ce cas, les méthodes stochastiques jouent un rôle important, car en général elles n'explorent pas complètement l'espace de recherche et elles sont aidées par des heuristiques qui guident d'une manière intelligente leur exploration et la réparation d'instanciations. Par exemple en [Sel95] GSAT a pu résoudre des problèmes difficiles avec plus de 2000 variables, par contre les méthodes systématiques actuelles ne sont capables de manipuler à peu près que 400 variables. GSAT a été aussi utilisé pour résoudre des problèmes hautement structurés, par exemple les problèmes de conception de circuits et des problèmes d'algèbre finie, quelques uns de leurs tests avaient 20000 variables et 500000 clauses. Les deux méthodes présentées dans ce chapitre commencent avec une instanciation initiale et elles essaient de la réparer en faisant des changements de valeurs de certaines variables en cherchant une escalade. Elles avancent sans revenir en arrière, donc elles sont dans la catégorie de méthodes irrévocables. Ces dernières années, d'autres méthodes stochastiques utilisant des *métaheuristiques* ont retenu l'attention des chercheurs. Ces méthodes permettent une "dégradation" de la solution, c'est à dire, elles acceptent une solution courante moins bonne que la précédente en terme de la fonction d'évaluation. Évidemment, cette permission de "descente" doit être conditionnée

à certaines règles, de façon d'augmenter la probabilité que l'algorithme puisse sortir d'un optimum local. Les deux méthodes les plus répandues qui utilisent ces idées sont le *Recuit Simulé* et la *Recherche Tabou*. Ici, nous ferons une brève description de chacune de ces méthodes.

1.5.1 Recuit simulé

Le Recuit simulé[5] (SA) fut proposé comme une technique d'optimisation par Kirkpatrick et al. [KGV83]. En métallurgie, l'obtention d'un cristal parfait se fait grâce à la méthode du recuit. On porte un métal à une température suffisamment élevée pour qu'il soit dans l'état liquide. Puis, à partir de cet état, on abaisse la température, et de ce fait les atomes se réorganisent en une autre nouvelle structure.
Une même structure initiale peut donner différentes structures finales selon la façon dont on baisse la température. Si celle-ci baisse trop brutalement, on risque d'atteindre un état métastable qui ne correspond pas à l'état fondamental, minimum absolu d'énergie interne, mais un minimum local de l'énergie. On a obtenu un verre. Il est donc essentiel que l'abaissement de la température se fasse très lentement et très régulièrement. Pour faire disparaître d'éventuels défauts dans la structure du cristal, on utilise la technique du recuit. On réchauffe un peu le métal, afin que les atomes aient plus de liberté de mouvement. En chauffant suffisamment, on donne assez d'énergie pour qu'ils sortent de l'optimum local. En abaissant à nouveau la température régulièrement, ils pourront atteindre l'optimum global.
C'est en s'inspirant de ce procédé que SA a été développé. Il s'agit d'un algorithme stochastique, permettant une détérioration de la valeur de la configuration courante. SA est une technique probabiliste de type escalade qui est basée sur le processus physique du "recuit". SA commence avec un point aléatoire de l'espace de recherche, accepte toute amélioration et permet une modification locale détériorant l'instanciation courante selon une certaine fonction de probabilité. Le niveau d'acceptation d'une "descente" dépend de la grandeur de la diminution de la valeur de la fonction objectif et d'un paramètre appelé température, qui diminue au cours de la recherche.

5. en anglais: Simulated Annealing

Procédure Algorithme Recuit Simulé Standard
Début
Choisir une solution initiale s
Choisir une température initiale T
tant que (que le système n'est pas gelé) [a] **faire**
  tant que (l'équilibre à T n'est pas atteint) **faire**
    Choisir aléatoirement un mouvement élémentaire pour obtenir s'
    Calculer la variation du coût $\Delta f := f(s') - f(s)$
    Si $\Delta f \leq 0$ (amélioration de la solution)
      alors $s := s'$ (modification acceptée)
    sinon $s := s'$ avec la probabilité $\exp^{-\frac{\Delta f}{T}}$
  Réduire la température
Fin /* procédure Algorithme Recuit Simulé Standard */

[a] la température est devenue très basse donc la solution ne peut plus évoluer

FIG. 1.6 - *Structure de l'Algorithme Recuit Simulé Standard*

La figure 1.6 montre l'algorithme de base. L'obtention d'un cristal sans défaut revient donc à minimiser une grandeur physique (l'énergie) en évitant les minima locaux, alors que le nombre d'états possible est immense (le nombre de molécules étant lui-même très élevé, de l'ordre de $10^{23}/cm^3$).

En Optimisation Combinatoire, on est confronté à un problème du même type: minimiser une fonction sur un grand nombre de combinaisons possibles (en général suffisamment grand pour empêcher l'examen de tous les possibilités en un temps raisonnable). Et là aussi, on essaie d'éviter les minima locaux. C'est sur cette idée que l'analogie entre la Mécanique Statistique et les problèmes d'Optimisation Combinatoire a été mise à profit. La solution optimale joue le rôle de l'état cristallin, et les minima locaux remplacent les états métastables. Une différence fondamentale apparaît alors: la température n'est plus une notion physique, elle devient simplement un paramètre de contrôle lorsqu'on simule un recuit.

Cette approche a été utilisée avec succès pour résoudre une grande gamme de problèmes tels que le voyageur de commerce, les tournées de véhicules, la conception de circuits, le traitement d'images, la reconnaissance des formes, incluant aussi les

COP[6] (Problèmes d'Optimisation sous Contraintes). Plus récemment elle a aussi été appliquée à la résolution de MCSP[7] (problèmes de satisfaction d'un maximum de contraintes), où le but est de trouver une solution qui satisfait le nombre maximum de contraintes. Pour ce problème, Hao et al. ont proposé un algorithme basé au recuit simulé qui ont comparé empiriquement, avec un algorithme qui réalise une recherche tabou, [HP98]. Le lecteur peut trouver une description détaillée et des applications de cette méthode en [VLA88] du recuit simulé.

1.5.2 Recherche tabou

L'idée d'une Recherche tabou[8] (TS) a été proposée par Glover en [Glo86] et indépendamment par Hansen en [Han86], pendant la même période. La méthode est une extension directe des procédures d'escalade vues précédemment. Elle est conçue pour aider ces méthodes itératives de recherche locale à sortir d'un optimum local. Pour cela elle gère des structures de données qui mémorisent des éléments de la recherche déjà effectuée.

Deux points sont les fondements de cette approche:

- L'utilisation de structures flexibles de mémorisation conçues pour exploiter l'historique de la recherche.
- Un mécanisme associé pour intensifier et diversifier le processus de recherche en utilisant les structures de mémorisation.

Tabou est actuellement une heuristique qui donne de bons résultats pour un bon nombre de problèmes d'optimisation. En particulier, le succès de cette approche pour le problème classique du voyageur de commerce (TSP) a motivé l'application de la méthode tabou à d'autres problèmes classiques ou plus généraux. Ainsi le coloriage de graphes [HdW87], l'affectation quadratique [Tai91], la tournée de véhicules [Tai93], [Lap92], [Reg96] et plus récemment les MCSP. Pour ces problèmes, Galinier et al. ont proposé un algorithme de recherche tabou qui s'est révélé être plus performant,

6. en anglais: Constraint Optimization Problems
7. Maximal Constraint Satisfaction Problems
8. en anglais: Tabu Search

empiriquement, qu'un algorithme qui utilise min-conflits avec la technique du chemin aléatoire [GH97].

Une fonction objectif f doit être minimisée dans un espace d'états X. Pour chaque point s de X, on définit un voisinage $N(s)$. Pour un CSP, c'est généralement l'ensemble des configurations différant de s par la valeur d'une variable. L'ensemble X et la définition du voisinage se traduisent par un graphe d'états G où deux états i et j sont liés par un arc (i, j) si et seulement si j appartient au voisinage $N(i)$. La méthode tabou est une procédure itérative qui, en partant d'un point initial de X, tente d'atteindre la solution optimale pour f en exécutant à chaque pas un mouvement dans le graphe d'états G. Chaque itération consiste d'abord à produire dans le voisinage $N(s)$ de la configuration courante s, l'ensemble V_a des configurations admissibles, c'est-à-dire celles qui sont obtenues à partir de s par l'exécution d'un mouvement non tabou. Puis on choisit pour nouvelle configuration courante dans V_a la meilleure configuration pour f, même si cela entraîne une augmentation de la fonction f.
La suite des configurations produites à chaque étape de la recherche crée donc un chemin dans G. Il arrive que la recherche de la meilleure solution dans le voisinage ne soit pas une tâche triviale et que l'on doive pour cela résoudre un problème d'optimisation "locale". Selon la taille du voisinage choisi, le problème d'optimisation locale peut se résoudre exactement ou de façon approchée.
En acceptant de détériorer la valeur de la solution courante, on permet de "s'éloigner" d'un optimum local, mais cela peut induire des phénomènes de circuits (parcours répétitif du même ensemble de solutions). Pour éviter ces circuits la méthode consiste en l'introduction de contraintes Tabou servant à interdire les mouvements précédents pendant un certain nombre d'iterations. Ces mouvements sont introduits dans une liste T constamment tenue à jour: la liste Tabou.
A chaque itération, l'élément le plus ancien est remplacé par le dernier mouvement (ou le mouvement inverse). La figure 1.7 montre l'algorithme dans sa version la plus simple qui utilise une liste taboue T et un critère d'aspiration qui accepte un mouvement tabou si celui-ci produit un état meilleur que le meilleur état trouvé jusqu'à présent.

L'élément fondamental de la méthode est l'utilisation de structures de mémoire

Procédure Algorithme Recherche Tabou Standard
Début
Choisir une solution initiale s
tant que (que le critère d'arrêt n'est pas vérifié) **faire**
Engendrer un échantillon $V_a \subseteq N(s) - T$
Rechercher $s' \in V_a$ telle que $f(s') = min_{x \in V_a} f(x)$
$s := s'$
Si $f(s') \geq f(s^*)$
alors $s^* := s'$
mettre a jour la liste T
Fin /* procédure Algorithme Recherche Tabou Standard */

FIG. 1.7 – *Structure de l'Algorithme Recherche Tabou Standard*

flexibles au lieu de structures de mémoire rigides (comme dans les recherches systématiques Branch & Bound et A^*) ou encore de systèmes sans mémoire(comme le Recuit Simulé ou autres approches aléatoires). Les structures de mémoire de la recherche Tabou se réfèrent à quatre types de mesures: ancienneté, fréquence, qualité et influence. Ces mesures sont évaluées à partir d'un ensemble de structures logiques construites au cours du processus de la recherche. La méthode est fondée sur trois structures de mémoire flexible, celles-ci correspondent à trois stratégies de recherche. La *mémoire à court terme* correspond à une stratégie d'exploration agressive recherchant le meilleur mouvement possible (une meilleure évaluation) sujet à certaines contraintes: c'est la liste taboue des mouvements interdits. La *mémoire à moyen terme* réalise une stratégie d'intensification, l'idée est d'intensifier la recherche dans une zone prometteuse. La *mémoire à long terme* correspond à une stratégie de diversification qui évite que certaines régions ne soient totalement négligées. On remarque que souvent les méthodes taboues implantées n'utilisent que la mémoire à court terme, comme l'algorithme de la figure 1.7. Pour une description complète de cette méthode, nous suggérons [Glo89], [Glo90], [GTdW93].

1.5.3 Autres méthodes stochastiques

Récemment, Dorigo en [Dor92] et Dorigo, Maniezzo et Colorni en [DMC96] ont présenté un nouvel algorithme réparti pour résoudre des problèmes d'optimisation combinatoire. Ils ont défini un Système de Fourmis[9] (AS) dans lequel un ensemble d'agents appelés *fourmis* réalisent un travail coopératif pour trouver la solution au problème du voyageur du commerce. Cette technique peut être interprétée comme un type particulier d'apprentissage réparti renforcé. La métaphore est basée sur le comportement naturel des fourmis dans le monde réel. Les vraies fourmis sont capables de trouver le chemin le plus court, à partir d'une source de nourriture pour arriver à leur fourmilière, en exploitant l'information de la phéromone. En marchant, les fourmis déposent de la phéromone sur le sol et elles suivent avec une certaine probabilité la phéromone déjà déposée par d'autres fourmis. Cette phéromone subit aussi un phénomène d'évaporation.

Le système des fourmis est un algorithme dans lequel un ensemble de fourmis artificielles coopère pour résoudre un problème, en interchangeant de l'information à travers la phéromone qui est déposée sur les arcs d'un graphe. La phéromone déposée sur les arcs joue le rôle d'une mémoire répartie à long terme. Cela permet une sorte de communication indirecte appelée *stigmergy*. AS a été principalement utilisé pour résoudre le problème du voyageur de commerce et le problème d'affectation quadratique. Pour une description complète de cette nouvelle méthode, nous suggérons [DG97].

1.6 Conclusion du chapitre

Nous voulons souligner l'effort effectué dans différentes lignes de recherche pour concevoir des algorithmes plus performants pour affronter la complexité des problèmes NP-complets. Chaque discipline a son espace et sa contribution. Par exemple, pour le problème du voyageur de commerce, Johnson et al. [JM97] ont évalué les algorithmes les plus répandus dans la littérature actuelle. Ils ont trouvé que parmi ces algorithmes,

9. Ant System en anglais

l'algorithme classique de Lin-Kernighan [LK73] donne les meilleurs résultats, pour les problèmes de grande taille d'un million de villes ou plus et aussi pour certaines instances réelles du TSP (dans le fichier TSPLIB). Ils ont montré que ni la recherche tabou, ni le recuit simulé, ni les algorithmes génétiques ne sont capables d'obtenir de résultats comparables pour cette taille du problème. Par contre, si le problème est de trouver des tours plus petits, le recuit simulé et les algorithmes génétiques peuvent trouver de meilleurs tours que ceux que trouve dans le même temps l'algorithme de Lin-Kernighan. Les approches avec métaheuristiques ont l'avantage supplémentaire de ne pas avoir besoin d'un code aussi compliqué que celui de Lin-Kernighan.

En ce qui concerne les problèmes de satisfiabilité (SAT), avant les années 90 la taille des problèmes résolus était de 20 à 50 variables avec 20 à 200 clauses. Maintenant, avec les nouvelles méthodes (telles que GSAT), on peut résoudre de problèmes entre 2000 à 20000 variables avec 10000 à 20000 clauses. Au début, la recherche avec GSAT était orientée vers la comparaison avec des problèmes générés d'une façon aléatoire, maintenant la recherche s'oriente vers la résolution de problèmes pratiques.

Pour les problèmes classiques des N-reines, Minton et al. [MJPL92] ont pu résoudre le problème avec 1000000 de reines dans un temps raisonnable, en utilisant leur heuristique de min-conflicts. Ils ont conclu que le problème des N-reines est donc un problème facile, bien qu'il soit impossible de le traiter par un algorithme de retour-en-arrière classique pour plus de cent reines. GSAT a pu aussi résoudre le problème des N-reines. Pour 100 reines codées avec 10000 variables et 1.6 10^6 clauses, il a trouvé la solution en 195 secondes.

Finalement, pour le problème de 3-coloriage Minton et al. [MJPL92] ont fait des expérimentations avec un nombre de variables entre 30 et 180 pour des graphes denses et peu denses, coloriables avec 3 couleurs. L'algorithme testé commence en appliquant l'algorithme classique de Brélaz pour trouver une solution initiale. Dans le cas où la solution initiale ne correspond pas à une solution du problème, il la répare en utilisant l'heuristique de min-conflits. Pour les graphes denses, l'heuristique de Brélaz a pu trouver la solution sans utiliser leur heuristique de réparation. En revanche, les graphes peu denses donnent plus de problèmes à l'heuristique pour trouver la solution. Plus particulièrement, en résolvant les graphes avec 30 variables, l'algorithme a mis

en évidence une dégradation des performances en utilisant l'heuristique de Brélaz comme procédure initiale, par rapport à l'algorithme pur, qui utilise l'heuristique de *min-conflits* avec une solution initiale générée aléatoirement.

Dans la catégorie des méthodes stochastiques, on peut inclure aussi les algorithmes évolutionnistes que nous décrivons dans le chapitre suivant. Ils ont été conçus pour réaliser une recherche stochastique pendant laquelle ils réparent des instanciations (exploration), mais ils ont aussi l'avantage d'être capables d'utiliser certaines informations accumulées pendant la recherche (exploitation), une sorte de mémoire implicite. Leur principale différence avec les méthodes citées jusqu'à présent est qu'ils font évoluer "une population" d'instanciations, ce qui leur permet d'explorer à la fois plusieurs zones de l'espace de recherche, en combinant les instanciations obtenues dans ces diverses zones.

Chapitre 2

Rapide survol des Algorithmes Génétiques

Les algorithmes génétiques appartiennent aux algorithmes basés sur la théorie de l'évolution nommés *algorithmes évolutionnistes (EA)*. Parmi eux, on trouve à présent trois autres lignes de recherche:

- La programmation évolutive (EP), proposée aux États Unis par L.J. Fogel, A.J. Owens and M.J.Walsh en [FOW66] et récemment affinée par D.B. Fogel en [Fog92].
- Les stratégies évolutives (ES) proposées en Allemagne par I. Rechenberg [Rec73] et H.P. Schwefel [Sch81].
- Le paradigme de la programmation génétique (GP) présenté aux États Unis par J.R.Koza [Koz89].

Chacune de ces quatre approches est basée sur le processus d'apprentissage collectif à partir d'une population d'individus, où chaque individu représente un point dans l'espace de recherche. La population initiale est générée d'une manière aléatoire. La population s'adapte à l'environnement en suivant un processus aléatoire de sélection, recombinaison et mutation. L'environnement évalue la qualité des individus et le processus de sélection préfère les individus de meilleure qualité. La recombinaison permet

l'échange d'information entre les individus et la mutation introduit une nouvelle information dans la population. En général, la qualité de la population augmente et on espère arriver à trouver l'optimum. Pour construire un algorithme évolutionniste, nous avons besoin de:

1. Une population initiale de pré-solutions,
2. Une représentation génétique pour les pré-solutions,
3. Des opérateurs génétiques qui font les transformations,
4. Une fonction d'évaluation,
5. Un algorithme de sélection,
6. Des paramètres: taille de la population, probabilités pour les opérateurs génétiques.

La figure 2.1 montre plus en détail la structure d'un algorithme évolutionniste.

```
Procédure Algorithme Évolutionniste
Début /* procédure Algorithme Évolutionniste */
 t = 0
 initialiser population P(t)                                  (1)
 évaluer les individus en P(t)                                (4)
 tant que (non condition de fin) faire
       t=t+1
       Parents = sélectionner parents à partir de P(t-1)      (5)
       Enfants = transformer Parents                          (3)
       P(t) = Enfants
       évaluer P(t)                                           (4)
Fin /* procédure Algorithme Évolutionniste */
```

FIG. 2.1 – *Structure d'un Algorithme Évolutionniste*

Les différences entre ces quatre lignes de recherche se situent au niveau plus profond de la conception de l'évolution, plus précisément au niveau de la représentation des individus, des mécanismes d'adaptation, de la fonction d'évaluation, du degré

d'utilisation des opérateurs de mutation et de recombinaison et du mécanisme de sélection. Pour une comparaison des différentes classes d'Algorithmes Évolutionnistes, le lecteur peut consulter [BS93].

2.1 L'origine des algorithmes génétiques

En 1975, John Holland publie [Hol75] "Adaptation in Natural and Artificial Systems". Son objectif était de mettre en évidence et d'expliquer rigoureusement les processus d'adaptation des systèmes naturels et de concevoir des systèmes artificiels (logiciels) qui utilisent ces mécanismes. Sa principale contribution était l'utilisation d'une chaîne de caractères binaires pour représenter des structures complexes et l'application de transformations pour les améliorer. L'algorithme génétique de Holland est une méthode pour évoluer depuis une population de "chromosomes" (chaîne de caractères de uns et zéros, ou bits) vers une nouvelle population en utilisant une sorte de "sélection naturelle" et en appliquant ensuite les opérateurs inspirés de la génétique: croisement, mutation. Chaque chromosome est composé par des gènes (bits), chaque gène est une instanciation d'un allèle (0 ou 1). Un chromosome ou individu représente une solution possible du problème à résoudre. Une population est formée par un ensemble d'individus généralement générés d'une façon aléatoire. Il s'agit d'un algorithme probabiliste qui maintient une population d'individus pour chaque génération. La façon d'évoluer est de sélectionner certains individus de la population "t" (les parents) et parmi eux d'en transformer quelques uns avec des opérateurs génétiques pour générer une nouvelle population "t+1" (les enfants) qui, nous espérons, aura des meilleurs individus que la population de la génération précédente. Chaque individu est évalué par une fonction d'évaluation, laquelle va nous permettre de guider la sélection des bons individus et aussi de savoir quand nous trouvons une solution. La figure 2.2 montre schématiquement, d'une façon très simplifiée, l'évolution d'une génération à la suivante. Nous avons un ensemble initial de pré-solutions dénommé Population Initiale. Chaque pré-solution est évaluée pour connaître sa qualité vis-à-vis de la fonction à optimiser. Ensuite, l'algorithme de sélection choisit un sous-ensemble de cette population. Ces chromosomes participeront à la reproduction. En moyenne,

les chromosomes les mieux adaptés se reproduisent plus souvent et contribuent davantage aux populations futures. Une partie des membres du sous-ensemble seront donc transformés et ils feront partie de la nouvelle population. Le processus finira une fois qu'il aura trouvé la solution au problème ou quand l'algorithme aura généré un nombre maximum de populations. Dans la même figure, nous montrons le processus de transformation standard. La transformation 1 prend une pré-solution sélectionnée et change aléatoirement quelques unes des valeurs des variables. Ce processus est la "mutation". La transformation 2 prend deux pré-solutions sélectionnées, et échange des sous-parties des deux chromosomes, comme une recombinaison biologique en choisissant un point de croisement d'une façon aléatoire. Ce processus est dénommé "croisement". Ensuite, il faut évaluer la nouvelle population.

Les algorithmes génétiques se sont révélés efficaces pour un grand ensemble de problèmes [Hol75]. De plus, ils n'étaient pas limités par des conditions analytiques sur l'espace de recherche telles que la continuité. Mais, Holland reconnaît que la non-linéarité ou la haute corrélation entre les variables représente un obstacle pour l'évolution.

Pour faire une recherche efficace, il faut trouver un équilibre entre découvrir une nouvelle connaissance et exploiter ce que nous connaissons déjà. Cela peut s'exprimer avec 2 concepts:

Définition 2.1.1 *(Exploration)*
*On dit qu'un algorithme réalise de l'*exploration *quand il cherche à découvrir de nouvelles parties de l'espace de recherche d'une façon aléatoire dans le but de trouver une solution.*

Définition 2.1.2 *(Exploitation)*
*On dit qu'un algorithme réalise de l'*exploitation *quand sa recherche vers la solution est guidée par les pré-solutions.*

Un algorithme qui ne fait que de l'*exploration* suit un comportement nettement offensif, mais en aveugle. Par contre, un algorithme qui fait que de l'*exploitation* sera plutôt conservateur. Holland a montré [Hol75], [Hol73] qu'un algorithme génétique peut garder un bon équilibre entre chercher dans l'espace de recherche pour découvrir

des nouvelles solutions (exploration), et utiliser comme l'*escalade* la connaissance acquise pendant la recherche (exploitation). On peut considérer en gros que l'opérateur de mutation fait plutôt de l'exploration et que le croisement fait de l'exploitation. Un algorithme génétique est basé sur la simple heuristique que les meilleures solutions se trouvent dans les régions de l'espace de recherche qui contiennent des bonnes solutions et que ces régions peuvent être identifiées. Dans la section suivante, nous allons présenter la façon de travailler d'un algorithme génétique standard.

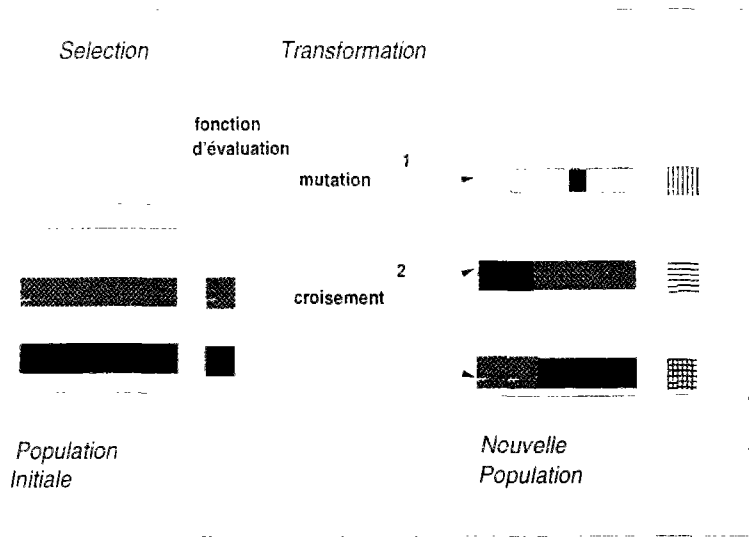


FIG. 2.2 – *Description d'un Algorithme Génétique*

2.2 Algorithme génétique standard

L'algorithme génétique standard [Gol89] développé à partir du travail de Holland, a trois opérateurs, reproduction, croisement et mutation. La structure de l'algorithme génétique standard est montrée sur la figure 2.3.

Le modèle le plus simple fait les hypothèses suivantes:

Définition 2.2.1 *(Chaîne de caractères binaire)*
Une chaîne de caractères binaires peut être représentée symboliquement par une chaîne de caractères C_{bin} *qui est composée de* l *entiers binaires.* $C_{bin} = b_1, \ldots, b_l$ *où* $b_i \in \{0, 1\}$ *pour* $i = 1, 2, \ldots, l$.

```
Procédure Algorithme Génétique Standard
Début
t = 0
Initialiser population P(t)
Évaluer les individus en P(t)
tant que (non condition de fin) faire
      t=t+1
      tant que (P(t) n'est pas complète) faire
            Parent_1 = sélection[a](P(t-1))
            Parent_2 = sélection(P(t-1))
            Enfants = transformation(Parent_1, Parent_2)
            P(t) = Enfants ∪ P(t)
      évaluer P(t)
Fin /* procédure Algorithme Génétique Standard */
```

[a] algorithme de sélection défini sur la section 2.2.1

FIG. 2.3 – *Structure de l'Algorithme Génétique Standard*

Les chaînes de caractères binaires sont de taille fixe.

Définition 2.2.2 *(Population de chaînes de caractères)*
À l'instant "t" (ou génération) il y a une population $P(t)$ de n chaînes de caractères binaires C^p_{bin} où $p = 1, 2, \ldots, n$.

Définition 2.2.3 *(Évaluation d'une chaîne de caractères)*
L'adéquation d'une chaîne de caractères est la valeur de la fonction d'optimisation nommée pour un algorithme génétique fonction d'évaluation.

Chaque chaîne de caractères a une capacité relative (adéquation) pour survivre et pour produire des enfants. Cette adéquation sera mesurée en utilisant la fonction à optimiser ou fonction d'évaluation (def 2.2.3). En plus de la fonction d'évaluation, un algorithme génétique a besoin d'un processus de sélection et de transformation.

2.2.1 Algorithme de sélection

Pour un algorithme génétique standard, la sélection est basée sur les valeurs d'une fonction d'évaluation a maximiser, de la manière suivante. Il construit une roue de loterie biaisée avec des sous-parties d'une taille proportionnelle à leur évaluation comme cela est montré sur la figure 2.5. La construction de la roue de loterie est faite de la façon suivante:

- Évaluer chaque chromosome $v_i (i = 1, \ldots, taille - pop)$
- Calculer la fonction d'évaluation totale de la population

$$F = \sum_{i=1}^{poptaille} eval(v_i), (i = 1, \ldots, taille - pop) \tag{2.1}$$

- Calculer la probabilité de sélection p_i pour chaque chromosome v_i en fonction de son évaluation (les meilleurs ont une probabilité plus forte d'être sélectionnés)

$$p_i = \frac{eval(v_i)}{F}, (i = 1, \ldots, taille - pop) \tag{2.2}$$

- Ordonner les chromosomes selon leur valeur p_i
- Calculer une probabilité cumulative q_i pour chaque chromosome v_i

$$q_i = \sum_{j=1}^{i} p_j, (i = 1, \ldots, taille - pop) \tag{2.3}$$

Le processus de sélection est basé sur cette roue, il choisit chaque fois un individu parmi les chromosomes de la population, comme cela est décrit dans la figure 2.4.

2.2.2 Transformation: opérateurs standards

L'algorithme génétique standard utilise l'opérateur de *croisement à un point* montré dans la figure 2.7 et l'opérateur de *mutation* de la figure 2.8 pour réaliser la transformation des chaînes de caractères binaires (chromosomes). La procédure de transformation standard est montrée sur la figure 2.6. Pour le croisement, deux individus

Procédure sélection roue de loterie biaisée (Population)
Début
Générer un nombre aléatoire r dans l'intervalle [0..1]
si $r < q_1$ **alors**
  sélectionner le premier chromosome (v_1)
sinon
  sélectionner le ième chromosome $v_i (2 \leq i \leq taille - pop)$
  tel que $q_{i-1} < r \leq q_i$
Fin /* sélection roue de loterie biaisée */

FIG. 2.4 – *Structure de la procédure roue de loterie biaisée*

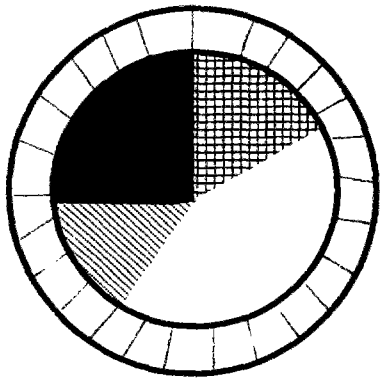


FIG. 2.5 – *Exemple: Roue de loterie utilisée pour sélectionner 4 individus avec des parties proportionnelles à leur évaluation*

ont la probabilité $Prob_c$ d'être croisés. Après avoir sélectionné à partir de la population $P(t)$ deux individus (parents), qui seront croisés, il tire un nombre aléatoire qui correspond à la position de la coupure en deux parties des parents. Les deux enfants seront formés à partir de la première partie d'un des parents et la deuxième partie de l'autre parent. Le concept plus général pour le croisement est la recombinaison définie comme:

Définition 2.2.4 *(Recombinaison)*
On dira qu'un fils est le résultat d'une recombinaison *de gènes quand il hérite ses gènes à partir de ses parents.*

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

FIG. 2.6 – *Structure de la procédure de transformation standard*

Pour la mutation, chaque allèle a la même probabilité $Prob_m$ d'être changé. La nouvelle valeur pour un gène qui sera muté sera aussi choisie aléatoirement. Nous pouvons regarder dans la figure 2.9 la façon d'évoluer en utilisant ces opérateurs. Dans cette figure, nous avons une population initiale $P(0)$ avec cinq chromosomes. Nous faisons une sélection depuis $P(0)$ de deux chromosomes qui ont une chance d'être croisés égale à $Prob_c$. S'ils ne sont pas croisés, ils passent directement à l'étape de mutation. S'ils sont croisés, ils seront remplacés par leurs enfants obtenus de leur échange de gènes, ensuite ces enfants passeront à la procédure de mutation. Une fois la procédure de mutation finie, les enfants résultants seront incorporés comme membres de la nouvelle population $P(1)$. Ce processus est répété jusqu'à compléter la taille de la population, jusqu'à avoir cinq enfants. Nous pouvons remarquer qu'un individu a une chance de rester lui-même dans la génération suivante. D'abord, il a une probabilité $(1 - Prob_c)$ de ne pas être croisé, et ensuite une probabilité $(1 - Prob_m)$ de ne pas subir une mutation.

En résumé, dans un algorithme génétique standard, les chromosomes ont une représentation binaire. Chacun est évalué par l'application de la fonction à optimiser. La sélection est faite en utilisant une roue de loterie biaisée et la transformation est réalisée en utilisant les deux opérateurs standards: *croisement à un point* et *mutation*. Les concepts semblent faciles à comprendre, en revanche nous imaginons que le lecteur se pose la même question que nous: comment les algorithmes peuvent-ils vraiment arriver à trouver une solution pour un problème d'optimisation combinatoire? Pour y

Procédure croisement à un point $(Parent_1, Parent_2)$
Début
Générer un nombre aléatoire r dans l'intervalle [0..1]
si r <probabilité de croisement **alors**
  Générer un nombre aléatoire pos dans l'intervalle [1..n-1]
  Remplacer les Parents
    $(b_1, b_2, \ldots, b_{pos}, b_{pos+1}, \ldots, b_n)$ $(c_1, c_2, \ldots, c_{pos}, c_{pos+1}, \ldots, c_n)$ par
    $(b_1, b_2, \ldots, b_{pos}, c_{pos+1}, \ldots, c_n)$ $(c_1, c_2, \ldots, c_{pos}, b_{pos+1}, \ldots, b_n)$
Fin /* croisement à un point */

FIG. 2.7 – *Structure de la procédure croisement à un point*

Procédure mutation (Chromosome)
Début
Pour chaque chromosome après croisement à un point
  Pour chaque gène
    Générer un nombre aléatoire r dans l'intervalle [0..1]
    si r <probabilité de mutation **alors**
      changement du gène
Fin /* procédure mutation */

FIG. 2.8 – *Structure de la procédure mutation*

répondre, nous aborderons dans la section suivante la base théorique des algorithmes génétiques.

2.2.3 Pourquoi les algorithmes génétiques marchent?

Un des concepts centraux développés dans l'analyse théorique des algorithme génétiques est le concept de "schéma" [Hol75]. Pour comprendre cette notion, nous devons adopter le point de vue suivant: pour guider sa recherche, un algorithme génétique qui agit sur une population traite en fait des entités appelées "schémas" et tente de découvrir les meilleurs schémas et de les combiner.

Définition 2.2.5 *(Schéma)*
Un schéma H décrit un sous-ensemble de chromosomes ayant des caractéristiques communes à certaines positions de la chaîne de caractères.

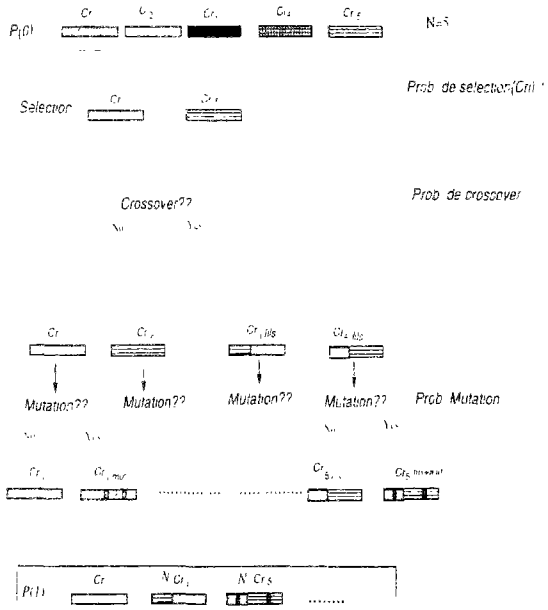


FIG. 2.9 – *Exemple avec des opérateurs génétiques standards*

C'est donc une hyper-surface particulière de l'espace de recherche. L'ensemble de tous les schémas couvre la totalité de l'espace de recherche avec une redondance élevée. Un schéma H peut être défini sur l'alphabet $(0, 1, \#)$, où le métasymbole "#" est le symbole de l'indifférence. Par exemple, le schéma $1\#\#\#0$ désigne tous les chromosomes qui commencent par 1 et qui finissent par 0. En conséquence, tout chromosome de longueur l est un représentant de 2^l schémas. La fonction d'évaluation ou performance d'un schéma H, $f(H)$, correspond à la valeur moyenne de la fonction d'évaluation du sous-ensemble de chromosomes qu'il décrit. Certains schémas sont plus spécifiques que d'autres. Par exemple, le schéma $01\#\#1$ est plus spécifique que le schéma $1\#\#\#0$. De plus, certains schémas couvrent une plus grande partie de la longueur totale de la chaîne de caractères que d'autres. Par exemple, le schéma $1\#\#\#0$ couvre une plus grande portion de la chaîne que $1\#0\#\#$. Pour quantifier ces notions, il est nécessaire de définir deux caractéristiques propres aux schémas: *ordre* et *longueur utile*

Définition 2.2.6 *(Ordre d'un schéma)*
L'ordre $o(H)$ d'un schéma H est le nombre de positions fixes (le nombre d'uns et de

zéros) dans H. Par exemple $o(\#1\#00) = 3$.

Définition 2.2.7 *(Longueur utile d'un schéma)*
La longueur utile $\delta(H)$ *d'un schéma H est la distance entre la première et la dernière position fixée dans H, donc correspond à l'écart maximal entre deux bits fixés. Par exemple,* $\delta(\#1\#00) = 5 - 2 = 3$.

Le concept de schéma et les propriétés d'ordre et de longueur permettent d'analyser l'effet de la reproduction sur le nombre attendu de schémas d'un certain type dans la population. Supposons qu'à l'instant t, il y ait m exemplaires d'un schéma H contenus dans la population $P(t)$ dénoté par $m(H,t)$ et soit $\hat{f}(H)$ la moyenne des adéquation des m individus. Lors de la reproduction, un chromosome C'_{bin} est copié en fonction de son adéquation (sa valeur de la fonction d'évaluation) avec la probabilité $Prob_p = \frac{f_p}{f_{moyen}}$, où:

$$f_{moyen} = \frac{(\sum_{p=1}^{n} f_p)}{n} \tag{2.4}$$

Soit n la taille de la population depuis la population $P(t)$, soit $m(H, t+1)$ l'espérance de nombre de représentants du schéma H dans la population à l'instant $t+1$. L'équation de développement par reproduction des schémas (avant de faire une transformation) est la suivante:

$$m(H, t+1) \geq m(H,t)\frac{\hat{f}(H)}{f_{moyen}} \tag{2.5}$$

Le développement d'un schéma dépend du rapport de la fonction d'évaluation du schéma dans la population sur la valeur moyenne de la fonction d'évaluation de la population. Les schémas au-dessus de la moyenne auront un nombre de représentants croissant exponentiellement, tandis que les schémas dont les valeurs sont en-dessous de la moyenne seront moins copiés. La survie d'un schéma dépend aussi du croisement et de la mutation. Considérons l'opérateur de croisement à un point qui choisit aléatoirement un point où couper les chaînes de caractères des parents. Il y a $l-1$ points possibles de croisement pour choisir. La probabilité qu'un schéma H dépérisse dépendra de son ordre $o(H)$ et de sa longueur utile $\delta(H)$. Plus les positions d'un

schéma sont fixées et plus elles se trouvent éloignées, plus sa probabilité de disparition augmente. La probabilité de disparition d'un schéma est $Prob_d = \frac{\delta(H)}{(l-1)}$, donc sa probabilité de survie est $Prob_s = (1 - Prob_d)$. Une borne inférieure de survie après croisement pour un schéma H est:

$$Prob_s \geq 1 - Prob_c \frac{\delta(H)}{l-1} \tag{2.6}$$

où $Prob_c$ est la probabilité d'effectuer un croisement.

Pour qu'un schéma H survive, toutes ses positions fixées doivent survivre. Si on considère la mutation, chaque allèle survit avec la probabilité $(1 - Prob_m)$, où $Prob_m$ est la probabilité de mutation. Un schéma quelconque survit quand chacune des $o(H)$ positions fixées dans le schéma n'a pas subi un changement, puisque chaque mutation est indépendante. En multipliant la probabilité de survie par elle-même $o(H)$ fois, on obtient la probabilité de la survie à la mutation $(1 - Prob_m)^{o(H)}$. Quand $Prob_m \ll 1$ la probabilité peut être approximée par $1 - o(H)Prob_m$, [Gol89]. En général, la probabilité de mutation dans un algorithme génétique est assez petite, cette condition est donc valable. En considérant les effets du croisement et de la mutation ensemble, nous obtenons l'expression suivante pour la survie d'un schéma H:

$$Prob_s \geq 1 - Prob_c \frac{\delta(H)}{l-1} - o(H)Prob_m \tag{2.7}$$

Finalement pour être plus précis, nous incorporons ces résultats dus aux opérateurs, dans l'équation 2.5, et nous obtenons le Théorème Fondamental des Algorithmes Génétiques, [Gol89]:

Théorème 2.2.1 *(Théorème des Schémas)*

$$m(H, t+1) \geq m(H, t) \frac{\hat{f}(H)}{f_{moyen}} \left(\frac{\delta(H)}{1 - Prob_c l} - (1 - o(H)Prob_m) \right) \tag{2.8}$$

Hypothèse 2.2.1 *(Blocs de Construction)*

Un algorithme génétique cherche à accroître sa performance près de l'optimum en utilisant la juxtaposition des schémas ayant une fonction d'évaluation au-dessus de la moyenne, courts, et peu spécifiques ($o(H)$ petit). Ces schémas sont dénommés blocs de construction.

En utilisant le Théorème des Schémas nous pouvons prévoir que le nombre d'occurrences d'un schéma court et performant augmentera en moyenne d'une manière exponentielle d'une génération à la suivante.

Définition 2.2.8 *(Parallélisme Implicite)*
L'évaluation de la performance d'un chromosome est également l'évaluation de tous les schémas auxquels il appartient.

L'augmentation exponentielle des blocs de construction est dénommée "parallélisme implicite". Holland [Hol75] a estime qu'avec une population de taille "n" pour chaque génération, un algorithme génétique utilise implicitement $O(n^3)$ schémas.

Malheureusement, l'hypothèse des blocs de construction pour certains problèmes est facilement violée. Prenons un problème qui a les deux schémas courts: S_1 = (111##########) et S_2 = (##########11). Supposons qu'ils sont au-dessus de la moyenne et que leur combinaison S_3 = (111########11) est moins adéquate que S_4 = (000########00). De plus, supposons que la chaîne optimale est (11111111111). Un algorithme génétique peut avoir quelques difficultés pour converger vers la solution, car il peut avoir tendance à converger vers des points comme (00011111100). Ce type de phénomène est dénommé déception, [Bet80], [Gol89]: Certains blocs de construction (courts et peu spécifiques) peuvent mal guider l'algorithme génétique et produire une convergence vers des points sous-optimaux.

Le Théorème des Schémas aide à comprendre comment les algorithmes génétiques travaillent, mais il ne constitue pas une preuve du succès et il n'existe pas encore d'analyse stochastique rigoureuse sous des circonstances réelles (telles qu'une population finie, une fonction d'évaluation non triviale, des opérateurs spécialisés). Rabinovich en [RW91] a prouvé la convergence des algorithmes génétiques vers un optimum global, d'une façon non triviale et développée sous un cadre très simplifié et idéalisé (par exemple quand l'évaluation d'un chromosome est le nombre d'uns qu'il contient).

La Théorie des Formae est une généralisation récente de la Théorie des Schémas qui remplace le schéma par des sous-ensembles de l'espace de recherche appelés *formae*. Elle peut-être utilisée pour l'analyse des algorithmes génétiques qui utilisent différents types de représentation (non-binaires) et d'autres types d'opérateurs que

l'opérateur de croisement à un point et la mutation classique. Elle a été presentée par N. Radcliffe en [Rad90], [Rad91]. La Théorie des Formae ne fait mention d'aucune représentation particulière et elle permet aux opérateurs d'opérer sur un espace de recherche abstrait. Pour elle, le choix de la représentation ne concerne que l'implémentation. La tache de trouver une représentation binaire appropriée de l'espace de recherche est remplacée pour la définition des *formae*, qui sont des ensembles de solutions qui partagent une certaine propriété, que l'on suppose être pertinente pour l'évaluation des solutions. La Théorie des Formae donne quelques guides pour les propriétés que devraient avoir les opérateurs par rapport à ces *formae*. Les guides peuvent être utilisés pour examiner les heuristiques des opérateurs. La Théorie des Formae suggère aussi des opérateurs standards (qui dépendent de l'ensemble des formae). Pour une discussion approfondie de cette théorie, nous suggérons l'article de Radcliffe [Rad92] et pour l'application de cette même théorie, le lecteur peut consulter [Hof93], où Hofmann a montré comment utiliser ce formalisme dans l'analyse d'opérateurs génétiques pour le problème du voyageur du commerce.

2.3 Autres opérateurs de croisement

En plus de l'opérateur de *croisement à un point* illustré dans la figure 2.7, d'autres opérateurs ont été proposés dans la littérature. Le *croisement à un point* est inspiré par le processus biologique, mais il présente certains inconvénients. Par exemple, supposons que pour un problème donné les schémas $S_1 = (001\#\#\#\#\#\#\#\#01)$ et $S_2 = (\#\#\#\#11\#\#\#\#\#\#\#)$ sont performants. De plus, supposons qu'il y a deux individus dans la population (0010001101001) et (1110110001000) qui coïncident respectivement avec les schémas S_1 et S_2. Il est évident que le croisement ne peut pas réaliser certaines combinaisons entre eux, par exemple il est impossible qu'avec le *croisement à un point* nous puissions obtenir un chromosome qui coïncide avec le schéma $S_3 = (001\#11\#\#\#\#01)$, puisque le premier schéma sera détruit. Une modification de cet opérateur est le *croisement à deux points*. La procédure est montrée sur la figure 2.10. Il utilise deux positions de croisement. Dans l'exemple, il pourrait fournir le chromosome (0010110001001).

Procédure croisement à deux points($Parent_1, Parent_2$)
Début
Générer un nombre aléatoire r dans l'intervalle [0..1]
si r <probabilité de croisement **alors**
Générer un nombre aléatoire $pos_1 \in [1..n-1]$
Générer un nombre aléatoire $pos_2 \in \{\{1..n-1\} - pos_1\}$
Remplacer les Parents
$(b_1, b_2, \ldots, b_{pos_1}, b_{pos_1+1}, \ldots, b_{pos_2}, b_{pos_2+1}, \ldots, b_m)$
$(c_1, c_2, \ldots, c_{pos_1}, c_{pos_1+1}, \ldots, c_{pos_2}, c_{pos_2+1}, \ldots, c_m)$ par
$(b_1, b_2, \ldots, b_{pos_1}, c_{pos_1+1}, \ldots, c_{pos_2}, b_{pos_2+1}, \ldots, b_m)$
$(c_1, c_2, \ldots, c_{pos_1}, b_{pos_1+1}, \ldots, b_{pos_2}, c_{pos_2+1}, \ldots, c_m)$
Fin /* croisement à deux points */

FIG. 2.10 – *Structure de la procédure croisement à deux points*

De même que le *croisement à un point*, lui non plus ne peut pas réaliser certaines combinaisons. C'est pour cela que les chercheurs ont aussi expérimenté des *croisements à points multiples*. Une généralisation du *croisement à un point, à deux points* et *à points multiples* est le *croisement uniforme* défini par Syswerda en [Sys89]. Le *croisement uniforme* est défini comme suit: pour chaque position dans le premier enfant, on décide (avec une certaine probabilité p) de quel parent il hérite la valeur pour cette position. Le deuxième enfant recevra donc la valeur de l'autre parent.

Dans la section suivante, nous montrerons que l'utilisation d'un bon opérateur de croisement est encore plus crucial quand il s'agit d'un problème avec contraintes.

2.4 Les algorithmes génétiques et les problèmes d'optimisation avec contraintes

Le phénomène de déception est fortement lié au concept d'*épistasie*. En génétique, un gène est appelé épistatique, par rapport à un autre gène s'il cache l'expression phénotypique du deuxième gène, [Sti68]. De cette manière l'*épistasie* exprime le lien entre des gènes qui se trouvent séparés dans un chromosome. La même notion dans le contexte des algorithme génétiques a été présentée par Rawlins en [Raw91]. Il

définit qu'un degré minimum d'epistasie correspond à la situation où chaque gène (ou bit) est indépendant de chaque autre gène. De la même manière, l'epistasie est maximale quand aucun sous-ensemble de gènes n'est indépendant d'un autre gène. Plus formellement

Définition 2.4.1 *(Epistasie)*
L'épistasie est la façon dont un gène modifie l'expression phénotypique d'un autre.

En d'autres mots, l'épistasie mesure l'importance des relations entre les valeurs des gènes. Pour un problème donné, un haut degré d'épistasie signifie que les blocs de construction ne peuvent pas se former, le problème est donc sujet aux phénomènes de déception décrit dans la section précédente. Le travail de Holland peut être utilisé avec succès sur les problèmes avec une faible épistasie [Dav85]. Dans le but d'utiliser le concept d'epistasie comme un guide pour déterminer la difficulté pour un algorithme génétique d'optimiser une fonction, Davidor en [Dav90] a proposé d'exprimer la notion d'epistasie pour une chaîne de bits $s = s_0 ... s_{l-1}$ dans l'espace de recherche $\Omega = \{0,1\}^l$ par:

$$\epsilon(s) = f(s) - \sum_{i=0}^{l-1} \frac{1}{2^{l-1}} \sum_{t \in \Omega, t_i = s_i} f(t) + \frac{l-1}{2^l} \sum_{t \in \Omega} f(t). \tag{2.9}$$

qui correspond à la différence de comportement entre l'évaluation et l'évaluation linéarisée au premier ordre. Pour la démonstration le lecteur peut consulter [Dav90], et pour une discution sur la relation entre épistasie et déception voir [NV97].

Les problèmes avec de contraintes rentrent dans la catégorie des problèmes avec un grand niveau d'épistasie. Chaque contrainte correspond à une relation qui représente en quelque sorte une "dépendance" ou un "compromis" entre ses variables pertinentes. Les problèmes avec contraintes ont donc besoin de modifier l'approche standard, parce qu'elle ne prévoit aucun mécanisme pour gérer la satisfaction des contraintes, [Gol89]. Parmi les problèmes d'optimisation avec contraintes on trouve des problèmes continus et discrets. Par exemple, le problème du voyageur de commerce, certains problèmes de planification et le problème de k-coloriage de graphe rentrent dans la catégorie de problèmes d'optimisation combinatoire (avec des domaines discrets). Le

cas le plus général d'optimisation avec des domaines continus est le *problème général de programmation non-linéaire* (NLP). Un NLP cherche trouver $\vec{x}$, tel que

$$\text{optimise } f(\vec{x}), \vec{x} = (x_1, \ldots, x_n) \in \mathcal{R}^n \tag{2.10}$$

où $\vec{x} \in \mathcal{F} \subseteq \mathcal{S}$. La fonction d'évaluation f est défini sur l'espace de recherche $\mathcal{S} \subseteq \mathcal{R}^n$ et l'ensemble $\mathcal{F} \subseteq \mathcal{S}$ define la région possible. Normalement, l'espace de recherche $\mathcal{S}$ est définit comme un rectangle de n dimensions sur $\mathcal{R}^n$ (les domaines des variables étant définis par leur borne supérieures et inférieures):

$$l(i) \leq x_i \leq u(i), 1 \leq i \leq n. \tag{2.11}$$

où la région possible $\mathcal{F} \subseteq \mathcal{S}$ est définie par un ensemble de m contraintes supplémentaires ($m \geq 0$):

$$g_j(\vec{x}) \leq 0, \text{ pour } j = 1, \ldots, q, \text{ et } h_j(\vec{x}) = 0, \text{pour } j = q+1, \ldots, m. \tag{2.12}$$

Pour tout $\vec{x} \in \mathcal{F}$, les contraintes g_k qui satisfont $g_k(\vec{x})$ sont appelées les contraintes actives de $\vec{x}$. Par extension, les contraintes d'égalités h_j sont aussi appelées actives sur tous les points de $\mathcal{S}$. Selon les caractéristiques des fonctions h_j et g_j le problème NLP peut être classé comme un problème d'optimisation linéairement contraint si g_j et h_j sont linéaires. Si, en plus, la fonction f est polynomiale au plus quadratique, le problème est un problème de programmation quadratique. Dans ce cas, si la fonction f est linéaire, il s'agit du problème bien connu de programmation linéaire. Dans le cas où $m = 0$, donc $\mathcal{F} = \mathcal{S}$ le problème est un problème d'optimisation non-contraint.

Dans la littérature, plusieurs méthodes qui incorporent les contraintes dans les algorithmes génétiques pour la résolution des problèmes d'optimisation combinatoire avec contraintes [War95], et pour NLP [MS96] ont été proposées. Elles peuvent être classifiées dans 4 catégories:

- Méthodes basées sur les fonctions de pénalisation
- Méthodes basées sur la préservation de la faisabilité des solutions
- Méthodes qui font une distinction claire entre solutions faisables et non-faisables

- D'autres méthodes hybrides.

Les deux premières catégories de méthodes sont les plus utilisées. La première ajoute à la fonction objectif une fonction de pénalisation, qui attribue un coût à la violation d'une contrainte, cette pénalisation peut être entre autres statique [HLQ94], dynamique [JH94], adaptative [ST93]. La seconde méthode modifie les opérateurs pour empêcher la production de solutions qui violent les contraintes, par exemple en faisant des réparations du chromosome ou en concevant des opérateurs génétiques qui travaillent implicitement avec les contraintes [MC91]. Les deux méthodes ont des avantages et des inconvénients. Si on utilise une grande pénalisation dans la fonction d'évaluation et s'il s'agit d'un problème qui a une grande probabilité de production d'individus qui violent des contraintes, on passera une grande partie du temps à évaluer des individus qui ne satisfont pas les contraintes. D'un autre côté, supposons qu'on trouve un individu qui satisfasse toutes les contraintes. L'algorithme pourrait converger vers lui sans avoir assez exploré l'espace pour trouver de meilleurs individus. Les trajectoires vers d'autres individus qui satisfont aussi les contraintes passent à travers la production d'individus illégaux comme structures intermédiaires et les pénalités peuvent rendre improbable la reproduction de ces structures. Si on impose des pénalités modérées, le système peut produire des individus qui violent les contraintes mais qui sont meilleurs que les autres qui les satisfont, parce que le reste de la fonction d'évaluation est mieux évalué. La troisième méthode inclut la conception de "décodeurs" incorporés dans la procédure d'évaluation (ou dans le processus de transformation), qui d'une façon intelligente soient capables d'empêcher la création d'un individu qui ne satisfait pas les contraintes. Malheureusement, le résultat de leur utilisation est fréquemment très coûteux en temps d'exécution. De plus, toutes les contraintes ne peuvent pas être facilement implémentées de cette manière [MN95].

Le problème de la prise en compte de contraintes dans les algorithmes génétiques est encore un sujet de recherche actuel, et l'efficacité de l'algorithme dépend beaucoup du problème à traiter, et aussi du type de représentation choisie pour les individus et par conséquent aussi des opérateurs génétiques utilisés. Michalewicz et Schoenauer ont fait une analyse des diverses méthodes existantes en [MS96]. Ils ont trouvé qu'il est difficile de réaliser une comparaison, car dans les formulations à l'origine de

ces méthodes, différents types de représentation (binaire ou réel) sont utilisés, et en conséquence elles utilisent différents opérateurs. Ils ont conclu que pour les problèmes d'optimisation avec contraintes, utiliser une représentation non-binaire (réelle) donne de meilleurs résultats. Enfin, ils ont signalé que le choix de la méthode à utiliser est encore un problème ouvert.

2.5 Les algorithmes génétiques et les problèmes de satisfaction de contraintes

2.5.1 Fonction d'évaluation

Un CSP est un problème où nous n'avons pas explicitement une fonction à optimiser, donc pour le résoudre en utilisant un algorithme génétique, nous avons besoin de définir une fonction d'évaluation adaptée.

Définition 2.5.1 *(Fonction D'Évaluation Standard pour CSP)*
Étant donné un CSP $P = (V, D, \zeta)$, *une instanciation* $\mathbf{I}$ *et* ψ *la fonction indicatrice telle que*

$$\psi = \psi(C_\alpha, \mathbf{I}) = \begin{cases} 0 & Si\ \mathbf{I} \models C_\alpha \\ 1 & sinon \end{cases}$$

On définit la Fonction d'Évaluation Standard pour un CSP *par:*

$$Z_{std}(\mathbf{I}) = \sum_{j=1}^{n} \psi(C_j, \mathbf{I}) \tag{2.13}$$

Typiquement, la fonction d'évaluation est définie comme le nombre de contraintes violées (ou le nombre de contraintes satisfaites), donc nous cherchons à minimiser (maximiser) cette fonction, dont le minimum (maximum) sera obtenu quand toutes les contraintes seront satisfaites [Tsa90]. D'autres fonctions d'évaluation ont été proposées dans la littérature. Par exemple, une fonction qui pénalise les violations en affectant des poids différents à chaque contrainte violée [ERR96] peut s'exprimer par:

Définition 2.5.2 *(Fonction d'évaluation pénalisante)*
Étant donné un CSP $P = (V, D, \zeta)$, une instanciation $\mathbf{I}$ *et* ψ *la fonction indicatrice de la définition 2.5.1. On définit la* Fonction d'évaluation pénalisante *par:*

$$Z_w(\mathbf{I}) = \sum_{j=1}^{\eta} w_j \psi(C_j, \mathbf{I}) \tag{2.14}$$

où w_j est le poids pour la contrainte C_j qui sera considérée dans la fonction d'évaluation quand elle n'est pas satisfaite.

Cette fonction d'évaluation est formulée du point de vue des contraintes. Un autre type de fonction d'évaluation prend en compte plus les variables que les contraintes. La fonction est alors basée sur l'évaluation des valeurs des variables qui ne font pas partie de la solution, c'est à dire les variables dont les valeurs violent au moins une contrainte. Nous pouvons exprimer plus formellement cette idée par:

Définition 2.5.3 *(Fonction d'évaluation basée sur les variables)*
Étant donné un CSP $P = (V, D, \zeta)$, une instanciation $\mathbf{I}$ *et C^i l'ensemble des contraintes qui ont comme variable pertinente $X_i, \forall i \in \{1..n\}$. On définit la* Fonction d'évaluation basée sur les variables *par:*

$$Z_v(\mathbf{I}) = \sum_{i=1}^{n} \mathcal{U}(X_i, \mathbf{I}) \tag{2.15}$$

où

$$\mathcal{U} = \mathcal{U}(X_i, \mathbf{I}) = \begin{cases} 0 & Si\ \mathbf{I} \models C^i \\ u_i & sinon \end{cases}$$

où u_i correspond à la pénalisation pour la variable X_i. Cette pénalisation sera prise en compte quand au moins une contrainte, parmi celles appartenant l'ensemble C^i n'est pas satisfaite.

La difficulté d'utiliser ces deux dernières définitions est l'estimation des poids (w_j ou u_i).

Une autre idée pour définir une fonction d'évaluation est de permettre le changement dynamique des poids w_j de la définition 2.5.2 ou u_i de la définition 2.5.3. L'idée est d'incorporer à l'algorithme génétique un mécanisme adaptatif qui "syntonise" les poids w_j ou u_i périodiquement, en suivant le déroulement de la recherche. L'analyse

est faite sur le meilleur des individus. Quand il ne satisfait pas une contrainte C_j ou quand une variable X_i est instanciée avec une valeur qui viole quelques contraintes, alors le poids w_j ou u_i est augmenté. La différence parmi les méthodes qui changent dynamiquement les poids de la pénalisation est le moment où elles appliquent le changement et aussi son importance [EvdH97], [Par94]. Pour plus de détails, le lecteur peut consulter [Eib96] où Eiben a fait une description d'autres fonctions d'évaluation qui ont été proposées dans la littérature pour résoudre les CSP.

2.5.2 Des opérateurs spécialisés pour les CSP

Différentes approches génétiques pour la résolution de CSP sont proposées dans la littérature, [Par94], [ERR94], [ERR95], [ERR96], [DBB94], [Tsa90], [FF95]. La plupart ont cherché à définir des nouveaux opérateurs qui permettent de guider la recherche stochastique pour obtenir une recherche plus performante. Ils essayent aussi d'empêcher l'algorithme de tomber dans un optimum local, c'est-à-dire, dans le contexte des CSP de l'empêcher de rester bloqué sur une instanciation qui ne satisfait pas toutes les contraintes du problème. Nous présentons ci-après quelques opérateurs parmi les plus connus et représentatifs pour la résolution de CSP, sans autre prétention que d'illustrer la recherche dans le domaine. Le but des trois opérateurs que nous allons décrire est d'affronter le problème d'épistasie dans les CSP. Cette caractéristique fait que les opérateurs standards de croisement ne sont plus efficaces, car dans ce cas l'ordre des variables dans le chromosome a une grande importance. Du moment où il y a deux variables qui sont fortement liées (par exemple par une contrainte) la valeur de l'une est "conditionnée" à la valeur de l'autre variable. Imaginons que la variable X_1 est fortement liée à la variable X_2 et qu'elles sont positionnées l'une à côté de l'autre dans le chromosome. De plus, supposons que la valeur 1 pour X_1 est incompatible avec la valeur 2 pour X_2, alors la probabilité de disparition du schéma 12## avec le *croisement à un point* sera inférieure à celle dans le cas où les deux variables se trouvent dans des positions plus écartées, par exemple dans un schéma 1##2. où la variable X_1 se trouve à la première position et la variable X_2 à la quatrième position du chromosome, (voir Théorème 2.2.1). En conséquence, l'algorithme

convergera plus lentement, ou il restera avec plus de probabilité dans un optimum local. De telles considérations sont prises en compte dans la conception des opérateurs spécialisés pour les CSP.

UAX : Uniform Adaptive Crossover

Uniform Adaptive Crossover (UAX) est proposé par Warwick et Tsang en [WT94] pour résoudre les problèmes PCSP (Partial Constraint Satisfaction Problem, [FW92]), c'est-à-dire des problèmes où on cherche à trouver une solution qui peut ne pas satisfaire toutes les contraintes et qui satisfait donc un sous-ensemble de contraintes du problème. Leur idée était de produire des points de croisement multiples pour affronter la caractéristique d'épistasie de ce type de problèmes et en conséquence d'augmenter la probabilité de convergence de leur algorithme. Ils arrivent à produire une certaine indépendance de l'ordre dans lequel se trouvent organisées les variables dans le chromosome. Pour cela, UAX ajoute à la représentation d'un chromosome une chaîne supplémentaire de bits. Chaque individu aura donc une représentation comme celle-ci:

Chromosome	1	2	3	4	2	5	4	1	3	5
Chaîne Suppl.	0	1	0	0	1	1	0	0	1	0

Elle correspond à une représentation non-binaire des chromosomes. Cette chaîne supplémentaire est conçue pour réaliser le contrôle de la création des enfants pendant le processus de croisement. UAX génère un enfant à partir de deux parents qui sont d'abord sélectionnés aléatoirement dans la population. Le fils au début hérite les bits d'un père, lequel est choisi aléatoirement parmi les deux parents. Les chaînes des parents (chromosomes) sont coupées aux points où les valeurs des chaînes binaires ont la même valeur et l'algorithme réalise une répartition alternée des valeurs, comme cela est illustré par un exemple sur la figure 2.11.

La structure de UAX est montrée sur la figure 2.12.

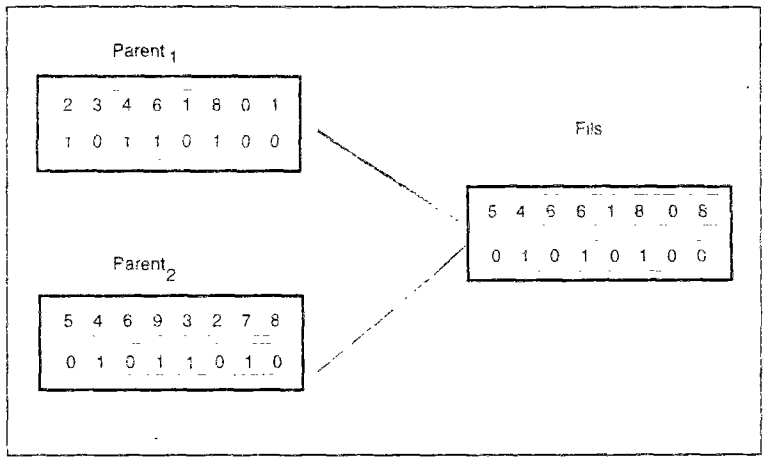


FIG. 2.11 - *UAX: Un exemple avec $Parent_2$ choisi comme premier légateur*

Opérateur Asexué :(n,p,g)

Un opérateur asexué produit un enfant à partir d'un père. (n,p,g) est un opérateur asexué défini par Eiben et al. en [ERR95]. Il est basé sur l'idée d'amélioration d'un individu en changeant quelques uns de ses gènes (on pourrait dire qu'il fait une mutation "multiple"). Il s'agit aussi d'un opérateur qui en quelque sorte est indépendant de l'ordre des variables dans le chromosome. L'algorithme choisit aléatoirement un parent. Il sélectionne des positions du parent à changer d'une façon aléatoire et il sélectionne ensuite de nouvelles valeurs pour ces positions. Le nombre de valeurs modifiées (n), le critère pour identifier la position des valeurs à modifier(p) et le critère pour définir les nouvelles valeurs pour le fils (g) sont les paramètres de l'opérateur asexué. Les valeurs p et g sont choisies dans l'ensemble {r,b}, où r indique une sélection aléatoire uniforme et b indique une méthode de sélection basée sur une heuristique biaisée. Eiben et al. ont trouvé que pour le problème de 3-coloriage de graphe les meilleurs paramètres étaient (n,p,g)=(#,r,b) où l'heuristique biaisée b correspond à la *minimisation de conflits* et # signifie que le nombre de valeurs à modifier est choisi aléatoirement, mais avec un maximum de $\frac{1}{4}$ du nombre total de positions.

Procédure UAX $(Parent_1, Suppl_1, Parent_2, Suppl_2)$
Début
si r <probabilité de croisement **alors**
légateur= Choisir-aléatoirement $(Parent_1, Parent_2)$
non-légateur = le parent non choisi
Pour i:=1 jusqu'à n **faire**
  si $Suppl\_legateur[i] = Suppl\_non - legateur[i]$ **alors**
    échanger(légateur, non-légateur)
  $fils[i] = Parent_{legateur}[i]$
  $Suppl\_fils[i] = Suppl\_legateur[i]$
Fin /* UAX */

FIG. 2.12 - *Structure de croisement UAX*

Knowledge-augmented crossover

Knowledge-augmented crossover est proposé par Fleurent et Ferland en [FF95], spécialement pour les problèmes de 3-coloriage de graphe. Il inclut une heuristique pour réaliser un meilleur coloriage du graphe. L'algorithme génère un fils à partir de deux parents, de la façon suivante. Il colorie les nœuds qui sont en conflit dans un père en utilisant les couleurs de l'autre père, s'il n'a pas le même nœud en conflit. Quand un nœud est en conflit dans les deux parents, l'opérateur choisit la couleur qui est la moins utilisée dans les nœuds adjacents de chaque père. Chacun des nœuds restants est colorié en utilisant une couleur de ses parents, choisie aléatoirement, comme avec le *croisement uniforme*. Évidemment, ce type d'opérateur est plus coûteux que les opérateurs standards de croisement et même que UAX. Mais ils ont montré que leur algorithme génétique est globalement plus performant. D'un certain point de vue, il réalise un héritage "intelligent" de gènes, en utilisant un certain niveau de connaissance propre au problème du coloriage de graphe. La structure de *knowledge-augmented crossover* est montrée sur la figure 2.13.

Le fait que les algorithmes génétiques fassent peu de suppositions sur le domaine du problème, les rend capables d'être utilisés pour une grande gamme de problèmes.

Procédure knowledge-augmented crossover($Parent_1, Parent_2$)
Début
si r <probabilité de croisement **alors**
Pour $X \in CN(Parent_1)$ et $(X \notin CN(Parent_2))$ **faire**
$fils[X] = Parent_2[X]$
Pour $X \in CN(Parent_2)$ et $(X \notin CN(Parent_1))$ **faire**
$fils[X] = Parent_1[X]$
Pour $X \in (CN(Parent_1) \cap CN(Parent_2))$ **faire**
fils[x] $= argmin^{a}{}_{l \in \{1,2,3\}} \left\{ \sum_{y \in N(X)} I^{b}{}_{l}(\{Parent_1[y], Parent_2[y]\}) \right\}$
Pour $X \in (V - (CN(Parent_1) \cup CN(Parent_2)))$ **faire**
si (U[0,1] ^{c}=1) **alors**
$fils[X] = Parent_1[X]$
sinon
$fils[X] = Parent_2[X]$
Fin /* knowledge-augmented crossover */

[a] $argmin_{l \in S}\{a_l\}$ donne l'indice l^* tel que $a_{l^*} \leq a_l, \forall l \in S$
[b] $I_l(S)$ est la fonction indicatrice de l'ensemble S = {1,2,3}
[c] distribution uniforme sur 0 et 1

FIG. 2.13 – *Structure de knowledge-augmented crossover pour le 3-coloriage de graphe*

En revanche, ils peuvent avoir une performance inférieure à celles des techniques spécialisées qui utilisent la connaissance du domaine du problème. La nouvelle tendance dans la communauté de recherche en algorithmes génétiques est de faire une distinction entre les algorithmes génétiques standards et ceux qui travaillent avec une représentation non-binaire et/ou qui incluent quelques connaissances du problème dans la conception d'opérateurs spécialisés. Ces algorithmes sont appelés algorithmes évolutionnistes.

2.6 Vers un algorithme évolutionniste pour CSP

On a défini dans les sections précédentes un algorithme génétique standard qui utilise une représentation binaire comme méthode de codage pour les individus. Si cette représentation est tout à fait appropriée pour certains problèmes d'optimisation combinatoire, il est maintenant courant d'utiliser d'autres représentations qui tirent

parti de la structure naturelle des solutions. Pour certains problèmes, la représentation binaire des solutions est tout à fait intuitive et bien adaptée. C'est le cas par exemple du problème de satisfiabilité (SAT). Étant donnée une expression logique F de n variables booléennes $x_1, \ldots, x_n$, ce problème fondamental consiste à trouver une affectation booléenne des variables de façon à satisfaire F. En établissant les correspondances $faux \Leftrightarrow 0, vrai \Leftrightarrow 1$, on peut représenter chaque solution par une chaîne binaire de longueur n et utiliser les opérateurs génétiques classiques décrits plus haut. Les représentations binaires ne sont pas limitées aux vecteurs à une dimension. Des structures plus complexes ont été récemment utilisées dans les cas de problèmes d'ordonnancement. Pour ces problèmes [BCSJ86], il s'agit de déterminer un ordre de traitement pour n tâches. Il est alors possible de coder les solutions à l'aide d'une matrice de précédence composée de 0 et de 1.

Pour d'autres types de problèmes, au contraire, une représentation binaire est contre-intuitive et semble peu appropriée. C'est pour cela qu'ont été proposés au cours des dernières années plusieurs algorithmes génétiques utilisant des systèmes de codage mieux adaptés aux structures naturelles des solutions. Par exemple pour le problème du voyageur de commerce Grefenstette a proposé en [Gre87], une représentation où chaque élément est une permutation de n-villes qui donne l'ordre dans lequel elles sont parcourues. Avec une telle représentation, il faut cependant construire de nouveaux opérateurs de croisement et mutation. Plusieurs autres représentations sont maintenant utilisées pour divers problèmes d'optimisation combinatoire. Pour certains problèmes, il n'est pas toujours évident de choisir la représentation la plus appropriée. Dans la plupart des cas, le choix de la meilleure méthode de codage pour les solutions demeure un art, et peu de travaux se sont intéressés à définir des critères rigoureux pouvant aider ces choix, [LV90]. Davis [Dav91] suggère toutefois d'emprunter les systèmes de codage employés par d'autres méthodes couramment utilisées pour le problème considéré.

La différence est que, par rapport aux algorithmes génétiques standard, nous pouvons dans un algorithme évolutionniste avoir différents types de structures de données pour la représentation d'un chromosome. De plus, nous augmentons l'ensemble des possibilités pour un opérateur génétique, cela veut dire que nous pouvons incorporer

de nouveaux opérateurs adaptés au problème.

Notre objectif est de concevoir un algorithme évolutionniste qui résout les problèmes de satisfaction de contraintes, en prenant en compte les caractéristiques propres des CSP. Pour cela, dans le prochain chapitre nous commencerons par la définition d'une nouvelle fonction d'évaluation, qui pourrait ne pas être exclusivement utilisée dans un algorithme évolutionniste, mais aussi être incorporée dans d'autres méthodes telles que *min-conflits* ou GSAT, ou dans une implémentation hybride par exemple dans la méthode SCORE [BC97]. Cette méthode réalise une recherche locale en réparant une configuration initiale. Pour ce faire, elle utilise deux heuristiques, la première pour choisir une variable à réparer et la deuxième pour le choix de sa valeur. L'heuristique de choix de valeur la plus usuelle consiste à choisir la valeur qui minimise le nombre total de conflits (**mc**). Ensuite, la méthode SCORE propage les contraintes en filtrant les domaines des variables non encore modifiées, qui partagent une contrainte avec la variable modifiée. On l'appelle hybride, car elle permet un retour-en-arrière informé, dans les cas où elle détecte une incohérence. On pourrait donc utiliser la fonction d'évaluation que nous allons définir dans le chapitre suivant au lieu de **mc** comme heuristique pour le choix de la valeur de la variable à réparer.

Chapitre 3

Fonction d'Évaluation

Dans le chapitre précédent, nous avons présenté les composantes d'un algorithme évolutionniste. Mais la question suivante demeure: Qu'est-ce que peut apporter de plus un algorithme évolutionniste, par rapport aux méthodes stochastiques traditionnelles pour la résolution des CSP? En premier lieu, un algorithme évolutionniste permet une recherche multiple (population de pré-solutions), on peut donc chercher en parallèle dans différents espaces de recherche. Au contraire, les méthodes stochastiques traditionnelles ne cherchent qu'avec une seule pré-solution. Une différence fondamentale, à notre avis, des algorithmes évolutionnistes par rapport aux méthodes stochastiques classiques (telles que GSAT ou *min-conflits* et leurs variantes) réside dans la coopération possible entre les pré-solutions de la population pour trouver une solution au problème. Une pré-solution n'évolue plus individuellement mais peut se combiner avec d'autres. Dans un algorithme évolutionniste qui fait évoluer de manière aléatoire les différents individus de la population (exploration), on ajoute de l'exploitation qui cherche à profiter de la connaissance contenue dans les différentes pré-solutions, en les combinant. Puisque notre but est de concevoir un algorithme évolutionniste pour résoudre des problèmes de satisfaction de contraintes, nous allons donc dans ce chapitre aborder la définition d'une fonction d'évaluation, en essayant de prendre en compte quelques caractéristiques des CSP. Pour profiter du mécanisme que fournissent les algorithmes évolutionnistes, il nous faut aussi de bons opérateurs qui soient capables de mettre en valeur la connaissance des pré-solutions par une bonne combinaison. Ce

sujet sera traité en détail dans les chapitres suivants.

Nous commencerons ce chapitre avec un exemple montrant les limitations de la fonction d'évaluation standard (voir définition 2.5.1), afin de motiver l'introduction de notre nouvelle fonction, après avoir défini quelques concepts. Ensuite, nous incorporerons cette fonction, comme heuristique dans la méthode de *min-conflits* et puis la comparerons à l'aide de quelques résultats. Enfin, nous evaluerons le comportement de cette fonction dans un algorithme génétique qui résout le problème de coloriage de graphe avec 3 couleurs. Finalement, nous proposerons une extension de cette fonction pour les problèmes n-aires.

3.1 Motivation: un exemple

Nous rappelons que les méthodes stochastiques telles que l'escalade, le recuit simulé, la recherche tabou (voir Chapitre 1), consistent à réparer une instanciation pour arriver à trouver une solution d'un CSP. Pour ce faire, elles utilisent quelques heuristiques pour guider le "parcours stochastique" dans l'espace de recherche. La fonction d'évaluation est utilisée pour mesurer l'amélioration de l'instanciation modifiée et pour conduire la recherche vers des instanciations plus prometteuses en termes de satisfaction des contraintes. Souvent, cette fonction est la fonction d'évaluation standard $Z_{std}(\mathbf{I})$ de la définition 2.5.1, qui correspond au nombre de contraintes violées. [Fre95]. Par exemple, considérons le CSP suivant dont le graphe et la matrice de contraintes sont illustrés par la figure 3.1:

variables $V=X_1, X_2, X_3, X_4$

domaines

- $D_1 = (10, 20, 35, 50)$ avec $m_1 = 4$
- $D_2 = (30, 25, 20, 10)$ avec $m_2 = 4$
- $D_3 = (25, 30, 20)$ avec $m_3 = 3$
- $D_4 = (40, 50, 60)$ avec $m_4 = 3$

- *L'ensemble* ζ *avec 4 contraintes*
  - $C_1 : X_1 \geq X_2$
  - $C_2 : X_3 \geq X_2$
  - $C_3 : X_4 \geq X_3$
  - $C_4 : X_4 \geq X_2$

	X_1	X_2	X_3	X_4
C_1	1	1	0	0
C_2	0	1	1	0
C_3	0	0	1	1
C_4	0	1	0	1

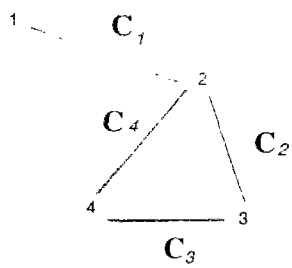


FIG. 3.1 – *Exemple: Graphe de Contraintes et Matrice de Contraintes*

L'ordre des variables dans l'instanciation est le suivant: X_1,X_2,X_3,X_4. Supposons que nous puissions choisir parmi les deux instanciations suivantes pour notre problème:

$\mathbf{I}_1$= | 35 | 30 | 25 | 40 | qui ne satisfait pas la contrainte C_2

$\mathbf{I}_2$= | 10 | 25 | 30 | 40 | qui ne satisfait pas la contrainte C_1

$Z_{std}(\mathbf{I})$ sera pour les deux égal à 1, car chacune ne viole qu'une seule contrainte. Regardons maintenant le graphe de contraintes: la contrainte C_2 est liée aux trois autres contraintes, ce qui veut dire que si nous changeons quelques valeurs des variables pertinentes pour C_2, cela pourrait éventuellement se traduire par une violation d'autres contraintes satisfaites à présent. En revanche, C_1 est liée à C_2 et C_4. On peut donc penser qu'une réparation de cette contrainte aurait moins de conséquences vis à vis des autres contraintes qui sont déjà satisfaites dans le réseau. C'est cette idée que nous allons incorporer dans la définition de la nouvelle fonction d'évaluation.

3.2 Fonction d'évaluation pour CSP binaire

A part $Z_{std}(\mathbf{I})$, d'autres fonctions d'évaluation, plus spécifiques au type de problème à traiter, ont été proposées dans la littérature pour les CSP binaires. Par exemple, pour le problème de N-reines, Eiben et al. [ERR94] ont utilisé comme fonction d'évaluation le nombre de contraintes non satisfaites sur la diagonale. Pour un problème de dessin mécanique, Thorton a proposé de minimiser $Y = \sum d_i^2$ où d_i est l'erreur normalisée pour la contrainte i, [Tho94]. Notre objectif ici est de définir une fonction indépendante du problème à résoudre. D'autre part, le fait de prendre en compte le seul nombre de contraintes violées signifie:

- D'un coté, qu'on considère que toutes les contraintes sont également facilement (difficilement) réparables dans la recherche d'une solution. Néanmoins, dans la plupart des problèmes de satisfaction de contraintes, des contraintes sont plus dures que d'autres à satisfaire.

  D'un autre coté qu'il n'y a pas de préférence parmi les contraintes. Avoir une préférence permet de relâcher les problèmes sous contraints pour obtenir une "bonne solution", quand on n'a pas le temps d'obtenir une solution satisfaisant toutes les contraintes. C'est notamment le cas des PCSP (Partial Constraint Satisfaction Problems) [FW92].

On se place dans le premier cas, c'est-à-dire, nous voulons obtenir une solution du CSP (et non une solution à un problème relâché). Nous souhaitons définir une fonction d'évaluation qui représente une sorte de "distance" entre une instanciation et une solution, en termes du nombre de changements effectués par un algorithme de recherche locale. Une fonction qui considère donc la difficulté d'obtenir une solution. Cela nous a conduite à la définition de la nouvelle fonction qui prend en compte la structure du réseau de contraintes, plus précisément le degré de liaison entre les différentes variables.

Pour cela, nous commençons avec un nouveau concept: nous voulons montrer que dans le cas où une contrainte est violée, les variables qui sont impliquées dans cette violation ne sont pas seulement les deux variables connectées par cette contrainte.

En effet, en plus des variables pertinentes, la réparation d'une contrainte violée peut affecter d'autres variables qui leur sont connectées à travers d'autres contraintes.

Définition 3.2.1 *(Ensemble de Variables Impliquées)*
Étant donné un CSP binaire $P = (V, D, \zeta)$, *et une instanciation* $\mathbf{I}$, *pour chaque contrainte* $C_\alpha(\alpha = 1, \ldots, \eta)$ *un ensemble de variables impliquées* $\mathbf{E}_{\alpha(\mathbf{I})} \subseteq V$ *est défini par:*

- $\mathbf{E}_{\alpha(\mathbf{I})} = \phi$ *ssi* C_α *est satisfaite*

  $X_i \in \mathbf{E}_{\alpha(\mathbf{I})}$ *si* $X_i \rhd C_\alpha$ *et* C_α *n'est pas satisfaite sous* $\mathbf{I}$

- $X_j \in \mathbf{E}_{\alpha(\mathbf{I})}$ *si* $\exists X_i \rhd C_\alpha$ *et* $\exists C_\beta \neq C_\alpha$ *tel que* X_i *et* $X_j \rhd C_\beta$, *et* C_α *n'est pas satisfaite sous* $\mathbf{I}$

Cette définition montre qu'en essayant de réparer une contrainte C_α sous une certaine instanciation $\mathbf{I}$, le changement de valeur d'une variable sera perçu par d'autres contraintes dans le réseau. C'est justement cet effet que nous voulons incorporer dans la fonction d'évaluation. Pour une instanciation $\mathbf{I}$ donnée, nous allons quantifier l'erreur attribuée à une contrainte C_α par:

Définition 3.2.2 *(Évaluation de l'erreur)*
Étant donné un CSP binaire $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$ *et une instanciation* $\mathbf{I}$, *la fonction de l'évaluation de l'erreur* $\mathbf{e}(C_\alpha, \mathbf{I})$ *pour une contrainte* C_α *est définie par:*

$$\mathbf{e}(C_\alpha, \mathbf{I}) = \textit{Nombre de variables dans } \mathbf{E}_{\alpha(\mathbf{I})}$$

Si une contrainte binaire non-satisfaite C_α a X_k et X_l comme variables pertinentes (elle en a juste deux, exactement ces deux), alors nous pouvons exprimer $\mathbf{e}(C_\alpha, \mathbf{I})$ de la manière suivante:

$\mathbf{e}(C_\alpha, \mathbf{I})$ = (Nombre de variables pertinentes pour C_α) + (Effet Propagé par X_k et X_l)

où l'effet propagé par X_k et X_l dans un réseau de contraintes binaires est défini comme le nombre de contraintes C_β, $\beta = 1, \ldots, \eta, \beta \neq \alpha$ qui ont comme variables pertinentes X_k ou X_l

Donc, en termes de la matrice de contraintes cela se traduit par:

$$\mathbf{e}(C_\alpha, \mathbf{I}) = \left(\sum_{j=1}^{n} \mathbf{R}[\alpha, j] \right) + \left(\sum_{\beta \neq \alpha, \beta=1}^{\eta} \mathbf{R}[\beta, k] + \sum_{\beta \neq \alpha, \beta=1}^{\eta} \mathbf{R}[\beta, l] \right) \tag{3.1}$$

Remarque 3.2.1 *Si C_γ est satisfaite alors $\mathbf{e}(C_\gamma, \mathbf{I}) = 0$*

Remarque 3.2.2 *La complexité du calcul de $\mathbf{e}(C_\alpha, \mathbf{I})$ pour chaque contrainte C_α est égal à $O(\eta)$.*

Chaque contrainte dans notre fonction contribue différemment à la valeur de la fonction d'évaluation, on peut définir donc la contribution d'une contrainte comme:

Définition 3.2.3 *(Contribution de C_α)*
Étant donné un CSP binaire $P = (V, D, \zeta)$, on dira que la Contribution de C_α à la fonction d'évaluation $\mathbf{c}(C_\alpha)$ sera:

$$\mathbf{c}(C_\alpha) = \mathbf{e}(C_\alpha, I_j), \text{ quand } C_\alpha \text{ est violée sous } I_j.$$

Cette fonction quantifie la contribution de C_α à la fonction d'évaluation quand elle n'est pas satisfaite. On remarque que cette contribution ne dépend pas des valeurs effectives de l'instanciation, mais seulement du fait que la contrainte est violée.

Finalement, la valeur de la fonction d'évaluation pour une instanciation $\mathbf{I}$ donnée sera la somme des évaluations des erreurs (équation 3.1) sur toutes les contraintes dans le CSP.

Définition 3.2.4 *(Fonction d'Évaluation pour CSP binaire)*
Étant donné un CSP binaire avec une matrice de contraintes $\mathbf{R}$, une instanciation $\mathbf{I}$ et la fonction d'Évaluation de l'erreur $\mathbf{e}(C_\alpha, \mathbf{I})$ pour chaque contrainte C_α, $(\alpha = 1, \ldots, \eta)$, la Fonction d'Évaluation $\mathbf{Z}(\mathbf{I})$ est définie par:

$$\mathbf{Z}(\mathbf{I}) = \sum_{\alpha=1}^{\eta} \mathbf{e}(C_\alpha, \mathbf{I}) \tag{3.2}$$

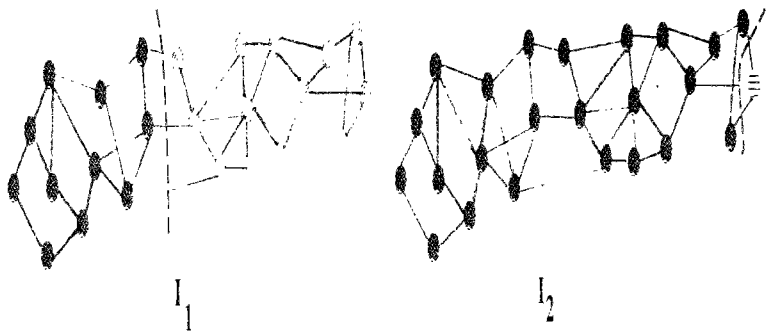


FIG. 3.2 – *Deux instanciations pour le même graphe*

Le but de la recherche d'une solution est de minimiser cette fonction $Z(\mathbf{I})$, laquelle sera égale à zéro quand toutes les contraintes seront satisfaites.

$Z(\mathbf{I})$ peut être interprétée comme une façon de quantifier notre préférence pour les instanciations qui satisfont plus de contraintes et de liaisons. Pour illustrer cela, regardons la figure 3.2. Il s'agit d'un problème de coloriage de graphe. Nous avons deux instanciations pour le même graphe: I_1 et I_2. Pour chaque instanciation, l'ensemble de nœuds à gauche de la ligne en pointillés représente un coloriage consistant, et l'ensemble à droite un autre coloriage consistant. Mais les deux coloriages sont inconsistants entre eux. À première vue, les deux instanciations peuvent paraître comme aussi bonnes, car dans les deux cas nous n'avons que trois contraintes violées. Par contre, en utilisant la fonction $Z(\mathbf{I})$ nous préférons la seconde instanciation, parce qu'elle a moins de variables impliquées (voir définition 3.2.1).

Fonction d'Évaluation et CSP aléatoires

Considérons les CSP générés aléatoirement, [Smi95], [Pro94] qui sont caractérisés par 4 paramètres: n le nombre de variables, m le nombre de valeurs de chaque domaine, égal pour toutes les variables, p_1 la probabilité qu'il existe une contrainte entre deux variables et p_2 la probabilité conditionnelle qu'une paire de valeurs soit inconsistante pour une paire de variables, étant donné qu'il y a une contrainte entre les variables. Pour ce type de CSP aléatoire, l'espérance du nombre de contraintes violées pour une instanciation quelconque $\mathbf{I}$ sera:

$$E[Z_{std}(\mathbf{I})] = \left(p_1 \frac{n(n-1)}{2}\right) p_2 \tag{3.3}$$

Le nombre maximum de contraintes pour un problème avec n variables est $\frac{n(n-1)}{2}$. Le CSP aléatoire aura donc $p_1 \frac{n(n-1)}{2}$ contraintes ou arêtes. C'est-à-dire que le nombre espéré de contraintes violées sera égal au nombre de contraintes sur le réseau, multiplié par la probabilité d'incompatibilité de valeurs des domaines p_2 pour chaque contrainte.

La valeur de notre fonction sera:

$$E[Z(\mathbf{I})] = 2Z_{std}(\mathbf{I})p_1(n-1) \tag{3.4}$$

Pour expliquer l'équation 3.4, voyons la figure 3.3. Chaque contrainte a deux variables pertinentes. Chaque variable du problème a une arité (voir définition 1.1.3) de $p_1(n-1)$. Alors, pour une contrainte C_a violée, nous allons compter ses deux variables pertinentes plus l'effet de propagation à partir d'elles, donc $2 + (2p_1(n-1) - 2)$. Maintenant nous devons faire la somme sur toutes les contraintes qui sont violées sur le reseau donc sur $Z_{std}(\mathbf{I})$.

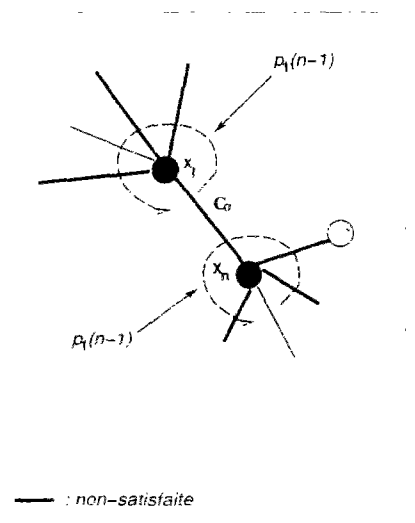


FIG. 3.3 – *Exemple:équation 3.4*

Nous avons généré les deux fonctions: $E[Z_{std}(\mathbf{I})]$ et $E[Z(\mathbf{I})]$ pour des graphes avec 8 variables (n=8) et leur taille de domaine égale à 3 (m=3), pour différentes valeurs

de p_1 et p_2. Sur la figure 3.4, calculée à partir de l'équation 3.3, nous observons que la variation de valeur de la fonction $E[Z_{std}(\mathbf{I})]$ est directement proportionnelle à p_1 et p_2, par contre sur la figure 3.5 déduite de la formule 3.4, notre fonction est directement proportionnelle à p_1^2 et à p_2. Cela veut dire que notre fonction dépend plus fortement de p_1 (e.g. connectivité) que $E[Z_{std}(\mathbf{I})]$.

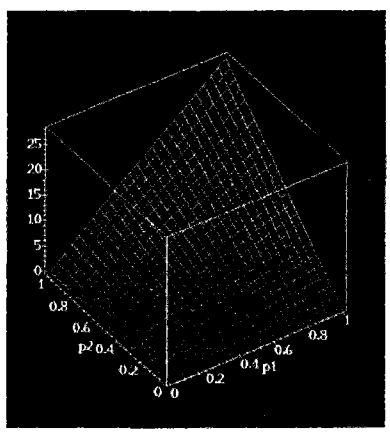


FIG. 3.4 – $E[Z_{std}(\mathbf{I})]$ *en fonction de* p_1 *et* p_2

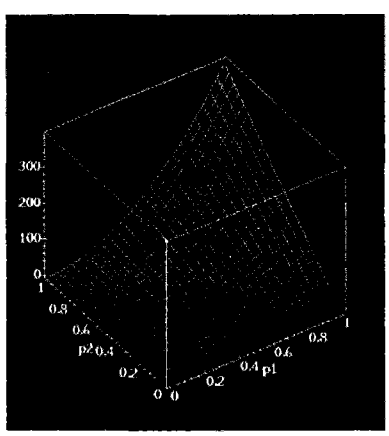


FIG. 3.5 – $E[Z(\mathbf{I})]$ *en fonction de* p_1 *et* p_2

3.3 Minimum-réseau-conflits

Nous voulons inclure l'idée développée dans le calcul de la fonction $Z(\mathbf{I})$ comme une nouvelle heuristique dans la méthode *min-conflits*, [MJPL92]. Pour cela, nous commençons avec la définition de l'ensemble de contraintes qui ne sont pas satisfaites et qui ont la même variable pertinente X_j, noté par N_j:

Définition 3.3.1 *($N_j(\mathbf{I})$)*
Étant donné un CSP binaire $P = (V, D, \zeta)$ avec une instanciation $\mathbf{I}$, pour une variable X_j, un ensemble de contraintes $N_j(\mathbf{I}) \subseteq \zeta$ est défini par $C_\alpha \in N_j(\mathbf{I})$ ssi:

- $X_j \rhd C_\alpha$
- *C_α est violée sous $\mathbf{I}$*

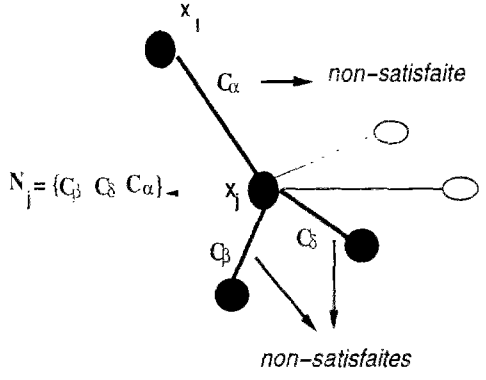


FIG. 3.6 – *Définition 3.3.1*

En utilisant cette définition, nous pouvons décrire l'heuristique de min-conflits (voir définition 1.3.1) du point de vue des contraintes comme:

Définition 3.3.2 *(Fonction de min-conflits)*
Soit un CSP binaire $P = (V, D, \zeta)$ avec une instanciation $\mathbf{I}$, et l'ensemble de contraintes $N_j(\mathbf{I})$ pour chaque variable X_j. La fonction de min-conflits, **mc** *pour X_j est définie par:*

$$\mathbf{mc}(X_j, \mathbf{I}) = \textit{Nombre de contraintes en } N_j(\mathbf{I})$$

En d'autres mots, $\mathbf{mc}(X_j, \mathbf{I})$ quantifie le nombre de conflits (contraintes violées) dans lesquels participe X_j.

Pour introduire l'idée de la connectivité dans l'heuristique, nous allons remplacer la fonction **mc** par la fonction suivante:

Définition 3.3.3 *(Fonction de Min-réseau-conflits)*
Étant donné un CSP binaire $P = (V, D, \zeta)$ avec une instanciation $\mathbf{I}$*, et l'ensemble de contraintes $N_j(\mathbf{I})$ pour chaque variable X_j, et les fonctions $\mathbf{e}(C_\alpha, \mathbf{I})$ pour toute contrainte C_α, la fonction de min-réseau-conflits,* **nc** *pour X_j est définie par:*

$$\mathbf{nc}(X_j, \mathbf{I}) = \sum_{C_\gamma \in N_j(\mathbf{I})} \mathbf{e}(C_\gamma, \mathbf{I}) \tag{3.5}$$

Remarque 3.3.1 *Cette fonction est calculée en prenant en compte seulement les arêtes impliquées (liées à X_j).*

Nous allons utiliser cette fonction dans l'algorithme de réparation *min-conflits*, décrit sur la figure 1.4, en introduisant une nouvelle heuristique que nous définissons dans la section suivante:

3.3.1 Heuristique: min-réseau-conflits

Comme pour *min-conflits*, l'heuristique commence avec une instanciation $\mathbf{I}$ complète devant être réparée. D'abord, la variable à changer est choisie aléatoirement parmi les variables en conflits, ensuite nous sélectionnons une valeur pour elle. La sélection de la valeur est faite en utilisant la *fonction min-réseau-conflits* **nc** (équation 3.5). Pour faire cela, la procédure calcule **nc** pour toutes les valeurs dans le domaine de la variable X_j et elle sélectionne la valeur x_j qui a une valeur de la *fonction min-réseau-conflits* la plus faible. Cela est possible parce que nous travaillons sur des réseaux de contraintes avec des domaine finis.
Cette heuristique guide le choix de la valeur de la variable sélectionnée en cherchant la plus grande descente, en termes des conflits directs et indirects du graphe

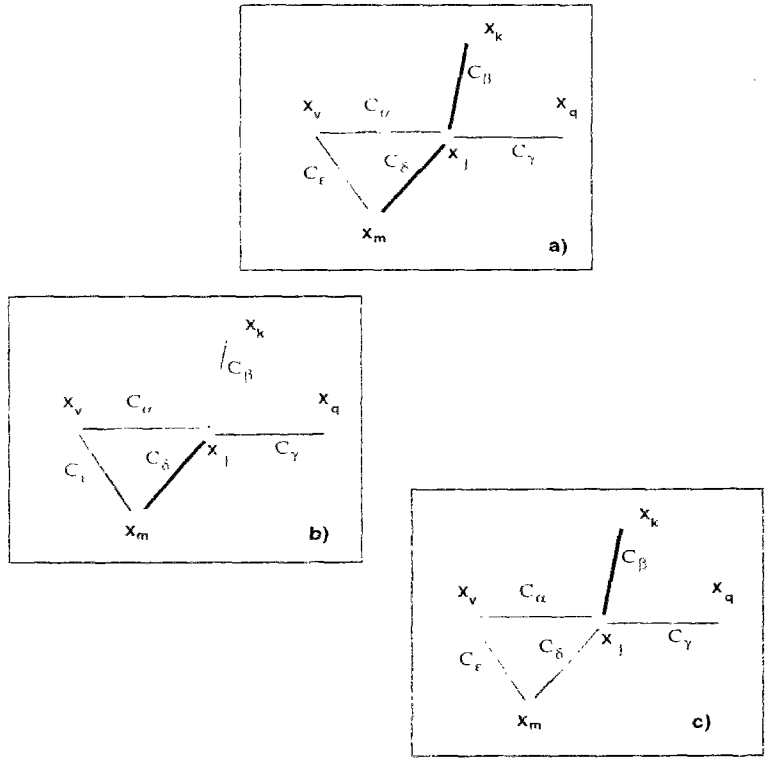


FIG. 3.7 – *Recherche d'une valeur pour* X_j

de contraintes. Pour montrer ce que cela veut dire regardons la figure 3.7. Supposons que le graphe a) représente la première pré-solution pour un problème donné. L'instanciation représentée viole C_β et C_δ (dessinées par une ligne noire). Soit X_j la variable en conflit sélectionnée d'une façon aléatoire pour être réparée. Le graphe b) et le graphe c) correspondent à deux choix de valeurs possibles pour cette variable. Pour l'heuristique *min-conflits*, les deux valeurs sont également bonnes, parce qu'après avoir modifié la valeur de X_j, dans les deux cas il ne reste qu'une contrainte violée. Mais pour *min-réseau-conflits* le graphe c) est meilleur que le graphe b) parce que l'instanciation représentée en c) pourrait être plus facilement réparée du point de vue de l'effet de propagation sur le réseau. Il suffirait que X_k soit sélectionnée et que l'algorithme instancie une valeur satisfaisant C_β.

Remarque 3.3.2 *Si le CSP est représenté par un graphe complètement connecté,*

avec chaque variable appartenant à n-1 *contraintes, alors min-réseau-conflits et min-conflits ont le même comportement. Plus précisément, pour un CSP où chaque variable appartient exactement à c contraintes, leur recherche est équivalente.*

Min-conflits a pour chaque contrainte la valeur 1 ou 0, selon si elle est satisfaite ou non-satisfaite, quel que soit le type du réseau de contraintes. Quand il s'agit d'un graphe où chaque variable est connectée à un ensemble c de contraintes, *min-réseau-conflits* ajoute la valeur $2c$ ou 0 selon que la contrainte est satisfaite ou non-satisfaite. Pour un graphe complètement connecté, *min-réseau-conflits* $\mathbf{nc}(X_j, \mathbf{I})$ vaudra pour le pire des cas $2(n-1)^2$. La priorité de contraintes implicite de *min-réseau-conflits*, dans ce cas n'existe pas, car elle est faite en prenant en compte la connectivité de chaque variable.

3.3.2 Algorithme de réparation en deux étapes

Nous voulons incorporer cette heuristique dans un algorithme de réparation. L'idée est de faire de l'escalade guidée par le réseau de contraintes. Nous avons réalisé plusieurs tests avec *min-conflits* et *min-réseau-conflits* et nous avons trouvé une sorte de complémentarité entre les deux heuristiques. Plus précisément, certains problèmes qui étaient faciles pour *min-conflits* ne l'étaient pas pour *min-réseau-conflits* et vice-versa. Cela peut être apprecié sur la fig 3.10. Cela a motivé la proposition d'algorithme en deux étapes. Dans la première étape, il utilise l'heuristique de *min-conflits*, donc avec la fonction **mc**, (définition 3.3.2). Dans le cas où il n'est pas capable de trouver une solution à la fin d'un nombre maximum d'itérations, il continue avec l'heuristique de *min-réseau-conflits*, donc avec la fonction **nc** (définition 3.3.3), jusqu'à trouver une solution ou avoir réalisé un nombre maximum d'itérations. Nous avons donc deux paramètres de contrôle, le nombre maximum d'itérations pour *min-conflits* et pour *min-réseau-conflits*. Nous pouvons interpréter cet algorithme comme une façon d'explorer d'abord les conflits directs. Si nous n'avons pas de succès dans la première étape, alors l'algorithme continue son exploration en utilisant les conflits propagés dans le réseau. La structure de l'algorithme est montrée sur la figure 3.8.

Pour évaluer notre algorithme, nous allons utiliser des problèmes générés d'une

Procédure Deux-étapes (V, D, ζ)
Début
Générer aléatoirement une pre-solution **I**
si **I** n'est pas une solution **alors**
  Répéter
    Sélectionner aléatoirement $X_j \in K(\mathbf{I})$ [a]
    $\mathbf{I}(X_j) = argmin_{x_j \in D_j}\{\mathbf{nc}(X_j = x_j, ((\mathbf{I} - x_{j_a}{}^{b}) \cup x_j))\}$ [c]
    itérations-conflits++
  Jusqu'à ((**I** soit solution) ou (max itérations-conflits))
si **I** n'est pas une solution **alors**
  Répéter
    Sélectionner aléatoirement $X_j \in K(\mathbf{I})$
    $\mathbf{I}(X_j) = argmin_{x_j \in D_j}\{\mathbf{nc}(X_j = x_j, ((\mathbf{I} - x_{j_a}) \cup x_j))\}$
    itérations-réseau-conflits++
  Jusqu'à ((**I** soit solution) ou (max itérations-réseau-conflits))
Fin /* procédure deux-étapes*/

[a] l'ensemble des variables en conflit
[b] sa valeur courante
[c] $argmin_{l \in S}\{a_l\}$ donne la valeur l^* tel que $a_{l^*} \leq a_l, \forall l \in S$

FIG. 3.8 – *Structure de la procédure deux étapes*

façon aléatoire, spécifiquement le problème de coloriage de graphe avec trois couleurs, à partir de maintenant dénommé par *3-coloriage*. La procédure utilisée est décrite dans la section suivante. Elle sera aussi rappelée dans les prochains chapitres.

Ensemble de problèmes: 3-coloriage

Le problème du coloriage de graphe est un problème NP-complet bien connu, qui est utilisé pour modéliser certains types de problèmes d'ordonnancement et d'allocation de ressources. Le but de nos essais est d'observer l'effet d'incorporer notre approche dans la performance d'un algorithme de réparation. Les problèmes de 3-coloriage sont générés de façon aléatoire, la contrainte est toujours la même: un nœud ne doit pas avoir la même couleur qu'un nœud voisin. La procédure de génération des graphes a été proposée par Adorf et al. en [AJ90]. Cette procédure génère des problèmes, qui ont au moins une solution, de la façon montrée sur la figure 3.9.

Les graphes générés par cette procédure auront n nœuds (variables) et m arêtes (contraintes).

Procédure Génération des Problèmes 3-Coloriage (n, m)
Début
Affecter $\frac{n}{3}$ des n nœuds à chacun des trois ensembles (A,B,C)
Répéter
    Ensemble_1 = random-entre(A, B, C)
    nœud_1 = random-dans(Ensemble_1)
    Ensemble_2 = random-entre(A,B,C - Ensemble_1)
    nœud_2 = random-dans(Ensemble_2)
    si ($\nexists C_\alpha$ tel que $\text{nœud}_1 \rhd C_\alpha$ et $\text{nœud}_2 \rhd C_\alpha$) **alors**
      Créer C_α entre nœud_1 et nœud_2
      Nombre-d'arêtes++
Jusqu'à Nombre-d'arêtes = m
Fin /* Procédure Génération des Problèmes 3-Coloriage */

FIG. 3.9 – *Procédure de Génération aléatoire des Problèmes de 3-Coloriage avec solution*

Évaluation de l'heuristique

Pour évaluer l'algorithme en deux étapes nous avons généré des graphes aléatoires, qui sont appelés "graphes peu denses[1]", c'est à dire, avec $\eta = 2n$, d'après leur structure. Ce type de graphes est connu pour être celui qui donne le plus de difficulté aux algorithmes de réparation stochastique (voir [MJPL92]). Ces graphes ont plusieurs solutions donc l'algorithme a plus de choix pour effectuer la sélection de valeurs, donc le nombre de combinaisons "prometteuses" de valeurs est plus important que pour les graphes denses. Nous avons essayé d'abord de résoudre chaque graphe en utilisant l'heuristique, définie par Brelaz [Bre79], laquelle est assez performante pour les problèmes de 3-coloriage. Il s'agit d'un algorithme glouton qui affecte une couleur à chaque nœud.

Définition 3.3.4 *(Heuristique de Brelaz)*
Trouver le nœud qui n'est pas colorié qui a le moins de couleurs consistantes avec

1. en anglais: sparse graphs

ses voisins. S'il y en a plusieurs, alors en choisir un qui a le degré maximum dans le sous-graphe non-colorié.

Nous allons faire les expériences seulement avec les problèmes que l'heuristique de Brelaz n'a pas pu résoudre, donc des problèmes particulièrement difficiles. La pré-solution trouvée par l'algorithme qui utilise cette heuristique, sera utilisée comme pré-solution initiale dans nos tests.

Pour chaque $\eta = [60, 90, 120, 150, 180]$ nous avons 100 graphes différents qui n'ont pas pu être résolus en appliquant l'heuristique de Brelaz (définition 3.3.4). Nous comparons trois algorithmes d'escalade, le premier utilise l'heuristique *min-conflits* avec un nombre maximum d'itérations égal à 5000. Le deuxième utilise l'algorithme en deux étapes. Pour chaque étape, le nombre maximum d'itérations est fixé à 2500. Le troisième utilise l'algorithme avec l'heuristique *min-réseau-conflits* pur, avec un maximum de 5000 itérations. Les résultats sont montrés sur la figure 3.10. En abscisse, nous avons le nombre de contraintes du graphe, en ordonnée nous avons le pourcentage de convergence de l'algorithme. Par exemple, pour les graphes avec 60 contraintes (30 variables), notre algorithme a trouvé une solution pour 70% des graphes, en revanche *min-conflits* a trouvé la solution seulement pour 20% des graphes. Au fur et à mesure que la taille du problème augmente, la probabilité de trouver une solution diminue, mais notre algorithme présente toujours un meilleur comportement que *min-conflits* pur. Il est important de remarquer que ce type de problèmes est particulièrement difficile pour les méthodes stochastiques. Ces expériences montrent que le fait de prendre en compte la structure du réseau de contraintes peut être un facteur important pour guider la recherche des méthodes stochastiques.

Maintenant que nous avons constaté que la fonction d'évaluation peut apporter une amélioration de la performance d'une recherche stochastique par réparation locale, nous allons introduire cette fonction dans un algorithme évolutionniste.

3.4 Z(I) dans un algorithme évolutionniste

Nous avons conçu un jeu de tests pour mesurer l'avantage d'utiliser Z(**I**) au lieu de $Z_{std}(\mathbf{I})$ dans un algorithme évolutionniste (EA). Dans le chapitre précédent, nous

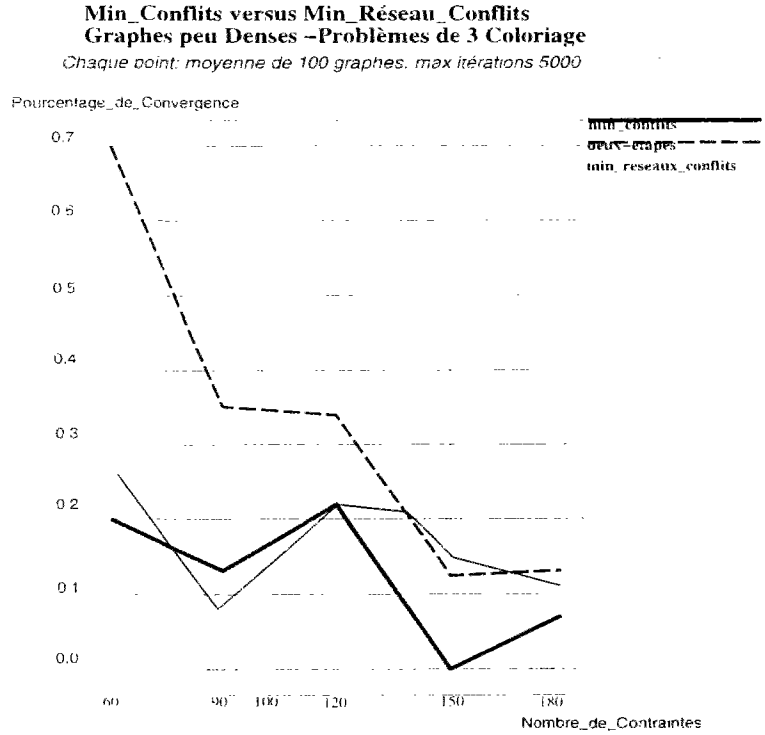


FIG. 3.10 – *Comparaison: Min-conflits v/s Min-réseau-conflits pour graphes peu denses*

avons affirmé que la conception d'un algorithme évolutionniste a besoin de six composantes (voir figure 2.1). Dans les sections suivantes, nous décrirons les différentes parties de notre EA.

3.4.1 Population initiale

La population initiale est générée aléatoirement. Chaque instanciation **I** est composée par des valeurs des variables qui sont choisies à partir de leurs domaines avec une distribution de probabilité uniforme.

3.4.2 Représentation génétique

Nous avons choisi une représentation génétique non-binaire, car ce type de représentation s'adapte naturellement aux CSP. Les domaines de variables dans un CSP

peuvent être de différents types, nous pouvons donc avoir dans un même CSP des variables booléennes, réelles, entières. Alors, la représentation génétique choisie a la structure de la figure 3.11

X1	X2	X3	.....	Xi	.....	Xn
A	1	1024	.....	0	.....	Z

FIG. 3.11 - *Représentation du Chromosome*

3.4.3 Algorithme de sélection

La fonction d'évaluation Z(**I**) nous a permis de concevoir un nouvel algorithme pour réaliser la sélection à partir de la population. Nous avons présenté dans la section 2.2.1 la méthode de sélection standard de la roue biaisée pour le problème de maximisation, son adaptation pour le calcul de la probabilité de sélection pour notre problème de minimisation est présentée dans l'annexe A. Nous souhaitons avantager les chromosomes qui ont une valeur de la fonction d'évaluation inférieure, plus fortement que ce que fait la méthode de sélection standard. Les mauvais individus auront dans notre algorithme quand même une probabilité d'être choisis, parce que pour obtenir des bons individus on a souvent besoin de produire de mauvais individus comme structures intermédiaires, [Mic94].

Nous avons conçu la stratégie de sélection de la figure 3.12. Nous divisons la population en trois sous-populations, la première est constituée par les individus qui ont une fonction d'évaluation inférieure à la moyenne, la deuxième ceux qui ont une fonction d'évaluation comprise entre la moyenne et la moyenne plus l'écart type, dans la dernière sous-population nous avons les individus qui ont une valeur supérieure à la moyenne plus l'écart type.

Pour déterminer α et β, nous avons comparé, pour différents réseaux de contraintes, le nombre de générations nécessaire pour trouver une solution. Pour chaque combinaison nous avons résolu 100 problèmes de 3-coloriage avec 30 variables générés de manière aléatoire. La table 3.1 montre le nombre de générations d'un des ensembles de test (des graphes avec une connectivité égale à 4) pour différentes valeurs de α et

α	$\beta - \alpha$	Générations
0.05	0.8	124.4
0.1	0.75	93.4
0.2	0.65	79.2
0.3	0.55	69.2
0.4	0.45	67.7
0.5	0.35	58.1
0.6	0.25	63.1
0.7	0.15	70.0
0.8	0.05	62.2

TAB. 3.1 - *Nombre de générations pour différentes valeurs de α et de $\beta - \alpha$, pour 3-coloriage avec une connectivité moyenne de 4.0*

de $\beta - \alpha$.

Parmi tous les tests générés, nous avons obtenu les meilleurs résultats avec $\alpha = 0.5$ et $\beta = 0.85$.

Propriétés statistiques de l'algorithme de sélection

En utilisant les valeurs α et β, la population est divisée en trois régions A, B, C. Cela est montré dans la figure 3.13. Supposons que nous ayons une population de taille N, avec n_1 chromosomes dans la region A, n_2 dans la région B et n_3 dans la région C, tel que $n_1 + n_2 + n_3 = N$, alors nous avons les probabilités de sélection suivantes pour un chromosome:

- à partir de la région $A = \alpha + (\beta - \alpha) * \frac{n_1}{n_1+n_2} + (1 - \beta) * \frac{n_1}{N}$
- à partir de la région $B = (\beta - \alpha) * \frac{n_2}{n_1+n_2} + (1 - \beta) * \frac{n_2}{N}$
- à partir de la région $C = (1 - \beta) * \frac{n_3}{N}$

La figure 3.13 montre que nous avons une préférence pour les chromosomes de la région A, c'est à dire, nous préférons les individus dont la fonction d'évaluation est inférieure ou égale à la moyenne. Néanmoins, la probabilité de sélectionner des individus à partir de la région C existe. Pour illustrer cela, regardons le problème de coloriage de graphe sur la figure 3.14. La population est composée par quatre

Procédure sélection(Population)
Début
Choisir j à partir de $U^{a}[0..1]$
si (j $< \alpha$) **alors**
  Choisir chromosome tel que $\mathbf{Z}(chromosome) \leq F^{b}$
sinon
  si (j $< \beta$) **alors**
  Choisir chromosome tel que $\mathbf{Z}(chromosome) < F + \sigma^{c}$
  sinon
    Choisir chromosome à partir $U[1..taille - population]$
Fin /* sélection */

[a] distribution uniforme
[b] moyenne de la population
[c] écart-type de la population

FIG. 3.12 – *Algorithme de Sélection*

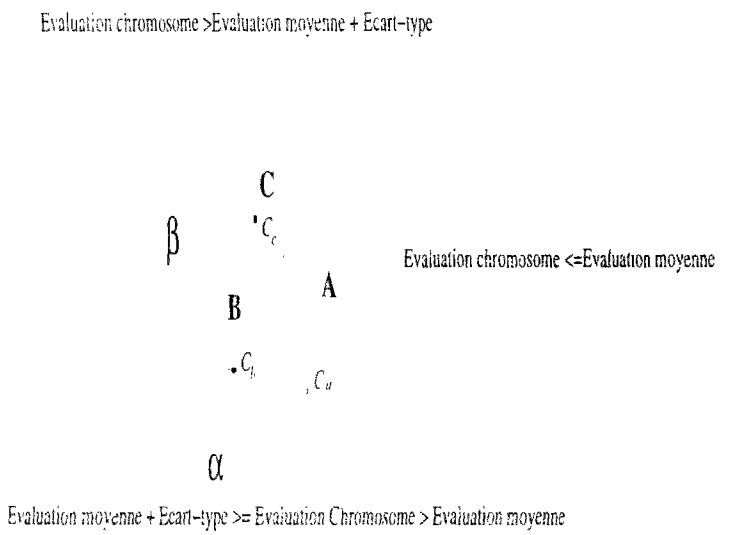
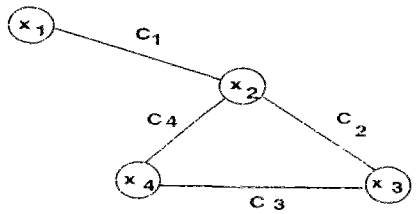


FIG. 3.13 – *Régions de Sélection*



		Fonction d'evaluation		Probabilite de sélection		
	contraintes violées	$Z_{std}(I)$	Z(I)	P_r	P	Région
Pré–solution 1	C 4	1	5	0.28	0.41	A
Pré–solution 2	C 1	1	4	0.30	0.41	A
Pré–solution 3	C 4 C 3	2	9	0.20	0.03	C
Pré–solution 4	C 1 C 3	2	8	0.21	0.15	B

FIG. 3.14 - *Exemple de Sélection*

chromosomes ou pré-solutions. Chacun des individus viole au moins une contrainte. Nous pouvons regarder la valeur de leur fonction d'évaluation $Z_{std}(\mathbf{I})$ et $Z(\mathbf{I})$. Les probabilités de sélection montrées sur la figure sont calculées en utilisant $Z(\mathbf{I})$ pour les deux algorithmes de sélection. P_r est la probabilité pour la méthode de la roue biaisée et P pour notre méthode. Nous pouvons conclure que notre algorithme est plus strict avec les mauvais chromosomes, par contre les bons chromosomes qui sont classés dans la région A sont traités avec égalité, il y a aussi plus de chance de choisir un chromosome de la région A qu'avec la méthode de la roue biaisée.

Dans l'annexe A, nous calculons la probabilité de sélection en utilisant la roue biaisée, pour le traitement des meilleurs individus (ceux qui sont classés par notre méthode dans la sous-population A).

3.4.4 Tests sur différents ensembles de problèmes de 3-coloriage

Nous avons choisi, pour évaluer notre fonction dans l'algorithme évolutionniste, le problème de 3-coloriage sur des graphes générés d'une façon aléatoire. Nous avons conçu différents tests qui sont classés selon les opérateurs génétiques utilisés. Le premier ensemble de problèmes utilise les opérateurs standards: *croisement à un point*

```
Procédure Algorithme Évolutionniste pour CSP
Début
t = 0
initialiser population P(t)
évaluer les individus en P(t) en utilisant Z(I)
tant que (non condition de fin) faire
      t=t+1
     tant que (P(t) n'est pas complète) faire
           Parent1 = sélection[a](P(t-1))
           Parent2 = sélection(P(t-1))
           Enfants = transformation(Parent1, Parent2)
           P(t) = Enfants ∪ P(t)
      évaluer P(t) en utilisant Z(I)
Fin /* procédure Algorithme Évolutionniste pour CSP */
```

[a] algorithme de sélection défini sur la figure 3.12

FIG. 3.15 – *Structure de l'Algorithme Évolutionniste proposé pour résoudre les CSP*

et *mutation*. Les résultats obtenus avec ces opérateurs nous ont motivée pour définir un nouvel opérateur nommé *permutation* qui aidera à améliorer ces résultats. Enfin, le dernier ensemble de problèmes utilise un algorithme avec l'opérateur asexué(n,p,g) avec les paramètres (#,r,b) spécialement défini par Eiben et al. en [ERR95] pour le problème de 3-coloriage (voir chapitre 2). Nous avons comparé les résultats obtenus en utilisant $Z_{std}(\mathbf{I})$ et $Z(\mathbf{I})$ comme fonctions d'évaluation. Pour tous les tests, nous avons utilisée une taille de la population de 20 individus, une probabilité de croisement de 0.9 et une probabilité de mutation de 0.1. La structure générale de l'algorithme proposé est montré sur la figure 3.15. Il utilise la fonction d'évaluation $Z(\mathbf{I})$ et l'algorithme de sélection défini par la figure 3.12. La transformation dépend des opérateurs utilisés sur les différents ensembles de problèmes.

Tests avec opérateurs standards: *croisement à un point, mutation*

Considérons d'abord le petit graphe pour 3-coloriage avec seulement sept variables et onze contraintes illustré par la figure 3.16. Ce graphe en particulier pourrait être réduit au sous-graphe composé par les nœuds 2, 4 et 6, en appliquant l'algorithme

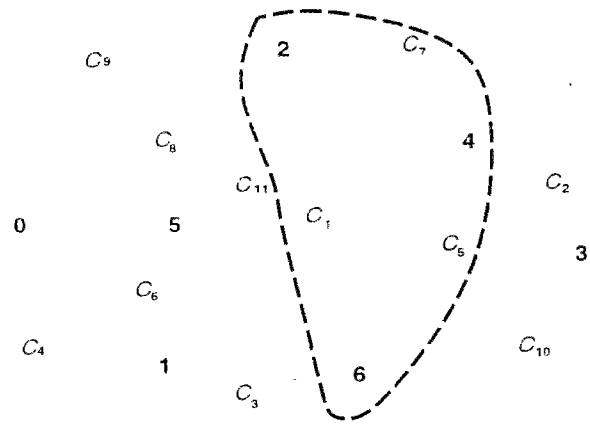


FIG. 3.16 – *Graphe 3-coloriage*

de réduction proposé par Cheeseman [CKT91]. Si nous avons une instanciation qui satisfait les nœuds 2,4 et 6 (lignes coupées), il sera facile de trouver une valeur pour les autres variables qui satisferont toutes les contraintes. Cependant, pour illustrer notre algorithme, la recherche d'une solution a été faite avec tous les nœuds en utilisant la nouvelle fonction d'évaluation. Avec cette fonction d'évaluation, nous donnons une certaine priorité aux contraintes pour les satisfaire. Par exemple dans le graphe, la contrainte C_1 entre le nœud 2 et le nœud 6 est plus importante à satisfaire que la contrainte C_{10} qui lie le nœud 3 avec le nœud 6, même en partageant le même nœud 6. En effet, la contrainte C_1 à travers le nœud 2 est connectée à C_9, C_8, C_{11} et C_7, par contre la contrainte C_{10} à travers le nœud 3 est seulement connectée à la contrainte C_2. Cela confirme la réduction qu'on pourrait faire en faisant la pré-analyse proposée par Cheeseman. Cet exemple est trivial, mais son but est de montrer que la structure est une chose importante à considérer pour améliorer la performance de la recherche stochastique faite par un algorithme évolutionniste. Nous avons généré six différentes topologies de graphes avec 7 variables et 11 contraintes. Nous avons résolu le 3-coloriage de ces graphes en utilisant un algorithme avec $Z_{std}(\mathbf{I})$, l'*Algorithme A*, et un algorithme avec $Z(\mathbf{I})$, l'*Algorithme B*, toutes les autres composantes de l'EA étant évidemment les mêmes pour les deux. Pour chaque graphe généré, nous avons changé l'ordre des variables dans la représentation du chromosome.

Les résultats sont montrés sur la figure 3.17 pour l'*Algorithme A* et sur la figure 3.18 pour l'*Algorithme B*.

En observant ces résultats, on peut conclure les faits suivants:

- Le nombre de générations en utilisant $Z(\mathbf{I})$ est inférieur à celui obtenu en utilisant $Z_{std}(\mathbf{I})$. En fixant le nombre maximum d'itérations à 100, $Z(\mathbf{I})$ arrive à trouver la solution avec un nombre de générations moyen de 20, en revanche avec $Z_{std}(\mathbf{I})$ le nombre d'itérations moyen est de 48.

- Le nombre de générations utilisé pour les deux algorithmes est influencé par l'ordre dans lequel les variables sont organisées dans le chromosome, c'est à dire, qu'il y a des ordres qui permettent de trouver plus facilement la solution que d'autres. Par exemple, pour la topologie 4, l'ordre 19 a empêché l'*Algorithme A* avec $Z_{std}(\mathbf{I})$ de trouver une solution. Pour le même problème et avec le même ordre des variables, l'*Algorithme B* a trouvé une solution.

- Le surcoût de l'utilisation de Z(**I**) à la place de $Z_{std}(\mathbf{I})$ est négligeable, car on détermine la valeur de la *contribution* (voir def 3.2.3) de chaque contrainte C_α au début de l'algorithme. Pour calculer la valeur de Z(**I**), on vérifie si la contrainte est violée ou pas (comme pour $Z_{std}(\mathbf{I})$). La seule différence reside dans le cas où la contrainte n'est pas satisfaite, car on ajoute à la fonction d'évaluation la valeur de sa *contribution* au lieu de 1 (comme le fait $Z_{std}(\mathbf{I})$).

Ensuite, nous avons généré un problème avec 30 variables et 40 contraintes, les résultats des deux algorithmes sont comparés sur la figure 3.19, toujours pour 30 ordres différents des variables dans le chromosome, un nombre maximum d'itérations égal à 100, donc nous pouvons conclure:

- Pour l'*Algorithme B*, en moyenne, il a été plus facile de trouver une solution. Néanmoins, pour l'*Algorithme A*, le "meilleur ordre" a permis de trouver une solution avec 12 itérations, en revanche pour l'*Algorithme B* l'ordre 11 a permis de trouver la solution en faisant 16 itérations.

- L'*Algorithme A* n'a pas pu trouver la solution pour un ensemble plus important d'ordres que l'*Algorithme B*, qui a échoué seulement avec l'ordre 13.

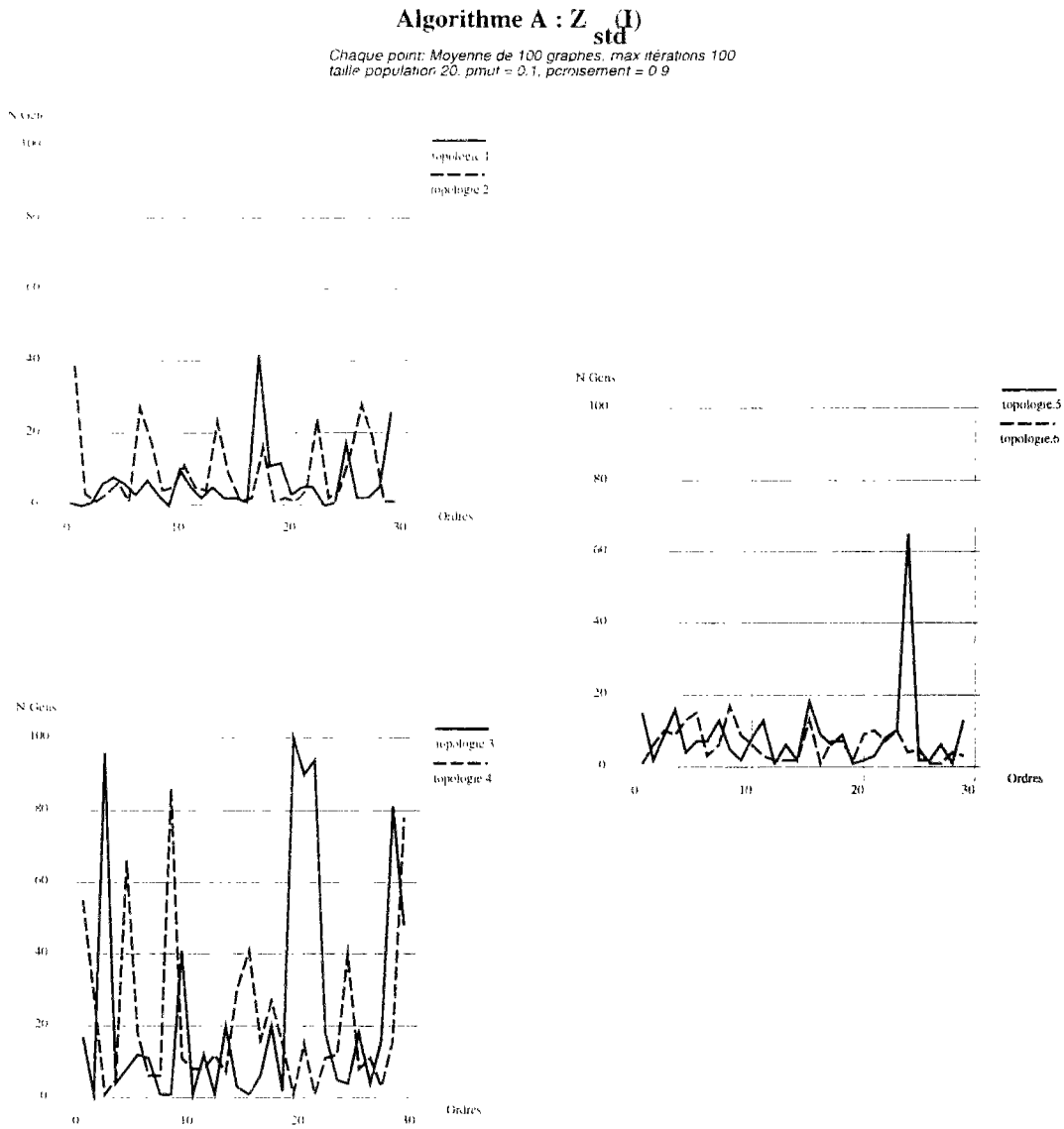


FIG. 3.17 – *Algorithme A:Graphes avec 7 variables et 11 contraintes. Echelle du nombre de générations entre 0 et 100*

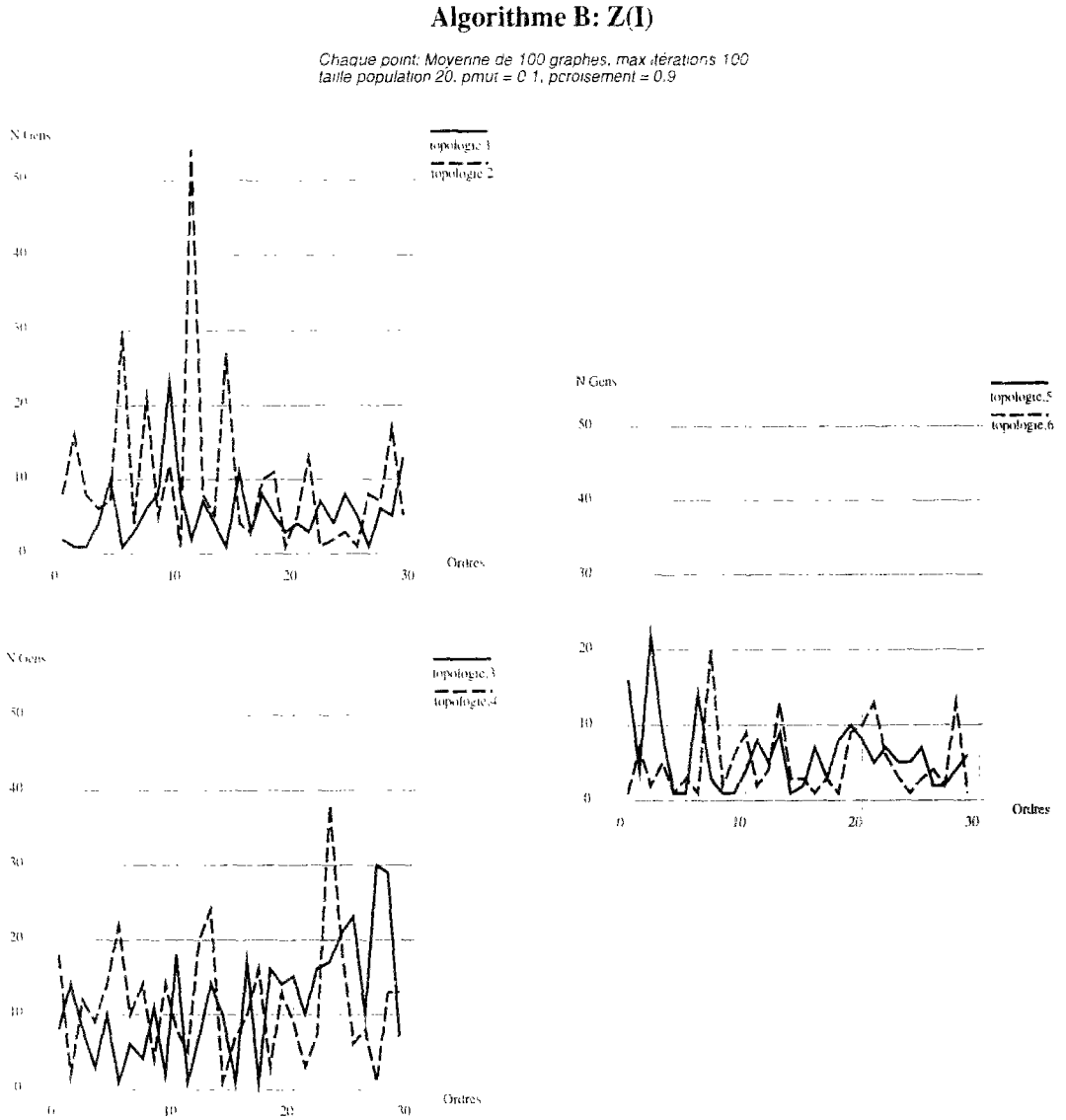


FIG. 3.18 – *Algorithme B: Graphes avec 7 variables et 11 contraintes. Echelle du nombre de générations entre 0 et 50*

Il est important de remarquer que nous n'avons pas une connaissance a priori du bon ordre des variables dans le chromosome. À partir de ces résultats, nous pouvons donc conclure finalement que considérer la structure dans la fonction d'évaluation peut conduire à une amélioration de la performance de l'algorithme, même en choisissant un mauvais ordre des variables dans le chromosome. L'ordre des variables dans le chromosome joue un rôle important aussi dans la performance de l'algorithme. Cela a été considéré par Goldberg [Gol89] et plus récemennt par Kargupta [Kar95] pour la conception des algorithme génétiques "messy". Cela motive l'introduction d'un opérateur de "permutation" qui nous permettra de modifier la valeur de la *longueur utile d'un schéma*, $\delta(Sch)$, (voir définition 2.2.7) en changeant de position certaines variables dans le chromosome, en conséquence la probabilité de destruction du schéma (équation 2.6) changera aussi. L'objectif de cet opérateur de "permutation" est d'augmenter la potentialité de l'opérateur de croisement à un point.

Opérateur de permutation L'opérateur de permutation sera activé seulement une fois que l'algorithme réalise que l'ordre des variables dans le chromosome n'est pas bon. S'il a choisi au début un bon ordre, l'algorithme converge naturellement sans une aide supplémentaire. Nous incorporons un autre paramètre, la *probabilité de permutation*, qui correspond à la probabilité de changer l'ordre des variables dans le chromosome. Contrairement aux opérateurs de croisement et mutation, pour cet opérateur, c'est toute la population qui est concernée, c'est-à-dire, si l'algorithme décide de réaliser une permutation, il changera de la même manière la position des variables dans la population entière. Si l'opérateur n'est pas activé, la probabilité de permutation est nulle. Nous avons conçu une simple heuristique pour identifier un mauvais ordre des variables. Avant de l'introduire, nous définissons le concept de stabilité:

Définition 3.4.1 *(Stabilité dans la Recherche)*
On dit qu'un ordre est stable si l'algorithme évolutionniste a trouvé le même meilleur chromosome pendant les S dernières générations.

La figure 3.20 montre l'instant pendant le processus d'évolution où l'opérateur pourrait être activé. La structure de l'algorithme de l'opérateur de permutation est

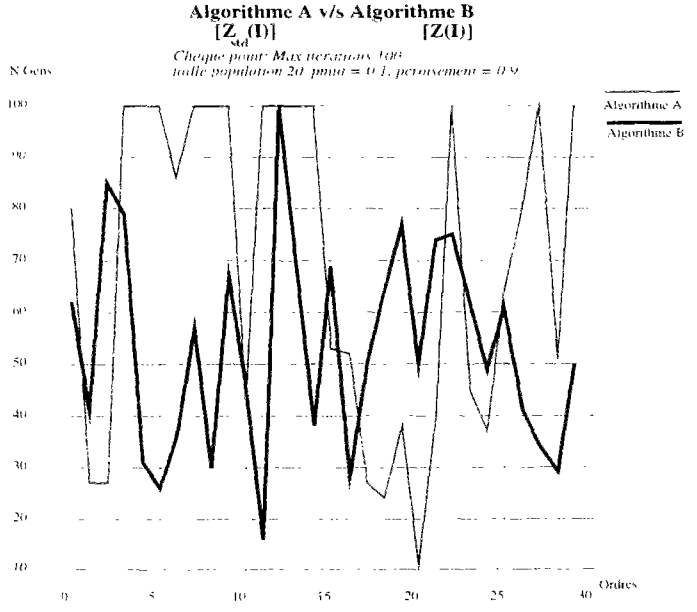


FIG. 3.19 – *Graphe avec 30 variables, 40 contraintes: Algorithme A v/s Algorithme B*

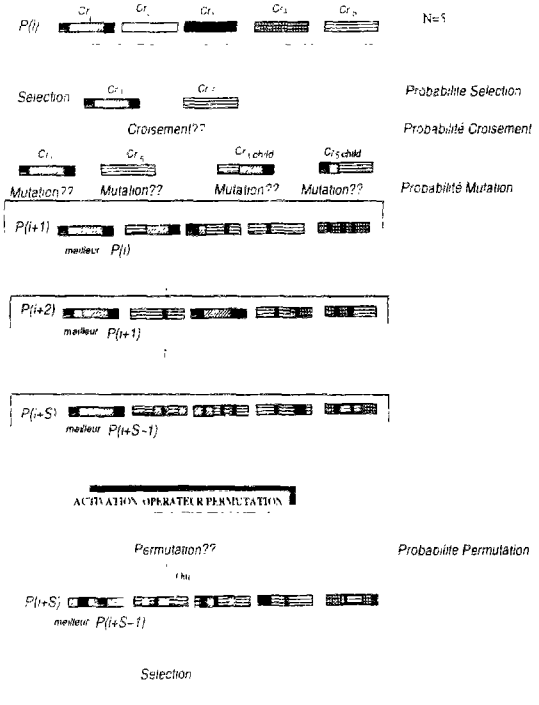


FIG. 3.20 – *Activation de l'opérateur de permutation*

Procédure Opérateur Permutation (Population)
Début
si l'opérateur de permutation est activé **alors**
Générer un nombre aléatoire r à partir $U[0..1]$
si r <probabilité de permutation **alors**
Générer un nouvel ordre du chromosome pour $[1..n]$
Pour tous les chromosomes dans la population
Positionner (X_j, x_j) suivant le nouvel ordre, $\forall j = 1, \ldots, n$
Fin /* opérateur permutation*/

FIG. 3.21 – *Structure de l'opérateur permutation*

montrée sur la figure 3.21. Une fois que l'opérateur est activé, l'algorithme l'applique pendant toutes les futures générations selon la *probabilité de permutation*. La valeur de la probabilité de permutation est faible pour ne pas perturber la recherche réalisée avec un nouvel ordre.

Nous avons incorporé cet opérateur dans les algorithmes A et B de la section précédente. Nous avons fait des tests avec S=25 et une probabilité de permutation de 0.3. Les résultats ont montré une augmentation de la performance pour les pires ordres de 20%, donc l'*algorithme B* cette fois-ci a été capable de trouver une solution pour tous les problèmes.

On pourrait aussi utiliser d'autres critères plus complexes et coûteux du point de vue du temps de calcul pour mesurer la stabilité, tels que l'entropie définie en [FF95] pour le 3-coloriage comme:

Définition 3.4.2 *(Entropie)*
Soit n_{ij} le nombre de fois que le nœud i prend la couleur j dans la population P. On définit la mesure de la diversité de la population entropie *par:*

$$E = \frac{-\sum_{i=1}^{|V|}\sum_{j=1}^{k}\frac{n_{ij}}{|P|}log(n_{ij}|P|)}{|V|logk} \tag{3.6}$$

Cette entropie $E \in [0, 1]$ et elle indique à quel point les couleurs sont uniformément affectées aux nœuds dans la population.

Algorithme	Fonction d'évaluation	Sélection
A	$Z_{std}(\mathbf{I})$	Roue Biaisée
B	Z(**I**)	Roue Biaisée
C	Z(**I**)	Nouvelle sélection

TAB. 3.2 – *Trois algorithmes qui diffèrent par leur fonction d'évaluation et par leur algorithme de sélection*

Nous pouvons décider que dans le cas où la population n'est pas assez diversifiée (petite entropie), alors on active la permutation.

Tests avec l'opérateur spécialisé: (#,r,b)

Dans cet ensemble de tests, nous avons comparé trois algorithmes, lesquels diffèrent par leur fonction d'évaluation et par leur algorithme de sélection, comme cela est montré dans le tableau 3.2. Tous les trois utilisent l'opérateur asexué (n,p,g) avec les paramètres spécifiques pour le problème de 3-coloriage (#,r,b) (pour sa définition voir section 2.5.2). Pour cela, nous avons généré 1000 CSP aléatoires avec différentes topologies avec un degré de connectivité entre 4 et 6 pour 30 variables. Pour chaque connectivité, nous avons généré 100 différents graphes aléatoires.

La figure 3.22 montre le pourcentage de solutions trouvées pour les trois algorithmes. Les meilleurs résultats ont été trouvés en utilisant $Z(\mathbf{I})$ avec le nouvel algorithme de sélection. Il a trouvé dans le pire des cas 70% de solutions contrairement à l'*Algorithme A* qui lui a trouvé seulement 20% de solutions. La figure 3.23 montre le nombre moyen de générations pour chaque connectivité. Le nombre moyen d'itérations réalisé par l'*Algorithme A* est supérieur à celui des deux autres algorithmes. De plus, on peut observer que la nouvelle fonction donne de meilleurs résultats quand elle est couplée avec le nouvel algorithme de sélection.

Les problèmes qui ont été les plus difficiles à résoudre pour les trois algorithmes se trouvent parmi les connectivités [4.5, 5.0], ce qui correspond à la zone connue des problèmes difficiles de 3-coloriage [CKT91], [Smi95].

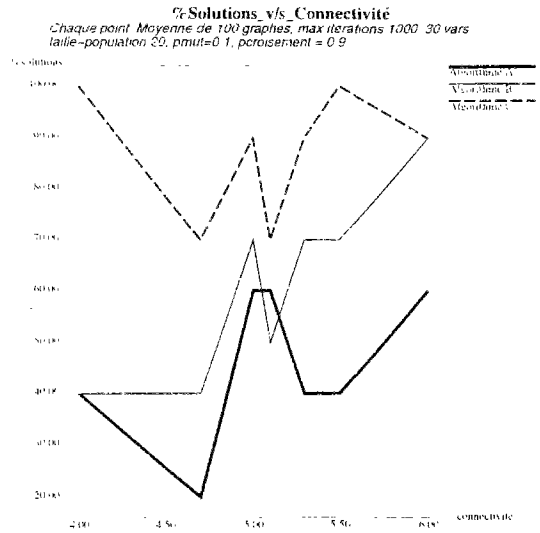


FIG. 3.22 – *Pourcentage de solutions trouvées par les trois algorithmes pour différentes connectivités*

3.5 Fonction d'évaluation pour CSP n-aire

Pour faire l'extension de la fonction d'évaluation présentée dans les sections précédentes aux CSP n-aires, il faut prendre en compte la réflexion suivante:

Si nous avons un chromosome dont les allèles ne satisfont pas une contrainte, pour le réparer, dans le pire de cas, nous devrions changer toutes les valeurs des variables qui appartiennent à cette contrainte, et les valeurs des variables qui leur sont connectées par d'autres contraintes dans le réseau. L'effet de propagation donc utilise la même idée que pour les réseaux binaires, comme on le montre dans les définitions suivantes:

Définition 3.5.1 *(Évaluation de l'erreur n-aire)*
Étant donné un CSP $P = (V, D, \zeta)$ *avec une matrice de contraintes* $\mathbf{R}$ *et une instanciation* $\mathbf{I}$, *la fonction de l'évaluation de l'erreur n-aire* $\mathbf{e_n}(C_\alpha, \mathbf{I})$ *pour une contrainte* C_α *qui a comme variables pertinentes* $X_{k_i}, i \in [1, \ldots, l], l \leq n$, $(\mathbf{R}[\alpha, k_i] = 1, \forall i)$, *est définie par:*

$$\mathbf{e_n}(C_\alpha, \mathbf{I}) = (\textit{Nombre de variables en } C_\alpha) + \sum_i (\textit{Effet de propagation par } X_{k_i})$$

où l'effet de propagation par X_{k_i} *dans un réseau de contraintes n-aires est défini comme le nombre de contraintes* $C_\beta, \beta = 1, \ldots, \eta, \beta \neq \alpha$ *qui ont aussi comme variable*

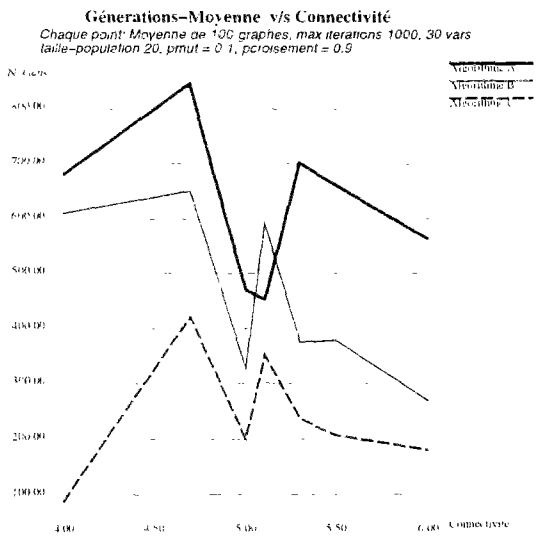


FIG. 3.23 – *Comparaison: Nombre moyen de générations des trois algorithmes pour différentes connectivités*

pertinente X_{k_i}. *Cela s'exprime en utilisant la matrice de contraintes par:*

$$\mathbf{e_n}(C_\alpha, \mathbf{I}) = \left(\sum_{w=1}^{n} \mathbf{R}[\alpha, w]\right) + \left(\sum_{w=1}^{n} \mathbf{R}[\alpha, w] \left[\sum_{\beta \neq \alpha, \beta=1}^{\eta} \mathbf{R}[\beta, w]\right]\right) \tag{3.7}$$

Remarque 3.5.1 *Si* C_γ *est satisfaite* $\mathbf{e_n}(C_\gamma, \mathbf{I}) = 0$

Nous illustrons l'effet de propagation n-aire sur la figure 3.24. Chaque contrainte est représentée par un carré, par exemple les variables pertinentes pour la contrainte C_α sont X_1, X_2 et X_3. Leur effet de propagation dans la matrice de contraintes $\mathbf{R}$, correspond à la somme des valeurs sur leurs colonnes (sans compter ceux de la file α).

Définition 3.5.2 *(Contribution n-aire de* C_α*)*
Étant donné un CSP $P = (V, D, \zeta)$, *on dira que la Contribution n-aire de* C_α *à la fonction d'évaluation* $c_n(C_\alpha)$ *sera:*

$$\mathbf{c_n}(C_\alpha) = \mathbf{e_n}(C_\alpha, I_j), \text{ quand } C_\alpha \text{ est violée sous } I_j.$$

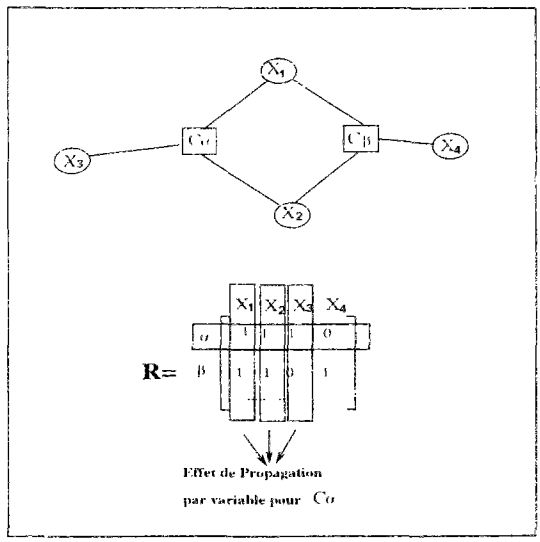


FIG. 3.24 – *Effet de Propagation pour CSP n-aire*

Finalement la fonction d'évaluation pour un CSP n-aire est définie par:

Définition 3.5.3 *(Fonction d'Évaluation pour CSP n-aire)*
Étant donné un CSP avec une matrice de contraintes $\mathbf{R}$*, une instanciation* $\mathbf{I}$ *et la fonction d'Évaluation de l'erreur n-aire* $\mathbf{e_n}(C_\alpha, \mathbf{I})$ *pour chaque contrainte* $C_\alpha, (\alpha = 1, \ldots, \eta)$*, la Fonction d'Évaluation n-aire* $\mathbf{Z_n}(\mathbf{I})$ *est définie par:*

$$\mathbf{Z_n}(\mathbf{I}) = \sum_{\alpha=1}^{\eta} \mathbf{e_n}(C_\alpha, \mathbf{I}) \tag{3.8}$$

Dans le cas où nous avons un CSP binaire, cette fonction est équivalente à la fonction de la définition 3.2.4. De la même manière, nous avons une préférence pour les chromosomes dont les valeurs des variables satisfont plus d'arêtes et de liaisons. Il reste à tester cette fonction d'évaluation.

3.6 Conclusion du chapitre

Nous avons défini une nouvelle fonction d'évaluation $Z(\mathbf{I})$, en prenant en compte la structure du graphe de contraintes, ce qui permet de mieux guider la recherche

stochastique que ne le fait la *fonction d'évaluation standard*. Nous avons proposé un nouvel algorithme du type escalade en deux étapes, qui combine *min-conflits* et *min-réseau-conflits*, heuristiques basées sur les fonctions d'évaluation $Z_{std}(\mathbf{I})$ et $Z(\mathbf{I})$. Cet algorithme a obtenu un pourcentage de convergence vers la solution plus élevé qu'avec *min-conflits* seul, sur des exemples.

Nous avons aussi proposé un algorithme évolutionniste, guidé par $Z(\mathbf{I})$ et avec une nouvelle méthode de sélection qui a fourni de bons résultats pour le problème de 3-coloriage.

L'opérateur standard de *croisement à un point* rend l'algorithme évolutionniste influençable par l'ordre dans lequel se trouvent les variables dans le chromosome. C'est cela qui nous a motivée à proposer un nouvel opérateur, activé seulement lorsque l'algorithme n'arrive pas à trouver une meilleure solution après un certain nombre d'itérations. Ceci a permis d'améliorer les résultats de l'algorithme de 20%.

La fonction d'évaluation $Z(\mathbf{I})$ définie dans ce chapitre est bien adaptée aux problèmes où le graphe est important, par exemple pour le 3-coloriage où toutes les contraintes sont les mêmes, et aussi pour les CSP aléatoires avec toutes les contraintes ayant la même difficulté.

Pour résoudre des CSP non uniformes, il faudrait probablement définir une fonction d'évaluation qui prenne en compte aussi la difficulté des contraintes[2].

Maintenant, notre but est d'incorporer la notion de structure dans la conception de nouveaux opérateurs génétiques, qui permettraient une amélioration de la performance de l'algorithme évolutionniste et qui de plus seraient indépendants de l'ordre des variables dans la représentation.

Dans le chapitre suivant, nous définirons deux nouveaux opérateurs qui prennent en compte cette conclusion et qui sont conçus pour résoudre des problèmes de satisfaction de contraintes généraux. Les travaux exposés dans ce chapitre ont été présentés dans les références [Rif96a], [Rif96b].

2. en anglais:constraint tightness

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 $X_j \rhd C_\alpha$.

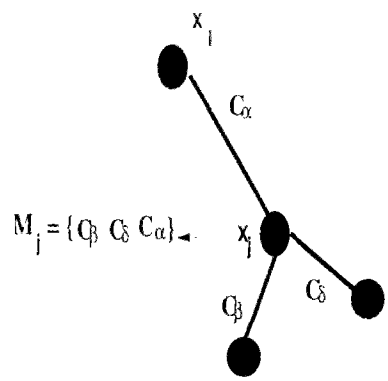


FIG. 4.2 - *Définition 4.2.1*

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.

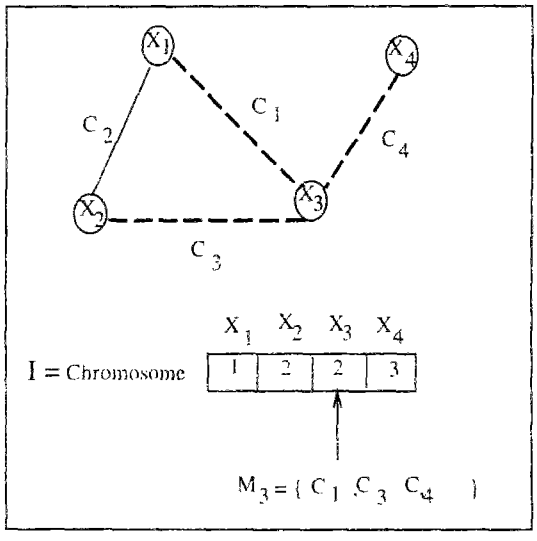


FIG. 4.4 – *Exemple: Arc-mutation*

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

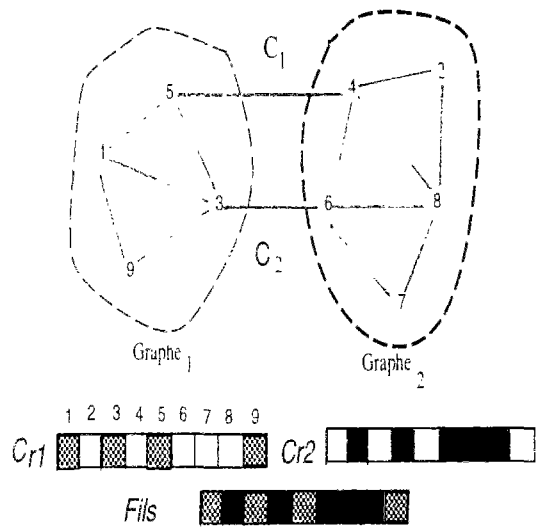


FIG. 4.5 - *Exemple: Bon croisement*

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

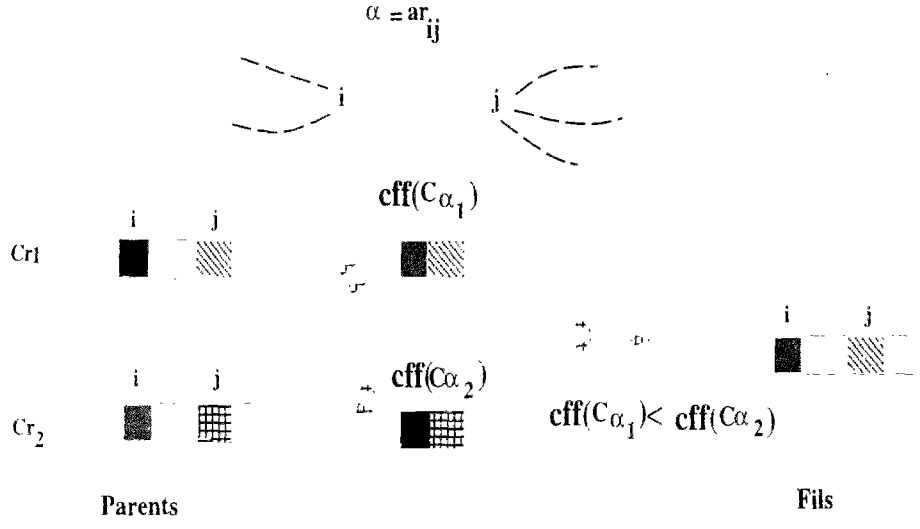


FIG. 4.7 – *Arc-crossover: Croisement pour une arête violée par les deux parents*

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

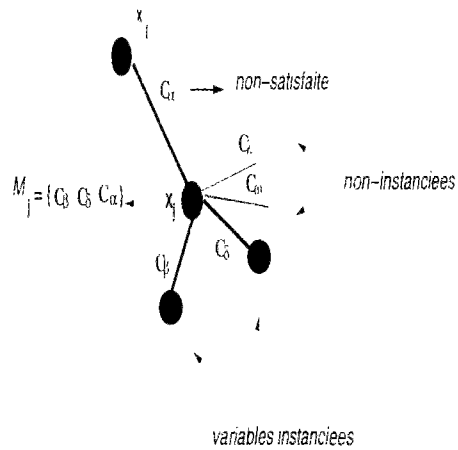


FIG. 4.8 – *Définition 4.2.7*

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:

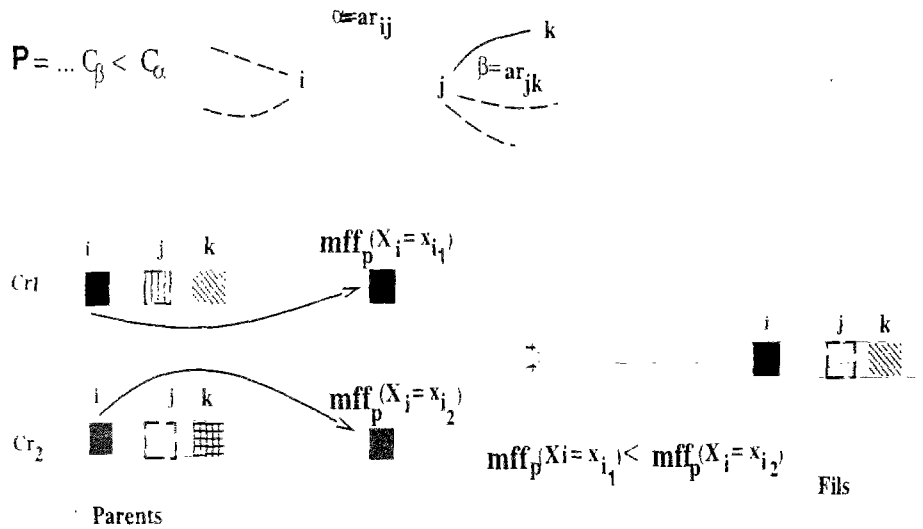


FIG. 4.9 – *Arc-crossover: Héritage de la valeur d'une seule variable qui viole la contrainte* C_α

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*

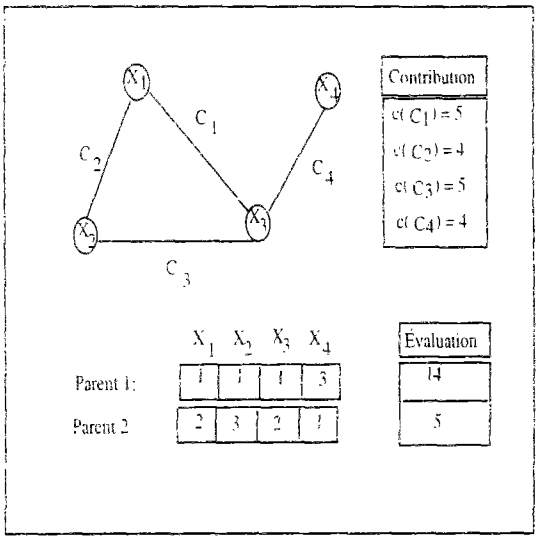


FIG. 4.11 – *Exemple: Arc-crossover*

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 $(\#, r, b)$.

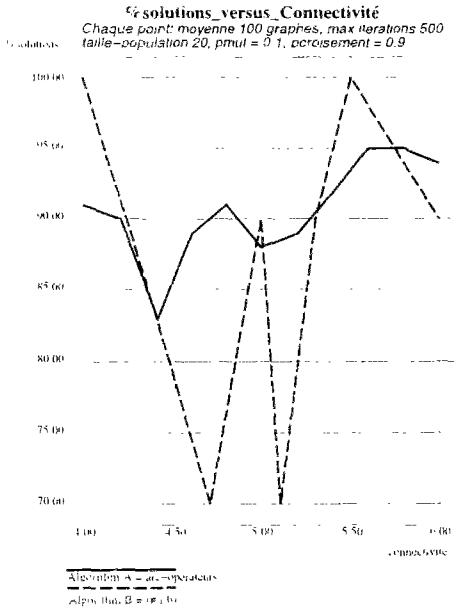


FIG. 4.12 – *Pourcentage de solutions trouvées par Algorithme A et Algorithme B pour différentes connectivités*

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

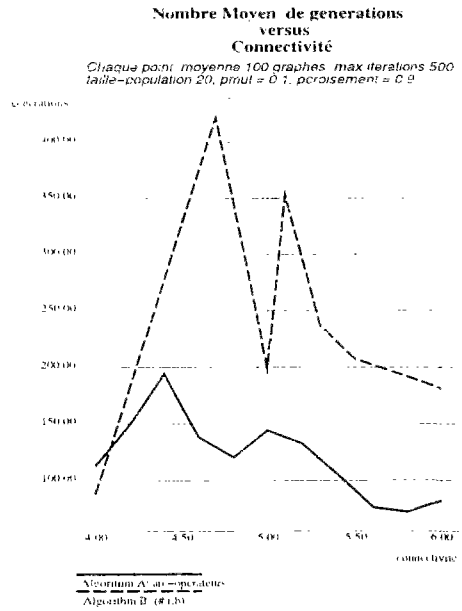


FIG. 4.13 – *Comparaison: Nombre moyen de générations pour Algorithme A et Algorithme B pour différentes connectivités*

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:

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

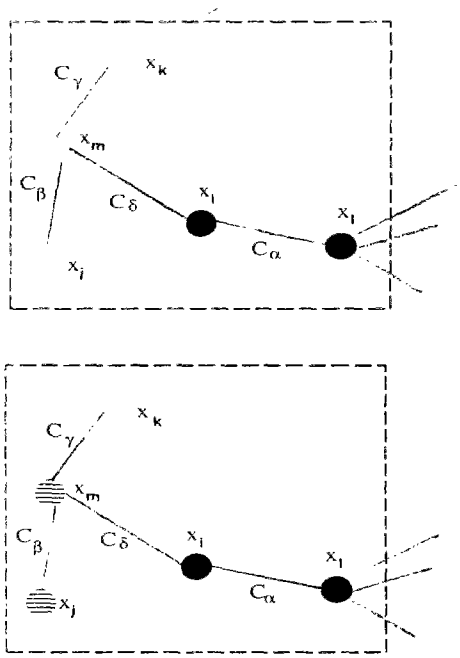


FIG. 4.14 – *Ex: vérifications de contraintes par arc-crossover*

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

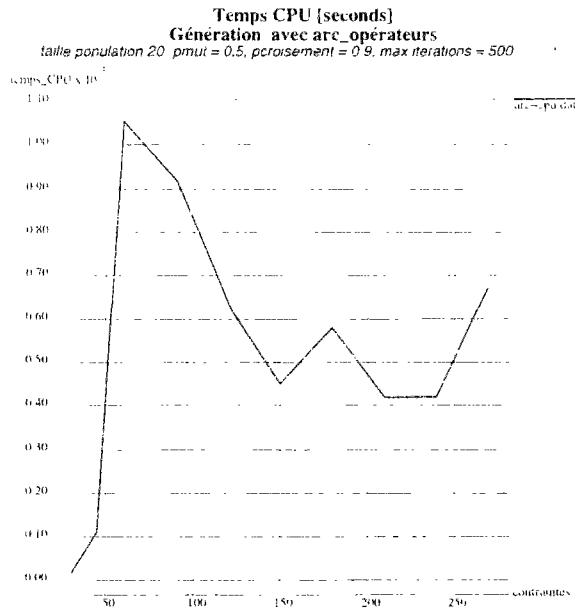


FIG. 4.15 – *Arc-opérateurs: Temps CPU pour résoudre 100 graphes en fonction du nombre de contraintes*

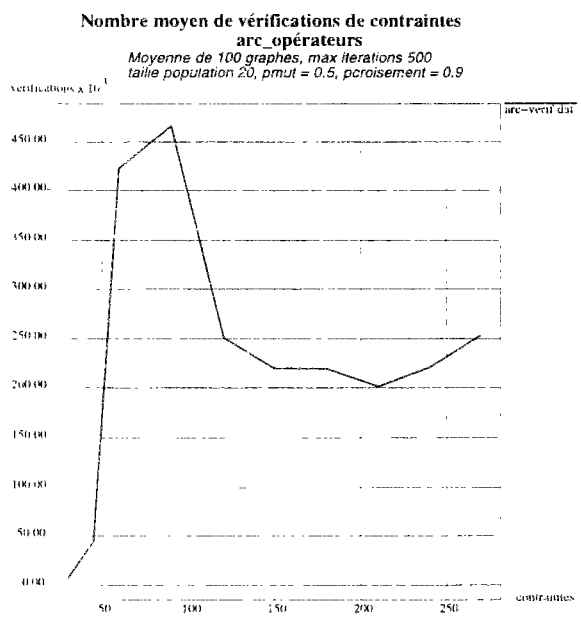


FIG. 4.16 – *Arc-Opérateurs: Nombre moyen de vérifications de contraintes*

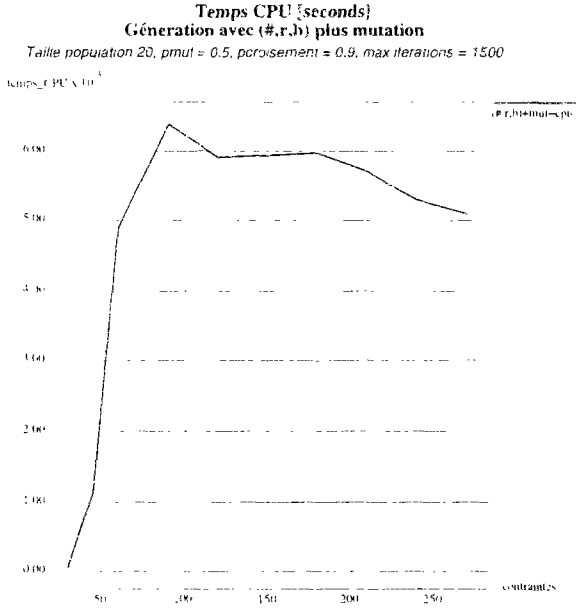


FIG. 4.17 – *(≠,r.b): Temps CPU pour résoudre 100 graphes en fonction du nombre de contraintes*

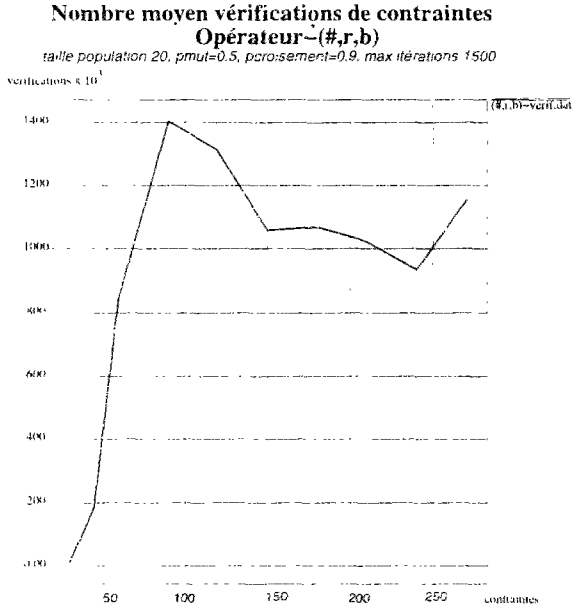


FIG. 4.18 – *(≠,r.b): Nombre moyen de vérifications de contraintes*

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.

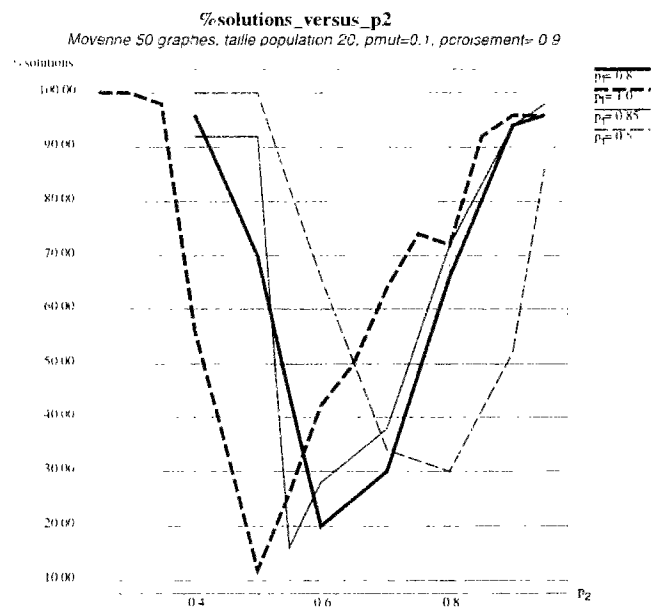


FIG. 4.20 – *%solutions trouvées en fonction de* p_2 *avec* $p_1 \in \{0.5, 0.8, 0.85, 1.0\}$

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

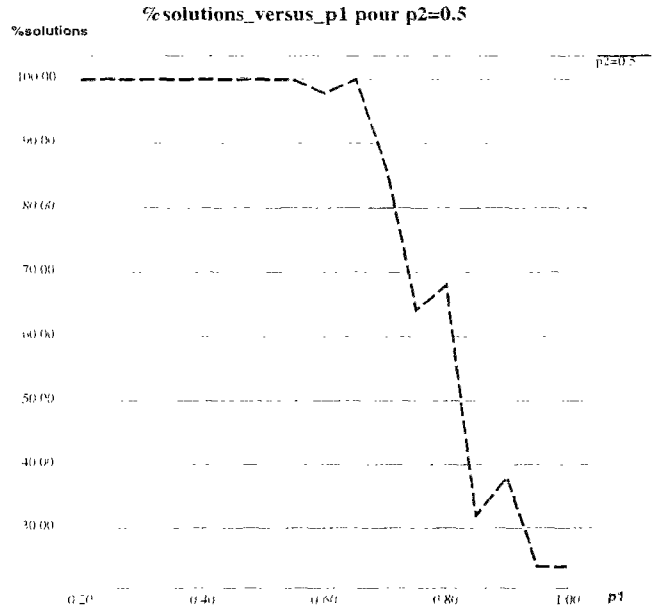


FIG. 4.21 – *%solutions trouvées en fonction de* p_1 *avec* $p_2 = 0.5$

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].

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

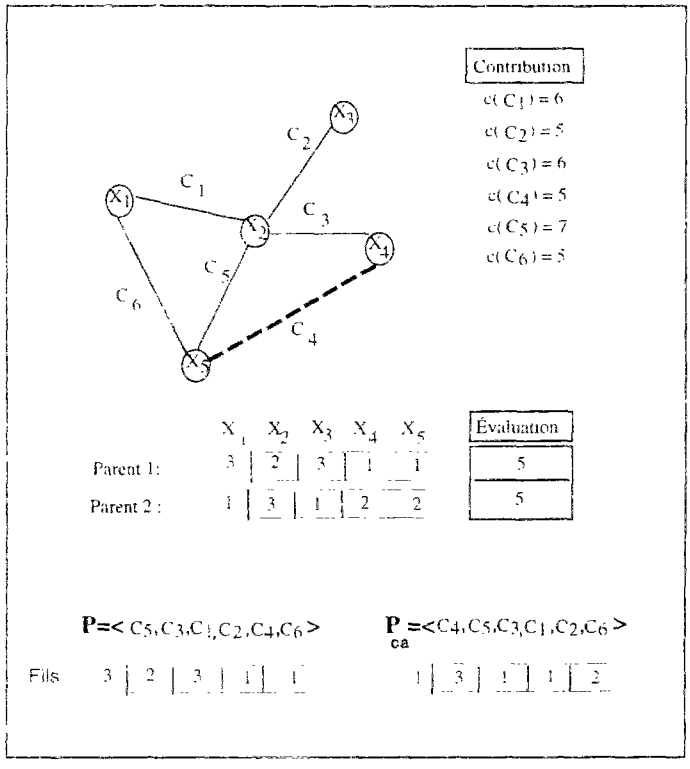


FIG. 5.3 – *Exemple: Différentes Priorités pour Crossover*

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.

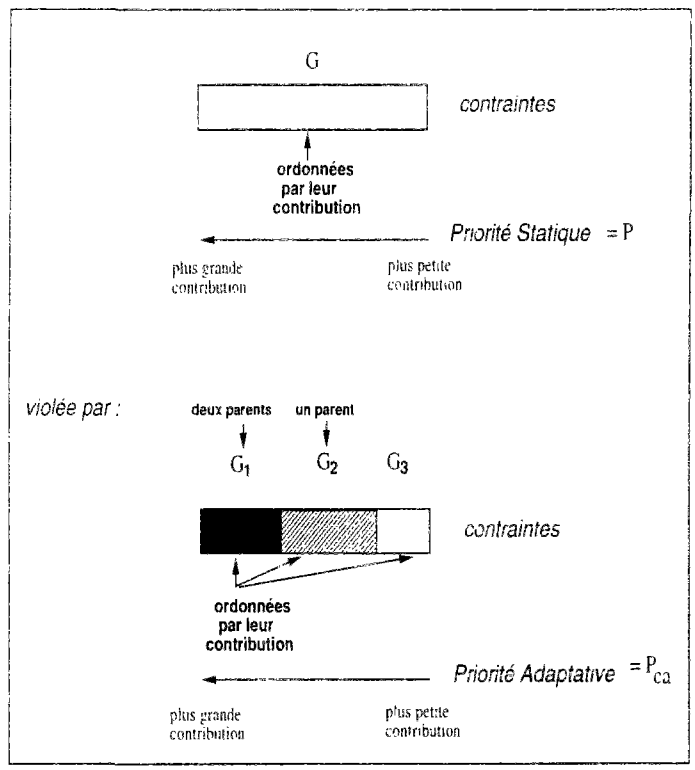


FIG. 5.4 - *Exemple: Priorité Statique et Adaptative pour Crossover*

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].

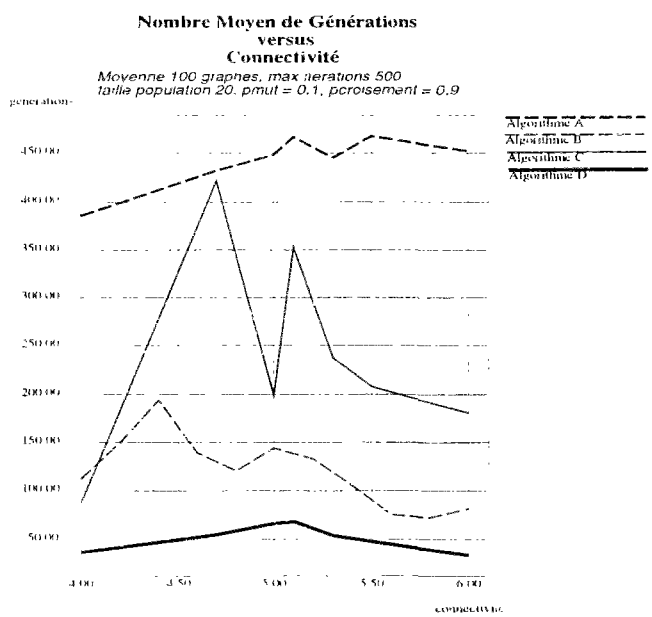


FIG. 5.7 – *Comparaison: Nombre moyen de générations par les Algorithmes A, B, C et D pour différentes connectivités*

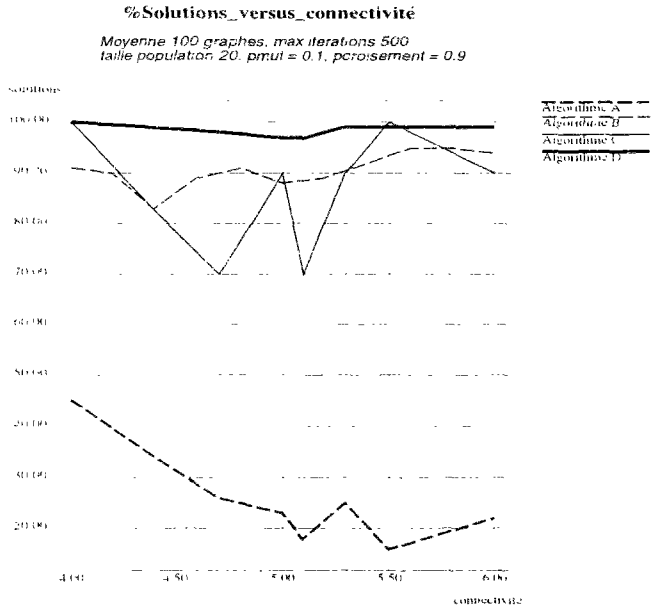


FIG. 5.8 – *Comparaison: Pourcentage de solutions trouvées par les Algorithmes A, B, C et D pour différentes connectivités*

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

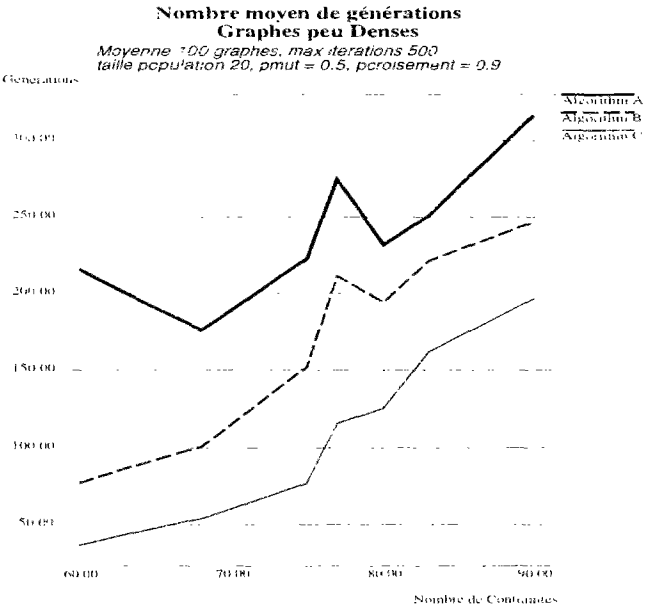


FIG. 5.9 - *Comparaison: Nombre moyen de générations par les Algorithmes A, B et C par nombre de contraintes*

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,

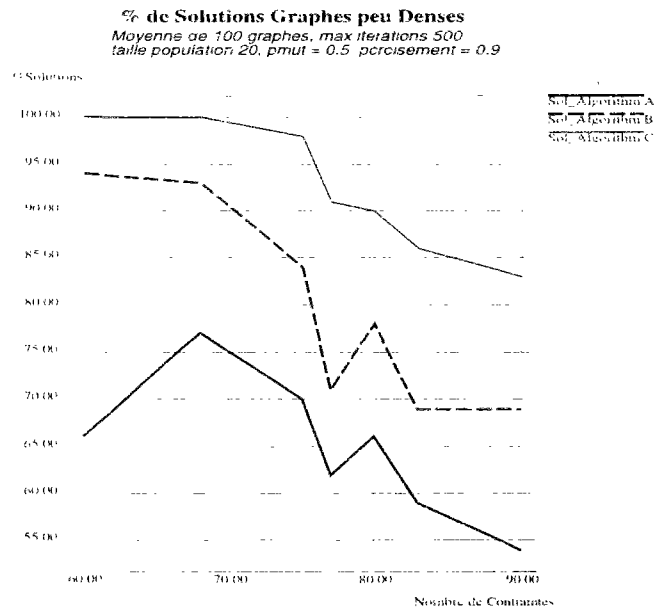


FIG. 5.10 – *Comparaison: Pourcentage de solutions trouvées par les Algorithmes A, B et C par nombre de contraintes*

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

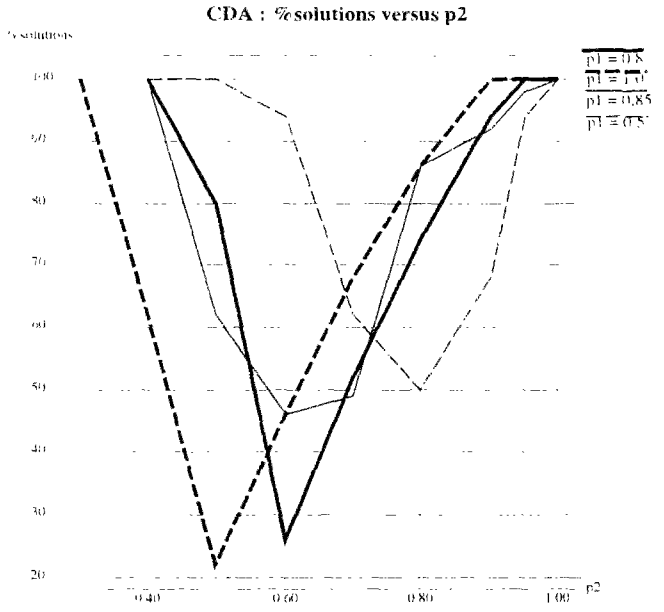


FIG. 5.11 - *CDA: %solutions trouvées en fonction de* p_2 *avec* $p_1 \in \{0.5, 0.8, 0.85, 1.0\}$

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

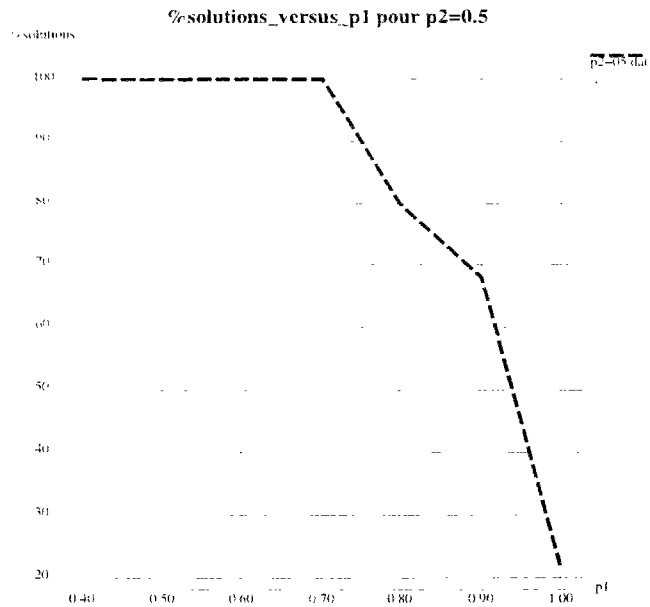


FIG. 5.12 – *CDA: %solutions trouvees en fonction de* p_1 *avec* $p_2 = 0.5$

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

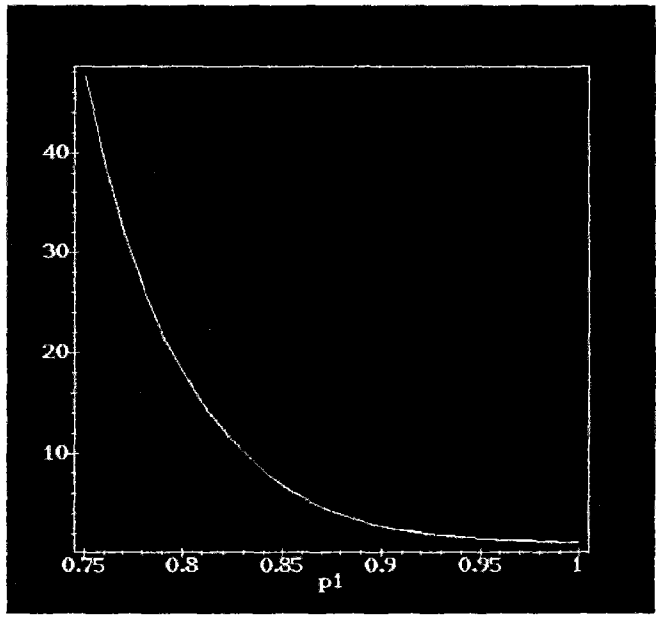


FIG. 5.13 – *Nombre de solutions espérées en fonction de* p_1 *avec* $p_2 = 0.5$

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].

Conclusion

Nous avons étudié dans cette thèse un sujet qui portait sur deux axes de recherche: les Problèmes de Satisfaction de Contraintes et les Algorithmes Évolutionnistes. Pour cette raison, les conclusions seront classées en trois sections. La première section aborde les conclusions dans le cadre des méthodes de résolution pour les Problèmes de Satisfaction de Contraintes. Ensuite, dans la deuxième section, les conclusions sont données du point de vue des Algorithmes Évolutionnistes. Puis, nous présentons des réflexions qui concernent les deux axes de recherche. Finalement, nous présentons des idées pour quelques perspectives de recherche à partir de nos travaux.

Les problèmes de satisfaction de contraintes

Nous avons montré dans le premier chapitre le besoin de concevoir des algorithmes plus performants que ceux existant actuellement pour la résolution des problèmes de satisfaction de contraintes. Dans ce but, plusieurs méthodes complètes et incomplètes de plus en plus efficaces sont proposées dans la littérature. Dans cette thèse, nous nous sommes intéressée plus particulièrement aux méthodes incomplètes. Ces méthodes possèdent l'avantage de se révéler être une bonne alternative pour la résolution de CSP de grande taille, qui peuvent difficilement être résolus par des méthodes systématiques. La principale difficulté pour ces méthodes est de savoir guider efficacement l'algorithme vers une solution en réalisant des réparations locales successives. Pour une méthode de réparation standard, ceci consiste à bien choisir les variables à modifier, ainsi que leurs valeurs et à avoir une bonne fonction d'évaluation, dont le minimum correspond à la satisfaction de toutes les contraintes. Dans cette thèse, nous

avons proposé une nouvelle fonction d'évaluation qui prend en compte la structure du graphe de contraintes, et à partir de laquelle nous avons proposé une nouvelle heuristique pour réaliser une recherche plus performante. Cette heuristique à tout d'abord été intégrée avec des bons résultats, dans une méthode de réparation locale classique. Ensuite, nous avons incorporé cette fonction d'évaluation dans un algorithme évolutionniste pour lequel nous avons obtenu expérimentalement une amélioration sensible de ses performances. Nous nous sommes intéressée plus spécifiquement aux algorithmes évolutionnistes pour la résolution des problèmes de satisfaction de contraintes, car leur méthodologie de résolution a une fonctionnalité supplémentaire par rapport aux méthodes stochastiques classiques. En effet, il est possible de combiner les pré-solutions. La question est alors de savoir comment les combiner avantageusement. Pour cela, nous avons proposé deux opérateurs *arc-mutation* et *arc-crossover*. *Arc-mutation* procède à une sélection de la valeur d'une variable en regardant le graphe de contraintes et en étudiant les conséquences que ce changement pourrait avoir sur les autres contraintes du réseau. *Arc-crossover* essaie de trouver une bonne combinaison des valeurs des pré-solutions en prenant en compte les contraintes. De façon analogue aux heuristiques utilisées dans les méthodes classiques pour la résolution de CSP pour la construction d'un ordre d'instanciation (l'ordonnancement des variables suivant la taille de leur domaine, le choix de la variable la plus connectée), nous avons incorporé une nouvelle heuristique dans l'opérateur de croisement. Celle-ci consiste à rendre prioritaire parmi les contraintes actuellement violées, celle qui contribue le plus fortement à la fonction d'évaluation. Notre approche a été influencée par des concepts proposés par la communauté de chercheurs en CSP, comme par exemple le concept de *structure du graphe* que nous avons considéré dans les définitions des fonctions d'évaluation y compris partielles.

Un autre concept important est celui de la *décomposition* en sous-graphes ou sous-problèmes, que nous avons utilisé lors de la conception des opérateurs de croisement: *arc-crossover*, *constraint-crossover* et *crossover dynamique adaptatif*. La minimisation du *nombre de vérifications de contraintes* est aussi un objectif fondamental pour tout algorithme efficace de résolution de 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 dans la conception de nos opérateurs.

En résumé, nous avons proposé un nouvel algorithme évolutionniste pour la résolution des CSP qui peut se presenter comme une alternative aux méthodes stochastiques traditionnelles (GSAT, *min-conflits*).

Les algorithmes évolutionnistes

Du côté des Algorithmes Évolutionnistes, un des sujets de recherche les plus intéressants est la résolution de problèmes avec contraintes. Les contraintes entraînent généralement une difficulté naturelle dans le déroulement de l'algorithme évolutionniste, à cause du phénomène d'épistasie. La théorie des Algorithmes Génétiques Standards suppose en effet l'indépendance des gènes dans le chromosome. Mais, les contraintes "lient" d'une certaine manière des gènes entre eux, ce qui crée une dépendance. Nous avons présenté une fonction d'évaluation permettant à l'algorithme d'améliorer la satisfaction de sous-graphes, et non plus d'une contrainte en particulier. Les idées sur lesquelles est basée notre fonction, comme la structure du graphe, pourraient être aussi utilisées dans la résolution des problèmes d'optimisation avec contraintes. En effet, ces problèmes sont souvent résolus en ajoutant à la fonction à optimiser une pénalité correspondant aux contraintes violées. Notre fonction pour les CSP pourrait donc jouer le rôle de cette pénalité. La conception de nos opérateurs nous a permis d'incorporer dans un algorithme évolutionniste deux types de méthodes. D'abord, l'opérateur de mutation *arc-mutation* au lieu d'effectuer un changement purement aléatoire fait une sorte d'*escalade* en changeant la valeur d'une variable. D'un autre coté, l'opérateur de croisement *arc-crossover* correspond à un *algorithme glouton* qui construit une pré-solution à partir des deux pré-solutions en essayant de faire le meilleur choix parmi leurs gènes. Finalement, nous avons incorporé le concept d'adaptation qui est un des sujets les plus prometteurs en évolution artificielle, dans l'opérateur de croisement. Et ceci nous a permis d'améliorer la performance de cet opérateur.

En résumé, nous avons conçu une fonction d'évaluation, ainsi que des opérateurs réalisant une recherche guidée par la structure du problème. Ce sont les composantes fondamentales d'une nouvelle méthode évolutionniste pour le traitement de problèmes

de satisfaction de contraintes.

Mises en commun des recherches sur les CSP et EA

Notre algorithme a été testé principalement sur le problème de 3-coloriage de graphe pour deux raisons. La première est que c'est un problème qui ne présente pas de préférence entre les contraintes: c'est un CSP binaire où seule la topologie du graphe importe. La deuxième raison est la possibilité de comparer avec d'autres méthodes proposées dans la littérature, en provenance des deux axes de recherche qui nous intéressent plus particulièrement: CSP et EA. Nous avons constaté que la recherche effectuée dans ces deux communautés peuvent collaborer pour parvenir à résoudre plus efficacement les problèmes de satisfaction de contraintes. Nous nous sommes placee dans le cadre des algorithmes évolutionnistes les plus récents (avec adaptation des opérateurs), et nous avons conçu nos opérateurs spécialisés en étant guidée par les diverses recherches sur les CSP (utilisation de la structure de graphe, heuristique d'ordonnancement, limitation de nombre de tests de contraintes).

Nous nous sommes rendu compte que l'idée d'incorporer des connaissances provenant d'autres domaines de recherche permet parfois une avancée vers l'obtention de meilleurs résultats.

Perspectives

De nombreux points pourraient faire l'objet de recherches supplémentaires. On peut citer parmi eux:

- Tout d'abord, la réalisation de tests sur d'autres types de CSP devrait valider en particulier les extensions proposées pour les CSP n-aires.

- Nos résultats concernent essentiellement la structure du graphe de contraintes et non la sémantique de ces dernières. La considération de la difficulté des contraintes est un sujet important qui devrait être inclus dans la conception

d'un algorithme évolutionniste pour résoudre des problèmes avec des contraintes hétérogènes.

- Les hiérarchies de contraintes sont étudiées pour résoudre les problèmes surcontraints en relâchant automatiquement les contraintes de moindre importance. Il serait intéressant de considérer le concept de hiérarchie dans un algorithme évolutionniste.

- Une autre thème important pour les deux communautés de recherche est la comparaison entre les méthodes évolutionnistes et d'autres méthodes de résolution de CSP, complètes et incomplètes, dans le but de définir un cadre d'applicabilité de chaque méthode.

- Finalement, inclure le concept d'adaptation des paramètres dans les chromosomes (auto-adaptation) semble être un concept puissant afin d'aider les algorithmes à s'échapper des optimums locaux où ils pourraient être piégés.

Annexe A

Probabilité de sélection standard pour la région des meilleurs individus

Le nouvel algorithme de sélection défini dans le chapitre 3, fait une classification des individus selon leur fonction d'évaluation par rapport à l'évaluation moyenne de la population et à son écart-type.

Nous voulons répondre à la question: quelle est la probabilité de sélection, en utilisant la méthode de la roue biaisée, d'un individu qui avec la nouvelle méthode de sélection est classé dans la région A (les meilleurs individus)?

Antécédents:

La nouvelle méthode fait la classification des individus de la façon suivante:

Un individu appartient à la région A si sa fonction d'évaluation $eval(v_i)$ varie entre:

$$0 \leq eval(v_i) \leq F \tag{A.1}$$

La probabilité de sélection avec la roue biaisée, pour un problème de maximisation est:

$$p_s = \frac{g_i}{GN} \tag{A.2}$$

avec g_i l'évaluation du chromosome i avec la fonction à maximiser, G la moyenne des g_i de tous les chromosomes dans la population et N la taille de la population.

Nous avons donc besoin d'une fonction à maximiser. Nous définissons

$$g_i = MAX - eval(v_i) \tag{A.3}$$

où $MAX = \sum_{\forall j} c(C_j)$, c'est à dire MAX correspond à la somme de la contribution à la fonction d'évaluation de toutes les contraintes du problème. Cela est la pire évaluation, c'est le cas où nous aurions toutes les contraintes insatisfaites. En conséquence

$$G = MAX - F \tag{A.4}$$

Nous pouvons réécrire la équation A.2 comme:

$$p_s = \frac{MAX - eval(v_i)}{(MAX - F)N} \tag{A.5}$$

Finalement, les valeurs de la probabilité de sélection pour un individu de la région A avec la méthode de la roue biaisée sera:
Si $eval(v_i) = F - \epsilon$

$$p_s = \frac{MAX - F + \epsilon}{(MAX - F)N}; F \geq \epsilon \geq 0 \tag{A.6}$$

Annexe B

CSP aléatoires avec solution

B.1 Algorithme pour générer des CSP aléatoires avec solution

Chaque jeu de CSP binaires est décrit par quatre paramètres: n, le nombre de variables, m le nombre de valeurs de chaque domaine, p_1 la probabilité qu'il y ait une contrainte entre deux variables, et p_2 la probabilité conditionnelle qu'une paire de valeurs soit inconsistante pour un couple de variables, étant donné qu'il y a une contrainte entre les deux variables. La procédure de génération des CSP aléatoires est montrée sur la figure 4.19.

B.2 Caractéristiques de l'algorithme

Paramètres:

- p_1=probabilité qu'il y a une contrainte entre une paire de variables, c.a.d., p_1= P(il y a une contrainte entre $X_i et X_j$)
- p_2= probabilité conditionnelle qu'une paire de valeurs est inconsistante pour une paire de variables, étant donné qu'il y a une contrainte entre les variables,

p_2= P(une paire de valeurs est inconsistante pour $X_i$et X_j/ il y a une contrainte entre $X_i$etX_j)

n = nombre de variables

- m = taille des domaines

Conditions:

- Toutes les variables sont pertinentes pour au moins une contrainte
- Le CSP il y a au moins une solution.
- Les valeurs de domaines sont indépendantes

B.3 Proprietés statistiques

De la première condition

E[Nombre de contraintes] = E[Nombre d'arcs dans le graphe de contraintes]

Supposons:
Z = Nombre de contraintes générés automatiquement
Y = Nombre de corrections pour les variables non-connectées

Si après avoir généré automatiquement le graphe, il reste des variables qui ne sont pas connectées. Pour chacune de ces variables nous créons une contrainte qui la lie à une autre qui est déjà dans le graphe de contraintes. On ajoute donc des contraintes supplémentaires au problème (une par variable isolée). On modifie donc la valeur initial de p_1

E[Nombre de contraintes] = E[η] = E[Z] + E[Y]

- $E[Z] = p_1 \frac{n(n-1)}{2}$. (voir [Smi95])

- $E[Y] = \sum_{i=1}^{n} P(\text{var i n'est pas connectée})$
  $E[Y] \leq \sum_{n} (1-p_1)^{n-1} = n(1-p_1)^{n-1}$

$$E[\eta] \leq \frac{p_1 n(n-1)}{2} + n(1-p_1)^{(n-1)} = \overline{\eta} \tag{B.1}$$

De la seconde condition

Le nombre espéré de solutions est calculé en utilisant la condition d'existance d'au moins une solution.
Soit:
X= Nombre de solutions, $x_i \in \{0, 1, 2...m^n\}$

$E[X/X \geq 1] = \sum_{x_i=0}^{m^n} x_i P(X = x_i/X \geq 1)$

$P(X = x_i/X \geq 1) = \frac{P(X=x_i \cap X \geq 1)}{P(X \geq 1)}$

$$P(X = x_i/X \geq 1) = \begin{cases} \frac{P(X=x_i)}{P(X \geq 1)} & \text{if } x_i \in \{1, 2, \ldots, m^n\} \\ 0 & \text{if } x_i = 0 \end{cases}$$

X suit une distribution binomiale avec les paramètres (m^n, p_s), ainsi $X \sim b(m^n, p_s)$ où p_s = probabilité d'une solution pour le CSP, donc

$p_s = P(\text{ Tous les arcs sont satisfaits})$

$$p_s = (1-p_2)^{\eta} \tag{B.2}$$

alors

$P(X \geq 1) = 1 - P(X = 0) = 1 - (1-p_s)^{m^n} \approx 1 - exp(-p_s m^n) = \alpha$

$E[X/X \geq 1] = \frac{1}{\alpha} \sum_{x_i=0}^{m^n} x_i P(X = x_i)$

$E[X/X \geq 1] = \frac{1}{\alpha} E[X] = \frac{m^n p_s}{\alpha}$

$$E[X/X \geq 1] = \frac{m^n p_s}{1 - exp(-p_s m^n)} \tag{B.3}$$

nous utilisons l'inégalité de Jensen. [Shi96], pour estimer la valeur de p_s, parce que nous ne connaissons pas la valeur exacte de η.

Par Jensen:

$$E[(1 - p_2)^{\eta}] \geq (1 - p_2)^{\overline{\eta}} \tag{B.4}$$

donc p_s est estimé par

$$\widehat{p_s} = (1 - p_2)^{\overline{\eta}} \tag{B.5}$$

avec $\overline{\eta} = \frac{p_1 n(n-1)}{2} + n(1 - p_1)^{n-1}$

B.4 CSP aléatoires qui peuvent ne pas avoir une solution

Le modèle utilisé typiquement dans la littérature de CSP a comme paramètres p_1, p_2, n, m sans conditions. Smith en [Smi95] a estimé le nombre espéré de contraintes comme $\frac{p_1 n(n-1)}{2}$, et le nombre espéré de solutions: $m^n(1 - p_2)^{\frac{n(n-1)p_1}{2}}$

Bibliographie

[AJ90] H.M. Adorf and M.D. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings International Joint Conference on Neural Networks*, pages 65–67, 1990.

[AS94] J.M. Alliot and T. Schiex. *Intelligence Artificielle & Informatique Theorique*. Cépaduès Éditions. 1994.

[BC97] I. Blot and J. Chabrier. Application de la méthode score(fd/i) aux csps binaires. In *3ème Conférence Nationale pour la Resolution Pratique de Problemes NP-Complets*, pages 75–80, 1997.

[BCSJ86] J. Blazewicz, W. Cellary, R. Slowinski, and Weglarz J. Scheduling under resource constraints: Deterministic models. In *Annals of Operations Research*, 1986.

[Bes94] C. Bessière. Arc-consistency and arc-consistency again. In *Artificial Intelligence*, volume 65, pages 179–190, 1994.

[Bet80] A.D. Bethke. *Genetic Algorithms as Function Optimizers*. PhD thesis, University of Michigan, 1980.

[BFR95] C. Bessière, E. C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *International Joint Conference on Artificial Intelligence*, pages 592–598, 1995.

[BL88] E. Bonomi and J.L. Lutton. Le recuit simulé. In *Pour la Science*, volume 129, 1988.

[Bre79] D. Brelaz. New method to color the vertices of a graph. In *Communications ACM*, volume 22, pages 251–256, 1979.

[BS93] T. Back and H. Schwefel. An overview of evolutionary algorithms for parameter optimization. In *Evolutionary Computation*, volume 1, pages 1–23, 1993.

[CKT91] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *International Joint Conference on Artificial Intelligence*, pages 163–169, 1991.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to ALGORITHMS*. The MIT Press, 1990.

[Dav85] L. Davis. Job shop scheduling with genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 136–140, 1985.

[Dav90] Y. Davidor. Epistasis variance: A viewpoint on representations, GA hardness, and deception. In *Complex Systems*, volume 4, pages 369–383, 1990.

[Dav91] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[DBB94] G. Dozier, J. Bowen, and D. Bahler. Solving small and large scale constraint satisfaction problems using a heuristic-based microgenetic algorithm. In Michalewicz et al. [MSS+94], pages 306–311.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. In *Artificial Intelligence*, volume 41, pages 273–312, 1990.

[Dec92] R. Dechter. Constraint networks. In *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley and Sons, 1992.

[DG97] M. Dorigo and L. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problems. In *IEEE Transactions on Evolutionary Computation*, volume 1, 1997.

[DJ75] K. De-Jong. *An analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan, USA, 1975.

[DMC96] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 26, 1996.

[Dor92] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[DP88] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. In *Artificial Intelligence*, volume 34, pages 1–38, 1988.

[Eib96] A.E. Eiben. *Handbook of Evolutionary Computation*, chapter Constraint Satisfaction Problems. 1996.

[ERR94] A.E. Eiben, P-E. Raué, and Zs Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In Michalewicz et al. [MSS+94], pages 542–547.

[ERR95] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Ga-easy and ga-hard constraint satisfaction problems. In Meyer [Mey95], pages 267–283.

[ERR96] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In Fogel [Fog96], pages 258–261.

[EvdH97] A.E. Eiben and J. van der Haw. Solving 3-sat by gas adapting constraint weights. In Porto [Por97], pages 81–86.

[FF95] C. Fleurent and J. Ferland. Genetic algorithms and hybrids for graph coloring. In *Annals of Operations Research*, 1995.

[Fog92] D. Fogel. *Evolving Artificial Intelligence*. PhD thesis, University of California, USA, 1992.

[Fog96] D. Fogel, editor. *Fourth IEEE International Conference on Evolutionary Computation*. IEEE Press., 1996.

[FOW66] L Fogel, A Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.

[Fre78] E. Freuder. Synthesizing constraint expressions. In *Communications of the ACM*, volume 21, pages 958–966, 1978.

[Fre82] E. Freuder. A sufficient condition of bactrack-free search. In *Journal of the ACM*, volume 29, pages 24–32, 1982.

[Fre95] E. Freuder. The many paths to satisfaction. In Meyer [Mey95], pages 103–119.

[FW92] E. Freuder and R. Wallace. Partial constraint satisfaction. In *Artificial Intelligence*, volume 58, pages 21–70, 1992.

[Gas77] J. Gaschnig. A general backtrack algorithm that eliminates most redundants tests. In *International Joint Conference on Artificial Intelligence*, pages 457–458, 1977.

[GH97] P. Galinier and J-K. Hao. Tabu search for maximal constraint satisfaction problems. In *Constraint Processing (CP97)*, LNCS. Springer-Verlag, 1997.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability*. W.H.Freeman, 1979.

[Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operations Research*, volume 13, pages 533–549, 1986.

[Glo89] F. Glover. Tabu search - part i. In *ORSA Journal of Computing*, volume 1, pages 190–206, 1989.

[Glo90] F. Glover. Tabu search - part ii. In *ORSA Journal of Computing*, volume 2, pages 4–32, 1990.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Gre87] J.J. Grefenstette. Incorporating problem specif knowledge into genetic algorithms. In *Genetic Algorithms and Simulated Annealing*, 1987.

[GTdW93] F. Glover, E. Taillard, and D. de Werra. A users guide to tabu search. In *Annals of Operations Research*, volume 41, pages 3–28, 1993.

[Han86] P. Hansen. The steepest ascend mildest descend heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimisation*, 1986.

[HdW87] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. In *Computing*, volume 39, pages 345–351, 1987.

[HE80] Haralick and Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Artificial Intelligence*, volume 14, pages 263–313, 1980.

[HLQ94] A. Homaifar, H.-Y Lai, and X. Qi. Constrained optimization via génetic algorithms. In *Simulation*, volume 62, pages 242–254, 1994.

[HME97] R. Hinterding, Z. Michalewicz, and A. Eiben. Adaptation in evolutionary computation: A survey. In Porto [Por97], pages 65–69.

[Hof93] R. Hofmann. Examinations on the algebra of genetic algorithms. Master's thesis, Technical University of Munich, 1993.

[Hol73] J.H. Holland. Genetic algorithms and the optimal allocations of trials. In *SIAM Journal of Computing*, volume 2, pages 88–105, 1973.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University Michigan Press, 1975.

[HP98] J-K. Hao and J. Pannier. Simulated annealing and tabu search for constraint solving. In *5th Intl. Symposium on Artificial Intelligence and Mathematics.*, 1998.

[Jeg90] P. Jegou. Cyclic-clustering: a compromise between tree-clustering and the cycle-cutset method for improving search efficiency. In *Proceedings of ECAI90*. pages 369–371. 1990.

[JH94] J. Joines and C. Houck. On the use of non-stationary penalty function to solve non-linear constraint optimization problems with gas. In Michalewicz et al. [MSS+94]. pages 579–584.

[JM97] D. Johnson and L. McGeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*. John Wiley and Sons, New York. 1997.

[Kar95] H. Kargupta. *{SEARCH}, polynomial complexity, and the fast messy genetic algorithm*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[KGV83] S. Kirkpatrick. C. Gellat, and M. Vecchi. Optimization by simulated annealing. In *Science*, volume 220, pages 671–680, 1983.

[Koz89] J. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 768–774, 1989.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems:a survey. In *Artificial Intelligence Magazine*, volume 13, pages 32–44, 1992.

[Lap92] G. Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. In *European Journal of Operational Research*, volume 59, pages 345–358, 1992.

[LK73] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. In *Operations Research*. volume 21. pages 498–516, 1973.

[LV90] G.E. Liepins and M.D. Vose. Representational issues in genetic optimization. In *Journal of Experimental and Theorical Artificial Intelligence*, volume 2, pages 4–30, 1990.

[MC91] Z. Michalewicz and J. Cezary. Handling constraints in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 151–157, 1991.

[Mey95] Manfred Meyer, editor. *Constraint Processing (CP95)*, LNCS 923. Springer-Verlag, 1995.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. In *Artificial Intelligence*, volume 28, pages 225–233, 1986.

[Mic94] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Artificial Intelligence. Springer-Verlag, 1994.

[MJPL92] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. In *Artificial Intelligence*, volume 58, pages 161–205, 1992.

[MN95] Z. Michalewicz and G. Nazhiyath. Genocop iii: A co-evolutionary algorithm for numerical optimization problems with nonlinear constraints. In *Second IEEE International Conference on Evolutionary Computation*, pages 647–651, 1995.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. In *Information Science*, volume 7, pages 95–132, 1974.

[Mor93] P. Morris. The breakout method for escaping from local optima. In *Proceedings of the AAAI*, pages 40–45, 1993.

[MS96] Z. Michalewicz and M. Schoenauer. *Handbook of Evolutionary Computation*, chapter Evolutionary Algorithms for Constrained Parameter Optimization Problems. 1996.

[MSS+94] Z. Michalewicz, J. Schaffer, H.-P. Schwefel, D. Fogel, and H. Kitano, editors. *First IEEE International Conference on Evolutionary Computation*. IEEE Press., 1994.

[Nad90] B. Nadel. Some applications of the constraint satisfaction problem. Csc-90-008. Computer Science Dept. Wayne State University. 1990.

[NV97] B. Naudts and A. Verschoren. Epistasis and deceptivity. In *Bull. Soc. Mathematiques Belges*. 1997.

[Par94] J. Paredis. Co-evolutionary constraint satisfaction. In *Third International Conference on Parallel Problem Solving Nature*, pages 46–55, 1994.

[Pea81] J. Pearl. Heuristic search theory: a survey of recent results. In *International Joint Conference on Artificial Intelligence*, pages 24–28, 1981.

[Pea90] J. Pearl. *Heuristique*. Addison-Wesley, 1990.

[Por97] B. Porto, editor. *Third IEEE International Conference on Evolutionary Computation*. IEEE Press., 1997.

[Pro91] P. Prosser. Hybrid algorithms for the constraint satisfaction problems. Aisl-46-91, University of Strathclyde, 1991.

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

[Rad90] N. Radcliffe. *Genetic Neural Networks on MIMD Computers*. PhD thesis, University of Edinburgh, England, 1990.

[Rad91] N. Radcliffe. Equivalence class analysis of genetic algorithms. In *Complex Systems*, volume 5, pages 183-205, 1991.

[Rad92] N. Radcliffe. The algebra of genetic algorithms. Epcc-tr-92-11, University of Edinburgh, 1992.

[Raw91] G.J. Rawlins. *Foundations of Genetic Algorithms*. Morgan Kauffmann, 1991.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologishen Evolution.* Frommann-Holzboog Verlag, 1973.

[Reg96] C. Rego. *Local Search and Neighborhood Structures for Vehicle Routing Problems: Sequential and Parallel Algorithms.* PhD thesis, Université de Versailles, France, 1996.

[Rif96a] M.-C. Riff. From quasi-solutions to solution: An evolutionary algorithm to solve csp. In Eugene Freuder, editor, *Constraint Processing (CP96)*, LNCS 1118, pages 367–381. Springer-Verlag, 1996.

[Rif96b] M.-C. Riff. Using the knowledge of the constraints network to design an evolutionary algorithm that solves csp. In Fogel [Fog96], pages 279–284.

[Rif97a] M.-C. Riff. Evolutionary search guided by the constraint network to solve csp. In Porto [Por97], pages 337–342.

[Rif97b] M.-C. Riff. A network-based adaptive evolutionary algorithm for csp. In *2nd Meta-Heuristic International Conference*, pages 23–24, 1997.

[Rif97c] M.-C. Riff. Self-adapting crossover operator for csp. In *3ème Conférence Nationale pour la Resolution Pratique de Problemes NP-Complets*, pages 41–48, 1997.

[RPD89] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problem. Act-ai-222-89, MCC Corporation, Austin, Texas, 1989.

[RW91] Y Rabinovich and A. Wigderson. An analysis of a simple genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 215–222. Morgan-Kaufmann, 1991.

[Sch81] H.P. Schwefel. *Numerical Optimization of Computer Models.* Whiley, 1981.

[Sel95] B. Selman. Stochastic search and phase de transitions: Ai meets physics. In *International Joint Conference on Artificial Intelligence*, pages 998–1002, 1995.

[Shi96] A.N. Shiryaev. *Probability*. Springer, 1996.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the AAAI*, pages 337–343, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satifiability problems. In *Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[Smi95] B. Smith. In search of exceptionally difficult constraint satisfaction problems. In Meyer [Mey95], pages 139–155.

[ST93] A. Smith and D. Tate. Genetic optimization using a penalty function. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 499–503, 1993.

[Sti68] M.M. Stickberger. *Genetics*. Collier Mc Millan, 1968.

[Sys89] G. Syswerda. Uniform crossover in genetic algorithmes. In J. Schaffer, editor, *Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kauffmann, 1989.

[Tai91] E. Taillard. Robust tabu search for the quadratic assignment problem. In *Parallel Computing*, volume 17, pages 443–455, 1991.

[Tai93] E. Taillard. Parallel iterative search methods for vehicle routing problems. In *Networks*, volume 23, pages 661–673, 1993.

[Tho94] A.C. Thorton. Genetic algorithms versus simulated annealing: Satisfaction of large sets of algebraic mechanical design constraints. In *Artificial Intelligence in Design*, pages 381–398, 1994.

[Tsa90] E. Tsang. Applying genetic algorithms to constraint satisfaction optimization problems. In *Nineth European Conference on Artificial Intelligence*, pages 649–654, 1990.

[VLA88] P. Van Laarhoven and E. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer, 1988.

[War95] T. Warwick. *A GA Approach to Constraint Satisfaction Problems*. PhD thesis, University of Essex, UK, 1995.

[WT94] T. Warwick and E. Tsang. Using a genetic algorithm to tackle the processors configuration problem. In *Proceedings of ACM Symposium on Applied Computing*, pages 217–221, 1994.