REPUBLIQUE DU CAMEROUN
REPUBLIC OF CAMEROON
Peace-Work-Fatherland
****
UNIVERSITE DE DSCHANG
UNIVERSITY OF DSCHANG
Scholae Thesaurus Dschangensis Ibi Cordum
BP 96, Dschang (Cameroun)
Tél. /Fax (237) 233 45 13 81
Website : http://www.univ-dschang.org.
E-mail : udsrectorat@univ-dschang.org

FACULTE DES SCIENCES
FACULTY OF SCIENCE
Département de Mathématiques et Informatique
Department of mathematics and Computer Science
BP 96, DSc hang (Cameroun)
Tél. /Fax (237) 233 45 13 81
Website : http://fs.univ-dschang.org.
E-mail : dept.math-info@univ-dschang.org

RAPPORT DE RECHERCHE OPERATIONNELLE

Noms et prénoms des membres du groupe	Matricules	SPECIALITE
FOAM TOUKAM CINDY LENA	CM-UDS-18SCI0092	IA
KENGNE WAMBO DARIL RAOUL	CM-UDS-18SCI0131	IA
NGANFANG VICTOIRE CABRELLE	CM-UDS-18SCI1828	IA
TEMFACK DERICK	CM-UDS-18SCI0797	IA

Enseignant : Dr. Soh Mathurin

Année académique
2021/2022

SOMMAIRE

Table des figures

Liste des tableaux

INTRODUCTION

La recherche opérationnelle est la discipline des méthodes scientifiques utilisable pour l'élaboration des meilleures décisions. Elle peut encore être définie comme un ensemble de méthodes et techniques orientées vers la recherche du meilleur choix dans la façon d'opérer en vue d'aboutir au résultat visé ou meilleur résultat possible. La recherche opérationnelle fait donc usage d'un ensemble de méthodes mathématiques, d'algorithmes et d'outils statistiques pour obtenir des solutions optimales aux problèmes. Ces problèmes pouvant se reformuler en problème de **programmation linéaire** (PL)**, de Satisfiabilité** (SAT)**, de satisfaction de contraintes** (CSP)**.**

Dans notre travail, nous allons tout d'abord donner une claire définition des différents problèmes de recherche opérationnel, puis modéliser quelques-uns présent en milieu universitaire pour être pragmatique et concret avec des exemples de problèmes solvables par les méthodes de recherches opérationnelle, par la suite nous mettrons en lumière les différences entre le problème SAT et le CSP ; Après quoi nous modéliserons les problèmes du milieu universitaires comme des CSP.

I. Définitions

1. Problème de satisfiabilité

Le problème de satisfiabilité d'une formule consiste à déterminer s'il existe une évaluation qui satisfait cette dernière. Il est le plus souvent connu sous le nom de problème **SAT.** Résoudre ce problème c'est déterminer si une formule conçue à partir de celui-ci est satisfaisante. Une méthode simple pour résoudre ce problème consiste à essayer toutes les évaluations possibles jusqu'à en trouver une qui satisfasse la formule considérée. Peut-on faire mieux ? On ne sait pas. Le problème de savoir si résoudre ce problème requiert dans le pire cas un nombre **exponentiel** d'étapes de calcul ou s'il peut se résoudre en un nombre **polynomial** d'étapes de calcul (à chaque fois en fonction du nombre de variables dans la formule) est inconnu. C'est un problème ouvert depuis le début des années 1970 et qui fait partie des problèmes du millénaire.

2. Le problème de satisfaction de contraintes

Le problème de satisfaction de contraintes s'apparente légèrement au problème SAT. Pour facilement gérer des CSP complexes et même obtenir la formule dont on va tester la satisfiabilité, nous

utiliserons un formalisme mathématique dans lequel un CSP sera modélisé par un triplet **(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 ≠ b, c ≠ 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 ;
- La somme de a et c doit être inférieure à b.

Des exemples plus concrets seront présentés dans les parties suivantes.

II. Exemples et applications en milieu universitaire

1. Problèmes dans la gestion des œuvres universitaires

1.1. Transport des étudiants

a. Définition générale

Le responsable de la navette inter-campus de l'université de Dschang désire minimiser les coûts de carburant lors du transport des étudiants en déployant à un instant donné un nombre de navettes optimale (ceci en évitant que ces étudiants ne soient en retard pour leur cours en les transportant dans une marge de 30 minutes maximum). En effet, l'on dispose d'exactement 3 navettes : la première et la seconde ayant une capacité de **25** personnes et la troisième une capacité de **40** personnes. Par ailleurs,

pour un aller et retour à la station de départ, la première navette consomme 0.2 L, la seconde 0.44L et la troisième 0.70L.

Ce problème s'apparente à un problème de programmation linéaire car est mieux adapté pour la résolution des problèmes d'allocation de ressources (dans notre cas les navettes inter-campus). Nous allons donc modéliser ce problème à l'aide des techniques de la **programmation linéaire** dans les sections suivantes.

b. Modélisation du problème

Tout d'abord, représentons les données que nous avons recueillies de l'énoncé du problème.

	Navette 1	Navette 2	Navette 3
Capacité	25 personnes	25 personnes	40 personnes
Consommation (en L)	0.2 L	0.44 L	0.7 L
Durée du trajet	6 minutes	7 minutes	6 minutes

Table 1: Tableau récapitulatif des paramètres du problème d'optimisation du carburant des navettes inter-campus

La modélisation d'un **programme linéaire** passe par plusieurs étapes :

- **Identification des variables principales ou alors variables de décision :** Comme variables, nous avons :
  - ✓ Le nombre de trajets (aller et retour) que fera la navette 1 ici noté **x**
  - ✓ Le nombre de trajets que fera la navette 2 noté **y**
  - ✓ Le nombre de trajets que fera la navette 3 noté **z**
  - ✓ Le nombre d'étudiants présents noté **n**

- **Expression de la fonction objective en termes de variables identifiées :** Notre fonction objective ici est le coût de carburant que nous devons dépenser pendant ces 30 minutes ; ce coût est donc fonction du nombre de trajets qu'a fait chaque navette. On obtient donc :

  $$S = 0.2\,x + 0.44\,y + 0.7\,z$$

- **Formulation des contraintes :**
  - ✓ Les trajets doivent être des entiers positifs ou nuls (mais non tous nuls) : **contraintes de positivité** $x, y, z \geq 0$
  - ✓ Le temps pris par chaque navette ne doit pas excéder 30 minutes :

6 x ≤ 30, 7 y ≤ 30, 6 z ≤ 30.

- ✓ La totalité des étudiants transportés pendant les différents trajets par toutes les navettes qui ont été déplacé doit être supérieure ou égale au nombre d'étudiants (n) présent initialement : **25 x + 25 y + 40 z ≥ n.**

c. Résolution du problème

Les étapes ainsi faites, nous obtenons le **Programme linéaire** suivant :

$$\begin{cases} \boldsymbol{Min\ S = 0.2x + 0.44y + 0.7z} \\ \boldsymbol{Sous\ contraintes\ (S.C)} \\ \boldsymbol{6x \leq 30} \\ \boldsymbol{7y \leq 30} \\ \boldsymbol{6z \leq 30} \\ \boldsymbol{25x + 25y + 40z \geq n} \\ \boldsymbol{x, y, z \geq 0} \end{cases}$$

1.2. Problème de rendu monnaie au niveau du restaurant universitaire

a. Définition générale

Dans le restaurant universitaire, le plat coûte **100FrCFA**, donc les étudiants doivent pour cela se munir d'un ticket à **100 FCFA de valeur** qui leur permettra d'être servis par la suite. Alors au cours de cet achat de ticket, la plupart du temps, les étudiants ne disposent pas des 100fr mais des sommes comme 500fr, 100fr, etc ; donc doivent être remboursé après avoir acheté un certain nombre de ticket. Ce qui pourrait être parfois facile à faire pour la dame en question chargée de vendre les tickets mais elle doit rembourser en tenant compte qu'elle pourrait également avoir le même cas avec l'étudiant suivant dans la file ou au pire avec tous. Or celle-ci dispose de monnaie limitée. Alors il est donc nécessaire dans l'optique de servir le maximum d'étudiant, de rembourser avec les plus grandes sommes qu'elle dispose. C'est à dire que si par exemple elle doit rembourser 700fr et dispose de 9 pièces de 100fr, 5 de 50fr et 3 billets de 500fr, il sera mieux de rembourser 1 billet de 500fr et deux pièces de 100fr que d'y introduire celles de 50fr ou d'utiliser directement 7 pièces de 100fr. Ainsi s'il faille rembourser le suivant 300fr, il sera possible et évident.

Donc ayant un ensemble de pièces ou de billets ici (**25, 50, 100, 500, 1000, 2000, 5000 et 10000**) où $\mathbf{m_i}$ sont respectivement le nombre d'unités de monnaie précédente dont elle dispose, l'on doit pouvoir rembourser en utilisant toutes ces unités là ou plusieurs fois.

b. Modélisation du problème

On définit le **CSP** ***(T, D, C)*** tel que

- $\boldsymbol{T = \{T_{25}, T_{50}, T_{100}, T_{500}, T_{1000}, T_{2000}, T_{5000}, T_{10000}\}}$, où $\boldsymbol{T_{25}}$ désigne le nombre de pièces de 25 FCFA à retourner, $\boldsymbol{T_{50}}$ désigne le nombre de pièces de 50 FCFA à retourner, $\boldsymbol{T_{100}}$ désigne le nombre de pièces de 100 FCFA à retourner, $\boldsymbol{T_{500}}$ désigne le nombre de pièces de 500 FCFA à retourner, $\boldsymbol{T_{1000}}$ désigne le nombre de pièces de 1000 FCFA à retourner, $\boldsymbol{T_{2000}}$ désigne le nombre de pièces de 2000 FCFA à retourner, $\boldsymbol{T_{5000}}$ désigne le nombre de pièces de 5000 FCFA à retourner, $\boldsymbol{T_{10000}}$ désigne le nombre de pièces de 10000 FCFA à retourner.
- La quantité de pièces retournées, pour un type de pièce donné, est comprise entre 0 et le nombre de pièces de ce type que l'on a en réserve :

  $D(T_{25}) = \{0, 1, ..., Max_{25}\}$

  $D(T_{50}) = \{0, 1, ..., Max_{50}\}$

  $D(T_{100}) = \{0, 1, ..., Max_{100}\}$

  $D(T_{500}) = \{0, 1, ..., Max_{500}\}$

  $D(T_{1000}) = \{0, 1, ..., Max_{1000}\}$

  $D(T_{2000}) = \{0, 1, ..., Max_{2000}\}$

  $D(T_{5000}) = \{0, 1, ..., Max_{5000}\}$

  $D(T_{10000}) = \{0, 1, ..., Max_{1000}\}$

  Les Max_i (i = 50,100,500,1000,2000,5000,1000) représentent le nombre maximal de pièces ou de billets i.
- **Les contraintes spécifient que la somme à retourner doit être égale à la somme insérée moins le prix à payer.** Si **Y** est le prix initial donné, **P** le prix de **x** tickets acheté, l'on devra rembourser

$$\text{Reste} = 25T_{25} + 50\,T_{50} + 100\,T_{100} + 500\,T_{500} + 1000\,T_{1000} + 2000\,T_{2000} + 5000\,T_{5000} + 10000\,T_{10000} = Y - P\ (S)$$

Ce reste sera soumis aux contraintes suivantes : $\mathbf{c_1: 25T_{25} + 50\,T_{50} + 100\,T_{100} + 500\,T_{500} + 1000\,T_{1000} + 2000\,T_{2000} + 5000\,T_{5000} + 10000\,T_{10000} = Y - P}$ $\mathbf{c_2: \leq T_i \leq Max_i}$ $\mathbf{c_3:}$ le 8-uplet de solution (T_i) du CSP sera tel qu'il soit le

plus optimale i.e pour tout 8-uplet **(T_i)** solution de l'équation **(S), il doit celui dont la somme des T_i soit minimale et dont les coefficients T_i sont tous les maximum que l'on pouvait utiliser pour résoudre (S).**

2. Problèmes dans d'autres domaines

2.1. Problème de coloriage des cartes

a. Définition

Ici, l'on voudrait par exemple colorier les différentes cartes de pays d'un continent de telle sorte que **deux pays adjacents n'aillent pas la même couleur (ceci représente la contrainte).** Étiquetons les différentes **np** pays par un acronyme en occurrence **P_i, $1 \leq i \leq np$** et choisissons un ensemble de **nc couleurs** avec lesquelles l'on va colorier les pays dans notre cas. Une fois les différentes données (les couleurs et les pays) du problème à résoudre définies, l'on doit à présent établir les variables propositionnelles avec lesquels on définira une formule logique.

b. Modélisation

Pour évaluer si un ensemble de combinaisons respectent la contrainte ci-haut, nous allons noter $X_{pi, cj}$ la coloration du pays P_i avec la couleur C_j. A l'aide des variables propositionnelles que l'on a défini précédemment, l'on peut résumer notre problème a une conjonction des couleurs prises par les pays définit comme suit : $\Phi = \bigwedge X_{p_i c_j}$

Ceci nous ramène au problème de satisfiabilité (SAT) où l'on doit trouver une combinaison qui vérifie cette formule.

La vérification de la validité de cette formule se fera dans ce cas en un temps polynomial. En effet, l'on itéra sur chacun des pays et on vérifiera que la couleur qu'il possède soit différente de celle de tous ses voisins. Mais par contre, dépendant du nombre de pays *np* et du nombre de couleurs *nc*, l'on aura donc **$np^{np}.nc^{nc}$ vérifications.** Une illustration de ceci est présentée dans la figure ci-dessous.

Figure 1:Carte de l'Afrique

2.2. Problème des reines

a. Définition et Description du problème

Il s'agit de placer 4 reines sur un échiquier comportant 4 lignes et 4 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.**

b. Modélisation

Pour modéliser un problème sous la forme d'un CSP (**Constraint Satisfaction problem)**, il s'agit tout d'abord d'identifier l'ensemble des variables **X** (les inconnues du problème), ainsi que la fonction **D** qui associe à chaque variable de X son domaine (les valeurs que la variable peut prendre). Il faut ensuite identifier les contraintes **C** entre les variables. Notons qu'à ce niveau, on ne se soucie pas de savoir comment résoudre le problème : on cherche simplement à le spécifier formellement. Cette phase de spécification est indispensable à tout processus de résolution de problème ; les CSP fournissent un cadre

structurant à cette formalisation. Un même problème peut généralement être modélisé par différents CSP. Cependant, la modélisation choisie peut avoir une influence sur l'efficacité de la résolution.

Les "**inconnues**" du problème sont les positions des reines sur l'échiquier. En numérotant les lignes et les colonnes de l'échiquier de la façon suivante :

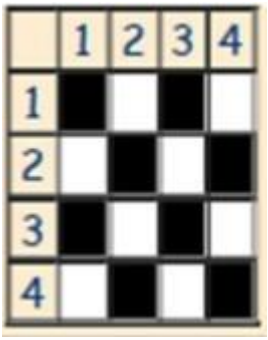

Figure 2: Représentation du problème des reines

On peut déterminer la position d'une reine par **un numéro de ligne** et un numéro de colonne. Ainsi, une première modélisation consiste à associer à chaque reine **i** deux variables $\mathbf{L_i}$ et $\mathbf{C_i}$ correspondant respectivement à la **ligne** et la **colonne** sur laquelle placer la reine. **Les contraintes spécifient alors que les reines doivent être sur des lignes différentes, des colonnes différentes et des diagonales différentes.** Notons pour cela que lorsque **2 reines sont sur une même diagonale montante, la somme de leurs numéros de ligne et de colonne est égale ; tandis que lorsqu'elles sont sur une même diagonale descendante (**en d'autres termes**,** il s'agit des cellules dont **l'indice de ligne est égale à l'indice de colonne contrairement aux diagonales précédentes) la différence de leurs numéros de ligne et de colonne est égale [EemmMfedNer].**

On en déduit le CSP suivant :

- **Variables** X = {L1, L2, L3, L4, C1, C2, C3, C4}
- **Domaines** D(L1) = D(L2) = D(L3) = D(L4) = D(C1) = D(C2) = D(C3) = D(C4) = {1,2,3,4}
- **Contraintes**

  On identifie 4 types de contraintes

  - **Les reines doivent être sur des lignes différentes.**

    C_{lig} = {L1≠L2, L1≠L3, L1≠L4, L2≠L3, L2≠L4, L3≠L4}
  - **Les reines doivent être sur des colonnes différentes.**

    C_{col} = {C1≠C2, C1≠C3, C1≠C4, C2≠C3, C2≠C4, C3≠C4}

- **Les reines doivent être sur des diagonales différentes.**
  C_{dm} = {C1+L1≠C2+L2, C1+L1≠C3+L3, C1+L1≠C4+L4, C2+L2≠C3+L3, C2+L2≠C4+L4, C3+L3≠C4+L4}
- **Les reines doivent être sur des diagonales descendantes différentes.**
  C_{dd} = {C1-L1≠C2-L2, C1-L1≠C3-L3, C1-L1≠C4-L4, C2-L2≠C3-L3, C2-L2≠C4-L4, C3-L3≠C4-L4

L'ensemble des contraintes est défini par l'union de ces 4 ensembles : **$C = C_{lig} \cup C_{col} \cup C_{dm} \cup C_{dd}$**

Cette contrainte c'est la formule de notre problème et donc, toutes les variables doivent la respecter pour être vraie. Les contraintes **C_{lig}** et **C_{col}** sont des **contraintes binaires** ; les contraintes **C_{dm}** et **C_{dd} sont des contraintes quaternaires**. L'énumération de toutes ces contraintes est ici un peu fastidieuse.

Pour être générique, on peut tout aussi bien les définir de la façon suivante :

Contraintes :

- **Les reines doivent être sur des lignes différentes**

  $$C_{lig}=\{Li \neq Lj \ \forall i \in \{1,2,3,4\}, \forall \ j \in \{1,2,3,4\} et\ i \neq j\}$$

- **Les reines doivent être sur des colonnes différentes**

  $$C_{col}=\{C\,i \neq C\,j \mid i \in \{1,2,3,4\}, j \in \{1,2,3,4\} et\ i \neq j\}$$

- **Les reines doivent être sur des diagonales montantes différentes**

  $$C_{dm}=\{Ci + Li \neq C\,j + L\,j \mid i\ \{1,2,3,4\}, j \in \{1,2,3,4\}\ et\ i \neq j\}$$

- **Les reines doivent être sur des diagonales descendantes différentes**

  $$C_{dd}=\{Ci - Li \neq C\,j - L\,j \mid i \in \{1,2,3,4\}, j \in \{1,2,3,4\}\ et\ i \neq j\}$$

2.3. Problème du Zèbre

a. Définition du problème

Attribué à Lewis Carroll, pasteur logicien et écrivain anglais auteur de nombreux autres puzzles. On considère ici cinq maisons, toutes de couleurs différentes (rouge, bleu, jaune, blanc, vert), dans

lesquelles logent cinq personnes de profession différente (peintre, sculpteur, diplomate, docteur et violoniste) de nationalité différente (anglaise, espagnole, japonaise, norvégienne et italienne) ayant chacune une boisson favorite (thé ,jus de fruits, café, lait et vin) et des animaux favoris (chien, escargots, renard, cheval et zèbre). On dispose des faits suivants:

- Le norvégien habite la première maison.
- La maison à côté de celle du norvégien est bleue.
- L'habitant de la troisième maison boit du lait.
- L'anglais habite la maison rouge.
- L'habitant de la maison verte boit du café.
- L'habitant de la maison jaune fume des kools.
- La maison blanche se trouve juste après la verte.
- L'espagnol a un chien.
- L'ukrainien boit du thé.
- Le japonais fume des cravens.
- Le fumeur d'Old Golds a un escargot.
- Le fumeur de gitanes boit du vin.
- Le voisin du fumeur de Chesterfield a un renard.
- Le voisin du fumeur de kools a un cheval.

Il s'agit de trouver le possesseur du zèbre et le buveur d'eau [EemmMfedNer].

b. Modélisation sous forme d'un CSP

On définit le CSP (X, D, C) tel que :

- **Variables du problème** : on associe une variable par attribut (couleur, animal, boisson, nationalité, cigarette)

X = {blanche, rouge, verte, jaune, bleue, norvégien, anglais, ukrainien, japonais, espagnol, cheval, renard, zèbre, escargot, chien, thé, eau, lait, café, vin, kools, chesterfield, old_golds, cravens, gitanes}

- **Domaines des variables** : **D(Xi)** = {1, 2, 3, 4, 5}, pour toute variable Xi de X
- **Contraintes** :
  - On pose tout d'abord une contrainte pour chaque assertion de l'énoncé :
    - norvégien = 1
    - bleue = norvégien + 1

- lait = 3
- anglais = rouge
- verte = café
- jaune = kools
- blanche = verte + 1
- espagnol = chien
- ukrainien = thé
- japonais = cravens
- gold_golds = escargot
- gitanes = vin
- (chesterfields = renard + 1) ou (chesterfields = renard - 1)
- (kools = cheval + 1) ou (kools = cheval – 1).

- De plus, toutes les variables de même "type" doivent avoir des valeurs différentes (il ne peut pas y avoir plusieurs maisons qui ont la même couleur, ou un même animal, ...) :
  - blanche ≠ rouge ≠ verte ≠ jaune ≠ bleue
  - thé ≠ eau ≠ lait ≠ café ≠ vin
  - norvégien ≠ anglais ≠ ukrainien ≠ japonais ≠ espagnol
  - cheval ≠ renard ≠ zèbre ≠ escargot ≠ chien
  - kools ≠ chesterfields ≠ old_golds ≠ cravens ≠ gitanes

III. Résolution des problèmes

A. Méthodes de résolution

1. Résolution des problèmes de programmation linéaire

Pour résoudre un problème de programmation linéaire, l'on dispose d'un ensemble de méthodes à savoir :

- **La méthode graphique** (possible seulement lorsque l'on a deux variables): celle-ci permet de résoudre le problème en utilisant des droites dans un repère orthonormé. Les étapes à suivre sont:

- **Représenter la région réalisable** : Ici, il s'agit de représenter le polygone des contraintes. En d'autres termes, il est question de tracer les droites qui représentent les contraintes et faire leur intersection.
- **Représenter la fonction objective**
- **Détermination de la solution optimale**. La solution optimale est l'un des sommets du polygone de contraintes obtenu s'il en existe; ou alors l'ensemble des solutions optimales est une droite (dans ce cas, la fonction objective est parallèle à un côté du polygone de contraintes).

- Méthode **Simplexe**

2. Résolution des CSP

Tout d'abord, il est nécessaire que nous définissions la notion de consistance d'une solution d'un CSP. Il s'agit d'une solution qui satisfait toutes les contraintes. Et donc si une solution est non consistante, cela signifie que celle-ci ne respecte pas toutes les contraintes du CSP.

a. La méthode « génère et test »

a.1. Principe

La façon la plus simple (et très naïve !) de résoudre un CSP sur les domaines finis consiste à énumérer toutes les affectations totales possibles jusqu'à en trouver 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.

a.2. Critique de "génère et teste" et notion d'espace de recherche d'un CSP

```
fonction genereEtTeste(A,(X,D,C)) retourne un booléen
Précondition :
(X,D,C) = un CSP sur les domaines finis A = une affectation partielle pour (X,D,C)
 Postrelation : retourne vrai si l'affectation partielle A peut être étendue en une solution pour
(X,D,C), faux sinon
début
                si toutes les variables de X sont affectées à une valeur dans A alors
                        /* A est une affectation totale */
                    si A est consistante alors /* A est une solution */
                        retourner vrai
                    sinon
                            retourner faux
                    finsi
                sinon /* A est une affectation partielle */
                        choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A

                            pour toute valeur Vi appartenant à D(Xi) faire
                                si genereEtTeste(A U {(Xi,Vi)}, (X,D,C)) = vrai alors retourner vrai
                            finpour retourner faux

finsi
fin
```

Table 2: test

L'algorithme "génère et teste" que nous venons de voir énumère l'ensemble des affectations complètes possibles, jusqu'à en trouver une qui soit consistante. L'ensemble des affectations complètes est appelé **l'*espace de recherche du CSP [1]***. Si le domaine de certaines variables contient une infinité de valeurs, alors cet espace de recherche est infini et on ne pourra pas énumérer ses éléments en un temps fini. Néanmoins, même en se limitant à des domaines comportant un nombre fini de valeurs, l'espace de recherche est souvent de taille tellement importante que l'algorithme "génère et teste" ne pourra se terminer en un temps "raisonnable". En effet, l'espace de recherche d'un CSP *(X,D,C)* comportant ***n*** variables ($X = \{X1, X2, ..., Xn\}$) est défini par

$$\boldsymbol{E = \{\{(X_1,v_1), (X_2,v_2), ..., (X_n, v_n)\}} \mid \forall i \in [1 ; n] \text{ , } \boldsymbol{v_i} \text{ est un élément de } D(X_i) \}$$

et le nombre d'éléments de cet espace de recherche est défini par

$$\boldsymbol{| E | = | D(X_1) | * | D(X_2) | * ... * | D(X_n) |}$$

de sorte que, si tous les domaines des variables sont de la même taille k (autrement dit, $| D(X_i) | = k$), alors la taille de l'espace de recherche est $\boldsymbol{| E | = k^n}$

Ainsi, le nombre d'affectations à générer croit de façon exponentielle en fonction du nombre de variables du problème. Considérons plus précisément un CSP ayant *n* variables, chaque variable pouvant prendre *2* valeurs (***k=2***). Dans ce cas, le nombre d'affectations totales possibles pour ce CSP est 2^n. En supposant qu'un ordinateur puisse générer et tester un milliard d'affectations par seconde (ce qui est une estimation vraiment très optimiste !), le tableau suivant donne une idée du temps qu'il faudrait pour énumérer et tester toutes les affectations en fonction du nombre de variables ***n***.

Nombre de variables n	Nombre d'affectations totales 2^n	Temps (si on traitait 10^9 affectations par seconde)
10	environ 10^3	environ 1 millionième de seconde
20	environ 10^6	environ 1 millième de seconde
30	environ 10^9	environ 1 seconde
40	environ 10^12	environ 16 minutes
50	environ 10^15	environ 11 jours
60	environ 10^18	environ 32 ans
70	environ 10^21	environ 317 siècles

Table 3:Tableau comparatif des temps de résolution d'un CSP en fonction de la taille des variables

La conclusion de ce petit exercice de dénombrement est que, dès lors que le CSP a plus d'une trentaine de variables, on ne peut pas appliquer "bêtement" l'algorithme "génère et teste". Il faut donc chercher à réduire l'espace de recherche. Pour cela, on peut notamment :

- **ne développer que les affectations partielles consistantes** : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à l'étendre en une affectation totale puisque celle-ci sera nécessairement inconsistante

  *Cette idée est reprise dans l'algorithme "**simple retour-arrière**" que nous étudierons dans la section suivante.*

- **réduire les tailles des domaines des variables** en leur enlevant les valeurs "incompatibles" : pendant la génération d'affectations, on filtre le domaine des variables pour ne garder que les valeurs "localement consistantes" avec l'affectation en cours de construction, et dès lors que le domaine d'une variable devient vide, on arrête l'énumération pour cette affectation partielle

*Cette idée est reprise dans l'algorithme "**anticipation**" que nous étudierons plus loin.*

- **Introduire des "heuristiques" pour "guider" la recherche** : lorsqu'on énumère les affectations possibles, on peut essayer d'énumérer en premier celles qui sont les plus "prometteuses", en espérant ainsi tomber rapidement sur une solution.

Il existe encore bien d'autres façons de (tenter de) réduire la combinatoire, afin de rendre l'exploration exhaustive de l'espace de recherche possible, que nous ne verrons pas ici. Par exemple, lors d'un échec, on peut essayer d'identifier la cause de l'échec (quelle est la variable qui viole une contrainte) pour ensuite "**retourner en arrière**" directement là où cette variable a été instanciée afin de remettre en cause plus rapidement la variable à l'origine de l'échec. C'est ce que l'on appelle le "**retour arrière intelligent" ("intelligent backtracking").**

Une autre approche particulièrement séduisante consiste à exploiter des connaissances sur les types de contraintes utilisées pour réduire l'espace de recherche. *considérons par exemple le CSP (X, D, C) suivant :*

- *X = {a, b, c},*
- *D(a)=D(b)=D(c) = {0, 1, 2, 3, 4, ..., 1000},*
- *C = {4*a - 2*b = 6*c + 3}*

*L'espace de recherche de ce CSP comporte 1 milliard d'affectations ; pour résoudre ce CSP, on peut énumérer toutes ces combinaisons, en espérant trouver une qui satisfasse la contrainte 4*a - 2*b = 6*c + 3... Ceci est très long et si on remplace la borne supérieure 1000 par l'infini, ça devient carrément impossible. En revanche un simple raisonnement permet de conclure très rapidement que ce CSP n'a pas de solution : la partie gauche de l'équation "**4*a - 2*b**" donnera toujours un nombre pair, quelles que soient les valeurs affectées à **a** et **b,** tandis que la partie droite **"6*c + 3"** donnera toujours un nombre impair, quelle que soit la valeur affectée à **c** ; par conséquent, on ne peut trouver d'affectation qui satisfasse la contrainte, et il est inutile d'énumérer toutes les affectations possibles pour s'en convaincre !*

Ce genre de raisonnement demande de l'intelligence, ou pour le moins des connaissances. De fait, l'homme est capable de résoudre des problèmes très combinatoires en raisonnant (en utilisant son "**expérience**" et des connaissances plus ou moins explicitées). Un exemple typique est le jeu d'échec :

les grands joueurs d'échecs n'envisagent pour chaque coup à jouer que très peu de combinaisons (les meilleures évidemment !), éliminant par des raisonnements souvent difficiles à expliciter un très grand nombre de combinaisons moins intéressantes.

b. L'algorithme "simple retour-arrière"

b.1. Principe de l'algorithme "simple retour-arrière"

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. Dans ce cas, on "retourne en arrière" ("**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.

Par exemple, sur la trace d'exécution de l'algorithme « génère et test » sur l'exemple de la partie précédente, on remarque que l'algorithme génère tous les prolongements de l'affectation partielle A={(a,0),(b,0)}, en énumérant toutes les possibilités d'affectation pour les variables ***c*** *et* ***d,*** *alors qu'elle viole la contrainte* ***a*** *≠* ***b****. L'algorithme "****simple retour-arrière****" ne va donc pas chercher à étendre cette affectation, mais va "****retourner en arrière****" à l'affectation partielle précédente A={(a,0)}, et va l'étendre en affectant 1 à b , ...*

Ce principe est repris dans la fonction récursive "simpleRetourArrière(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.

b.2. Algorithme

c. Méthode d'anticipation

c.1. Notions de filtrage et de consistance locale

Table 4: Algoritme retour arriere

```
fonction simpleRetourArrière(A,(X,D,C)) retourne un booléen
Précondition :
A = affectation partielle
(X,D,C) = un CSP sur les domaines finis
Postrelation : retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon
début
    si A n'est pas consistante alors retourner faux finsi
    si toutes les variables de X sont affectées à une valeur dans A alors
            /* A est une affectation totale et consistante = une solution */
            retourner vrai
    sinon
            /* A est une affectation partielle consistante */
            choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
    pour toute valeur Vi appartenant à D(Xi) faire
        si simpleRetourArrière(A U {(Xi,Vi)}, (X,D,C)) = vrai alors
            retourner vrai
    finpour
    retourner faux
    finsi
fin
```

Pour améliorer l'algorithme "simple retour-arrière", on peut tenter d'anticiper ("look ahead" en anglais) les conséquences de l'affectation partielle en cours de construction sur les domaines des variables qui ne sont pas encore affectées : si on se rend compte qu'une variable non affectée *Xi* n'a plus de valeur dans son domaine *D(Xi)* qui soit "localement consistante" avec l'affectation partielle en cours de construction, alors il n'est pas nécessaire de continuer à développer cette branche, et on peut tout de suite retourner en arrière pour explorer d'autres possibilités **[1].**

Pour mettre ce principe en œuvre, on va, à chaque étape de la recherche, filtrer les domaines des variables non affectées en enlevant les valeurs "localement inconsistantes", c'est-à-dire celles dont on peut inférer qu'elles n'appartiendront à aucune solution. On peut effectuer différents filtrages,

correspondant à différents niveaux de consistances locales, qui vont réduire plus ou moins les domaines des variables, mais qui prendront aussi plus ou moins de temps à s'exécuter : considérons un CSP *(X, D ,C)*, et une affectation partielle consistante *A*,

- Le filtrage le plus simple consiste à anticiper d'une étape l'énumération : pour chaque variable *Xi* non affectée dans *A*, on enlève de *D(Xi)* toute valeur *v* telle que l'affectation *A U {(Xi,v)}* soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X1 à 1, on peut enlever du domaine de X2 la valeur 1 (qui viole la contrainte X1 ≠ X2) et la valeur 2 (qui viole la contrainte 1-X1 ≠ 2-X2).

Un tel filtrage permet d'établir ce qu'on appelle la ***consistance de noeud***, aussi appelée ***1-consistance***.

- un filtrage plus fort, mais aussi plus long à effectuer, consiste à anticiper de deux étapes l'énumération : pour chaque variable *Xi* non affectée dans *A*, on enlève de *D(Xi)* toute valeur *v* telle qu'il existe une variable *Xj* non affectée pour laquelle, pour toute valeur *w* de *D(Xj)*, l'affectation *A U {(Xi,v),(Xj,w)}* soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X1 à 1, on peut enlever la valeur 3 du domaine de X2 car si X1=1 et X2=3, alors la variable X3 ne peut plus prendre de valeurs : si X3=1, on viole la contrainte X3 ≠ X1 ; si X3=2, on viole la contrainte X3+3 ≠ X2+2 ; si X3=3, on viole la contrainte X3 ≠ X2 ; et si X3=4, on viole la contrainte X3-3 ≠ X2-2.

Notons que ce filtrage doit être répété jusqu'à ce plus aucun domaine ne puisse être réduit.Ce filtrage permet d'établir ce qu'on appelle la ***consistance d'arc***, aussi appelée ***2-consistance***.

- Un filtrage encore plus fort, mais aussi encore plus long à effectuer, consiste à anticiper de trois étapes l'énumération. Ce filtrage permet d'établir ce qu'on appelle la ***consistance de chemin***, aussi appelée ***3consistance***.

- ... et ainsi de suite... notons que s'il reste *k* variables à affecter, et si l'on anticipe de *k* étapes l'énumération pour établir la k-consistance, l'opération de filtrage revient à résoudre le CSP, c'est-

à-dire que toutes les valeurs restant dans les domaines des variables après un tel filtrage appartiennent à une solution.

c.2. Principe de l'algorithme "anticipation"

Le principe général de l'algorithme "anticipation" reprend celui de l'algorithme "simple retour-arrière", en ajoutant simplement une étape de filtrage à chaque fois qu'une valeur est affectée à une variable. Comme on vient de le voir au point 4.1, on peut effectuer différents filtrages plus ou moins forts, permettant d'établir différents niveaux de consistance locale (nœud, arc, chemin, ...). Par exemple, la fonction récursive "anticipation/nœud (A,(X,D,C))" décrite ci-dessous effectue un filtrage simple qui établit à chaque étape la consistance de nœud. Dans cette fonction, *A* contient une affectation partielle consistante 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.

Nous n'étudierons pas dans le cadre de cette analyse l'algorithme de filtrage qui établit la consistance d'arc.

fonction anticipation/noeud(A,(X,D,C)) retourne un booléen **Précondition**

:

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

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

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

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

retourner vrai

sinon /* A est une affectation partielle consistante */

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

toute valeur Vi appartenant à D(Xi) faire

/* filtrage des domaines par rapport à A U {(Xi,Vi)} */ **pour**

toute variable Xj de X qui n'est pas encore affectée faire

Dfiltré(Xj) <- { Vj élément de D(Xj) / A U {(Xi,Vi),(Xj,Vj)} est consistante }

si Dfiltré(Xj) est vide alors retourner faux **finpour**

si anticipation(A U {(Xi,Vi)}, (X,Dfiltré,C))=vrai alors retourner vrai

finpour retourner faux

finsi

fin

d. Intégration d'heuristiques

Les algorithmes que nous venons d'étudier choisissent, à chaque étape, la prochaine variable à instancier parmi l'ensemble des variables qui ne sont pas encore instanciées ; ensuite, une fois la variable choisie, ils essayent de l'instancier avec les différentes valeurs de son domaine. Ces algorithmes ne disent rien sur l'ordre dans lequel on doit instancier les variables, ni sur l'ordre dans lequel on doit affecter les valeurs aux variables. Ces deux ordres peuvent changer considérablement l'efficacité de ces algorithmes : imaginons qu'à chaque étape on dispose des conseils d'un "oracle-qui-sait-tout" qui nous dise quelle valeur choisir sans jamais se tromper ; dans ce cas, la solution serait trouvée sans jamais retourner en arrière... Malheureusement, le problème général de la satisfaction d'un CSP sur les domaines finis étant NP-complet, il est plus qu'improbable que cet oracle fiable à 100% puisse jamais être "programmé". En revanche, on peut intégrer des heuristiques pour déterminer l'ordre dans lequel les variables et les valeurs doivent être considérées : une heuristique est une règle non systématique (dans le sens où elle n'est pas fiable à 100%) qui nous donne des indications sur la direction à prendre dans l'arbre de recherche.

Les heuristiques concernant l'ordre d'instanciation des valeurs sont généralement dépendantes de l'application considérée et difficilement généralisables. En revanche, il existe de nombreuses heuristiques d'ordre d'instanciation des variables qui permettent bien souvent d'accélérer considérablement la recherche. L'idée générale consiste à instancier en premier les variables les plus "critiques", c'est-à-dire celles qui interviennent dans beaucoup de contraintes et/ou qui ne peuvent prendre que très peu de valeurs. L'ordre d'instanciation des variables peut être :

- ***statique***, quand il est fixé avant de commencer la recherche

  Par exemple, on peut ordonner les variables en fonction du nombre de contraintes portant sur elles : l'idée est d'instancier en premier les variables les plus contraintes, c'est-à-dire celles qui participent au plus grand nombre de contraintes.

- Ou ***dynamique***, quand la prochaine variable à instancier est choisie dynamiquement à chaque étape de la recherche.

  Par exemple, l'heuristique "échec d'abord" ("first-fail" en anglais) consiste à choisir, à chaque étape, la variable dont le domaine a le plus petit nombre de valeurs localement consistantes avec l'affectation partielle en cours. Cette heuristique est généralement couplée

avec l'algorithme "anticipation", qui filtre les domaines des variables à chaque étape de la recherche pour ne garder que les valeurs qui satisfont un certain niveau de consistance locale.

3. Résolution des problèmes SAT

La résolution du problème SAT comme décrit dans l'exemple de la coloration des cartes se part le test de combinaisons de valeurs prises dans un espace précis.

B. Résolution des Applications

1. Résolution du problème de minimisation des dépenses en carburant de la navette inter-campus

$$\begin{cases} \boldsymbol{Min\ S = 0.2x + 0.44y + 0.7z} \\ \boldsymbol{Sous\ contraintes\ (S.C)} \\ \boldsymbol{6x \leq 30} \\ \boldsymbol{7y \leq 30} \\ \boldsymbol{6z \leq 30} \\ \boldsymbol{25x + 25y + 40z \geq n} \\ \boldsymbol{x, y, z \geq 0} \end{cases}$$

Nous avons à faire à un problème de minimisation avec plusieurs paramètres. La méthode de résolution graphique ne peut donc pas être appliquée ici. Nous utiliserons la méthode du simplexe.

Pour un problème de minimisation, la forme canonique doit uniquement contenir des contraintes avec les signes $\geq$. De cette forme canonique, nous obtenons la **forme standard** suivante :

$$\begin{cases} \boldsymbol{Min\ S = 0.2x + 0.44y + 0.7z} \\ \boldsymbol{Sous\ contraintes\ (S.C)} \\ \boldsymbol{6x + e_1 = 30} \\ \boldsymbol{7y + e_2 = 30} \\ \boldsymbol{6z + e_3 = 30} \\ \boldsymbol{25x + 25y + 40z - e_4 = n} \\ \boldsymbol{x, y, z, e_1, e_2, e_3, e_4 \geq 0} \end{cases}$$

Comme les coefficients de la base $(\boldsymbol{e_1, e_2, e_3, e_4, e_5})$ sont négatifs dans notre forme standard, nous allons introduire des variables artificielles pour corriger le problème.

$$\begin{cases} \boldsymbol{Min\ S = 0.2x + 0.44y + 0.7z + Mt} \\ \boldsymbol{Sous\ contraintes\ (S.C)} \\ \boldsymbol{6x + e_1 = 30} \\ \boldsymbol{7y + e_2 = 30} \\ \boldsymbol{6z + e_3 = 30} \\ \boldsymbol{25x + 25y + 40z - e_4 + t = n} \\ \boldsymbol{x, y, z, e_1, e_2, e_3, e_4, t \geq 0} \end{cases}$$

A partir du système ci-dessus, on peut aisément résoudre le problème en appliquant juste l'algorithme du simplexe en débutant par le tableau initial suivant [2] :

Min	**C_i**		**0,2**	**0,44**	**0,7**	**0**	**0**	**0**	**0**	**M**
C_j	**B**	**b**	**x**	**y**	**z**	**e_1**	**e_2**	**e_3**	**e_4**	**t**
0	**e_1**	**30**	**6**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
0	**e_2**	**30**	**0**	**7**	**0**	**0**	**1**	**0**	**0**	**0**
0	**e_3**	**30**	**0**	**0**	**6**	**0**	**0**	**1**	**0**	**0**
M	**t**	**n**	**25**	**25**	**40**	**0**	**0**	**0**	**-1**	**1**
	S_i	**nM**	**25M**	**25M**	**40M**	**0**	**0**	**0**	**-M**	**M**
	C_i-S_i		**0.2-25M**	**0.44-25M**	**0.7-40M**	**0**	**0**	**0**	**M**	**0**

Table 5: Tableau initial de la méthode Simplex pour la résolution du problème de la navette

Ici, il est question que nous puissions déterminer la ligne pivot et donc, nous devons avoir $\min\left\{\frac{30}{6}, \frac{n}{40}\right\}$

Premier cas : $n < 200$

Pour ces valeurs de n, on aura: $\boldsymbol{\min\left\{\frac{30}{6}, \frac{n}{40}\right\} = \frac{n}{40}}$

Min	C_i		0,2	0,44	0,7	0	0	0	0	M
C_j	B	b	x	y	z	e_1	e_2	e_3	e_4	t
0	e_1	30	6	0	0	1	0	0	0	0
0	e_2	30	0	7	0	0	1	0	0	0
0	e_3	30	0	0	6	0	0	1	0	0
M	t	n	25	25	40	0	0	0	-1	1
	S_i	nM	25M	25M	40M	0	0	0	-M	M
	C_i-S_i		0.2-25M	0.44-25M	0.7-40M	0	0	0	M	0

Table 6: Tableau initial de la méthode Simplex pour la résolution du problème de la navette itération 1

Min	C_i		0,2	0,44	0,7	0	0	0	0	M
C_j	B	b	x	y	z	e_1	e_2	e_3	e_4	t
0	e_1	30	6	0	0	1	0	0	0	
0	e_2	30	0	7	0	0	1	0	0	
0	e_3	30	-15/4	-15/4	0	0	0	1	3/20	
0,7	z	n/40	5/8	5/8	1	0	0	0	-1/40	
	S_i	7n/400	0,43	0,43	0,7	0	0	0	-0,01	
	C_i-S_i		-0,23	0,01	0	0	0	0	0,01	

Table 7: Tableau initial de la méthode Simplex pour la résolution du problème de la navette itération 2

Toujours dans le cas où $\boldsymbol{n < 200}$, prenons le cas où $\boldsymbol{n < 125}$, car l'on voudrait avoir le $\min\left\{\frac{30}{6}, \frac{n}{25}\right\} = \frac{n}{25}$

Min	C_i		0,2	0,44	0,7	0	0	0	0	M
C_j	B	b	x	y	Z	e_1	e_2	e_3	e_4	t
0	e_1	30	6	0	0	1	0	0	0	
0	e_2	30	0	7	0	0	1	0	0	
0	e_3	30	-15/4	-15/4	0	0	0	1	3/20	
0,7	z	n/40	5/8	5/8	1	0	0	0	-1/40	
	S_i	7n/400	0,43	0,43	0,7	0	0	0	-0,01	
	C_i-S_i		-0,23	0,01	0	0	0	0	0,01	

Table 8:Tableau initial de la méthode Simplex pour la résolution du problème de la navette itération 3

Min	C_i		0,2	0,44	0,7	0	0	0	0	M
C_j	B	b	x	y	Z	e_1	e_2	e_3	e_4	t
0	e_1	30	0	-6	-8.4	1	0	0	0.24	
0	e_2	30	0	7	0	0	1	0	0	
0	e_3	30	0	0	6	0	0	1	-0.3	
0,2	x	n/25	1	1	8/5	0	0	0	-1/25	
	S_i	2n/250	0.2	0.2	0.32	0	0	0	-0.008	
	C_i-S_i		0	0.24	0.38	0	0	0	0.008	

Table 9Tableau initial de la méthode Simplex pour la résolution du problème de la navette itération 4

Alors, la solution optimale est le 7-uplet (n/25, 0, 0, 30, 30, 30, 0).Donc la solution à notre problème dans ce cas est $\begin{cases} x = n/25 \\ y = 0 \\ z = 0 \end{cases}$

L'on poursuivra le même cheminement et les raffinements avec les autre cas (les valeurs ou intervalles que peuvent prendre n).

2. Résolution su problème de monnaie

Alors, nous sommes là dans un CSP où l'on doit choisir des combinaisons devant minimiser les **ti** et respecter les contraintes ces contrainte. Dans cette modélisation, nous avons utilisé une contrainte arithmétique linéaire sur les entiers. Cette contrainte est globale dans le sens où elle fait intervenir toutes les variables du problème. Une autre modélisation possible consisterait à définir le domaine des variables par l'ensemble des entiers positifs ou nuls, et d'ajouter en contraintes que la quantité de pièces retournées doit être supérieure ou égale au nombre de pièces en réserve (pour chaque type de pièce différent).

Pour exprimer le fait que l'on souhaite que le distributeur rende le moins de pièces possibles, on pourrait ajouter à ce CSP une fonction "objectif" à minimiser

$$f(T) = 25T_{25} + 50\ T_{50} + 100\ T_{100} + 500\ T_{500} + 1000\ T_{1000} + 2000\ T_{2000} + 5000\ T_{5000} + 10000\ T_{10000} = Y - P$$

Dans ce cas, résoudre le CSP revient à chercher une affectation de *X* complète et consistante qui minimise ***f(X)***. En utilisant en occurrence l'algorithme « génère et test ».

3. Résolution du problème de coloriage des cartes

En se restreignant juste à un ensemble de pays bien précis, l'on pourra avoir la solution suivante en disposant pour cela du couleur rouge, bleu, jaune et **vert pour le coloriage de la figure ci-bas**.

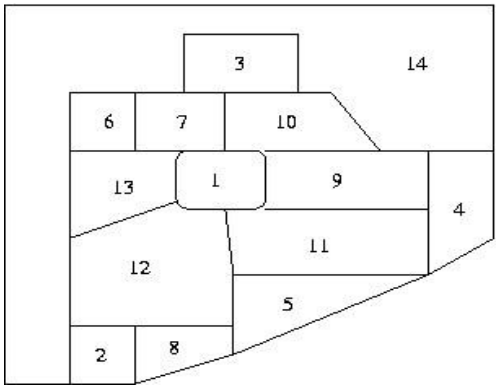

Figure 3: Catre pour la resolution du CSP de la coloration des cartes

On définit le CSP *(X, D, C)* tel que

- *X = {X1, X2, ..., X14}*

  (On associe une variable *Xi* différente par région *i* à colorier.)

- *pour tout Xi élément de X, D(Xi) = {bleu, rouge, vert, jaune}*

  (Chaque région peut être coloriée avec une des 4 couleurs.)

- *C = {Xi ≠ Xj / Xi et Xj sont 2 variables de X correspondant à des régions voisines}*

  (2 régions voisines doivent être de couleurs différentes.)

Pour être plus précis, on peut définir explicitement les relations de voisinage entre régions, par exemple à l'aide d'un prédicat *voisines/2*, tel que *voisines(X,Y)* soit vrai si *X* et *Y* sont deux régions voisines. Ce prédicat peut être défini en extension, en listant l'ensemble des couples de régions ayant une frontière en commun :

*voisines(X, Y) <=> (X,Y) élément-de {(1,7), (1,9), (1,10), (1,11), (1,12), (1,13), (2,8), (2,12), (2,14), (3,7), (3,10), (3,14), (4,9), (4,11), (4,14), (5,8), (5,11), (5,12), (6,7), (6,13), (6,14), (7,1), (7,3), (7,6), (7,10), (7,13), (7,14), (8,2), (8,5), (8,12), (9,1), (9,4), (9,10),(9,11), (10,1), (10,3), (10,7), (10,9), (10,14), (11,1), (11,4), (11,5), (11,9), (11,12), (12,1), (12,2), (12,5), (12,8), (12,11), (12,13), (12,14), (13,1), (13,6), (13,7), (13,12), (13,14), (14,2), (14,3), (14,4), (14,6), (14,7),(14,10), (14,12), (14,13)}.O*n peut alors définir l'ensemble des contraintes *C* de la façon suivante :

C = { Xi ≠ Xj / Xi et Xj sont 2 variables différentes de X et voisines(Xi,Xj) = vrai }

Ce problème de coloriage d'une carte est un cas particulier du problème du coloriage des sommets d'un graphe (deux sommets adjacents du graphe doivent toujours être de couleurs différentes). De nombreux problèmes "réels" se ramènent à ce problème de coloriage d'un graphe : problème des examens, d'emploi du temps, de classification, ...

III. Différence entre CSP et problème de satisfiabilité

Les différences fondamentales entre le CSP et le problème SAT sont:

- Dans le problème SAT, la notion phare est la formule contrairement au CSP où la notion phare est la contrainte.
- Dans le CSP, le but est de trouver une solution optimale qui doit satisfaire un ensemble de contrainte or dans un problème SAT, la préoccupation est juste de trouver une solution qui satisfait la formule donnée.
- Dans un problème SAT, l'on a à faire à une formule booléenne alors dans un CSP, la fonction du problème est une application
- Dans un CSP, l'on dispose d'une fonction à optimiser et des contraintes, qui combinée de manière conjonctive ou disjonctive, donne une formule; contrairement au problème SAT qui n'a qu'une seule formule.

Bibliographie

[EemmMfedNer] *Rapport de recherche operationnelle, la recherche operationnnelle en intelligence artificielle*,**Elouti Emmanuel, Mfendem Daina, Nebou Richie**

Webograpie

[1] *https://perso.liris.cnrs.fr/christine.solnon/Site-PPC/session3/e-miage-ppc-sess3.htm*

[2] *apprendre l'informatique, Recherche Opérationnelle, htpps://youtu.be/cvpBRNAYvQk*