RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieure et de la Recherche Scientifique

Université Mira Abderrahmane de Bejaia
Faculté de sciences Exactes
Département d'informatique

ÉCOLE DOCTORALE RESAUX ET SYSTEMES DISTRIBUES

Mémoire de Magister

En Informatique

Option : Réseaux et Systèmes Distribués

Thème

Calcul des hypertrees décompositions pondérées

Présenté par :

ABBACHE Farid

Devant le jury composé de :

Président : Tari Abdelkamel, MCA à l'Université de Bejaïa, Algérie

Rapporteur : Dahmani Abdennassar, Professeur à Université de Bejaïa, Algérie

Examinateurs : Melit Ali , Professeur à l'Université de Jijel, Algérie

Examinateurs : Boukeram Abdellah, MCA à l'Université de Sétif, Algérie

Invité : Amroun Kamal, Chargé de cours, Université de Bejaïa, Algérie

Promotion 2009-2010

Tableau des matières

Liste des Tableaux

Liste des figures

Remerciements

Je loue Dieu de la grâce qu'il m'a faite en me donnant la santé, la patience et la détermination sans les quelles je n'aurais jamais pu porter mon projet à son terme.

Je voudrais tout d'abord exprimer mes remerciements et ma gratitude à Mr Dahmani pour m'avoir encadré durant ce projet, son aide et sa confiance m'ont grandement aidé à mener à bien mon travail, ainsi que Mr Amroun pour ses conseils et critiques.

Je remercie les membres de jury qui ont accepté de juger ce travail. J'adresse mes très sincères remerciements à Mr TARI, Mr Melit et Mr Boukeram de me faire l'honneur de s'intéresser à ce travail et d'avoir accepté de faire partie de jury.

Je remercie également Mr TARI Kamel, le chef de département d'informatique de l'université de Béjaia pour son aide, ses conseils et sa disponibilité. Un merci bien distingué à mes enseignants de l'école doctorale ReSyD. J'aimerai remercier également tous les étudiants de l'école doctorale pour l'environnement de travail très agréable durant ces deux dernières années.

Enfin, et surtout, je remercie vivement ma famille qui m'a beaucoup soutenu et encouragé, en particulier mes parents et surtout ma mère qui a été toujours derrière moi pour m'encourager et me soutenir. Merci encore.

Introduction générale

L'optimisation combinatoire occupe une place très importante en recherche opérationnelle, en mathématiques discrètes et en informatique. Son importance se justifie d'une part par la grande difficulté des problèmes d'optimisation [44] et d'autre part par de nombreuses applications pratiques pouvant être formulées sous la forme d'un problème d'optimisation combinatoire [45]. Bien que les problèmes d'optimisation combinatoire soient souvent faciles à définir, ils sont généralement difficiles à résoudre. En effet, la plupart de ces problèmes appartiennent à la classe des problèmes NP-difficiles et ne possèdent donc pas à ce jour de solution algorithmique efficace valable pour toutes les données [46]. Etant donnée l'importance de ces problèmes, de nombreuses méthodes de résolution ont été développées en recherche opérationnelle (RO) et en intelligence artificielle (IA). Ces méthodes peuvent être classées sommairement en deux grandes catégories : les méthodes exactes (complètes) qui garantissent la complétude de la résolution et les méthodes approchées (incomplètes) qui perdent la complétude pour gagner en efficacité.

Le principe essentiel d'une méthode exacte consiste généralement à énumérer, souvent de manière implicite, l'ensemble des solutions de l'espace de recherche. Pour améliorer l'énumération des solutions, une telle méthode dispose de techniques pour détecter le plus tôt possible les échecs (calculs de bornes) et d'heuristiques spécifiques pour orienter les différents choix. Parmi les méthodes exactes, on trouve la plupart des méthodes traditionnelles telles les techniques de séparation et évaluation progressive (SEP) ou les algorithmes avec retour arrière. Les méthodes exactes ont permis de trouver des solutions optimales pour des problèmes de taille raisonnable. Malgré les progrès réalisés (notamment en matière de la programmation linéaire en nombres entiers), comme le temps de calcul nécessaire pour trouver une solution risque d'augmenter exponentiellement avec la taille du problème, les méthodes exactes rencontrent généralement des difficultés face aux applications de taille importante.

Les méthodes approchées constituent une alternative très intéressante pour traiter les problèmes d'optimisation de grande taille si l'optimalité n'est pas primordiale. En effet, ces méthodes sont utilisées depuis longtemps par de nombreux praticiens, souvent appelées méta-heuristiques [47, 48]. Une méta-heuristique est constituée d’un ensemble de concepts fondamentaux (par exemple, la liste tabou et les mécanismes d’intensification et de diversification pour la méta-heuristique tabou), qui permettent d'aider à la conception de

méthodes heuristiques pour un problème d'optimisation. Ainsi les méta-heuristiques sont adaptables et applicables à une large classe de problèmes.

Parmi les formalismes adaptés aux modélisation des problèmes d'optimisation combinatoire en peut citer les *CSP(Constraint Satifation Problems)* ou problème de satisfaction des contraintes.

D'une façon informelle, un *CSP* est constitué de variables et de valeurs qui peuvent être affectées à ces variables. Ces variables sont liées par une ou plusieurs contraintes. Une solution d'un *CSP* est une affectation des valeurs aux variables qui ne viole aucune contrainte.

Nous intéressons dans notre travail à une nouvelle approche de résolution des *CSPs*, dite résolution basée sur la décomposition structurelle pondérée, elle consiste à décomposer le problème pour simplifier sa résolution, puis résoudre chaque sous problème en cumulant chaque sous problème nous obtenons la solution globale.

Ce mémoire est organisé en quatre chapitres :

- **Le chapitre 1** présente en première lieu les différentes notions liées au formalisme *CSP*. Ensuite, nous évoquons les principales approches de résolution séquentielle.
- **Le chapitre 2** présente les méthodes principales de décompositions structurelles des problèmes CSP, puis il expose leurs hiérarchies en termes de généralisation.
- **Le chapitre 3** présente les méthodes méta-heuristiques principales avec leurs principaux éléments qui doivent être maitrisé afin de bien utiliser chaque méthode dans le bon lieu.
- **Le Chapitre 4** présente notre nouvelle méthode à deux phases, la première phase est celle de la décomposition hyper-arborescente pondérée du problème qui base sur l'hybridation entre la méthode structurelle pondérée et la méthode méta-heuristique dite recherche taboue et la deuxième phase s'occupe de la résolution du problème.

Chapitre 1

Les CSPs et les méthodes de résolutions séquentielles

1.1 Introduction

La notion de contrainte est très naturellement présente dans notre vie quotidienne, qu'il s'agisse d'affecter des stages à des étudiants en respectant leurs souhaits(le raisonnement temporel [20]), de ranger des pièces de formes diverses dans une boîte rigide(le maintien de la cohérence [19]), de planifier le trafic aérien (l'ordonnancement [18]) pour que tous les avions puissent décoller et atterrir sans se percuter. La notion de "Problème de Satisfaction de Contraintes" (ou CSP en abrégé, pour *Constraint Satisfaction Problem*) désigne l'ensemble de ces problèmes, définis par des contraintes, et consistant à chercher une solution les respectant.

Dans ce chapitre, Nous rappelons d'abord les concepts essentiels que nous devons connaitre pour bien assimiler les CSP [21]. Ensuite, nous présentons les principaux algorithmes utilisés pour la résolution des problèmes représentés par ce formalisme, en limitant essentiellement nos rappels aux approches énumératives et structurelles.

1.2 Définitions

1.2.1 Contrainte

Une contrainte est une relation logique, cette relation peut être définie en extension ou en intension, (une propriété qui doit être vérifiée) entre différentes inconnues, appelées variables, chacune prenant ses valeurs dans un ensemble donné, appelé domaine. Ainsi, une contrainte restreint les valeurs que peuvent prendre simultanément les variables. Par exemple, la contrainte "x + 3*y = 12" restreint les valeurs que l'on peut affecter simultanément aux variables x et y.

1.2.1.1 Contrainte en extension

Pour définir une contrainte en extension, on énumère les tuples de valeurs appartenant à la relation.

Par exemple, si les domaines des variables x et y contiennent les valeurs 0, 1 et 2, alors on peut définir la contrainte "x est plus petit que y" en extension par "(x=0 et y=1) ou (x=0 et y=2) ou (x=1 et y=2)" , ou encore par "(x,y) élément-de {(0,1),(0,2),(1,2)}"

1.2.1.1 Contrainte en intention

Pour définir une contrainte en intention, on utilise des propriétés mathématiques connues.

1.2.1.1 Quelques caractéristiques de contrainte

- Une contrainte est relationnelle : elle n'est pas "dirigée" comme une fonction qui définit la valeur d'une variable en fonction des valeurs des autres variables.
- Notons également qu'une contrainte est déclarative : elle spécifie quelle relation on doit retrouver entre les variables, sans donner de procédure opérationnelle pour effectivement assurer/vérifier cette relation.

1.2.2 Arité d'une contrainte

L'arité d'une contrainte est le nombre de variables sur lesquelles elle porte. On dira que la contrainte est :

- **Unaire** : si son arité est égale à 1 (elle ne porte que sur une variable), par exemple "x*x = 4" .
- **Binaire** : si son arité est égale à 2 (elle met en relation 2 variables), par exemple "x ≠ y" ou encore "A U B = A"
- **Ternaire** : si son arité est égale à 3 (elle met en relation 3 variables), par exemple "x+y < 3*z-4" ou encore "(non x) ou y ou z = vrai"
- ...
- **n-aire** : si son arité est égale à n (elle met en relation un ensemble de n variables). On dira également dans ce cas que la contrainte est globale.

1.3 Différents types de contraintes

On distingue différents types de contraintes en fonction des domaines de valeurs des variables:

- **Les contraintes numériques** : portant sur des variables à valeurs numériques : une contrainte numérique est une égalité (=) , une différence (≠) ou une inégalité (<, ≤, >,≥) entre 2 expressions arithmétiques.

- **Les contraintes numériques linéaires** : quand les expressions arithmétiques sont linéaires, par exemple "4*x - 3*y + 8*z < 10".
- **Les contraintes numériques non linéaires** : quand les expressions arithmétiques contiennent des produits de variables, ou des fonctions logarithmiques, exponentielle…, par exemple "x*x = 2" ou "sinus(x) + z*log(y) = 4".
- **Les contraintes booléennes** : portant sur des variables à valeur booléenne (vrai ou faux) : une contrainte booléenne est une implication (=>), une équivalence (<=>) ou une non équivalence (<≠>) entre 2 expressions logiques. Par exemple "(non a) ou b => c" ou encore "non (a ou b) <=> (c et d)".

1.4 Définition d'un CSP

Un CSP (Problème de Satisfaction de Contraintes) est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine. De façon plus formelle, on définira un CSP par un triplet [1] (X,D,C) tel que

- $X = \{ X_1, X_2, ..., X_n\}$ est l'ensemble des variables (les inconnues) du problème ;
- D est la fonction qui associe à chaque variable X_i son domaine $D(X_i)$, c'est-à-dire l'ensemble des valeurs que peut prendre X_i ;
- $C = \{C_1, C_2, ..., C_k\}$ est l'ensemble des contraintes. Chaque contrainte C_j est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

Par exemple, on peut définir le CSP (X,D,C) suivant :

$$X = \{a,b,c,d\}$$

$$D(a) = D(b) = D(c) = D(d) = \{0,1\}$$

$$C = \{ a \neq b, c \neq d, a+c < b \}$$

Ce CSP comporte 4 variables *a, b, c* et *d*, chacune pouvant prendre 2 valeurs (0 ou 1). Ces variables doivent respecter les contraintes suivantes : *a* doit être différente de *b ; c* doit être différente de *d* et la somme de *a* et *c* doit être inférieure à b.

1.5 Réseau de contraintes

Un réseau de contraintes est un ensemble de contraintes sur des variables ayant chacune un domaine fini et discret.

Un réseau de contraintes binaires peut être représenté par un graphe spécial :

- Les sommets représentent les variables.
- Les arêtes représentent les contraintes.

1.6 Solution d'un CSP

Etant donné un CSP (X,D,C), sa résolution consiste à affecter des valeurs aux variables, de telle sorte que toutes les contraintes soient satisfaites. On introduit pour cela les notations et définitions suivantes :

- On appelle **affectation** le fait d'instancier certaines variables par des valeurs (évidemment prises dans les domaines des variables). On notera A = { (X_1,V_1), (X_2,V_2), ..., (X_r,V_r) } l'affectation qui instancie la variable X_1 par la valeur V_1, la variable X_2 par la valeur V_2, ..., et la variable X_r par la valeur V_r.

  Par exemple, sur le CSP précédent, A={(b,0),(c,1)} est l'affectation qui instancie b à 0 et c à 1.

- Une **affectation** est dite **totale** si elle instancie toutes les variables du problème ; elle est dite partielle si elle n'en instancie qu'une partie.

  Par exemple, A_1 = {(a,1),(b,0),(c,0),(d,0)} est une affectation totale ; A_2 = {(a,0),(b,0)} est une affectation partielle.

- Une **affectation** A **viole** une contrainte C_k si toutes les variables de C_k sont instanciées dans A, et si la relation définie par C_k n'est pas vérifiée pour les valeurs des variables de C_k définies dans A.

  L'affectation partielle $A2$ = {(a,0),(b,0)} viole la contrainte a≠b ; en revanche, elle ne viole pas les deux autres contraintes dans la mesure où certaines de leurs variables ne sont pas instanciées dans $A2$.

- Une **affectation** (totale ou partielle) est **consistante** si elle ne viole aucune contrainte, et inconsistante si elle viole une ou plusieurs contraintes.

L'affectation partielle {(c,0),(d,1)} est consistante, tandis que l'affectation partielle {(a,0),(b,0)} est inconsistante.

- Une **solution** est une **affectation totale consistante**, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.

  A = {(a,0),(b,1),(c,0),(d,1)} est une affectation totale consistante : c'est une solution.

1.7 CSP sur-contraint et sous-contraint

Lorsqu'un CSP n'a pas de solution, on dit qu'il est **sur-contraint** : il y a trop de contraintes et on ne peut pas toutes les satisfaire. Dans ce cas, on peut souhaiter trouver l'affectation totale qui viole le moins de contraintes possibles.

Un tel CSP est appelé **max-CSP** (max car on cherche à maximiser le nombre de contraintes satisfaites).

Une autre possibilité est d'affecter un poids à chaque contrainte (une valeur proportionnelle à l'importance de cette contrainte, et de chercher l'affectation totale qui minimise la somme des poids des contraintes violées. Un tel CSP est appelé **CSP valué (VCSP).**

Il existe encore d'autre types de CSPs, appelés CSPs basés sur les semi-anneaux **(Semiring based CSPs)** permettant de définir plus finement des préférences entre les contraintes.

Inversement, lorsqu'un CSP admet beaucoup de solutions différentes, on dit qu'il est **sous-contraint**. Si les différentes solutions ne sont pas toutes équivalentes, dans le sens où certaines sont mieux que d'autres, on peut exprimer des préférences entre les différentes solutions. Pour cela, on ajoute une fonction qui associe une valeur numérique à chaque solution, valeur dépendante de la qualité de cette solution. L'objectif est alors de trouver la solution du CSP qui maximise cette fonction. Un tel CSP est appelé **CSOP** (Constraint Satisfaction Optimisation Problem).

1.8 Modélisation en CSP

1.8.1 Le problème des n-reines

Description du problème

Il s'agit de placer *n* reines sur un échiquier comportant *n* lignes et *n* colonnes, de manière à ce qu'aucune reine ne soit en prise. On rappelle que 2 reines sont en prise si elles se trouvent sur une même diagonale, une même ligne ou une même colonne de l'échiquier.

Variables :

$X = \{X_i / i$ est un entier compris entre 1 et $n\}$

Domaines :

Quelque soit X_i l'élément de X, $D(X_i) = \{j / j$ est un entier compris entre 1 et $n\}$

Contraintes :

Les reines doivent être sur des lignes différentes :

- $Clig = \{X_i \neq X_j / i$ et j sont 2 entiers différents compris entre 1 et $n\}$

Les reines doivent être sur des diagonales montantes différentes

- $Cdm = \{X_{i+i} \neq X_{j+j} / i$ et j sont 2 entiers différents compris entre 1 et $n\}$

Les reines doivent être sur des diagonales descendantes différentes

- $Cdd = \{X_{i-i} \neq X_{j-j} / i$ et j sont 2 entiers différents compris entre 1 et $n\}$

L'ensemble des contraintes est défini par l'union de ces 3 ensembles

- $C = Clig\ U\ Cdm\ U\ Cdd$

1.8.2 Le problème des colorations du graphe

Description du problème

Il s'agit de colorier toutes les régions d'une carte, de telle sorte que deux régions ayant une frontière en commun soient coloriées avec deux couleurs différentes. La figure1.1 illustre une instance possible de 3-coloration avec 4 régions :

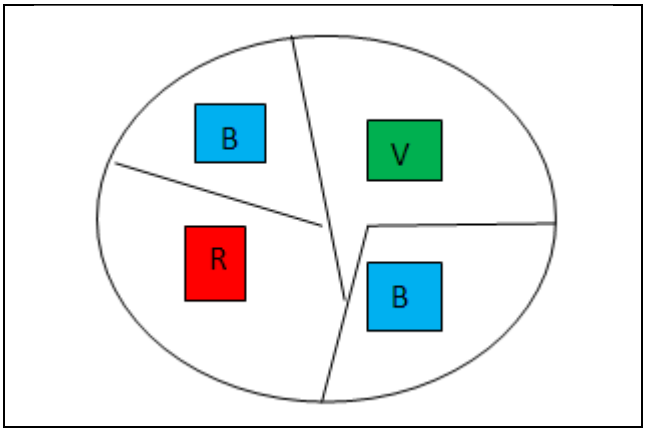


Figure1.1. Problème de coloration

Ici notre carte est divisée en 4 régions et l'on dispose des trois couleurs Vert, Rouge et Bleu pour le coloriage. Donc, on associe une variable X_i différente par région i à colorier, alors le modèle est défini comme suit :

- $P = (X,D,C)$
- $X = \{X_1,X_2,\ldots,X_4\}$ où X_i correspond au numéro de la région à colorier.
- $D = \{D_1,D_2,\ldots,D_4\}$ avec $D_i = \{\text{Vert,Bleu,Rouge}\}$
- $C = \{(X_i,X_j) \mid X_i$ à une frontière avec X_j ,et la couleur de X_i est différente de la couleur de $X_j\}$

1.9 Résolution de problèmes d'optimisation sous contraintes

"Résoudre un CSP" peut signifier autre chose que chercher simplement une solution. En particulier, il peut s'agir de chercher la "meilleure" solution selon un critère donné. Ces problèmes d'optimisation sous contraintes, où l'on cherche à optimiser une fonction objectif donnée tout en satisfaisant toutes les contraintes, peuvent être résolus en explorant l'ensemble des affectations possibles selon la stratégie de "Séparation & Evaluation " ("Branch & Bound") bien connue en recherche opérationnelle.

1.9.1 Notions de consistances

1.9.1.1 Consistance locale

On vérifie la consistance avec un sous-ensemble des contraintes (généralement une).

Un **CSP** est en **consistance locale** si toutes les valeurs dans les domaines des variables sont consistantes localement.

Quand un CSP atteint une consistance locale, l'ensemble de ses solutions ne change pas. Donc, le CSP reste équivalent.

1.9.1.2 Consistance de Nœud (1-consisance)

Un CSP est nœud consistant si pour chaque variable x ϵ X, et pour toute valeur v de D(x), l'affectation partielle {(x,v)} satisfait toutes les contraintes unaires de C. (consistance locale).

1.9.1.3 Arc-Consistance (2-consistance)

La valeur *a* de la variable x est arc-consistante ssi elle possède au moins une valeur compatible (un support) dans chaque domaine voisin.

Formellement :

‹x,a› est arc-consistante $\Rightarrow \forall C(x,y)\ \exists b \in Dy : C(a,b)$

Une contrainte est arc-consistante ssi toutes les valeurs dans les domaines de ces variables sont arc-consistantes.

Un CSP est arc-consistant ssi toutes ces contraintes sont arc-consistantes.

1.9.1.4 K-Consistance

Un CSP est K-consistant si chaque ensemble de valeurs de K -1 variables qui satisfait toutes les contraintes entre eux peut être « étendu » à la K-ème variable (il existe une valeur pour cette K-ème variable telle que toutes les contraintes entre ces K variables sont satisfaites).

Les algorithmes pour rendre un CSP K-consistant sont exponentiels en K. c.à.d **NP-complet**.

1.9.1.5 Problème P-complet et NP-complet

- Un algorithme est traitable (P-complet) si sa complexité [22] est polynomiale.
- Une classe de problème est dite traitable ssi il existe un algorithme polynomiale qui la résout, et intraitable (NP-complet) sinon.

1.10 Les algorithmes complets et incomplets pour la résolution de CSPs

Les algorithmes que nous allons étudier dans ce chapitre sont dits **complets**, dans le sens où l'on est certain de trouver une solution si le CSP est consistant. Cette propriété de complétude est fort intéressante dans la mesure où elle offre des garanties sur la qualité du résultat. En revanche, elle impose de parcourir exhaustivement l'ensemble des combinaisons possibles, et même si l'on utilise différentes techniques et heuristiques pour réduire la combinatoire, il existe certains problèmes pour lesquels ce genre d'algorithme ne termine pas "en un temps raisonnable". Par opposition, les algorithmes **incomplets** étudiées dans le chapitre trois ne cherchent pas à envisager toutes les combinaisons, mais cherchent à trouver le plus vite possible une affectation "acceptable" : ces algorithmes permettent de trouver rapidement de bonnes affectations (qui violent peu ou pas de contraintes) ; en revanche, on n'est pas certain que la meilleure affectation trouvée par ces algorithmes soit effectivement optimale ; on est par ailleurs certain que l'on ne pourra pas prouver l'optimalité de l'affectation trouvée. Il existe différents algorithmes incomplets pour résoudre de façon générique des CSPs sur les domaines finis, généralement basés sur des techniques de recherche locale (éventuellement combinées avec des méthodes tabou, du recuit simulé,...) où évolutionnaires (algorithmes génétique, des algorithmes à base de fourmis,…).

1.11 Algorithmes Complets

1.11.1 Algorithmes Enumératifs

1.11.1.1 génère et teste

Principe :

La façon la plus simple de résoudre un CSP sur les domaines finis consiste à énumérer toutes les affectations totales possibles jusqu'à on trouve une qui satisfasse toutes les contraintes. Ce principe est repris dans la fonction récursive « *genereEtTeste(A,(X,D,C)* » décrite ci-dessous. Dans cette fonction, *A* contient une affectation partielle et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La

fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon (ligne 12 et 13).

Algorithme 1 : genereEtTeste

fonction *genereETteste*(*A*,(*X*,*D*,*C*)) **retourne** un booléen

1 : entrée : (*X*,*D*,*C*) = un CSP sur les domaines finis, *A* = une affectation partielle pour (*X*,*D*,*C*)

2 : sortie : retourne vrai si l'affectation partielle A peut être étendue en une solution pour (*X*,*D*,*C*), faux sinon

3 : début

4 : **si** toutes les variables de X sont affectées à une valeur dans A **alors**

/ A est une affectation totale */*

5 : **si** A est consistante **alors**

/ A est une solution */*

6 : **retourner** vrai

7 : **sinon**

8 : **retourner** faux

9 : **fin si**

10 : **sinon** */* A est une affectation partielle */*

choisir une variable *Xi* de *X* qui n'est pas encore affectée à une valeur dans *A*

11 : **pour chaque** valeur *Vi* appartenant à *D*(*Xi*) **faire**

12 : **si** *genereETteste* (*A U* {(*Xi*,*Vi*)}, (*X*,*D*,*C*)) = vrai **alors**

13 : **retourner** vrai

14 : **fin pour**

15 : **retourner** faux

16 : **fin si**

fin fonction.

1.11.1.2 Simple retour arrière (Backtrack)

Principe :

Une première façon d'améliorer l'algorithme « génère et teste » consiste à tester au fur et à mesure de la construction de l'affectation partielle sa consistance : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à la compléter (ligne 4 à 7). Dans ce cas, on "retourne en arrière"[2][3] (« Backtrack » en anglais) jusqu'à la plus récente instanciation partielle consistante que l'on peut étendre en affectant une autre valeur à la dernière variable affectée .Ce principe est repris dans la fonction récursive « *backtrack* (A,(X,D,C)) » décrite ci-dessous. Dans cette fonction, *A* contient une affectation partielle et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon.

Algorihme 2 : Backtrack

fonction *Backtrack* (*A,(X,D,C)*) **retourne** un booléen

1 : entrée : *A* = affectation partielle,(*X,D,C*) = un CSP sur les domaines finis

2 : sortie : retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon

3 : début

4 : si *A* n'est pas consistante **alors retourne** faux ;

5 : si toutes les variables de *X* sont affectées à une valeur dans *A* **alors**

6 : */* A est une affectation totale et consistante = une solution */*

7 : retourner vrai **sinon** */* A est une affectation partielle consistante */*

9 : Choisir une variable *Xi* de *X* qui n'est pas encore affectée à une valeur dans *A*

10 : pour toute valeur *Vi* appartenant à *D(Xi)* **faire**

11 : si *Backtrack (A U {(Xi,Vi)}, (X,D,C))* = vrai **alors retourne** vrai

12 : fin pour

13 : fin si

14 : retourner faux

15 : fin si

fin fonction.

Complexité temporelle dans le pire des cas : $O(e.d^n)$. *e*:nombre de contrainte, *d* : taille de domaine, *n* : nombre de variable

Il existe plusieurs variantes de backtracting, en peut citer Back- Jumping[5], Conflict Directed Back Jumping[6], Nogood Recording[7], Learning Tree-Solve[8], Dynamic Backtracking[9].

1.11.2 Algorithmes de Filtrages

La complexité du Backtrack est exponentielle en la taille du problème, ce qui rend intéressant de réduire l'espace de recherche en utilisant des techniques de filtrage. Ce filtrage consiste soit à supprimer les valeurs des domaines où plus élaboré en filtrant sur valeur et contrainte.

1.11.2.1 Arc-consistance (AC-1)

Principe :

Son principe est basé sur la diminution du domaine des variables en supprimant chaque valeur qui y n'est pas arc-consistante (ligne 5 et 6).

Algorithme3 : AC-1

1 : répéter

2 : terminé VRAI;

3 : pour chaque contrainte C(x, y) **faire**

4 : si il existe dans *Dx* des valeurs qui n'ont pas de support dans *Dy* **alors**

5 : Supprimer les;

6 : terminé FAUX;

7 : fin si

8 : Fin Pour

9 : Jusqu'à terminé = VRAI ;

Complexité temporelle dans le pire des cas : $O(e.n.d^3)$.

1.11.2.2 Arc-consistance (AC-3)

Principe :

C'est une amélioration de AC-1 qui diminue la complexité en restreignant le filtre sur les valeurs à utiliser du domaine sur leur contrainte en respectant l'arc-consistance, il se base sur la propagation de contraintes, c'est-à-dire, il consiste à supprimer de chaque domaine D_i d'une variable X_i, les valeurs qui ne vérifient pas l'arc consistance, cette suppression peut être propagée vers le voisinage de X_i dans le réseau de contraintes, si la valeur supprimée est la seule valeur avec laquelle une valeur d'une autre variable vérifie la consistance d'arc, cette dernière sera a son tour supprimée (ligne 5 et 6). La propagation se termine s'il n' y a plus de suppression.

Algorithm 4 : AC-3

1 : àTester ← {C(x,y)} | C(x,y)ϵ C;

2 : **pour chaque** C(x,y) ϵ àTester **faire**

3 : àTester ← àTester \{C(x,y)};

4 : Supprimer les valeurs de D_x qui n'ont pas de support dans D_y ;

5 : **si** au moins une valeur a été supprimée **alors**

6 : àTester ← àTester ∪ {C(z,x) : ∃C(z,x) ϵ C; z ≠x};

Complexité temporelle dans le pire des cas : $O(e.d^2)$.

Complexité spatiale dans le pire des cas : $O(d.n + e)$.

Il existe plusieurs version de l'AC en peut citer :AC-4[11],AC-5[12],AC-6,AC-7[13].

1.11.2.3 Principe de l'algorithme d'anticipation (Forward Checking)

Le principe général de l'algorithme "anticipation" reprend celui de l'algorithme " retour-arrière", en ajoutant simplement une étape de filtrage anticipé c.-à-d. on évalue l'impact d'une instanciation avant même que la décision ne soit prise, puis il le supprime de l'ensemble du combinaison possible [15](ligne 10 à 15). Chaque fois qu'une valeur est affectée à une variable, on peut effectuer différents filtrages plus ou moins forts permettant d'établir différents niveaux de consistance locale (nœud, arc, chemin, ...).

Algorithme 5 : ForwardCheck

fonction *ForwardCheck* (*A,(X,D,C)*) **retourne** un booléen

/* cette algorithme applique anticipation /nœud ;

1 : entrée : *A* = affectation partielle consistante,*(X,D,C)* = un CSP sur les domaines finis

2 : sortie : retourne vrai si *A* peut être étendue en une solution pour*(X,D,C),* faux sinon

3 : début

4 : si toutes les variables de *X* sont affectées à une valeur dans A **alors**

5 : */* A est une affectation totale et consistante = une solution */*

6 : retourne vrai

7 : sinon */* A est une affectation partielle consistante */*

8 : Choisir une variable X_i de *X* qui n'est pas encore affectée à une valeur dans *A*

10 : pour chaque valeur V_i appartenant à $D(X_i)$ **faire**

11 : */* filtrage des domaines par rapport à A U {(X_i,V_i)} */*

12 : pour chaque variable X_j de *X* qui n'est pas encore affectée **faire**

13 : $Dfiltré(X_j) \leftarrow \{ V_j$ élément de $D(X_j)$ / *A* U $\{(X_i,V_i),(X_j,V_j)\}$ est consistante $\}$

14: **si** $Dfiltré(X_j)$ est vide **alors** retourner faux

15 : fin pour

16 : si *ForwardCheck* (A U $\{(X_i,V_i)\}$, (X,Dfiltré,C))=vrai **alors**

17 : retourne vrai

18 : fin pour

19 : retourner faux

20 : fin si

fin fonction.

Complexité temporelle dans le pire des cas : $O(e.d^n)$.

Il existe d'autre version de FC telle que nFC2, nFC3, nFC4, nFC5 [17].mais FC1reste la plus simple à implémenter et moins couteux en temps d'exécution.

1.11.3. Algorithmes d'assignations

En plus de ce que nous avons vu précédemment, Des heuristiques générales peuvent améliorer l'algorithme significativement en basant sur l'ordre :

- Choisir judicieusement l'ordre d'assignement des variables(DVFC).
- Choisir judicieusement l'ordre d'assignement la prochaine valeur à assigner (*least-constraining- value).*
- Détecter plutôt les assignations conflictuelles (algorithme *min-conflicts*).

Nous restreignant notre étude sur l'assignation sur ordre variable.

1.11.3.1 Assignation sur ordre de variables

1.11.3.1.1 L'algorithme d'anticipation à variable dynamique

(Dynamic variable forward checking DVFC)

Principe:

On a la solution partial $\overrightarrow{a_i} = (a_1, \dots, a_i)$,l'algorithme met à jours le domaine de chaque future variable pour garder que les valeur consistantes avec $\overrightarrow{a_i}$.puis la variable qui a le domaine le plus petit est sélectionnée.si la future variable contienne un domaine vide, elle est placée à la fin de l'ordre, et si cette variable soit la future variable sélectionné une notification d'erreur est signalé (aucune autre variable existe).

***Procédure** SELECT-VALUE-FORWARD-CHECKING*

1 : tant que $D_i^{'}$ est ┐ vide **faire**

2 : Sélectionner un élément arbitraire $a \in D_i^{'}$, et supprimer *a* de $D_i^{'}$

3 : pour chaque k, $i < k < n$ **faire**

4 : pour chaque valeur b dans $D_k^{'}$ **faire**

5 : si ┐ CONSISTANT$\langle \overrightarrow{a_{i-1}}, X_i = a, X_k = b \rangle$ **alors**

6 : Supprimer *b* de $D_k^{'}$

7 : fin pour

8 : si $D_k^{'}$est vide **alors** ($X_i = a$ ramène à l'inconsistant, ne sélectionner pas *a*)

9 : Rétablit chaque $D_k^{'}$, i < k < n à ça valeur avant que *a* est sélectionné.

10 : sinon

11 : retourne *a*

12 : fin pour

13 : fin tant que

14 : retourne ϕ (valeur non consistante)

fin procédure ;

Algorithme 6 : DVFC

procédure DVFC

1 : entrée : un réseau consistant $\mathcal{R}$=(X,D,C).

2 : sortie : soit une solution, soit une notification qui indique l'inconsistance de $\mathcal{R}$.

3 : $D_i' \leftarrow D_i$ pour $1 \leq i \leq n$ (copie tous les domaines)

4 : $i \leftarrow 1$ *(initialiser le compteur des variables)*

5 : $s \leftarrow \min_{i \leq j \leq n} |D_j'|$ *(trouver le future variable qui a le plus petit domaine)*

6 : $X_{i+1} \leftarrow X_s$ (réarrange les variable telle que X_s suit X_i)

7 : tant que $1 \leq i \leq n$ **faire**

8 : X_i Instantié ← SELECT-VALUE-FORWARD-CHECKING (procédure au-dessus)

9 : si $X_i = \phi$ **alors** (aucune valeur soit retourner)

10 : Rétablit chaque D' à sa valeur avant que X_i est instancié

11 : $i \leftarrow i - 1$

12 : sinon

13 : si $i < n$ **alors**

14 : $i \leftarrow i + 1$

15 : $s \leftarrow \min_{i \leq j \leq n} |D_j'|$ *(trouver le future variable qui a le plus petit domaine)*

16 : $X_{i+1} \leftarrow X_s$ (réarrange les variables telles que X_s suit X_i)

17 : $i \leftarrow i + 1$

18 : **fin si**

19 : **fin si**

20 : **fin tant que**

21 : **si** $i = 0$ **alors retourne** « inconsistant »

22 : **sinon retourne** valeur instancier de $\{ X_i, \ldots, X_n \}$

fin procédure.

Complexité temporelle de la procédure :$O(e.d^2)$;

Complexité temporelle du DVFC :$O(e.d^3)$;

1.12. Conclusion

Dans ce chapitre, nous avons présentés quelque notions importantes dans le domaine des CSPs ainsi que les algorithmes des résolutions de base utilisés pour résoudre ces CSPs, dites algorithmes complets à base énumératives et à base de filtre .Malgré leurs complétudes elles restent déconseillées avec les CSP de grande taille, aussi il se trouve que la structure des CSPs peuvent contribuer à des bonnes résultats concernant la résolution des ces CSPs, alors il fallait une autre approche basé sur cette structure. L'approche qui prend en considération la structure du problème s'appelle méthode de décomposition structurelle.

Dans le deuxième chapitre nous allons introduire les principaux méthodes de décompositions structurelles existe dans la littérature en présentant leurs principes.

Chapitre 2

Les Méthodes de décompositions structurelles

Nous avons présenté dans le chapitre précédent des approches à base énumératives est à base filtre et nous avons conclue que avec des CSP à grandes échelles ces méthodes ne fonctionnent plus, alors pour faire face à ce problème nos avons appliquée la décomposition du problème en sous problèmes en respectant la structure du problème initiale, dans le but d'avoir un temps d'exécution acceptable.

La modélisation de cette structure est appliquée en utilisant les hypergraphes (généralisation des graphes) où les sommets représentent les variables du problème est les hyper-arêtes représentent ces contraintes.

2.1 Introduction

Dans ce chapitre nous allons étudier les principaux méthodes de décomposition structurelle qui existe dans la littérature puis en va voire la relation entre une décomposition structurelle et celle de requête conjonctives, puis quelle est la modification qu'on va apporter aux décompositions hyper-arborescentes pour qu'elle support efficacement l'environnement des bases de données qui est riche en informations quantitatives qu'il fallait les exploiter afin d'avoir une méthode hybride structuro-quantitative qui exploite la puissance des méthodes structurelles et soit adaptable dans les requête conjonctives.

2.2 Définitions

Dans cette section en va définit les concepts essentiels pour bien avaler les méthodes structurelles.

2.2.1 Arête (arc)

Une arête se décrit par deux nœud connectés, un nœud de départ un autre d'arrivé (source et amont).

2.2.2 Graphe

Un graphe G est constitué par un ensemble finis des nœud $V=\{v_1,\ldots,v_n\}$ et un ensemble des arêtes $E=\{e_1,\ldots e_k\}$ telle que chaque nœud de V doit être couvrir par au moins une arête de E.

2.2.3 Chaine

Soit G(V,E) un graphe :

Une chaine joignant deux sommets x_0 et x_k dans un graphe G est une suite de sommets reliés par des arêtes tels que, deux sommets successifs ont une arête commune.

On la note : $(x_0,\ldots,x_n)$.

2.2.4 Cycle

Un cycle est une chaine simple dont les deux extrémités coïncident (x_0 coïncide avec x_k). on le note : $(x_0,x_1,x_2,\ldots,x_k=x_0)$.

2.2.5 Arbre (Tree)

Un arbre T de graphe G(V,E) est le paire $\langle T, x\rangle$,tels que $T = (V(T), E(T))$ Où $V(T)$ est l'ensemble des nœud de T , $E(T)$ est l'ensemble des arêtes de T et x est une fonction qui associe à chaque nœud $p \in V(T)$ l'ensemble de variables $x(p) \subseteq V$.

2.2.6 Hypergraphe

On définit un hypergraphe $\mathcal{H}$ par le paire (*V*,*H*),où *V* est l'ensemble des sommets et *H* est l'ensemble des hyper-arêtes tels que pour chaque h $\in$ *H*, h$\subseteq$ *V*.(fig2.1).

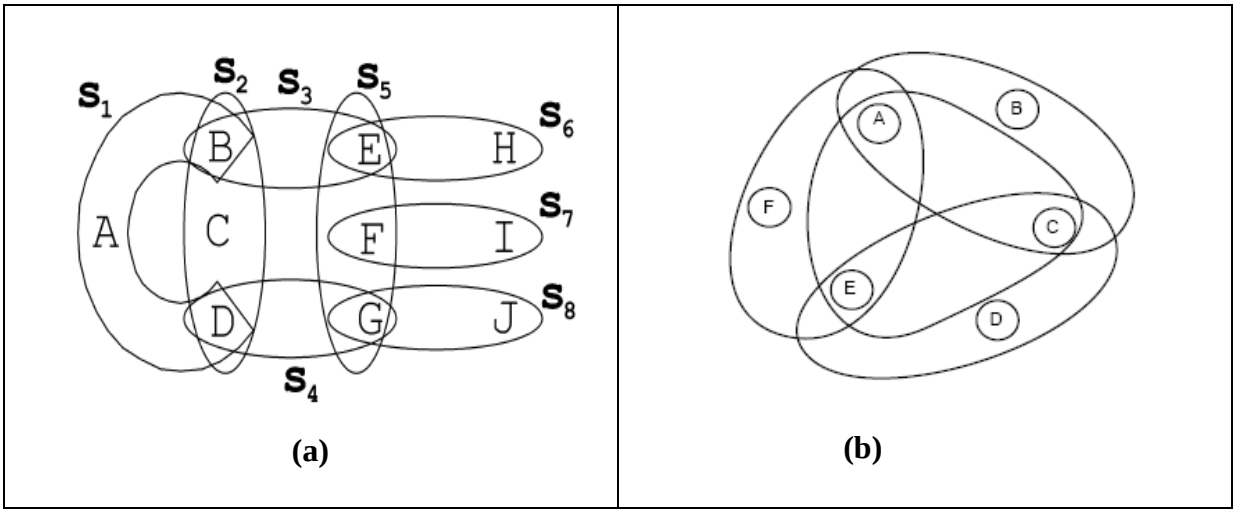


Figure 2.1 (a) [35].et (b) [1] Des hypergraphes où les S_i représentent les hyper arêtes

2.2.7 Graphe primal (Gaifgman)

Le graphe primal est la présentation transformé d'un hypergraphe $\mathcal{H}$ (*V*, *H*) tels que :

- Tous les nœuds de l'hypergraphe doit être contenu dans le graphe primal.
- Chaque nœud de graphe primal contient une variable de V, et une arête entre deux variables.

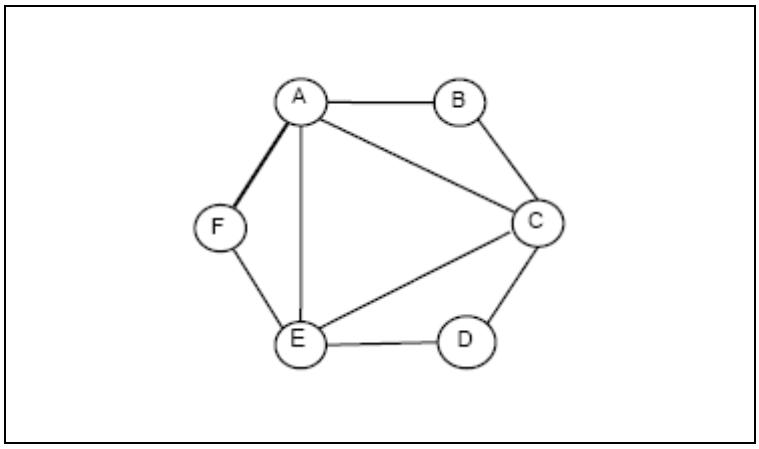


Figure 2.2 un graphe primal de l'hypergraphe (b)[1]

2.2.8 Gaphe Dual

Le graphe dual d'un hypergraphe $\mathcal{H}$ (V,H) est un graphe dont les nœuds sont les hyper-arêtes appartient à H est les arêtes sont étiquetées par au moins un variable commun entre ces hyper-arêtes.

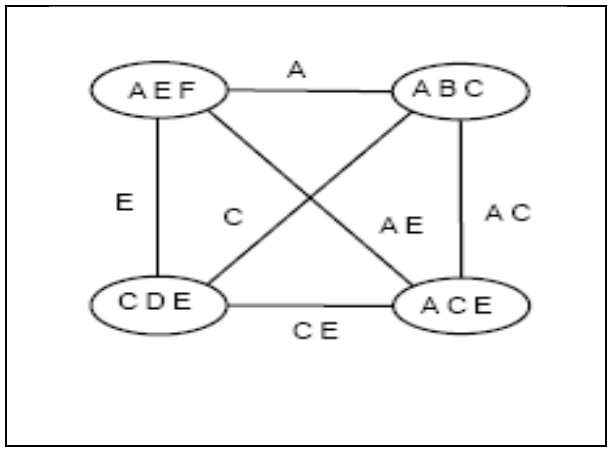


Figure 2.3 un graphe dual d'l'hypergraphe (b) [1]

2.2.9 Arbre de jointure (joint tree)

Une arbre de jointure *JT(ℋ)* d'un hypergraphe $\mathcal{H}$ (*V*,*H*) est une arbre où les sommets sont les arête de $\mathcal{H}$ tels que ,si une variable X ϵ *V* apparait dans deux arête A1 et A2 de *H*, alors X apparait dans chaque nœud avec un chemin unique liant A1 et A2 dans *JT(ℋ)*(condition de connectivité).

2.2.10. Hypergraphe acyclique

Un hypergraphe est dit acyclique [14, 23, 24,25] si seulement si s'il possède un arbre jointé.

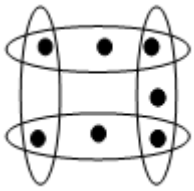
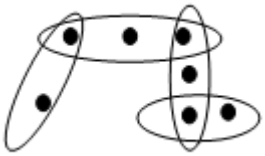
		
1. cyclique	2. cyclique	3. acyclique

Figure 2.4. Hypergraphes cycliques et acycliques [25]

2.3 Les méthodes de décompositions structurelles

2.3.1 Décomposition arborescente (tree decomposition)

Soit le graphe G=(V,E),une Décomposition arborescente[26,27] de G est le paire $(T, x), où\ T = (I, F)$ est l'arbre avec l'ensemble de nœuds I et l'ensemble des arêtes F,et $x = \{x_i : i \in I\}$ est la famille de sous ensembles de V pour chaque nœud de T,tels que :

1. $\bigcup_{i\in I} x_i = V$,
2. Pour chaque arête $(v, w) \in E$, il y a un $i \in I\ avec\ v \in x_i\ et\ w \in\ x_i$,
3. Pour tout $i, j, k \in I$ si j dans le chemin de i à k dans T, alors $x_i \cap x_k \subseteq x_j$.

La largeur (*width*) de la décomposition arborescente est $\max_{i\in I}|x_i| - 1$.

La largeur de l'arborescence **(**treewidth***)*** *d*e graphe G, noté par *tw(G)*, est la largeur minimum parmi toutes les décompositions arborescentes possibles de G.

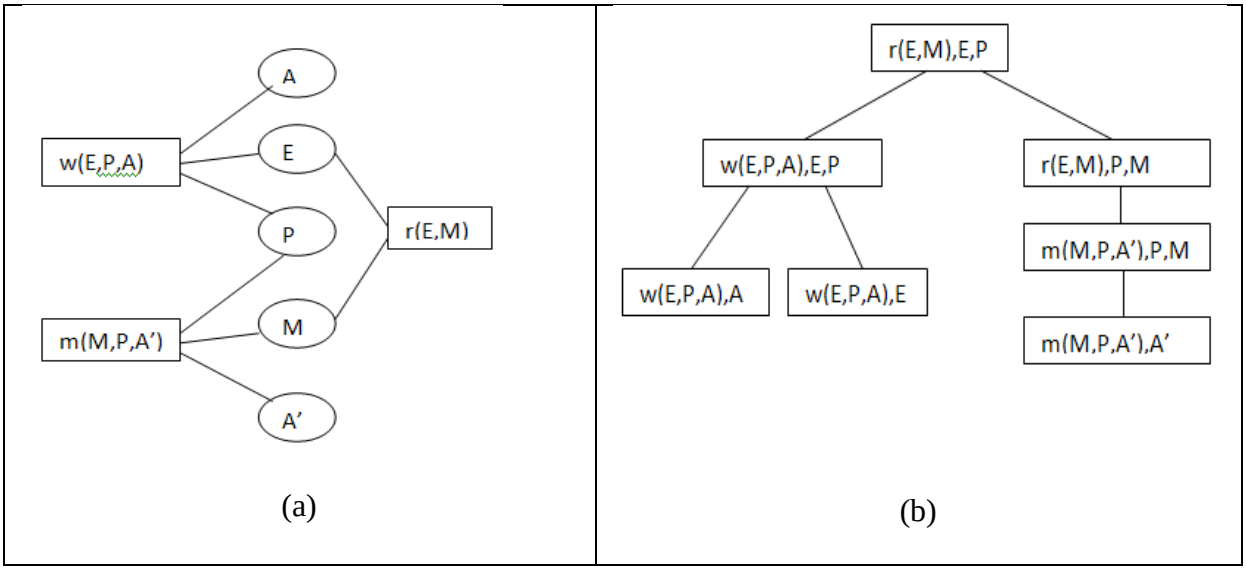


Figure 2.5 (a) Un graphe (b) sa décomposition arborescente de largeur deux [25]

Notes

- Le but de la décomposition est de transférer un CSP cyclique en un autre CSP équivalent acyclique.
- L'acyclicité permet d'assurer la résolution de n'importe quel problème NP-complet en un temps polynomiale [25].

2.3.2 Composants biconnéctés (biconnected components)

Soit un Graphe G= (V, E), connecté, est unidirectionnel. Un sommet a est nommé un sommet d'articulation ou sommet de partition si la suppression de a dans G (avec ces arcs incidents) le transforme en un graphe déconnecté (crie deux composant connectés ou plus). D'une autre manière, il existe deux sommets v et w different de a tels que pour chaque chemin allons de v à w passe par a.

Un graphe G est dit biconnecté si pour tout triple différent v, w, a il existe un chemin entre v et w ne contient pas a. Ou un graphe est biconnecté si seulement si s'il ne contient pas des sommets d'articulation.

Dans le cas des CSPs n-aires on utilise ce théorème :

Théorème *:*

Pour $1 \leq i \leq k, G_i = (V_i, E_i)$ soit composants biconnectés (biconnected component) du graphe unidirectionnel $G = (V, E)$. Il faut que :

1. $G_i$est un composant biconnecté (biconnected component).
2. Pour tout $i \neq j, V_i \cap V_j$ contient au plus un sommet.
3. a est un sommet d'articulation ssi $a \in V_i \cap V_j$ pour $i \neq j$.

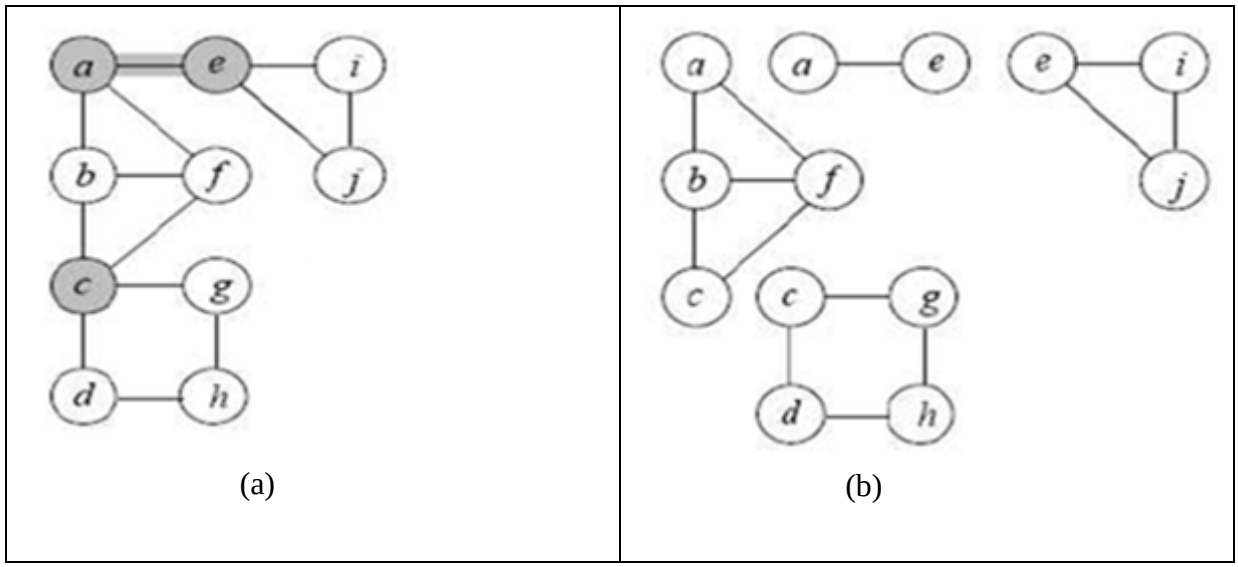


Figure 2.6 (a) un Graphe (b) ses composants biconnectés

2.3.3 Cycle cutset

Principe :

Le principe de la méthode de décomposition cycle-cutset [1] est basé sur l'identification de l'ensemble des nœuds tels que leurs suppressions rend le graphe sans cycle.

Cette méthode cherche un point x' tel que son instanciation change effectivement la connectivité de graphe de contraintes.

Exemple :

Considérant le CSP définit sous le graphe de la Figure 2.7(a). Pour ce problème, l'instanciation de x_2 avec une valeur a rend le choix des valeurs de x_1 et x_5 indépendant, comme si le chemin (x_1, x_2, x_5) est bloqué à x_2. D'une manière similaire, cette instanciation bloque la dépendance dans le chemin (x_1, x_2, x_4), on quitte seulement un seule chemin entre n'importe quels deux variables .D'une autre manière, si on a la variable x_2 assigné par une valeur spécifique, le graphe à contrainte résultant pour le reste des variables est démontré dans la Figure 2.7(b). Ici, x_2 instancié et ces arcs incidents sont les premier à supprimer du graphe, et par conséquent x_2 est dupliquée avec chaqu'un de ces voisins. On peut lire les deux graphes comme suit :

Le CSP qui à le graphe sur la Figure 2.7(a) si on $x_2 = a$ alors elle est identique au CSP qui à le graphe sur la Figure 2.7(b) ($x_2 = a$).

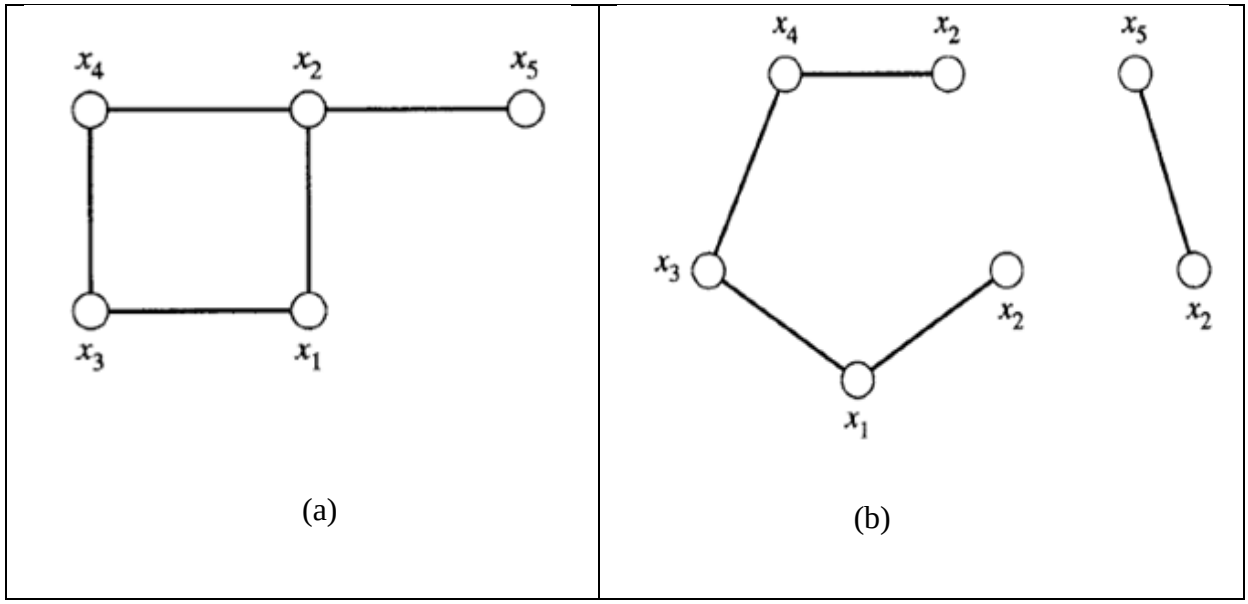


Figure 2.7 (a) [1] un graphe avec cycle

(b) [1] ses composants cutset associés en utilisant le coupage par rapport à x_2

2.3.4 Cycle hyper cutset

Principe :

Cette méthode [30] est basée sur le même principe que la méthode cycle cutset mais elle est généralisée sur les hypergraphes.

Exemple :

Considérant l'hypergraphe (a) de la Figure 2.8 avec six hyper-arêtes ternaire, $e_1, \dots, e_6$, si on supprime les hyper-arêtes e_1 et e_3 on obtient un hyper cutset acyclique (b).

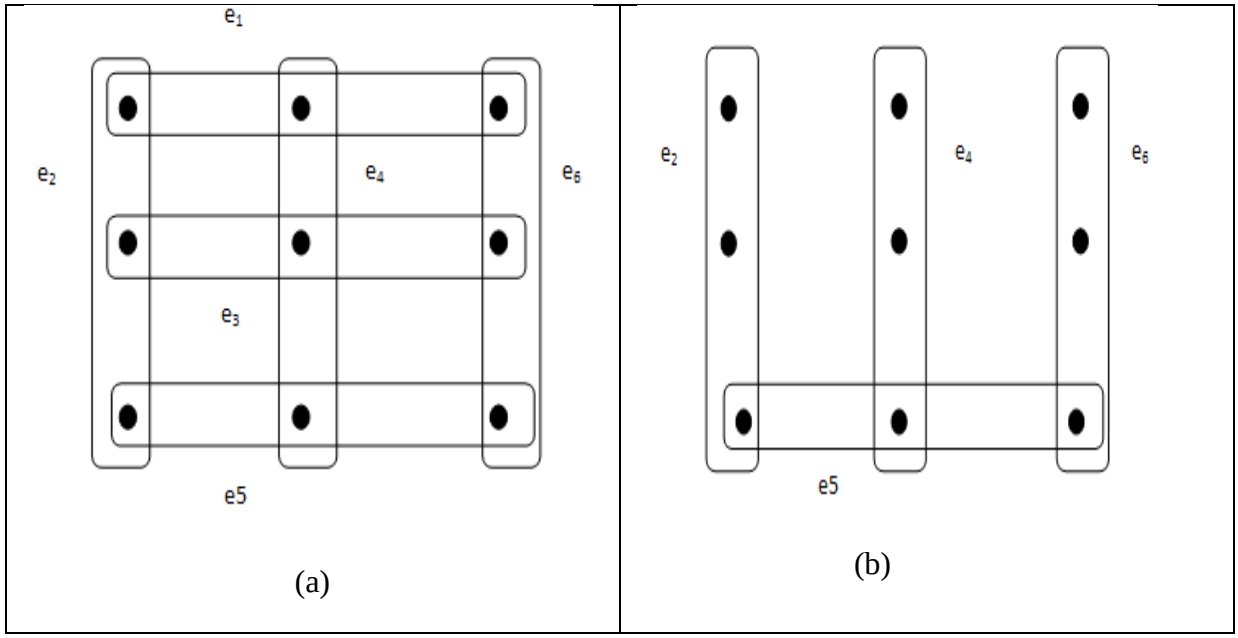


Figure 2.8 (a)un hypergraphe (b)l'hyper cutset acyclique [51]

2.3.5 Arborescence à base de cluster (Tree base cluster)

Principe :

Son principe est basé sur le graphe primale et un ordre des variables calculé par une méthode prédéfinis (max cardinalité) qui est utilisé pour générer une arbre jointure pour grouper des sous ensembles des contraintes(scopes) dans des clusters, leurs scopes constituent l'hyper- arborescence, puis on transforme l'hypergraphe avec contrainte à une hyper-arborescence avec contrainte. En remplaçant chaque sous problème avec leur ensemble de solutions [1]. Ce traitement est appelé l'arbre de jointure en cluster (joint tree clustring).

Exemple :

Considérant le graphe dans la figure2.9(a) et en suppose que c'été le graphe primale. Nous considérons l'ordre $d_1 = (F, E, D, C, B, A)$ dans la figure2.9 (b). L'exécution de l'arborescence en cluster (tree-clustring) connecte les parents récursivement du dernier variable au premier, en créant l'ordre induit de graphe en ajoutant les arêtes si nécessaire (dans notre figure ces arêtes sont représentées par des arcs pointés) figure 2.9 (b). Les cliques induisent de ce graphe sont Q_1={A,B,C,E},Q_2={B,C,D,E},*et* Q_3={D,E,F}.

Alternativement si l'ordre $d_2 = (A, B, C, D, E, F)$ dans la figure2.9(c) est utilisé, le graphe induit à une seule arête. Les cliques dans ce cas sont Q_1={D,F},Q_2={A,B,E},Q_3={B,C,D},et Q_4={A,B,C}. L'arbre de jointure correspondant dans les deux ordres sont dessinés dans la figure2.10(les arcs pointés n'appartient pas au l'arbre de jointure).Puis prenant l'arbre de jointure dans la figure2.10 (b). Nous partitionnons les contraintes sur les nœuds de l'arbre .Nous plaçons les sous problèmes dans les nœuds comme suit: P_1={R_{FD}} est placé dans le nœud (FD),P_2={R_{BD},R_{CD}} est placé dans le nœud (*BCD*),P_3={R_{AB},R_{AC}} est placé dans le nœud (ABC),et P_4={R_{AB},R_{BE},R_{AE}} est placé dans le nœud (*ABE*).Alors la résolution du problème revient à résoudre les sous problèmes P_1,P_2,P_3,P_4, en remplaçant chacun par R_1,R_2,R_3,R_4,où R_i est la solution de P_i ,nous obtenons notre CSP acyclique.

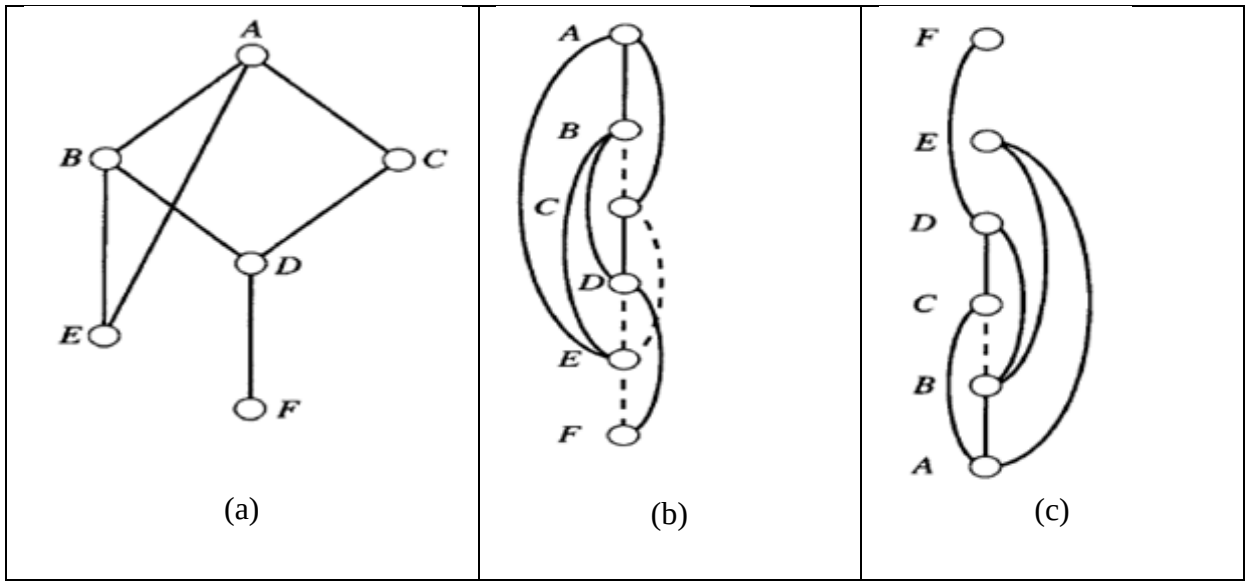


Figure 2.9 (a) un graphe (b) graphe induit par l'ordre d_1 (c) graphe induit par l'ordre d_2 [1]

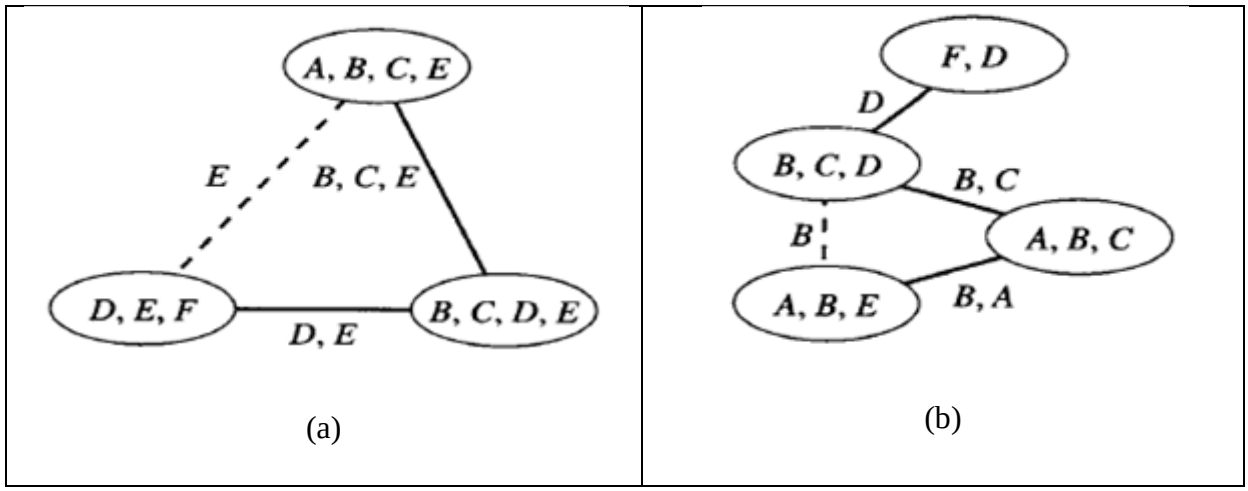


Figure 2.10 (a) l'arbre de jointure de graphe induit à partir de la Figure 2.9 (b).

(b) l'arbre jointure de graphe induit à partir de la Figure 2.9 (c).

(Les arcs pointés n'appartient pas à l'arbre de jointure [1])

2.3.6 Décomposition en hinge (Hinge decomposition)

Définition:

Soit $\mathcal{H} = \langle V, E \rangle$ un hypergraphe et $H \subseteq E$ un sous ensemble d'hyper-arête .un sous ensemble d'hyper-arête $F \subseteq E \backslash H$ est **connecté** si pour chaque deux hyper-arête $e, f \in F$ il existe une séquence $e_1, \dots, e_n$ de l'hyper-arête dans F tels que :

1- $e_1 = e$

2- Pour $i = 1, \dots, n-1,$ on a $e_i \cap e_{i+1} \nsubseteq \bigcup H$

3- $e_n = f$

On définit le **composant connectés** de $E \backslash H$ par le sous ensemble connecté maximale de $E \backslash H$.

Un hinge, H , pour l'hypergraphe $\mathcal{H}$,est un ensemble de deux hyper-arête au minimum tels que pour chaque composant connecté de $E \backslash H$ intersectionne avec H au plus sur un seul hyper-arête dans H.

Un hinge peut contient d'autres hinges plus petits, le hinge minimal ne contient pas d'autre hinges.

La décomposition en hinge (*hinge tree)* permet de décomposer un hypergraphe en une collection de hinge dont leurs interaction est acyclique en respectant les propriétés [31] suivante :

1- chaque nœud est un hinge minimal et chaque arc est étiqueté avec un seul hyper-arête de E.
2- tout hyper-arête dans E existe dans au moins un nœud.
3- pour un arc étiqueté, $\langle \{n_i, n_j\}, e \rangle$, l'étiquète e est précisément l'hyper-arête partagé par n_i et n_j.Ce qui y est l'ensemble des sommets dont l'hyper-arête e est précisément l'intersection des ensembles de sommets de l'hyper-arêtes de n_i et n_j.
4- les sommets partagés par deux nœuds sont contenu dans tout nœud existe dans leurs chemin de liaison.

La largeur d'une décomposition en hinge T est la taille du nœud le plus large de T de la décomposition en hinge minimale.est égal à **quatre** dans Figure 2.11

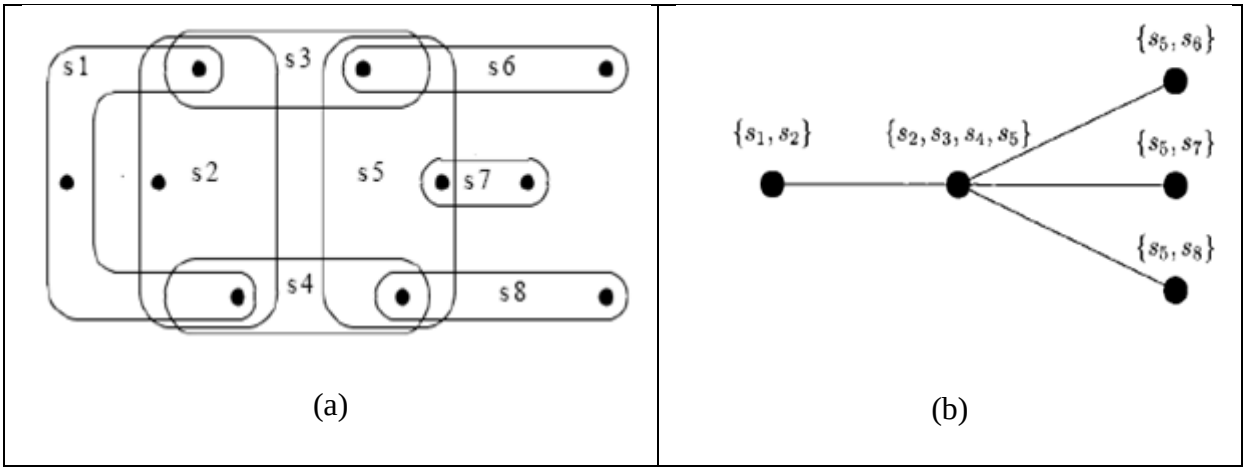


Figure 2.11 (a) un hypergraphe (b) sa décomposition en hinge [25]

2.3.7 Décomposition hyper-arborescente (hypertree decomposition)

Une décomposition hyper-arborescente HD [28] de l'hypergraphe $\mathcal{H}(V,H)$ est le triplet $\langle T, x, \lambda \rangle$ qui vérifier les conditions suivantes :

1. pour chaque arête $h \in H$,il existe $p \in sommets(T)$ tels que $h \in x(p)$ $(p\ couvre\ h)$ où $x(p)$ est la fonction qui associe un ensemble des variables de $\mathcal{H}$ au nœud p.
2. pour chaque variable $Y \in V$,l'ensemble $\{p \in sommets(T) \mid Y \in x(p)\}$ produit une sous arbre (connectée) de T.
3. pour chaque $p \in sommets(T), x(p) \subseteq var(\lambda(p))$;où *var* est l'ensemble des variables dans λ(p).et λ(p) est une fonction qui associe à chaque nœud p un ensemble d'hyper arêtes de $\mathcal{H}$.
4. pour chaque $p \in sommets(T), var(\lambda(p)) \cap x(T_p) \subseteq x(p)$. où $T_p$est le sous arbre de T routé à p.

   Où sommets() est une fonction qui retourne les sommets de l'arbre T

La largeur de la décomposition hyper-arborescente $\langle T, x, \lambda \rangle$ est $\max_{p \in sommets(T)} |\lambda(p)|$.

Largeur de la décomposition hyper-arborescente (hypertree width) noté $hw(\mathcal{H})$ de $\mathcal{H}$ est la largeur minimum parmi toutes leurs décompositions hyper-arborescentes.

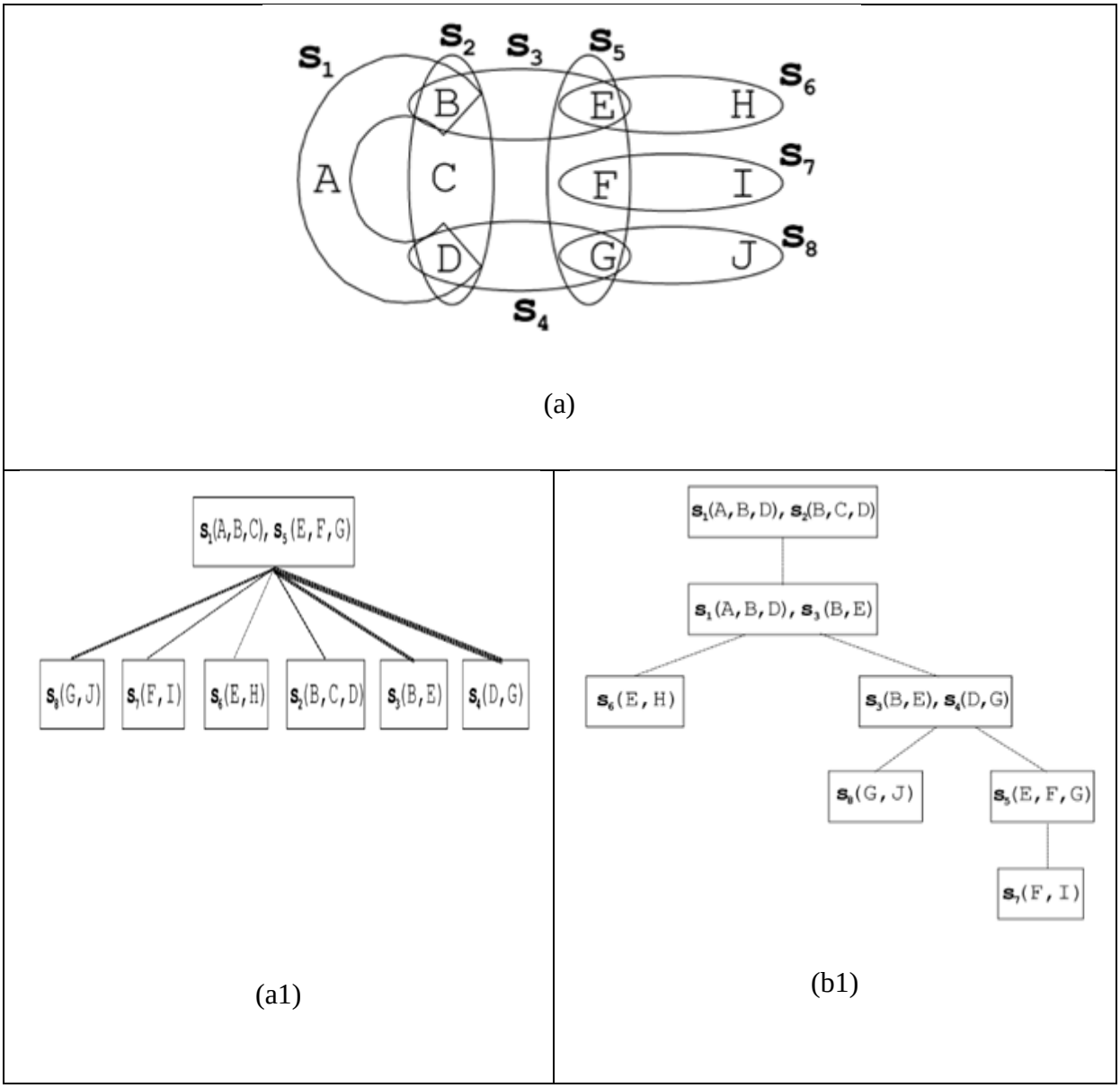


Figure 2.12 (a1) et (b1) deux décompositions hyper-arborescentes de l'hypergraphe (a)[35]

2.3.8 Décomposition arborescente à base retour arrière (BTD_{TD})

BTD [49] peut être considérer comme une approche efficace qui exploite une décomposition arborescente pour résoudre un problème CSP donnée. Cette méthode applique un algorithme énumératif guidé par un ordre de variable induit par la décomposition arborescente. Notant que cette décomposition est optimale (w^*).

Sa complexité temporelle est m^{w+1} où m est taille max de domaine, w est la largeur de l'arborescence ainsi que sa complexité spatiale est m^S Où S est la taille de CSP.

2.3.9 Décomposition hyper-arborescente à base retour arrière (BTD-2009$_{HD}$)

BTD-2009 est une extension de BTD en appliquant le concept de l'hyper-arborescence à l'algorithme de retour arrière [50], notant que cette décomposition est optimale (h^*).

Sa complexité temporelle est min (m^{w+1}, r^h) où r est la taille de la relation maximale est h est la largeur de l'hyper-arborescence et la complexité spatiale est m^S.

2.3.10 Décomposition arborescente à base cluster (TC$_{TD}$)

Cette méthode exploite une décomposition arborescente [50] pour résoudre chaque sous problème (cluster) où chaque cluster est constitué d'ensemble de variable de CSP. sa complexité temporelle est m^{w+1} ainsi que sa complexité spatiale est m^{w+1}.

2.3.11 Décomposition hyper- arborescente à base cluster (TC-2009$_{HD}$)

Cette méthode est une extension de la méthode TC$_{TD}$ où la résolution des cluster est basé sur les hyper-arborescences [50] où chaque cluster est constitué d'ensembles de contraintes.sa complexité temporelle est min (m^{w+1}, r^h) et la complexité spatiale est m^{w+1}.

2.4 L'hiérarchie des traitabilités des contraintes

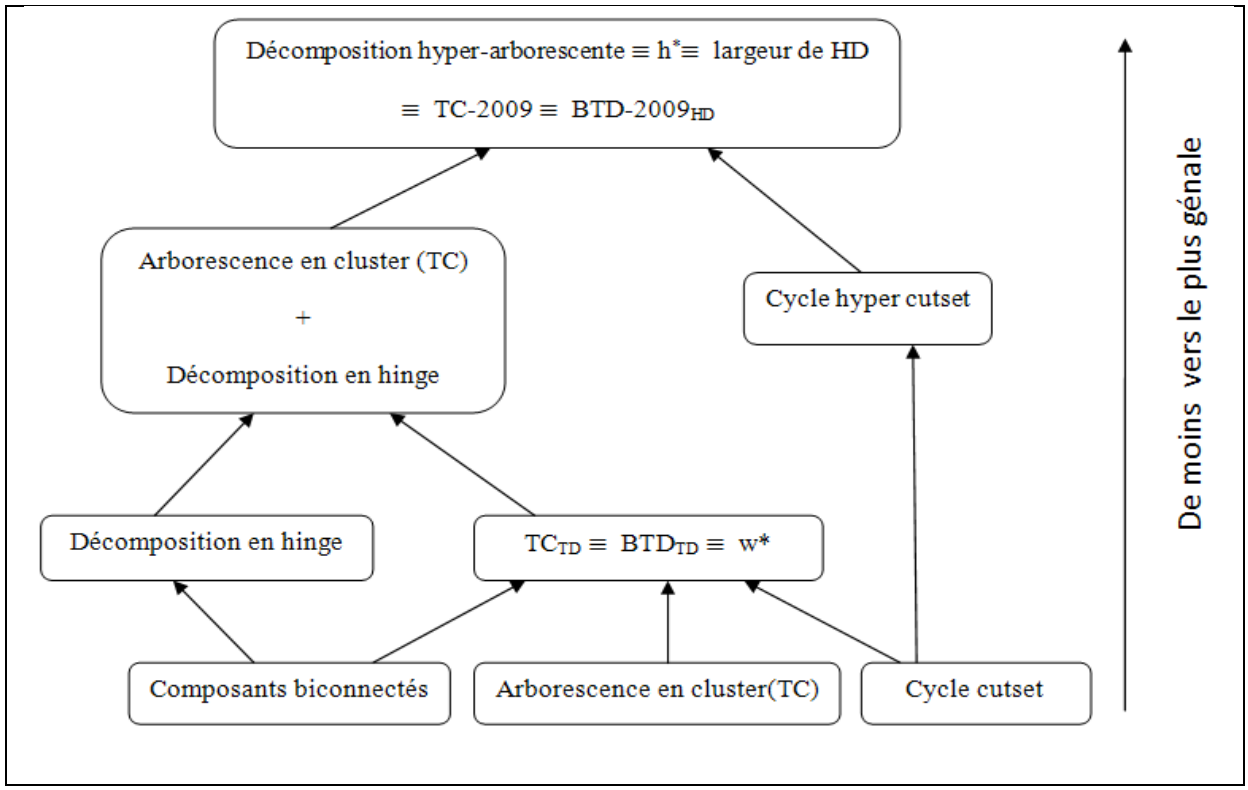


Figure 2.13 L'hiérarchie des traitabilités de contraintes de [50]

2.5 Décomposition hyper-arborescente pondérée (weighted hypertree decomposition)

2.5.1 Proposition

Résoudre un problème des requêtes conjonctives revient à résoudre un problème de satisfaction de contrainte [32, 33,34].

Nous avons vue des méthodes purement structurelle qui profite de la structure du problème afin de résoudre les problèmes posés, ce pondant avec des problèmes structurés riches d'information comme les requêtes conjonctives, ces méthodes ne rependent pas .Par exemple l'évaluation des requêtes de base de données dans un contexte pratique où nous avons plus d'information à l'intérieur de la structure des requêtes tels que le nombre de tuples ,la sélectivité des attributs ,…. Afin de prendre ces informations en considération une approche de combinaison entre les informations quantitatives et structurelle est introduite. Cette méthode est appelée « méthode de décomposition hyper-arborescente pondérée (weighted hypertree decomposition) ».

2.5.2 Définitions

Soit $\mathcal{H}$ un hypergraphe, $V \subseteq var(\mathcal{H})$ l'ensemble de variable de $\mathcal{H}$ et $Y \in var(\mathcal{H})$.

$[V]$-adjacent

- X est $[V]$-adjacent à Y s'il existe une arête $h \in aretes(\mathcal{H})$ tels que $\{X, Y\} \subseteq (h - V)$.

$[V]$-path

- $[V]$-path de X à Y est la séquence $X = X_0, \dots, X_\ell = Y$ de variables tels que : X_i est $[V]$-adjacent à X_{i+1},pour chaque $i \in [0 \dots \ell - 1]$.

[V]-connected

- un ensemble $W \subseteq var(\mathcal{H})$ de variables est $[V]$-connected si $\forall X, Y \in W$ il existe un $[V]$-path de X à Y.

[V]-component

- un $[V]$-component est $[V]$-connected maximal non vide de l'ensemble des variables $W \subseteq (var(\mathcal{H}) - V)$.

Pour tout $[V]$-component C, $aretes(C) = \{h \in aretes(\mathcal{H}) | h \cap C \neq \emptyset\}$.

On note $[v]$-component au lieu de $[x(v)]$-component où $v \in T$

2.5.3 Définition de la forme normale

Une décomposition hyper-arborescente $HD = \langle T, x, \lambda \rangle$ d'un hypergraphe $\mathcal{H}$ est en *forme normal* (NF) si pour chaque sommet $r \in sommets(T)$,et pour chaque fils s de r, toutes ces conditions doit être vérifiées :

1- il y a exactement un $[r]$-composant tels que $x(T_s) = C_r \cup \big(x(s) \cap x(r)\big)$
2- $x(s) \cap C_r \neq \emptyset$,où C_r est le $[r]$-composant satisfaisant la conditions (1)
3- pour chaque $h \in \lambda(s), h \cap var(aretes(C_r)) \neq \emptyset$, C_r est le $[r]$-composant satisfaisant la conditions (1).
4- $x(s) = var(aretes(C_r)) \cap var(\lambda(s))$, où C_r est le $[r]$ -composant satisfaisant la condition (1)

Définition

Une décomposition hyper-arborescence pondérée est une décomposition hyper-arborescente définit dans (2.3.7) on lui ajoutant une fonction hyper-arborescente pondéré (WHF) qui mappe chaque décomposition hyper-arborescente $\mathrm{HD} = \langle \mathrm{T}, \mathrm{x}, \lambda \rangle$ d'un hypergraphe $\mathcal{H}$ à une valeur numérique, appelé pondération (weight) de HD.

2.5.4 Fonction d'agrégation arborescente (Tree aggreation function)

Soit $\langle \mathbb{R}^+, \oplus, min, \perp, \infty \rangle$ un semi anneau ,où $\oplus$ est commutative ,associative, et opérateur binaire fermé, $\perp$ est l'élément neutre de $\oplus$ et l'élément absorbant de min,et min distribué sur $\oplus$ .soit une fonction g et un ensemble d'élément $S = \{p_1, \dots, p_n\}$,on notant avec $\bigoplus_{p_i \in S} g(p_i)$ la valeur $g(p_1) \oplus \dots \oplus g(p_n)$.

Définition

Soit $\mathcal{H}$ un hypergraphe.une fonction d'agrégation arborescente noté *TAF* est toute fonction hyper-arborescente pondérée de la forme suivante :

$F_{\mathcal{H}}^{\oplus, v, e}(HD) = \bigoplus_{p \in N} \left(v_{\mathcal{H}}(p) \oplus \left(\bigoplus_{(p,p') \in E} e_{\mathcal{H}}(p, p')\right)\right)$,

En associant tout valeur $\mathbb{R}^+$ avec la décomposition hyper arborescente $HD = \langle T, x, \lambda \rangle$ où $v_{\mathcal{H}} : N \mapsto \mathbb{R}^+$ et $e_{\mathcal{H}} : N \times N \mapsto \mathbb{R}^+$sont deux fonctions polynomiales qui évaluent les sommets et les arêtes respectivement.

Exemple :

Soit une requête Q,et une décomposition hyper arborescente $HD = \langle T, x, \lambda \rangle$ dans une forme normale pour $\mathcal{H}(Q)$. Pout tout sommet p de T ,soit $E(p)$ une expression relationnelle tels que $E(p) = \bowtie_{h \in \lambda(p)} \prod_{x(p)} rel(h)$,(i.e la jointure de toutes les relations qui correspondent aux hyper-arêtes de $\lambda(p)$ projetées sur les variables de $x(p)$).

Avec un nœud p' fils d'un nœud p dans la decomposition HD ,nous définissons $v^*_{\mathcal{H}(Q)}(p)$ et $e^*_{\mathcal{H}(Q)}(p, p')$ comme suit :

- $v^*_{\mathcal{H}(Q)}(p)$ est l'estimation de cout de l'évaluation de l'expression $E(p)$
- $e^*_{\mathcal{H}(Q)}(p, p')$ est l'estimation de cout de l'évaluation de la semi jointure $E(p) \ltimes E(p')$.

2.5.5 Algorithme exacte pour le calcul de la décomposition hyper-arborescente pondérée (minimal-k-decomposition)

L'algorithme qu'on va présenter est un algorithme exacte [35] qui permet de calculer la décomposition pondérée minimal appelé minimal-k-decomposition.

Principe :

Cet algorithme maintien un graphe bipartie orienté soit CG(condidate Graph). Leur nœuds sont partitionnés on deux ensembles N_{sub} et N_{sol},qui représentent respectivement les sous problèmes à résoudre et leurs solutions candidates . Les nœuds dans N_{sub} ont la forme (R, C),où R est l'ensemble de k-aretes au plus de $\mathcal{H}$,appelé k-sommets,et C est un $[var(R)]$-component. Le nœud $(\emptyset, var(\mathcal{H}))$ représente le problème total. Les nœuds dans N_{sol} ont la forme (S, C'),où S est le k-sommets, C' est le composant qu'on va le décomposé ,avec $var(S) \cap C' \neq \emptyset$ et, $\forall h \in S, h \cap var(aretes(C')) \neq \emptyset$. Le nœud (S, C'),a un arc qui pointe vers tous les nœuds de la forme $(R', C') \in N_{sub}$ pour le quel il soit une solution candidate ,en vérifiant la condition $var(aretes(C')) \cap var(R') \subseteq var(S)$. Aussi pour le nœud (S, C') il y a un nombre de nœuds qui lui connectent de la forme $(S, C'') \in N_{sub}$,pour chaque $[var(S)]$-component C'' qui est inclue dans C' tels que chaqu'un de ces nœuds représente un sous problème de (S, C') (ligne 10 à 14).

On a pour tout nœud $p' \in N_{sol}$, $weight(p') := v_{\mathcal{H}}(p')$ .et pour tout p descendant de p' , $weight(p') := weight(p') \oplus (weight(p) \oplus e_{\mathcal{H}}(p', p))$,si on a plus d'un nœud $q \in N_{sub}$ descendant de p' tels que : $q \in Descendant(p')$, $p \in N_{sol}$ alors on applique $weight(p) \oplus e_{\mathcal{H}}(p', p) = min_{p'' \in Descendant(q)} \left(weight(p'') \oplus e_{\mathcal{H}}(p', p'') \right)$.

Algorithme 7 : minimal-k-decomposition

1 : entrée : un hypergaphe, une fonction d'agrégation d'arbre $F_{\mathcal{H}}^{\oplus,v,e}$.

2 : sortie : la décomposition $[F_{\mathcal{H}}^{\oplus,v,e}, kNFD_{\mathcal{H}}]$-hypertree minimal de

$\mathcal{H}$,s'il existe, échec sinon.

3 : var $CG = (N_{sol} \cup N_{sub}, A, weight)$; (un graphe bipartie orienté et pondéré).

4 : $HD = \langle (N_{sol}, E), x, \lambda \rangle$; (hypertree de $\mathcal{H}$).

5 : begin

6 : (*construction de graphe candidat*)

7 : $N_{sub} \coloneqq \{(\emptyset, var(\mathcal{H}))\} \cup \{(R, C) | R$ est un k-component et C est un

$[var(R)]$-component.

8 : $N_{sol} \coloneqq \{(S, C) | S$ est un k-sommets, C est tout $[var(R)]$-

component, $var(S) \cap C \neq \emptyset$ et, $\forall h \in S, h \cap var(aretes(C)) \neq \emptyset\}$;

9 : $A \coloneqq \emptyset$;

10 : **pour chaque** $(R, C) \in N_{sub}$ **faire**

11 : **pour chaque** $(S, C) \in N_{sol}$ tels que $var(aretes(C)) \cap$

$var(R) \subseteq var(S)$ **faire**

12 : Ajouter un arc de (S, C) vers (R, C) dans A ;

13 : **pour chaque** $(S, C') \in N_{sub}$ qui satisfait $C' \subseteq C$ **faire**

(*connecter leur sous problèmes*)

14 : Ajouter un arc de (S, C') à (S, C) dans A ;

15 : **fin pour ;**

16 : **fin pour ;**

17 : **fin pour ;**

18 : (*Evaluer le graphe candidat*)

19 : **pour chaque** $p = (S, C) \in N_{sol}$ **faire**

20 : $\lambda(p) \coloneqq S; x(p) \coloneqq var\big(aretes(C)\big) \cap var(S)$, et $weight(p) \coloneqq v_{\mathcal{H}}(p)$;

21 : $weighted \coloneqq \{p \in N_{sol} | desendant(p) = \emptyset\}$;

22 : $A\_Calculé \coloneqq N_{sub}$;

23 : **tant que** $A\_Calculé \neq \emptyset$ **faire**

24 : Extraire un nœud q de $A\_Calculé$ tels que $Descendants(q) \subseteq weighted$;

25 : **si** $Descendants(q) = \emptyset$ **alors** (*pas solution à ce sous problème *)

26 : Supprimer tout nœud $p' \in Acsendants(q)$;

27 : **sinon** (* ce sous problème est résolu avec succès *)

28 : **pour chaque** $p^{'} \in Ascendants(q)$ **faire**

29 : $weight(p^{'}) \coloneqq weight(p^{'}) \oplus min_{p \in Descendants(q)}($

$weight(p) \oplus e_{\mathcal{H}}(p, p^{'}))$;

30 : **si** $Descendants(p^{'}) \cap A\_Calculé = \emptyset$ **alors**

31 : Ajouter p' à $weighted$;

32 : **fin tant que** (*$A\_Calculé$ *)

33 : **fin pour ;**

34 : (* Identifier la décomposition hyper-arborescente minimal s'il existe *)

35 : **si** $Descendants\Big(\big(\emptyset, var(\mathcal{H})\big)\Big) = \emptyset$ **alors**

36 : Afficher Echec ;

37 : **sinon**

38 : $E \coloneqq \emptyset$; (* initialement, l'arbre de décomposition n'a pas d'arêtes *)

39 : Choisir le $minimum - weighted\ p \in Descendants(\big(\emptyset, var(\mathcal{H})\big))$;

40 : $Select\text{-}hypertree(p)$;

41 : Supprime tous les Sommets isolés dans l'arbre (N_{sol}, E) ;

42 : Afficher HD ;

43 : **procédure** $Select\text{-}hypertree(p \in N_{sol})$

44 : **pour chaque** $q \in Descendant(p)$ **faire**

45 : Choisir le $minimum\text{-}weighted\ p' \in Descendants(q)$;

46 : Ajouter l'arête $\{p, p'\}$ à E ;

47 : $Select\text{-}hypertree(p')$;

48 : **fin procédure ;**

fin.

Complexité temporelle : $O(\Psi^2 m^2 c_{\oplus} + \Psi^2 m\, c_e c_{min} + \Psi^2 m c_v)$.

$\Psi = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n!}{i!(n-i)!}$; Ψ dénote le nombre de k-sommets qu'on peut les construire.

Où n : est le nombre des arêtes, m : nombre de sommets et k : est une constante prédéfinit qui dénote la largeur de la décomposition hyper-arborescente (définit en entrée de l'algorithme).

$c_{\oplus}$, c_v,c_e et c_{min}, soit les couts maximum de l'évaluation des opérateurs v,e,min et $\oplus$.

2.6 Conclusion :

Nous avons étudiés dans ce chapitre les méthodes de décompositions structurelles qui permettent de transformer un hypergraphe cyclique en un autre acyclique (arborescence, hyper-arborescence) afin de simplifier la résolution en utilisant les algorithmes de résolutions tels que le simple retour arrière et la résolution arborescente (*tree solving[36]).*

A la fin de ce chapitre en a étudié une méthode complète de décomposition hybride (basée sur le traitement structurel et quantitatif), mais le problème qui se pose avec les méthodes complètes est qu'elle doit balayer toute l'espace de recherche, chose qui présente une complexité exponentielle.

Dans le chapitre suivant, nous allons étudier les méthodes dites méta-heuristiques (algorithmes génétique, recherche taboue, colonie de fourmis,…). Ce sont des méthodes incomplètes dont le but est d'explorer l'espace de recherche efficacement afin de déterminer des solutions optimales.

Chapitre 3

Les Méthodes méta-heuristiques

Dans le chapitre précédant nous avons étudiés les méthodes dites complètes et on a constaté que leurs problème majeur est le temps de résolution qui est exponentielle au diriment de problème étudié, afin d'écarter ce problème, des méthodes méta-heuristiques ont proposées comme solution.

Dans ce chapitre nous allons étudiés les déférents méthodes méta-heuristiques, celles qui basent sur une seul solution (tel que la recherche taboue), et celles qui basent sur une population de solution (tel que les algorithmes génétiques)

3.1 Définition

Le mot méta-heuristique est dérivé de la composition de deux mots grecs:

- *heuristique* qui vient du verbe *heuriskein* et qui signifie « *trouver* ».
- *méta* qui est un suffixe signifiant « *au-delà* », « *dans un niveau supérieur* ».

« Les méta-heuristiques sont des stratégies qui permettent de guider la recherche d'une solution optimale »

3.2 Quelques propriétés

- Les techniques qui constituent des algorithmes de type méta-heuristique vont de la simple procédure de recherche locale à des processus d'apprentissage complexes.
- Les méta-heuristiques sont en général non-déterministes et ne donnent aucune garantie d'optimalité.
- Les méta-heuristiques peuvent contenir des mécanismes qui permettent d'éviter d'être bloqué dans des régions de l'espace de recherche.
- Les concepts de base des méta-heuristiques peuvent être décrits de manière abstraite, sans faire appel à un problème spécifique.

3.3 Classification des méta-heuristiques

On peut distinguer celles qui travaillent avec une *population de solutions* de celles qui ne manipulent qu'une *seule solution* à la fois. Les méthodes qui tentent itérativement d'améliorer une solution sont appelées méthodes de recherche locale ou méthodes de trajectoire. La méthode Tabou, le Recuit Simulé et la Recherche à Voisinages Variables sont

des exemples typiques de méthodes de trajectoire. Ces méthodes construisent une trajectoire dans l'espace des solutions en tentant de se diriger vers des solutions optimales. L'exemple le plus connu de méthode qui travaille avec une population de solutions est l'algorithme génétique.

3.4 Définitions

- **Diversification**

Par diversification, on sous-entend généralement une *exploration* assez large de l'espace de recherche, l'exploration à pour but d'identifier rapidement les régions de l'espace de recherche qui contiennent des solutions de bonne qualité.

- **Intensification**

Par intensification, on sous-entend par une *exploitation* de l'information accumulée durant la recherche à pour but de fouiller la bonne solution dans une seule région.

- **Structure de voisinage**

Soit S l'ensemble de solution à un problème d'optimisation. Soit f une fonction qui mesure la valeur f(s) de toute solution s dans S.On veut déterminer une solution $s \in S$ de valeur f(s) minimale. Le problème à résoudre est donc le suivant : $\min_{s \in S} f(s)$

Un voisinage est une fonction N qui associe un sous-ensemble de S à toute solution $s \in S$ Une solution $s' \in N(s)$ est dite voisine de s.

- **Minimum local**

Une solution $s \in S$ est un minimum local relativement à la structure de voisinage N si $f(s) \leq f(s')$ pour tout $s' \in N(s)$.

- **Minimum global**

Une solution $s \in S$ est un minimum global si $f(s) \leq f(s')$ pour tout $s' \in S$.

3.5 Méthodes de recherches locales

3.5.1 Méthode de descente

Elle peut être décrite comme suit [37] :

Algorithme 8 : Méthode de descente

1. Choisir une solution $s \in S$
2. Déterminer une solution s' qui minimise f dans N(s).
3. **si** f(s′) < f(s) **alors** poser s ≔ s′ et retourner à 2. **sinon** stop.

Une variante consiste à parcourir N(s) et à choisir la première solution s' rencontrée telle que f(s') < f(s) (pour autant qu'une telle solution existe). Le principal défaut des méthodes de descente est qu'elles s'arrêtent au premier minimum local rencontré. Tel que déjà mentionné, un minimum local pour une structure de voisinage ne l'est pas forcément pour une autre structure. Le choix de N peut donc avoir une grand influence sur l'efficacité d'une méthode de descente.

Pour éviter d'être bloqué au premier minimum local rencontré, on peut décider d'accepter, sous certaines conditions, de se déplacer d'une solution s vers une solution $s' \in N(s)$ telle que $f(s') \geq f(s)$. C'est ce que font les méthodes que nous décrivons ci-dessous.

3.5.2 Recuit simulé (Simulate annealing)

Qui peut être définit comme suit [38]:

Algorithme 9 : Méthode du Recuit Simulé

1. Choisir une solution $s \in S$ ainsi qu'une température initiale T.
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Choisir aléatoirement $s' \in N(s)$;
4. Générer un nombre réel aléatoire r dans [0,1];
5. **si** $r < p(T, s, s')$ **alors** poser $s \coloneqq s'$;
6. Mettre à jour T;
7. **fin du tant que**.

La fonction $p(T, s, s')$ est généralement choisie comme étant égale à la distribution de Boltzmann $e^{\frac{f(s)-f(s')}{T}}$.ainsi :

- Si $f(s') < f(s)$ alors $e^{\frac{f(s)-f(s')}{T}} > 1$, ce qui signifie que $r < p(T, s, s')$ et on accepte donc s'
- Si T a une très grande valeurs alors $e^{\frac{f(s)-f(s')}{T}} \cong 1$, et on est donc presque sûr d'accepter s'
- Si T a une tès petite valeur et si $f(s') > f(s)$ alors $e^{\frac{f(s)-f(s')}{T}} \cong 0$ et on va donc probablement refuser s'

En général, on choisit une température initiale suffisamment élevée qui donne une plus grande liberté pour l'exploration de l'espace de recherche. Puis, petit à petit, la température décroît jusqu'à atteindre une valeur proche de 0, ce qui signifie que la méthode n'acceptera plus de détériorer une solution. Pour choisir la température initiale, on peut générer aléatoirement un ensemble de paires (s, s') où $s' \in N(s)$, et choisir ensuite la température initiale de telle sorte qu'une proportion r de ses solutions auraient été acceptées à l'étape 5 de l'algorithme.

Sous certaines conditions de décroissance de la température, l'algorithme du Recuit Simulé converge en probabilité vers un optimum global lorsque le nombre d'itérations tend vers l'infini. Plus précisément, soit P(k) la probabilité que l'optimum global soit trouvé après k itérations, et soit T_k la température à la $K^{ème}$ itération.

Il existe $L \in \mathbb{R}$ tel que $\lim_{k\to\infty} P(K) = 1$ si et seulement si $\sum_{k=1}^{\infty} e^{\frac{L}{T_k}} = \infty$ On peut, par exemple, définir $T_k = \frac{L}{\log(k+c)}$ où c est une constante. Mais de telles stratégies de décroissance de la température sont trop lentes et nécessitent donc des temps de calcul astronomiques avant d'atteindre la convergence vers l'optimum global. En pratique, on préfère donc utiliser d'autres stratégies qui ne garantissent plus la convergence vers l'optimum global, mais qui convergent plus rapidement vers un état stable. La méthode la plus utilisée consiste à définir $T_{k+1} = \alpha T_k$ où α est un nombre réel choisi entre 0 et 1 (typiquement α =0.95).

La décroissance de la température peut également être réalisée par paliers. Certains préconisent l'utilisation de stratégies non monotones. On peut ainsi rehausser la température lorsque la recherche semble bloquée dans une région de l'espace de recherche. On peut alors considérer une grande augmentation de la température comme un processus de diversification alors que la décroissance de la température correspond à un processus d'intensification.

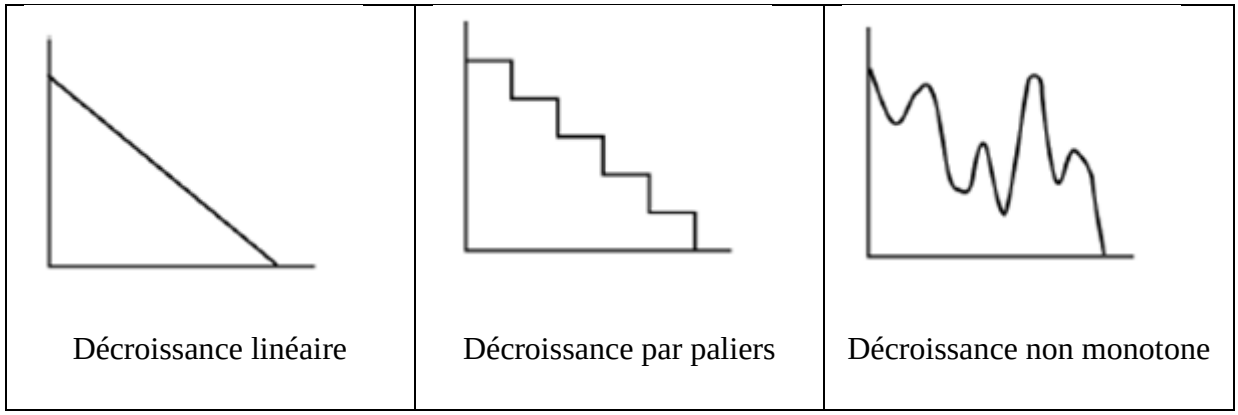


Figure 3.1 type de décroissance celons l'ajustement de la température

La méthode du Recuit Simulé est une méthode sans mémoire. On peut facilement l'améliorer en rajoutant une mémoire à long terme qui stocke la meilleure solution rencontrée. En effet, l'algorithme du Recuit Simulé décrit ci-dessus peut converger vers une solution s^* en ayant visité auparavant une solution s de valeur $f(s) < f(s^*)$ Il est donc naturel de mémoriser la meilleure solution rencontrée durant le processus de recherche.

Comme critère d'arrêt, on peut choisir une limite sur le temps, ou une limite sur le nombre d'itérations sans modification de la solution courante. Lorsqu'on stocke la meilleure solution rencontrée, on peut décider de stopper la recherche dès qu'un certain nombre d'itérations a été effectué sans amélioration de cette solution.

L'un des principaux défauts de la méthode du Recuit Simulé est l'étape 3 où s' est choisi aléatoirement dans $N(s)$ .On peut donc être proche de l'optimum et passer juste à côté sans le voir.

3.5.3 La recherche taboue

Le principe de la recherche taboue [37] est de choisir à chaque itération la meilleure solution $s' \in N(s)$ même si $f(s') > f(s)$. Lorsqu'on atteint un minimum local s par rapport au voisinage N, la recherche taboue va donc se déplacer vers une solution s' plus mauvaise que s. Le danger est alors de revenir à s immédiatement si $s \in N(s')$ puisque s est meilleure que s', pour éviter de tourner ainsi en rond, on crée une liste T qui mémorise les dernières solutions visitées et interdit tout déplacement vers une solution de cette liste. Cette liste T est appelée *liste taboue*.

Les solutions ne demeurent dans T que pour un nombre limité d'itérations. La liste T est donc une mémoire à court terme. Si une solution s' est dans T on dit que s' est une solution taboue. De même, tout mouvement qui nous mène de la solution courante à une solution de T est appelé mouvement tabou.

Il se peut que l'on visite une même solution à plusieurs reprises, à intervalles plus petits que $|T|$ À titre d'illustration, supposons que l'on cherche à ordonner les quartes premières lettres de l'alphabet $\{a, b, c, d\}$,pour passer d'un ordre à un autre, on permute les positions de deux lettres et on rend tabou une nouvelle permutation de ces deux lettres. On pourrait donc avoir la suite de solutions suivante :

$$abcd \rightarrow bacd \rightarrow bcad \rightarrow dcab \rightarrow acdb \rightarrow abdc \rightarrow abcd$$

Les permutations qui ont été effectuées sont les suivantes : ab, ac, bd, ad, bc et cd.On voit donc qu'aucune permutation n'a été réalisée deux fois et on retombe cependant sur la solution initiale.

Il se peut que la liste T interdise la visite de solutions qui n'ont jamais été visitées (de telles solutions pouvant être des optimaux globaux). Par exemple, supposons à nouveau que l'on cherche à ordonner les premières lettres de l'alphabet. Comme ci-dessus, on permute les positions de deux lettres pour se déplacer d'une solution à une autre et on rend tabou une nouvelle permutation de ces deux lettres. On pourrait avoir la suite de solutions suivante : $abcd \rightarrow bacd \rightarrow dacb \rightarrow dcab$

On a permuté les lettres ab, puis bd, et enfin ac. Si $|T| \geq 3$, on n'a donc plus le droit de permuter ab. Ainsi, on interdit la solution $dcba$ qui pourrait être la solution optimale.

Pour éviter le deuxième défaut mentionné ci-dessus, il est d'usage de lever le statut tabou d'une solution si celle-ci satisfait certains *critères d'aspiration*. En règle générale, le statut tabou d'une solution est levé si celle-ci est meilleure que la meilleure solution s^* rencontrée jusqu'ici.

Étant donnée une solution courante s , définissons l'ensemble $N^T(s)$ comme l'ensemble des solutions de $N(s)$ vers lesquelles la recherche taboue accepte de se diriger. L'ensemble $N^T(s)$ contient donc toutes les solutions qui ne sont pas taboues ainsi que celles qui le sont mais dont le statut tabou est levé en raison du critère d'aspiration. Ainsi :

$N^T(s) = \{s' \in N(s)$ tel que $s' \notin T$ ou $f(s') < f(s^*)\}$

La recherche taboue peut être décrite comme suit :

Algorithme 10 : Recherche Tabou

1. Choisir une solution $s \in S$, poser $T \coloneqq \emptyset$ et $s^* \coloneqq s$;
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Déterminer une solution s' qui minimise $f(s')$ dans $N^T(s)$
4. **si** $f(s') < f(s^*)$ **alors** poser $s^* \coloneqq s'$
5. Poser $s \coloneqq s'$ et mettre à jour T
6. **fin du tant que**

Comme critère d'arrêt on peut par exemple fixer un nombre maximum d'itérations sans amélioration de s^*, ou on peut fixer un temps limite après lequel la recherche doit s'arrêter.

Les types de la liste taboue qu'on peut trouver sont :

- liste taboue de longueur variable [38], listes taboues réactives, dont la taille varie selon les résultats de la recherche.

La liste taboue est une mémoire à court terme. On peut rajouter une mémoire à plus long terme. Quatre types de mémoire sont alors envisageables :

- Les mémoires basées sur la *récence* mémorisent pour chaque solution (attribut ou mouvement) l'itération la plus récente qui l'impliquait. On peut par exemple favoriser les mouvements vers des solutions contenant des attributs peu récents, afin d'éviter de rester bloquer dans une même région de l'espace de recherche.

- Les mémoires basées sur la *fréquence* mémorisent le nombre de fois que les solutions (attributs ou mouvements) ont été visitées. Cette information permet d'identifier les régions de l'espace de recherche qui ont été le plus visitées. On peut exploiter cette information pour diversifier la recherche.

- Les mémoires basées sur la *qualité* mémorisent les meilleures solutions rencontrées afin d'identifier les composantes communes de ces solutions. On peut ensuite faire usage de cette information pour créer de nouvelles solutions qui contiennent autant que possible ces apparemment bonnes composantes.

- Les mémoires basées sur *l'influence* mémorisent les choix qui se sont avérés les plus judicieux ou les plus critiques durant le processus de recherche. Ceci permet d'essayer de répéter les bons choix et d'éviter de répéter les mêmes erreurs.

3.5.4 La méthode GRASP (Greedy Randomized Adaptive Search Procedure)

Cette méthode est une procédure itérative composée de deux phases : une phase constructive et une phase d'amélioration [38]. En supposant qu'une solution est constituée d'un ensemble de composantes :

- **la phase constructive** *:* génère une solution pas à pas, en ajoutant à chaque étape une nouvelle composante. La composante rajoutée est choisie dans une liste de candidats. Chaque composante est en fait évaluée à l'aide d'un critère heuristique qui permet de mesurer le bénéfice qu'on peut espérer en rajoutant cette composante à la solution partielle courante. La liste de candidats, notée *RCL* (Restricted Candidate List) contient les *R* meilleures composantes selon ce critère.

- **La phase d'amélioration** : consiste généralement en l'application d'une méthode de descente, de recuit simulé ou de recherche taboue.

La méthode GRASP peut être décrite comme suit :

Algorithme 11 : Méthode GRASP

1. $f^* = \infty$ (valeur de la meilleure solution rencontrée)
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Poser $s \coloneqq \emptyset$
4. **tant que** la solution courante s n'est pas complète **faire**
5. Évaluer l'ensemble des composantes pouvant être rajoutées à s
6. Déterminer la liste RCL des R meilleures composantes
7. Choisir aléatoirement une composante dans RCL et l'ajouter à s
8. **fin du tant que**
9. Appliquer une méthode de descente, de Recuit Simulé ou de recherche taboue à s et soit s' le résultat.
10. **si** $f(s') < f^*$ **alors** poser $s^* \coloneqq s'$ et $f^* \coloneqq f(s')$;
11. **fin du tant que**

3.6 Méthodes basées sur les populations (méthodes évolutives)

Les méthodes basées sur des populations de solutions sont parfois appelées méthodes évolutives parce qu'elles font évoluer une population d'individus selon des règles bien précises. Ces méthodes alternent entre des périodes d'adaptation individuelle et des périodes de coopération durant lesquelles les individus peuvent échanger de l'information.

Une méthode évolutive peut être décrite comme suit :

Méthode Évolutive

1. Générer une population initiale d'individus
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Exécuter une procédure de coopération;
4. Exécuter une procédure d'adaptation individuelle
5. **fin du tant que**

3.6.1 Caractéristiques des méthodes évolutives

Nous décrivons ci-dessous les principales caractéristiques qui permettent de faire la différence entre diverses méthodes évolutives.

a) Types d'individus

Les individus qui évoluent dans une méthode basée sur les populations ne sont pas nécessairement des solutions. Il peut aussi s'agir de morceaux de solutions ou tout simplement d'objets que l'on peut facilement transformer en solutions. Considérons par exemple un problème de tournées de véhicules. Un individu peut être la tournée d'un véhicule qui visite un sous-ensemble de clients alors qu'une solution est un ensemble de tournées (et donc d'individus) qui visitent tous les clients. Pour la coloration de graphes, on peut considérer toute permutation des sommets comme un individu. En appliquant l'algorithme séquentiel de coloration, on peut transformer une permutation en une coloration et donc un individu en solution.

b) Type d'évolution

À chaque itération d'une méthode évolutive, de nouveaux individus sont créés et la population de l'itération suivante sera ainsi constituée d'anciens individus (qui auront survécu) et de nouveaux individus. La méthode évolutive doit indiquer comment décider de la

survie des individus et comment choisir les nouveaux individus qui vont entrer dans la population. Lorsqu'on change totalement de population d'une itération à l'autre (c'est-à-dire que seuls les nouveaux individus sont conservés pour l'itération suivante), on parle de remplacement générationnel. Par contre, lorsque seulement une partie de la population peut varier d'une itération à la suivante, on parle de remplacement stationnaire.

La plupart des méthodes évolutives utilisent des populations de taille fixe, et on décide généralement de garder les p meilleurs individus (parmi l'union des anciens et des nouveaux). Des populations de taille variable (où on décide par exemple de manière aléatoire de la survie des individus) sont cependant également possibles.

c) Structure de voisinage

À chaque individu on associe un sous-ensemble du d'autres individus avec lesquels il peut échanger de l'information. Si chaque individu peut communiquer avec tous les autres individus de la population, on parle de population non structurée. Par contre, si chaque individu ne peut communiquer qu'avec un sous-ensemble d'individus, on parle de population structurée. Les structures les plus communes sont la grille et le cercle.

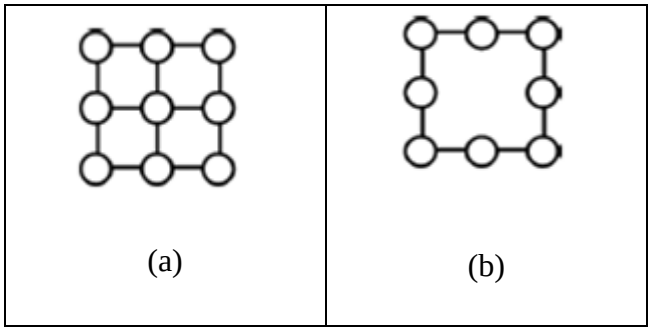


Figure 3.2 (a)structure en grille (b)structure en cercle.

d) Sources d'information

Le nombre d'individus qui coopèrent pour créer un nouvel individu est souvent égal à deux. On parle alors de parents qui génèrent des enfants. On peut cependant également combiner plus de deux solutions pour créer des enfants.

Certaines méthodes évolutives utilisent par exemple l'information contenue dans toute la population pour créer un nouvel individu. D'autres méthodes utilisent même toutes

les populations de toutes les itérations précédentes pour créer des enfants : on dit alors que la source d'information est l'historique de la recherche. Par historique on entend généralement toute information qui ne peut pas être obtenue en analysant les individus de la population courante; la connaissance des compositions des populations précédentes est nécessaire pour accéder à cette information.

e) Irréalisabilité

Un individu est un objet défini avec des règles bien précises. En combinant des individus pour en créer des nouveaux, il peut arriver que le nouvel objet résultant de l'échange d'information ne soit pas un individu admissible.

Par exemple, définissons un individu comme une coloration sans conflit des sommets d'un graphe. Étant données deux colorations C_1 et C_2, on peut combiner celles-ci pour en créer une nouvelle en choisissant la couleur de chaque sommet aléatoirement dans C_1 ou C_2. Une telle coloration peut avoir des conflits et n'est donc pas nécessairement un individu. Dans une telle situation, on peut réagir d'au moins de trois façons :

- rejeter l'enfant;
- accepter l'enfant et rajouter une composante à la fonction objectif qui pénalise les violations de contraintes;
- développer des procédures de combinaisons qui garantissent l'obtention d'enfants admissibles (sans violation).

f) Intensification

Idéalement, on devrait pouvoir localiser l'information pertinente qui rend un individu meilleur qu'un autre. Lorsque ceci est possible, il reste alors à développer une procédure de coopération qui crée des enfants en combinant adéquatement les informations pertinentes de chacun des parents. La phase d'adaptation individuelle n'est alors pas vraiment indispensable au bon fonctionnement de la méthode évolutive.

C'est le cas par exemple pour les problèmes de tournées de véhicules où la bonne qualité d'une solution peut être expliquée par l'utilisation d'arcs de faible coût.

Il existe cependant de nombreux problèmes pour lesquels une telle information pertinente est difficile à localiser.

Par exemple, si $\prod_1$ et $\prod_2$ sont deux permutations des sommets d'un graphe et si C_1 et C_2 sont les deux colorations résultant de l'algorithme séquentiel de coloration, avec C_1 utilisant moins de couleurs que C_2, il est difficile de localiser dans $\prod_1$ les raisons qui font que $\prod_1$ est meilleure que $\prod_2$. Il est donc également difficile de transmettre une information pertinente lors de la phase de coopération. Il est alors important d'utiliser une bonne procédure d'adaptation individuelle. Il est courant de faire appel à une technique de recherche locale qui est appliquée à chacun des individus afin d'explorer les régions de l'espace de recherche qui leur sont proches.

g) Diversification

Une difficulté majeure rencontrée lors de l'utilisation des méthodes évolutives est leur convergence prématurée. Certains utilisent des procédures de bruitage qui modifient légèrement les individus de manière aléatoire. Ce bruitage est appliqué indépendamment sur chaque individu. Il diffère de l'utilisation d'une recherche locale par le fait que son effet sur la qualité de la solution n'est pas prévisible. L'opérateur de bruitage le plus connu est la mutation dans les algorithmes génétiques. Au lieu de modifier les individus aléatoirement, certains préfèrent créer de nouveaux individus différents des individus déjà rencontrés en faisant usage d'une mémoire à long terme basée par exemple sur la récence et la fréquence.

Définition

Bruitage est une méthode qui fait appel à une notion de bruitage d'une donnée, qui est définie de la façon suivante : la donnée bruitée est produite à partir de la donnée initiale en ajoutant à chacun des réels une composante calculée comme le produit de trois éléments : *a)* une fonction aléatoire à valeurs sur l'intervalle [0,1], *b)* un paramètre permettant de contrôler le niveau du bruit, *c)* le plus grand des réels concernés, afin de normaliser le niveau du bruit par rapport à la donnée.

3.6.2 Les méthodes évolutives

3.6.2 .1 Les algorithmes génétiques

Les algorithmes génétiques [38] sont inspirés de la théorie de l'évolution et des processus biologiques qui permettent à des organismes de s'adapter à leur environnement.

Dans l'algorithme ci-dessous qui n'est qu'une variante des algorithmes génétiques, la coopération a lieu à l'étape 5 alors que l'adaptation individuelle a lieu à l'étape 6. Une description des 7 caractéristiques est également donnée.

Algorithme 12 : Algorithme génétique avec remplacement générationnel

1. Générer une population initiale P_i de $p \geq 2$ individus et poser $i := 0$;
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Poser $i := i + 1$ et $P_i := \emptyset$;
4. **répéter** p fois les deux lignes suivantes :
5. Créer un enfant E en croisant deux individus I_1 et I_2 de P_{i-1};
6. Appliquer l'opérateur de mutation à E et rajouter l'enfant modifié à P_i
7. **fin du tant que**

Types d'Individus : en général il s'agit de solutions

Type d'évolution : remplacement générationnel avec une population de taille constante

Structure de voisinage : population possiblement structurée

Sources d'information : deux parents

Irréalisabilité : l'opérateur de croisement évite la génération de solutions non admissibles

Intensification : aucune

Diversification : mutation qui est une procédure de bruitage

3.6.2.2 La recherche dispersée (scatter search)

La recherche dispersée [38] consiste à générer un ensemble D_i de points dispersés à partir d'un ensemble R_i de points de références. Ces points dispersés sont obtenus en effectuant tout d'abord des combinaisons linéaires des points de référence. Ces combinaisons linéaires peuvent avoir des coefficients négatifs, ce qui veut dire que les points résultant peuvent être à l'extérieur de l'enveloppe convexe des R_i. L'ensemble C_i des points résultant de ces combinaisons linéaires n'est pas forcément un ensemble de solutions admissibles.

C'est pourquoi, on applique une procédure de réparation sur chaque point de C_i pour obtenir un ensemble A_i de points admissibles. Les points de A_i sont finalement optimisés à l'aide d'une recherche locale pour obtenir l'ensemble D_i des points dispersés. Le nouvel ensemble R_{i+1} de points de référence est obtenu en sélectionnant des points dans $R_i \cup D_i$.

Voici le pseudo-code et les caractéristiques de cette méthode :

Algorithme 13 : Algorithme de recherche dispersée

1. Générer une population initiale R_i d'individus et poser $i \coloneqq 0$
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Créer un ensemble C_i de points candidats en effectuant des combinaisons linéaires des points de R_i;
4. Transformer C_i en un ensemble $A_i$de points admissibles (à l'aide d'une procédure de réparation);
5. Appliquer une recherche locale sur chaque point de A_i; soit D_i l'ensemble résultant.
6. Sélectionner des points dans $R_i \cup D_i$ pour créer le nouvel ensemble R_{i+1} de référence et poser $i \coloneqq i + 1$;
7. **fin du tant que**

Types d'Individus : solutions admissibles

Type d'évolution : remplacement stationnaire avec une population de

taille généralement constante

Structure de voisinage : population non structurée

Sources d'information : au moins deux parents

Irréalisabilité : les points non admissibles sont réparés

Intensification : recherche locale

Diversification : combinaison non convexe des points de référence.

3.6.2.3 L'optimisation par colonies de fourmis (ACO pour Ant Colony Optimisation)

L'optimisation par colonies de fourmis [38] est une méthode évolutive inspirée du comportement des fourmis à la recherche de nourriture. Il est connu que les fourmis sont capables de déterminer le chemin le plus court entre leur nid et une source de nourriture.

Ceci est possible grâce à la phéromone qui est une substance que les fourmis déposent sur le sol lorsqu'elles de déplacent. Lorsqu'une fourmi doit choisir entre deux

directions, elle choisit avec une plus grande probabilité celle comportant une plus forte concentration de phéromone. C'est ce processus coopératif que l'ACO tente d'imiter.

Chaque fourmi est un algorithme constructif capable de générer des solutions. Soit D l'ensemble des décisions possibles que peut prendre une fourmi pour compléter une solution partielle. La décision $d \in D$ qu'elle choisira dépendra de deux facteurs, à savoir la force gloutonne et la trace.

- la force gloutonne est une valeur η_d qui représente l'intérêt qu'a la fourmi à prendre la décision d. Plus cette valeur est grande, plus il semble intéressant de faire le choix d. En général, cette valeur est directement proportionnelle à la qualité de la solution partielle obtenue en prenant la décision d.
- la trace τ_d représente l'intérêt historique que peut avoir la fourmi, de prendre la décision d. Plus cette quantité est grande, plus il a été intéressant dans le passé de prendre cette décision.

Étant donné deux paramètres α et β qui donnent plus ou moins d'importance à ces deux facteurs, la fourmi va prendre la décision d avec une probabilité

$$\frac{(\eta_d)^\alpha (\tau_d)^\beta}{\sum_{r \in D} (\eta_r)^\alpha (\tau_r)^\beta}$$

Lorsqu'une fourmi termine la construction de sa solution, elle laisse une trace sur le chemin emprunté. Cette trace est proportionnelle à la qualité de la solution construite. De plus, il est important de mettre en place un processus d'évaporation de la trace afin d'oublier les choix réalisés dans un lointain passé et de donner plus d'importance aux choix réalisés récemment. Soit ρ un paramètre d'évaporation choisi dans l'intervalle]0,1[. Soit A l'ensemble des fourmis, et soit $f(a)$ la valeur de la solution produite par la fourmi a. La qualité de la solution produite par la fourmi a est donc inversement proportionnelle à $f(a)$. La mise à jour de la trace sur la décision d est réalisée comme suit :

$$\tau_d = (1-\rho)\tau_d + c \sum_{a \in A} \Delta_d(a)$$

Où c est une constante et

$$\Delta_d(a) = \begin{cases} 1/f(a) & si\ la\ fourmis\ \boldsymbol{a}\ a\ réalisé\ le\ choix\ d \\ 0 & sinon \end{cases}$$

Voici le pseudo-code d'un système de fourmi et ses caractéristiques :

Algorithme 14 : Système de fourmis

1. Initialiser les traces τ_d à 0 pour toute décision possible d
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Construire |A| solutions en tenant compte de la force gloutonne et de la trace
4. Mettre à jour les traces τ_d ainsi que la meilleure solution rencontrée;
5. **fin du tant que**

Types d'Individus : solutions admissibles obtenues à l'aide d'un algorithme constructif

Type d'évolution : remplacement générationnel avec une population de taille constante

Structure de voisinage : population non structurée

Sources d'information : historique de la recherche (mémorisé dans la trace)

Irréalisabilité : l'algorithme constructif ne produit pas de solution non admissible

Intensification : aucune

Diversification : aucune.

On appelle Optimisation par Colonies de Fourmis (ACO pour Ant Colony Optimisation) toute variation ou extension du système de fourmis décrit ci-dessus.

Cet algorithme peut être amélioré de diverses manières.

- Le premier type d'extension consiste à réaliser une recherche locale sur chaque solution produite par l'algorithme constructif.

L'étape 3 et la caractéristique Intensification deviennent donc :

Construire |A| solutions en tenant compte de la force gloutonne et de la trace;

Appliquer une recherche locale sur chacune de ces solutions;

Intensification : recherche locale

- Une deuxième variation est de prendre un certain pourcentage de décisions en ne tenant compte que de la force gloutonne (les autres décisions étant prises en combinant la force gloutonne et la trace).
- Une troisième variation consiste à faire une mise à jour supplémentaire des traces en tenant compte des choix ayant abouti à la meilleure solution s^*rencontrée.

3.6.2.4 La méthode à mémoire adaptative

La méthode à mémoire adaptative est une extension de la recherche taboue qui permet de réaliser automatiquement une diversification et une intensification de la recherche. Cette méthode fonctionne avec une mémoire centrale chargée de stocker les composantes des meilleures solutions rencontrées. Ces composantes sont combinées afin de créer de nouvelles solutions. Si la combinaison ne produit pas une solution admissible, un processus de réparation est mis en œuvre. Un algorithme de recherche locale est ensuite appliqué et les composantes de la solution ainsi obtenues sont considérées pour éventuellement faire partie de la mémoire centrale.

Au début de la recherche, la mémoire centrale contient des composantes provenant de solutions très diverses et le processus de combinaison aura donc tendance à créer une diversité de nouvelles solutions. Plus la recherche avance et plus la mémoire centrale aura tendance à ne mémoriser que les composantes d'un sous-ensemble très restreint de solutions. La recherche devient donc petit à petit un processus d'intensification.

Voici le pseudo-code et les caractéristiques de la méthode à mémoire adaptative

Algorithme 15 : Méthode à mémoire adaptative

1. Générer un ensemble de solutions et introduire leurs composantes dans la mémoire centrale;
2. **tant qu**'aucun critère d'arrêt n'est satisfait **faire**
3. Combiner les composantes de la mémoire centrale pour créer de nouvelles solutions;
4. Réparer si nécessaire les solutions non admissibles ainsi générées;
5. Appliquer une recherche locale sur chaque nouvelle solution Admissible ;
6. Mettre à jour la mémoire centrale en ôtant certaines composantes et en y insérant de nouvelles provenant des nouvelles solutions générées à l'étape 5.
7. **fin du tant que**

Types d'Individus : composantes de solutions admissibles

Type d'évolution : remplacement stationnaire avec une population de taille constante

Structure de voisinage : population non structurée

Sources d'information : au moins deux parents

Irréalisabilité : les solutions non admissibles sont réparées

Intensification : recherche locale + Intensification implicite durant les dernières itérations

Diversification : Diversification implicite durant les premières itérations.

3.7 Conclusion

Dans ce chapitre nous avons étudiées les différentes classes des méthodes méta-heuristiques (à base d'une solution, à base du population de solution), ainsi que le principe des principales méthodes méta-heuristiques et les caractéristiques de chaque méthode, cette étude va nous permettre de choisir la méthode à utiliser pour résoudre le problème posé par les méthodes de décompositions structurelles complètes.

Dans le chapitre suivant nous présenterons la partie implémentation et expérimentation afin d'introduire une amélioration concernant la résolution des benchmarks reconnus de déférentes classes de problèmes.

Nous notons que la technique que nous allons intégrées dans notre travail est celle de la recherche taboue avec une méthode de décomposition hyper-arborescente pondérée.

Chapitre 4

Recherche taboue pour le calcul de la décomposition hyper-arborescente pondérée (Tabu Search for computing Weighted Hypertree Decomposition(TSWHD))

Nous avons introduit dans les chapitres un et deux un état de l'art sur les principales méthodes des résolutions, que ce soit à base énumératives, filtrages où décompositions structurelles et nous avons remarqués leurs complexités exponentielles en la taille du problème, cependant dans le chapitre trois, nous avons étudiés les principales méthodes méta-heuristiques, dont le but de choisir une pour l'intégrer dans notre implémentation afin d'améliorer le résultat de la résolution en un temps polynomiale. La méthode que nous avons choisi est celle dite « la recherche taboue ».

Dans ce chapitre nous allons présenter un nouveau algorithme appelé TSWHD (Tabu Search for weighted Hypertree Decomposition) basé sur le mécanisme de la recherche taboue qui génère en une première phase la décomposition hyper-arborescente pondérée puis il exploite cette décomposition afin de générer une solution pour un CSP donné. L'utilité de la décomposition hyper-arborescente pondérée est de transformer le CSP cyclique en autre acyclique équivalent pour garantir la résolution en un temps polynomiale, en exploitant avec le paramètre de la largeur de la décomposition hyper-arborescente, les autres paramètres quantitatifs du problème tels que le nombre de tuple inclut dans les relations(contraintes) des CSPs.

4.1 Algorithme de résolution d'un CSP

4.1.1 Pour un CSP binaire acyclique

Principe :

L'algorithme de la résolution arborescente (Tree-solving [1]) accepte en entrée un réseau de contraintes acyclique d'un CSP binaire. Chaque nœud de cet arbre, hormis la racine, possède un nœud père et peut avoir plusieurs nœuds fils. La première étape de cet algorithme est la construction d'un arbre enraciné orienté, tout en spécifiant un ordre entre les nœuds de telle sorte que chaque nœud père doit apparaître avant ses nœuds fils dans l'ordre (ligne 3). De ligne 5 à 10, l'algorithme traite chaque arc et la contrainte associée, ceci de nœuds feuilles aux nœud racine, en vue de réaliser un arc consistance orientée. Pour chaque arc orienté de X_i vers X_j , la procédure *Révise* est appelée pour enlever les valeurs du domaine de X_j qui n'ont pas de support (valeur consistante) dans le domaine de X_i. Après avoir traité tous les nœuds jusqu'à la racine, et si aucun domaine des variables n'est vide, l'algorithme Backtrack free est utilisé pour trouver une solution.

Algorithme 16 Algorithme Tree Solving

1: entrée : Un arbre de contrainte T d'un *CSP P = (X,D,C,R).*
2: sortie : Un Backtrack free sur l'ordre d spécifié.
3: Générer un ordre $d = X_1,\ldots, X_n$ tel que un nœud père précède toujours ses nœuds fils.
4: Soit $X_p(i)$ le nœud père du nœud X_i.
5: pour $i = n$ à 1 **faire**
6: *Révise* $(X_p(i),X_i)$
7: si le domaine de $X_p(i)$ est vide **alors**
8: Exit /*Pas de solution*/
9: fin si
10: fin pour
procédure Révise $(X_p(i),X_i)$
11: **pour** chaque valeur x du $dom(X_p(i))$ **faire**
12: **si** x n'a pas de support dans dom(Xi) **alors**
13: Supprimer x du $dom(X_p(i))$
14: **fin Si**
15: **fin pour**
16 : **fin procédure**
17 : **fin**

La complexité de Tree-solving est linéaire en le nombre de variables. Elle est de l'ordre de $O(nd^2)$, où n est le nombre de variables et d la taille des domaines des variables (la procédure Révise est limité par d^2 et elle est exécutée n fois).

4.1.2 Pour un CSP *n*-aire acyclique

Principe :

Etant donné un CSP acyclique et son arbre de jointure (*JoinTree*), l'algorithme de résolution acyclique (Acyclic solving) est capable de générer une solution quelque soit l'arité du problème. Dans une première étape, il traite l'arbre de jointure du bas vers le haut (4 à 11), et à chaque étape, il fait l'élimination des tuples de la relation père de la relation courante qui ne peuvent pas participer à une jointure avec les tuples de cette dernière, en appliquant une semi jointure (6). Si le CSP est inconsistant, une relation vide est créée dans l'un des nœuds de l'arbre de jointure, et l'algorithme retourne problème inconsistant (7).

Après la fin de cette première phase (bas-haut) (4 à 11), on aura une arc consistance dirigée (DAC, Directed Arc Consistancy) du nœud racine vers les nœuds feuilles. Dans une deuxième étape (12 à 14), Acyclic solving procède à la recherche d'une solution en parcourant l'arbre de jointure de la racine vers les feuilles (haut-bas) (13) et à chaque nœud, il prend un tuple consistant avec ceux déjà pris.

Algorithme 17 Acyclic Solving

1: **entrée** : Un *CSP* acyclique $P = (X,D,C,R)$, $R = \{R_1, \dots, R_t\}$,Un join-tree T de P.
2: **sortie** : Vérifier la consistance et générer une solution.
3: $d = \{R_1, \dots, R_t\}$ est un ordre tel que chaque relation apparaît avant ses fils dans l'arbre T enraciné en R_1.
4: **pour** j = t à 1 **faire**
5: **pour chaque** arête (j, k), $k < j$, dans l'arbre **faire**
6: $R_k \leftarrow Semi\_Join(R_k , R_j)$
7: **si** une relation vide est créée **alors**
8: Exit /*pas de solution pour le problème*/.
9: **fin si**
10: **fin pour**
11: **fin pour**
12: Retourne les relations mises à jour et la solution est cherchée comme suit, Sélectionner un tuple dans R_1
13: **pour** chaque relation R_i , $i = 1$ à t **faire**
14: Prendre un tuple qui est consistant avec toutes les relations qui la Précèdent $(R_1,\dots,R_{i-1})$
15: **fin pour**
fin.

La complexité de Acyclic solving est de l'ordre de $O(nmLog(m))$, avec n le nombre de relations et m le nombre de tuples par relation.

4.2 L'Algorithme TSWHD(Tabu Search for Weighted Hypertree Decomposion)

Principe:

Notre algorithme est constitué de deux étapes :

1-construction de la décomposition hyper-arborescente pondérée:

Dans cette étape nous basons sur l'idée de la recherche taboue. La recherche taboue [37] est une puissante technique méta-heuristique, qui a été utiliser avec succès dans des problèmes réel. L'idée basique de la recherche taboue est d'éviter le cycle (visité la même solution) pendant la recherche d'une solution en utilisant une structure de donnée qui s'appelle liste taboue. Dans la liste taboue des informations spécifiques sur l'historique de recherche (solution visitée) sont stockées pendant un nombre d'itération fixé. L'acceptante d'une solution (une contrainte dans notre cas) est basée essentiellement sur le nombre de tuple engendré par cette solution (contrainte).

Notre algorithme est équipé d'un processus itératif qui permet de parcourir l'ensemble de voisinage par rapport au contrainte de démarrage sélectionnée initialement d'une manière aléatoire par l'algorithme cette ensemble va constituer la racine de l'hyper-arborescence (solution initiale).

L'algorithme va prendre une contrainte de la solution initiale (initialisation de la liste taboue) et calcule le cout (pondération) de la jointure entre ces voisins, le meilleur voisin qui va être sélectionné est celui qui à le minimum de nombre de tuples engendrés (ligne 4, 5, 6). ce voisin va constituer le nœud suivant (relation père-fils). L'algorithme va prendre cette contrainte et génère d'une manière récursive les autres nœuds de l'hyper-arborescence .

2-résolution basé sur la décomposition hyper-arborescente pondérée :

Cette étape est constituée de deux étapes :

a-résolution de bas en haut :

dans cette étape l'algorithme commence par les nœuds feuilles en calculant pour chaque nœud les relations résultantes du jointures de toutes les contraintes dans chaque nœud puis il calcule les relations résultantes entre les pères et les fils en appliquant les semi-jointures et en gardant le résultat dans le nœud père ,en fait le même processus jusqu'au nœud racine.

b-résolution de haut en bas :

Dans cette étape l'algorithme applique le processus de calcul inverse entre fils et père en appliquant une jointure entre eux et garde le résultat dans le nœud fils (garde que les tuples qui en un support dans le nœud père « arc-consistance »).

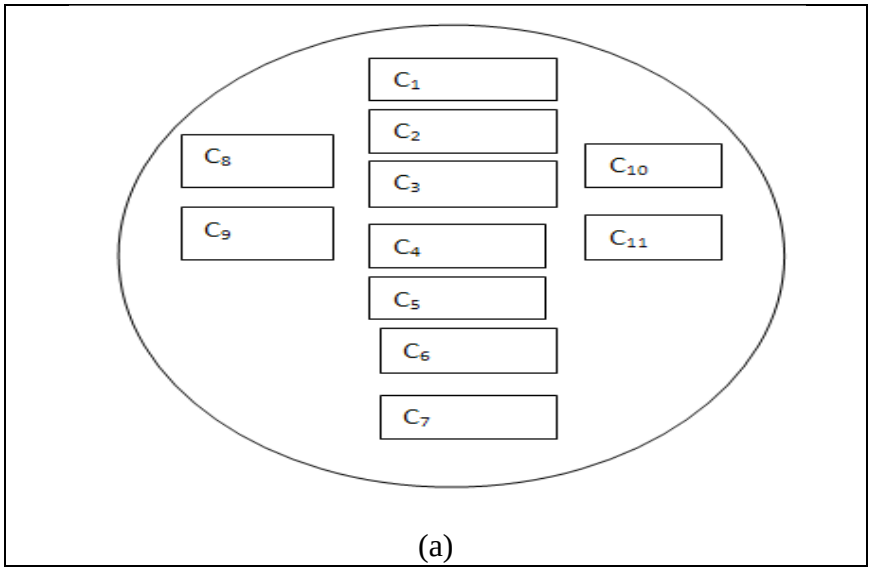
Note : les voisins d'une contrainte sont toutes les contraintes qui ont une intersection non vide avec cette contrainte.

Voici le pseudo-code de l'algorithme TSWHD :

Algorithme 18 : TSWHD

1 : **entree** : un Hypergraphe $\mathcal{H}$ associé à un CSP donné, Fonction de pondération $F^{\oplus,v,e}$.
/* v représente une fonction de jointure et e de semi- jointure.*/
2 : **sortie** : une solution $\mathcal{A}$ au CSP s'il existe, sinon afficher pas de solution
3 : **begin**
/* étape 1 : construction de la décomposition hyper- arborescente pondérée */
4 : - générer la solution initiale en basant sur la première contrainte
choisi d'une manière aléatoire.
5 : - accepte cette solution et construit son nœud racine
6 : - initialise la liste taboue
7 : **tant que** la condition de terminaison n'est pas vérifier **faire**
8 : - sélectionner un élément à partir de la solution initiale
9 : - générer ces voisins et évaluer chaque voisin avec son père
10 : - maitre à jours la liste taboue
11 : - sélectionner la contrainte qui génère le nombre de tuple minimum
12 : - construit un nouveau nœud incluant contrainte
13 : - prendre cette contrainte et faire le même travail récursivement pour
construire le reste des nœuds.
14 : **fin tant que**
/* Soit *HD* l'hyper-arborescence pondérée résultant de l'étape 1
15 : **tant que** la condition de terminaison n'est pas vérifier **faire**
/* étape 2.1 de résolution de bas en haut
16 : - appliquer une jointure pour tous les nœuds de HD
17 : - appliquer une semi-jointure entre chaque père et fils en Commençant de bas en haut
en stockant le résultat dans le nœud père jusqu'à l'arrivée au nœud racine.
/* étape 2.2 de résolution de haut en bas
18 : - appliquer une semi-jointure entre chaque fils est sont père et stocke le résultat dans
le nœud fils.
19 : **fin tant que**
/* soit $\mathcal{A}$ la solution et $C_1, \dots, C_n$ les contraintes qui contiennent
les résultats après l'étape de la résolution de haut en bas*/
20 : $\mathcal{A} :=$ sélectionner tous les tuples de $C_1, \dots, C_n$;
21 : **end**.

Exemples Illustratif :



(a)

Figure 4.1 (a)un problème CSP

En suppose qu'on a l'hypergraphe $\mathcal{H}$ qui représente le CSP au dessus, avec les informations de connexions décrit dans la Figure 4.2 :

	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}
C_1	x	x									
C_2	x	x									
C_3			x		x			x			
C_4		x		x		x					
C_5			x		x	x				x	
C_6			x	x	x	x					
C_7							x		x		x
C_8		x	x			x	x	x	x		
C_9							x	x	x		
C_{10}					x					x	x
C_{11}	x	x								x	x

Figure 4.2 : schéma de connexion de $\mathcal{H}$

Etape 1 (construction de l'hyper-arborescence) :

On suppose que le mouvement de la recherche taboue égal à 2 la mémoire utilisé et de court terme ;

1. On suppose que l'algorithme sélectionne C_5 aléatoirement dans l'étape initiale.
2. L'algorithme va chercher les voisins de C_5 (toutes les contraintes qui en une intersection non vide avec) soit C_{10}, C_3 et C_6 les voisins de C_5, alors l'ensemble de {C_5, C_{10}, C_3, C_6} vont construire la solution initiale, et en même temps la racine de l'hyper-arborescence puis initialise la liste taboue (enregistre cette solution comme taboue).
3. L'algorithme va chercher d'une manière itérative les voisins de C_{10}, C_3, Puis C_6 Soient {C_7, C_4, C_{11}} les voisins de C_{10}.

- une opération de calcule de pondération (nombre de tuple) est lancé entre {C_{10}, C_7},on suppose par exemple que, Cou_1=100 tuples, puis enregistre les comme taboue, puis calcule le pondération entre {C_{10},C_4} soit Cou_2=150 tuples et enregistre les comme taboue, pour la troisième contrainte en calcule la pondération entre {C_{10},C_{11}} soit Cou_3=50 tuples, l'algorithme va sortir C_7 de la liste (mouvement égal 2) et faire entrer C_{11} et comme on a pas d'autre voisin en choisi celui avec du cout minimum. Alors le voisin C_{11} qui va être choisi. Un nœud {C_{11}} connectant à sont père {C_{10} } est crée.
- Ce calcule va permettre l'affectation des contrainte sur des nœuds de l'hyper-arborescence on assurant un cout minimum .

4. L'algorithme va prendre C_{11} est refaire le processus depuis 2 jusqu'à ce que tous les contraintes soient affectées.

L'algorithme va générer la décomposition hyper-arborescente pondérée suivante :

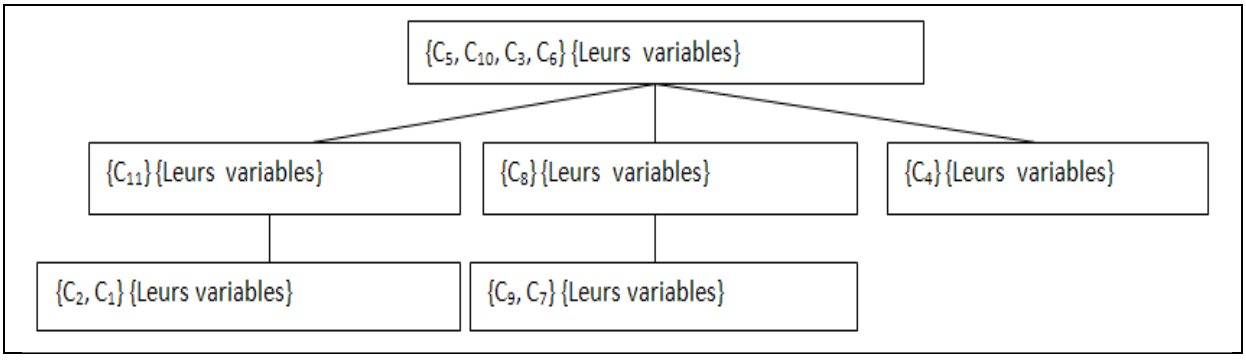


Figure 4.3 l'hyper-arborescente pondérée de problème $\mathcal{H}$

Etape 2 (résolution basé sur l'hyper-arborescence pondérée) :

a-Etape de bas en haut :

Nous calculons $R_1 = C_2 \bowtie C_1$ puis $R_2 = C_9 \bowtie C_7$ .en suite, nous appliquons une semi-jointure. Alors, $S_1 = C_{11} \ltimes R_1$ et $S_2 = C_8 \ltimes R_2$ .enfin, nous calculons $R_3 = C_5 \bowtie C_{10} \bowtie C_3 \bowtie C_6$, puis $S_3 = R_3 \ltimes S_1$, $S_3 = S_3 \ltimes S_2$ et $S_3 = S_3 \ltimes C_4$.

b-Etape de haut en bas :

Nous appliquons une semi-jointure dans l'ordre inverse. Alors nous avons :

$S_1 = S_1 \ltimes S_3$, $S_2 = S_2 \ltimes S_3$ et $C_4 = C_4 \ltimes S_3$,puis $R_1 = R_1 \ltimes S_1$ et $R_2 = R_2 \ltimes S_2$

alors la solution de CSP est l'ensemble des tuples dans $\{S_3, S_2, S_1, C_4, R_1, R_2\}$.

Détail de L'Algorithme TSWHD

1 : entrée : un hypergraph $\mathcal{H}$(V,E),une fonction de pondération $F^{\oplus, v_{\mathcal{H}}, e_{\mathcal{H}}}$
2 : sortie : une solution au problème représenté par $\mathcal{H}$,sinon afficher pas de solution .
3 : var Contraintes, Variable,Voisins, Sous_Voisin, Solutions_Initial**:**Tableau de Chaine;
4 : Nœud_Accepté,Pere : Nœud ;
5 : Les_Noeuds_Acceptés : Tableau des Nœuds ;
6 : debut
7: Contraintes =selectionner_ contraintes ($\mathcal{H}$) ;
8: Variable= selectionner_Variable($\mathcal{H}$) ;
9 : /* choisir une contrainte aléatoirement parmis l'ensemble total des contraintes */
10 : A= choisir_ Aléatoirement (Contraintes) ;
11 : Solutions_Initial=choisir_Voisin(A, Contraintes) ;
12 : Nœud_Accepté =créer_Noeud(A∪Solutions_Initial) ;
13 : Père= Nœud_Accepté ;
14 : Ajouter(Nœud_Accepté , Les_Noeuds_Acceptés) ;
15: Contraintes = Contraintes -{A ∪Solutions_Initial};
16 : tant que Contraintes!=ϕ **faire**
17 : pour chaque Sol_i de Solutions_Initial **fair**
18 : Voisins=choisir_Voisin(Sol_i, Contraintes) ;
19 : Nœud_Sol_i=créer_Noeud(Sol_i) ;
20: Tabu_List=créer_Liste(Voisins.length) ;
21 : /* accepter l'element0 choisi aléatoirement et ajouter le comme taboue*/

22 : element0=choisir_ Aléatoirement(Voisins);
23 : Nœud_Optimum =créer_Noeud(element0) ;
24 : Position=choisir_Position(element0,Voisins) ;
25 : Cout_Opt=Pondérer_(Nœud_Sol_i, Nœud_Optimum) ;
26 : Ajouter_Tabu(element0, Cout_Opt) ;
27 : Itr=1 ;
28 : tant que itr < NBR_Itération_définit (*soit dans notre cas = à voisin.length) **fair**
29 : Element_T=choisir_Aléatoirement(Voisins) ;
30 : si (Element_T n'est pas Tabu) **alors**
31 : Nœud_Tempo=créer_Noeud(Element_T) ;

```
32 :            Cout=Pondérer_( Nœud_Sol_i, Nœud_Tempo) ;
33 :              si (Cout_Opt>Cout)  alors
34 :                Cout_Opt=Cout ;
35 :                Position=chercher_Position(Element_T,Voisin) ;
36 :              fin si
37 :          Ajouter_Tabu(Element_T,Cout) ;
38 :          Maitre_Ajour_ListTabu ;      /*faire sortir l'ancien élément */
39 :      sinon
40 :        - Appliquer critère d'espérance et retourner un élément Taboue ;
41 :   fin tant que ;
42 :    Nœud_Optimum = créer_Noeud(Voisin[Position] ) ;
43 :    Nœud_Optimum .Pere=Pere;
44 :    Ajouter(Nœud_Optimum  ,Les_Noeuds_Acceptés ) ;
45 :    Contraintes = Contraintes -{ Voisin[Position] };
46 :     /*donner toujours la priorité de recherché au element optimum*/
47 :    Voisin_T=choisir_Voisin(Voisin[Position] , Contraintes);
48 :       pour  chaque  Element de Voisin_T   faire
49 :        Recursive_ Affectation (Element, Contraintes);
50 :       fin pour ;
51 :   fin tant que ;
52 :         i:=0 ;
53 :        répéter
54 :          Resoudre (Les_Noeuds_Acceptés [i]) ;
55 :               i=i+1 ;
56 :         jusqu'à ce que tout les Nœuds  soient Résolus ;
       fin.


 procédure  Recursive_ Affectation (element0 ∈ Contraintes0, Contraintes0⊆ Contraintes)
1 :    var   Voisins, Sous_Voisin ,Solutions:Tabeau de Chaine;
2 :    Nœud_Accepté :Node;
3 :   tant que   Contraintes0!=ϕ    faire
4 :     Solutions =choisir_Voisin(element0, Contraintes0) ;
5 :     Nœud_Accepté =créer_Noeud(element0∪Solutions) ;
6 :     Nœud_Accepté .Pere=Pere;
7 :     Pere= Nœud_Accepté ;
8 :     Ajouter(Nœud_Accepté , Les_Noeuds_Acceptés ) ;
9:      Contraintes0= Contraintes0-{ element0 ∪Solutions };
10 :    tant que  Contraintes0 !=ϕ  faire
11 :      pour  chaque  Sol_i de Solutions  fair
12 :       Nœud_Sol_i= créer_Noeud(Sol_i) ;
13 :       Voisins=chercher_Voisin(Sol_i, Contraintes0) ;
14 :      Tabu_List= créer_Liste(Voisins.length) ;
15 :      /*accepter l'element0  choisi aléatoirement et ajouter le comme taboue*/
16 :     element0= choisir_ Aléatoirement (Voisins);
17 :     Nœud_Tempo =créer_Noeud(element0) ;
18 :     Position=chercher_Position(element0,Voisins) ;
19 :    Cout_Opt=Pondérer_( Nœud_Sol_i, Nœud_Tempo) ;
20 :    Ajouter_Tabu(element0,Cout) ;
21 :    Itr=1 ;
22 :   tant que   itr < NBR_Itération_définit   fair (*soit dans notre cas = à  voisin.length*)
```

```
23 :      Element_T=choisir_Aléatoirement(Voisins) ;
24 :      si (Element_T n'est pas Tabu  ) alors
25 :        Nœud_Tempo= créer_Noeud(Element_T) ;
26 :        Cout=Pondérer_( Nœud_Sol_i, Nœud_Tempo) ;
27 :       si (Cout_Opt>Cout)  alors
28 :         Cout_Opt=Cout ;
29 :         Position=chercher_Position(Element_T,Voisin) ;
30 :       fin si
31 :         Ajouter_Tabu(Element_T,Cout) ;
32 :         Maitre_Ajour_ListTabu ;    /*faire sortir l'ancien élément
33 :     sinon
34 :      Appliquer critère d'espérance et retourner un élément Taboue
35 :  fin tant que ;
36 :   Nœud_Optimum = créer_Noeud(Voisin[Position] ) ;
37 :   Nœud_Optimum .Pere=Pere;
38 :  Ajouter(Nœud_Optimum  ,Les_Noeuds_Acceptés ) ;
39 :  Contraintes0= Contraintes0-{ Voisin[Position] };
40 :   /*donner toujours la priorité de recherché au élément optimum;
41 :    Voisin_T=choisir_Voisin(Voisin[Position] , Contraintes0);
42 :     pour  Chaque  Elm de Voisin_T   faire
43 :        Recursive_Affectation(Voisin[Position] , Contraintes0);
44 :     fin pour ;
45 :    fin pour ;
46 :   fin tant que ;
48 : fin tant que;
     fin procédure ;

procédure  Résoudre(Node[ ] Nodes) :longue Entier ;
1 :  debut

        /* résolution de bas en haut jusqu'au nœud père
        /* on commence par les feuilles
2 :       pour chaque  Node  N  de Nodes  faire
3 :            si !( N.à des Fils) alors
                N.Weight=Pondérer(N,null)
4 :        fin pour

5 :    tant que (Nodes [0].Weight==0) faire
6 :         pour chaque  Node Np de Nodes  faire
7 :            si  (Np.à des Fils)alors
8 :             si Tout_Les_Fils_Sont_Resolu(Np) alors
9:                pour (i=0 ;i< Np.Fils.Taille ;i++) faire
10 :                   Np.Weight=Pondérer(Np, Np.Fils[ i ])
11:               fin pour
12 :            fin si
13 :          fin si
14 :       fin pour
15 :   fin tant que

         /* résolution de haut en bas
15 :   Solution=false ;j=0
```

```
16 :  tant que ( !solution) faire
17 :     Node père=Nodes[0]
18 :     label1 : pour (i=0 ;i< père.Fils.Taille ;i++) faire
19 :                 Fils[i].Weight= Pondérer(Fils[i],père)
20 :                 File_attente[j]= Fils[i]
21 :                 j=j+1
22 :               fin pour
23 :                si File_attente == null alors
24 :                   Solution=true
25 :               sinon
                       /* affecter le 1er élément et décaler la file
26 :                  père= File_attente[0] ;
27 :                  Décaler(File_attente);
28 :                go to label1
29:               fin si
30 :       fin tant que
          fin procédure ;


 fonction Pondérer(Node n1,Node n2) :Longue Entier ;
1 :  var     Resultat :Longue Entier ;
2 :  debut
3 :       si n2 !=null  alors
4 :          Resultat = e_H(n1,n2);
5:        sinon
6 :          Resultat =v_H(n1) ;
7 :       fin si ;
    fin fonction ;

  /*la jointure est appliquée entre toutes les relations de même nœud

fonction v_H (Node n) : Longue Entier ;
     var Résultat : Longue Entier ;
1 :  début
       /* en note une contrainte par une relation
2 :       pour (i=0 ;i<n.Relations.Taille ;i++) faire
3 :          n. Relations[0]= n. Relations[0] ⋈ n. Relations[i]
4 :        fin pour
5 :       Résultat = n. Relations[0].Taille
fin fonction

   /*la semi-jointure est appliquée après la jointure

fonction e_H (Node n1,Node n2) : Longue Entier ;
     var résultat : Longue Entier ;
1 :  début
2 :     n1.Relations[0]= n1.Relations[0] ⋉ n2.Relations[0]
3 :     résultat =n1. Relations[0].Taille
 fin fonction
```

4.3 La complexité de TSWHD

En a deux étapes dans notre calcule :

En suppose qu'on a N contraintes (C_1,…,C_n) dans notre hypergraphe et r_i est le nombre de tuple de chaque contrainte C_i.

1-construction de la décomposition hyper-arborescente pondérée:

Au démarrage on a une contrainte sélectionnée aléatoirement, alors :

Pour le Noeud_0 -------> (N-1) parcours, et K_0 voisins acceptés.

Pour le Noeud _1 -------> (N-K_0) parcours, et K_1 voisins acceptés.

Pour le Noeud _2 -------> (N- K_0-K_1) parcours, et K_2 voisins acceptés.

⋮

Pour le Noeud _e-1-------> (N-(K_0+K_1+…+K_{n-1})) parcours, p voisins acceptés .

Alors la construction de la table des voisins va couter :

$N^2 – (1+(n-1)K_0+…+K_{n-1}) <= N^2 \Rightarrow O(N^2)$.

La construction de la décomposition hyper-arborescente pondérée va couter :

$O(r^{h*p})$ où h la largeur de la décomposition hyper arborescente , r le nombre de tuples dans la plus grande relation et p nombre de nœud de l'hyper-arborescence. Alors si on mais :

Alors la complexité de cette phase est $O(N^2 + r^{h*p})$.

2-Résolution basé sur la décomposition hyper-arborescente pondérée:

La parcourt de bas en haut va couter au pire des cas pr^h ,et de haut en bas presque le même cout.

Alors la complexité de cette phase $O(2\, r^{h*p})$

Alors la complexité de l'algorithme est $O(N^2 + r^{h*p} + 2\, r^{h*p})$.

Soit S la taille de CSP et r la taille de la grande relation de CSP. Alors La complexité spatiale est $O(S.|C|.r)$ où $|C|$ nombre de contrainte de CSP.

4.4 Expérimentation

4.4.1 Principe de la méthode de décomposition hyper-arborescente issue de Bucket Elimination (BE)

Bucket Elimination [43] est une méthode issue du domaine de satisfaction de contrainte. Cette méthode utilise la structure topologique de problème pour aider à trouver une solution efficacement. En induit une largeur minimale issue de son graphe primal.la propriété de cette méthode est qu'elle impose un ordre optimal de ces variables en entrée.

En suppose un ordre $x_1, \dots, x_n$ de variables de CSP. BE commence par créer un seau pour chaque variable x_i. Pour chaque contrainte $r_i(x_{i_1}, \dots, x_{i_k})$ de problème, nous plaçons la r_i avec ces variables dans le seau $\max\{i_1, \dots, i_k\}$, ensuite, nous itérant sur i de n à 1, eliminant chaque seau. Dans l'itération i, nous trouvons dans le seau i plusieurs relations (contraintes), où x_i est la variable commune entre toutes ces relations. Nous calculons leur jointure, et retirant la colonne x_i, soit r'_i le résultat. Si r'_i est vide, alors le résultat de CSP est vide aussi. Sinon, soit j l'index de valeur maximal inferieur de i tel que x_j est un variable de r'_i nous déplaçons r'_i au seau j.si aucune relations retourne un vide alors CSP est résolu.

4.4.2 Considérations expérimentales

Dans cette partie en va présenter les résultats obtenus par notre méthode décrit au dessus. Le test est fait sur des déférents problèmes téléchargés depuis [40].cette collection de problèmes contiennent des représentations des hypergraphes des déférentes classes des instances de CSPs. elles sont représentées sous le format XCSP 2.1 (format étendu pour la représentation des réseaux de contraintes basé sur le langage XML[41]) . Notre programme utilise un fichier XCSP 2.1 de type extension en entrées et génère un fichier GML(Graph Modeling Language File Foramt [42]) en sortie pour l'approche résolution, et un fichier de format GML en entrée dans le cas d'analyse des autres méthodes. Dans notre cas nous allons analyser la méthode BE (en utilisant un fichier GML généré par le programme BE[43]).

Toutes les expérimentations de notre méthode sont exécutées sur un système d'exploitation Windows xp pro, une machine avec Intel Pentium® 4CPU 3.00 GHz, 1.00 Go de RAM. Le temps maximum pour chaque exécution est de 500 secondes.

HW : est la largeur de la décomposition hyper-arborescente pondérée, DT : Temps de construction de la décomposition hyper-arborescente pondérée, ST : Temps de résolution, TT : Temps total

Les résultats sont présentés dans la table Table.1.

Table.2 : Résultats expérimentaux de TSWHD && BE

	Benchmarcks (E/V)	BE				TSWHD			
		HW	DT	ST	TT	HW	DT	ST	TT
Domino	domino-100-100_ext (100/100)	2	0	1.111	1.111	3	0.111	0.999	1.11
	domino-100-200_ext (100/100)	2	0	3.109	3.109	3	0.096	3.061	3.157
	domino-100-300_ext (100/100)	2	0	6.001	6.001	3	0.095	5.984	6.079
Prêt	pret-60-25_ext (40/60)	5	0	0.218	0.218	4	0.049	0.39	0.439
	pret-60-40_ext (40/60)	5	0	66.626	66.626	4	0.001	89.328	89.329
Renault	renault_ext (134/101)	3	0.01	?	?	?	?	?	?
Pigeons	pigeons-5_ext (10/5)	3	0	0.048	0.048	7	0.001	0.016	0.017
	pigeons-6_ext (15/6)	3	0	0.032	0.032	9	0.001	0.015	0.016
	pigeons-7_ext (21/7)	4	0	0.062	0.062	11	0.001	0.031	0.032
Coloring	dsjc-125-1-4-ext (736/125)	35	0.09	19.64	19.73	28	18.61	0.124	18.735
	dsjc-125-1-5-ext (736/125)	35	0.09	18.58	18.67	28	19.19	0.158	19.344
	geom-30a-3-ext (81/30)	4	0	0.313	0.313	11	0.063	0.031	0.094
	geom-30a-4-ext (81/30)	4	0	0.172	0.172	14	0.001	0.063	0.064
	geom-30a-5-ext (81/30)	4	0	0.095	0.095	18	0.001	0.062	0.063
	geom-30a-6-ext (81/30)	4	0	0.095	0.095	14	0.001	0.062	0.063
	myciel-5g-3-ext(236/47)	12	0.1	0.879	0.979	31	0.752	0.14	0.892
	myciel-5g-4-ext(236/47)	12	0.02	0.814	0.834	31	0.752	0.046	0.798
	myciel-5g-5-ext(236/47)	12	0.01	0.751	0.761	25	0.595	0.03	0.625
	myciel-5g-6-ext(236/47)	12	0	0.798	0.798	27	0.643	0.061	0.704
	queens-5-5-3-ext (160/25)	10	0.01	0.313	0.323	23	0.19	0.014	0.204
	queens-5-5-4-ext (160/25)	10	0.02	0.297	0.317	25	0.205	0.015	0.22
	queens-5-5-5-ext (160/25)	10	0.01	0.282	0.292	23	0.206	0.014	0.22
Mug	mug100-1-3_ext (166/100)	3	0	0.438	0.438	6	0.502	0.139	0.641
	mug100-1-4_ext (166/100)	3	0	0.437	0.437	6	0.47	0.046	0.516
	mug100-25-3_ext (166/100)	3	0.01	0.501	0.511	6	0.377	0.031	0.408
	mug100-25-4_ext (166/100)	3	0.01	0.5	0.51	6	0.283	0.156	0.439
	mug88-1-3_ext (146/88)	3	0.01	0.361	0.371	6	0.252	0.03	0.282
	mug88-1-4_ext (146/88)	3	0.01	0.36	0.37	6	0.237	0.014	0.251
	mug88-25-3_ext (146/88)	3	0	0.329	0.329	6	0.268	0.092	0.36
	mug88-25-4_ext (146/88)	3	0	0.36	0.36	6	0.205	0.186	0.391

4.4.3 Analyse des résultats

La méthode BE (Bucket Elimination[43]) est connue comme une meilleur méthode parmi ceux du domaine de décomposions structurelle qui produit une décomposition hyper-arborescente de largeur minimum en basant que sur la structure de problème, donc elle ne donne aucune importance aux informations quantitatives tels que le nombre de tuples malgré que ces informations sont très importantes dans le processus de résolution. Ce qui est prouvé dans nos expérimentations. Notre méthode a donnée des bonnes résultats par rapport au BE dans presque tous les problèmes (voir Table-1) parce qu'elle ordonne l'exécution en profitant de la taille de chaque relation (nombre de tuple) et elle minimise le nombre d'opérations « semi jointure » dans sa décomposition hyper-arborescente pondérée afin de générer une solution dans un temps optimale par rapport au BE.

Notre résolveur a trouvé un problème d'espace mémoire avec le fichier « Renault » due aux grand nombre de tuples générés par les opérations des jointures. Nous pouvons résumer les résultats par le graphe de comparaison entre les deux méthodes dans la figure 4.4

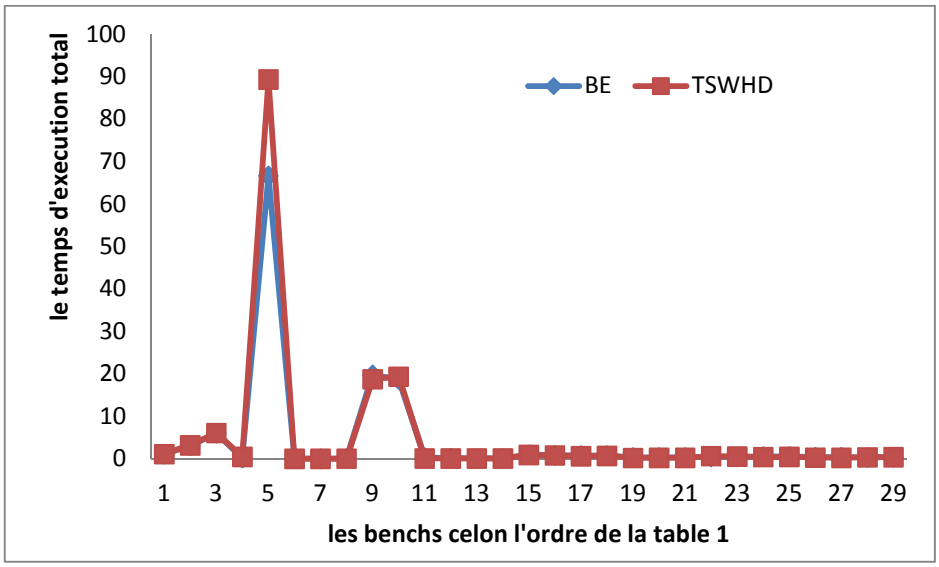


Figure 4.4 Graphe de comparaison entre BE et TSWHD

4.5 Conclusion

Dans ce chapitre, nous avons présentés notre nouvelle méthode dite TSWHD. Cette méthode applique une hybridation entre une méthode méta-heuristique et structurelle pour résoudre problème (CSP) en prend en considération leur informations quantitatives affectées à leur variables. Cette méthode est constituée de deux phases « la phase de construction de la décomposition hyper-arborescente pondérée et la phase résolution en exploitant cette décomposition » .Nous avons évaluées notre méthode avec 30 problèmes (voir Table.1) issue de monde industrielle et littérature.

D'après les résultats de la table.1 , nous pouvons dire qu'une bonne décomposition hyper-arborescente peut assurer un temps de résolution optimal, et que l'optimalité d'une décomposition hyper-arborescente est basée sur les informations quantitatives, qui est n'est pas le cas dans la méthode Bucket Elimination, qui tient en compte uniquement la largeur de la décomposition hyper-arborescente.

En conclusion, par comparaison de BE(méthode structurelle) et TSWHD(méthode hybride) nous pouvons valider notre attitude sur la nécessité d'inclure d'autres paramètres avec la largeur afin d'avoir une décomposition hyper-arborescente qui génère une solution en un temps optimal.

Conclusion générale et perspectives

Les méta-heuristiques avec les méthodes de décompositions structurelles pondérées peuvent constituer une classe de méthodes approchées adaptables à un très grand nombre de problèmes combinatoires et de problèmes d'affectation sous contraintes. Elles ont révélé leur grande efficacité pour fournir des solutions approchées de bonne qualité pour un grand nombre de problèmes d'optimisation classiques et d'applications réelles.

Afin de résoudre des instances de taille et de difficulté croissantes, il faut mettre au point des méthodes toujours plus puissantes. Pour atteindre cet objectif, au moins deux voies privilégiées doivent être parcouru : l'hybridation des méthodes et la distributivité. Dans notre travail nous avons exploités la voie d'hybridation qu'il a donnée des résultats prometteuses sauf pour les problèmes qui ont de grande nombre de tuples dans leurs relations(contraintes),c'est pour cela nous percevrons d'exploiter une nouvelle voie basée sur la distributivité afin d'implémenter un résolveur hybride sur un système distribué.

Bibliographie

[1] Rina Dechter ,« *constraint processing* » edtion2003.

[2] S. Golumb and L. Baumert, *Backtrack programming*, Journal of the ACM, pages 516-524 (1965).

[3] E.C Freuder, *A sufficient condition for backtrack free search*, Journal of the ACM Vol29 N°1 (1982).

[4] Roman Bartak « *User's Guide Constraint Programming* ».

[5] J. Gaschnig, *Performance measurement and analysis of certain search algorithms*,Technical Report CMU-CS-79-124, Carnegie-Mellon University, (1979).

[6] R. Dechter and D. Frost, *Backjump-based backtracking for constraint satisfaction problems*, Artificial Intelligence 136 :147-188 (2002).

[7] T. Schiex and G. Verfaillie, *Nogood recording for static and dynamic constraint satisfaction problems*, Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence (1993).

[8] R. J. Bayardo and D. P. Miranker, *A complexity analysis of space-bounded learning algorithms for the constraints satisfaction problem*, Proceedings of 13th National Conference on Artificial Intelligence, pages 298-304, (1996).

[9] M. Ginsberg, *Dynamic backtracking*, Journal of Artificial Intelligence Research,1 :25-46 (1993).

[10] K. Mackworth, *Consistency in networks of relations*, Artificial intelligence 8 :99-118(1977).

[11] R. Mohr and T. C. Henderson, *Arc and path consistency revisited*, Artificial Intelligence 28 :225-233 (1986).

[12] Y. Deville P. V. Hentenryck and C. M. Teng, *A generic arc-consistency algorithm and its specializations*, Artificial Intelligence 57 :291-321 (1992).

[13] C. Bessière, *Arc-consistency and arc-consistency again*, Artificial Intelligence 65 :179-190 (1994).

[14] C. Beeri, R. Fagin, D. Maier, M. Yannakakis, On the desirability of acyclic database schemes, J. ACM 30 (3) (1983) 479–513.

[15] R. Haralick and G. Elliot, *Increasing tree search efficiency for constraint satisfaction problems*, Artificial Intelligence, 14 :263-313, (1980).

[16] E.C. Freuder C. Bessiere, P. Meseguer and J. Larrosa, *On forward checking for non binary constraints satisfaction problems*, Artificial Intelligence (2002).

[17] E.C. Freuder C. Bessiere, P. Meseguer and J. Larrosa, *On forward checking for non binary constraints satisfaction problems*, Artificial Intelligence (2002).

[18] N. Sadeh M. S. Fox and C. Baykan, *Constrained heuristic search*, In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, pages 309-315, Detroit, MI, (1989).

[19] R. Dechter, *A constraint-network approach to truth-maintenance*, Technical Report 870009 (R-80), UCLA Cognitive Systems Laboratory (1987).

[20] I. Meiri R. Dechter and J. Pearl., *Temporal constraint networks*, Artif. Intell., 49(1-3) :61-95 (1991).

[21] U.Montanari, *Networks of constraintes : Fondamental properties and application to picture*, Proc. of Information Sciences (1974).

[22] Thomas H. Cormen,Charles E. Leiserson,Ronald L. Rivest,Clifford Stein *Introduction to Algorithms Second Edition.*

[23]P.A. Bernstein, N. Goodman, *The power of natural semijoins*, SIAM J. Comput. 10 (4) (1981) 751–771.

[24] D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1986.

[25] G. Gottlob, N. Leone, F. Scarcello, *The complexity of acyclic conjunctive queries*, J. ACM 48 (3) (2001) 431–498.

[26] H.L. Bodlaender. A tourist guide through treewidth. Acta Cybernetica, 11:1–21, 1993.

[27] N. Robertson and P. D. Seymour. *Graph minors. II. algorithmic aspects of tree-width.* Journal Algorithms,7:309–322, 1986.

[28] G. Gottlob, N. Leone, F. Scarcello, *Hypertree decompositions and tractable queries*, J. Comput. System Sci. 64 (3) (2002) 579–627.

[29] Rina Dechter and Judea Pearl. *Tree clustering for constraint networks*. AI, 38(3):353–366, 1989.

[30] G. Gottlob, N. Leone, F. Scarcello, *A comparison of structural CSP decomposition methods*, Artificial Intelligence 124 (2) (2000) 243–282.

[31] P.G. Jeavons, D.A. Cohen, and M. Gyssens. *A structural decomposition for hypergraphs*.Contemporary Mathematics, 178:161–177, 1994.

[32] A.K. Chandra, P.M. Merlin, *Optimal implementation of conjunctive queries in relational databases*, in: Proc. of the STOC '77, 1977, pp. 77–90.

[33] R. Dechter, *Constraint networks*, in: Stuart C. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, vol. 1, second ed., Wiley, 1992, pp. 276–285.

[34] J. Pearson, P.G. Jeavons, *A survey of tractable constraint satisfaction problems*, CSD-TR-97-15, Royal Holloway, Univ. of London, 1997.

[35] Francesco Scarcello a,∗, Gianluigi Greco b, Nicola Leone , *Weighted hypertree decompositions and optimal query plans.*

[36] *Tractable structures for constraint satisfaction problems*, Chapitre dans "Handbook of constraint programming", Elsevier (2006).

[37] Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[38] Günther Zäpfel, Michael Bögl,Roland Braune , *Metaheuristic Search Concepts - A Tutorial with Applications to Production and Logistics.*

[40] http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html#

[41] O. C. *of the Third International Competition of CSP Solvers, Xml representation of constraint networks format xcsp 2.1*, Disponible sur http ://www.cril.univartois. fr/CPAI08/XCSP2-1.pdf, dernière mise à jour 15 janvier 2008.

[42] http://www.uni-passau.de/Graphlet/GML.

[43] Artan Dermaku Tobias Ganzow Georg Gottlob Benjamin McMahan Nysret Musliu Marko Samer, *Heuristic Methods for Hypertree Decompositions DBAI-TR-2005-53.*

[44] C.H. Papadimitriou, K. Steiglitz, *Combinatorial optimization – algorithms and complexity. Prentice Hall, 1982.*

[45] C.C. Ribeiro, N. Maculan (Eds.), *Applications of combinatorial optimization.Annals of Operations Research 50, 1994.*

[46] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness, W.H. Freeman and Company, New York, 1979.*

[47] E.H.L. Aarts, J.K. Lenstra (Eds.*), Local search in combinatorial optimization, John Wiley & Sons, 1997.*

[48] C.R. Reeves (Ed.*) Modern heuristic techniques for combinatorial problems,Blackwell Scientific Publications, Oxford, 1993.*

[49] P. Jégou and C. Terrioux. *Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence,146:43–75, 2003.*

[50] P.Jégou,C.Terrioux, *Combined Strategies for Decomposition-Based Methods for Solving CSPs.*

[51] Martin J. Green, *New Methods for the Tractability of Constraint Satisfaction Problems PhD Thesis.*

Abstract

The *CSP* (constraints satisfaction problems) make it possible to model and solve many problems of the real-world. However, they are known to be NP -complete problems. Their theoretical complexity is exponential in the size of the problem. However, it is well-known that the *CSP* having a cyclic structure can be solved in polynomial time using their equivalent acyclic structure using the techniques of structural decompositions. These methods gather variables or constraints in clusters so that the *CSP* obtained has a structure of tree. Among these method, the most general method is the method known as "*Hypertree Decomposition*" where the quality of the decomposition is based only over the width of the decomposition .However, it exists others parameters which can led to a better decomposition, thing which has led to introduce a new method called "weighted hypertree decomposition" which deals with this parameters. However, the algorithm presented for the calculation of this method has an exponential complexity. For that, we propose in this work a new algorithm "*Tabu Search for Weighted Hypertree Decomposition*" based on a metha-heuristics method (tabu search). This algorithm is tested and evaluated on various problems known in the literature and the industry.

Keywords: CSP, Weighted hypertree decomposition, Meta-heuristics.

Résumé :

Les problèmes de satisfaction de contraintes (*CSP*) permettent de modéliser et de résoudre beaucoup de problèmes du monde réel. Cependant, ils sont connus pour être des problèmes NP complets. Leur complexité théorique est exponentielle en la taille du problème. Cependant, il est bien connu que les *CSP* ayant une structure acyclique en déterminant des sous classes parmi ces *CSP* cycliques traitables de manière polynomiale. Les techniques de décompositions structurelles permettent de déterminer des sous classes traitables en temps polynomial. Ces méthodes regroupent des variables ou des contraintes en des clusters de telle sorte que le *CSP* obtenu ait une structure d'arbre. Parmi ces méthode, la méthode la plus générale est la méthode dite « hypertree decomposition » où la qualité de la décomposition est basé uniquement sur la largeur de la décomposition hors qu'il y a d'autre paramètre qui peuvent guider à une meilleur décomposition, chose qui a permet à introduire une nouvelle méthode appelé « weighted hypertree decomposition ».Cependant, l'algorithme présenté pour le calcul de cette méthode présente une complexité exponentielle. Pour cela, nous proposant un nouvel algorithme « Tabu Search for Weighted Hypertree Decomposition » basé sur une méthode méta-heuristique (recherche taboue). Cet algorithme est testé et évalué sur des différents problèmes connus dans la littérature et l'industrie.

Mots clés : CSP, weighted hypertree decomposition, méta-heuristiques.

ملخص :

المشاكل الخاضعة للمعايير(م.خ.م) تسمح بقولبة و حل كثير من المشاكل المعاشة في واقعنا.في هذه الأثناء,يعتبر حل هذه المشاكل جد صعب و تعقيدها النظري يتناسب أسيا مع حجم المشكل لهذا تحويل ال م خ م الدائري إلى أخر لا دائري يسمح بحله في وقت ذو عبارة كثير حدود. من اجل ذلك لجانا إلى تقسيمات هيكلية تسمح بتعيين أقسام جزئية تعالج في المشكل في وقت ذو كثير حدود, هذه الطرائق تجمع المتغيرات و المعايير في فئات ذات شكل شجري. من بين هذه الطرائق, نجد الطريقة الأكثر شمولية " تقسيمة على شكل متعدد الأشجار" , وفيه كفاءة التقسيمة تعتمد على عرض التقسيمة رغم انه يوجد وسائط أخرى مهمة يجب أخذها بعين الاعتبار لتحسين كفاءة التقسيمة أو ما يسمى " تقسيمة على شكل متعدد الأشجار المقدرة " . لكن الخوارزمية المقدمة لذلك تعد ذات تعقيدة أسية. لهذا نقدم خوارزمية جديدة ذات الاسم " بحث متأصل عن تقسيمة على شكل متعدد الأشجار المقدرة ". هذه الخوارزمية مجربة و مقيمة مع مشاكل معروفة في المجال العلمي و الصناعي.

الكلمات الأساسية : CSP, weighted hypertree decomposition, méta-heuristiques.