TD2 – Graphe de décomposition et CSP

Chloé Conrad, Nathalie Guin, Marie Lefevre & Maëva Somny

PARTIE 1 – RESOLUTION PAR SATISFACTION DE CONTRAINTES

Ce paradigme de résolution de problèmes a été présenté en cours. Nous avons vu que pour faire de la résolution de problèmes par CSP, nous devons définir un problème selon ce formalisme :

- $X = \{X_1 , X_2 , ..., X_n\}$ l'ensemble des variables caractérisant le problème ;
- $D(X_i)$ le domaine de chaque variable X_i = l'ensemble des valeurs que X_i peut prendre théoriquement ;
- $C = \{C_1 , C_2 , ..., C_k\}$ l'ensemble des contraintes, sachant que chaque contrainte C_j est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

L'exemple de la coloration de carte vu lors du dernier TD est classiquement un problème que l'on peut résoudre en le modélisant ainsi.

Pour travailler durant ce TD, nous allons utiliser un **cryptarithme**, c'est-à-dire un casse-tête numérique et logique qui consiste en une équation mathématique où les lettres représentent des chiffres à trouver. L'équation comporte habituellement des opérations mathématiques de base, telles l'addition et la multiplication. L'exemple le plus connu, publié en juillet 1924 est dû à Henry Dudeney :

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

Chaque lettre représente un seul chiffre et le chiffre le plus significatif est différent de zéro. Idéalement, le casse-tête doit avoir une solution unique. La solution ici est O=0, M=1, Y=2, E=5, N=6, D=7, R=8 et S=9.

Pour résoudre à la main un cryptarithme, il faut faire des déductions astucieuses et une recherche extensive parmi les possibilités. Par exemple, le M du résultat est 1, puisqu'il s'agit de la retenue de la somme de deux nombres S et M et d'une éventuelle retenue. Etc.

MODELISATION DU PROBLEME

Plusieurs modélisations respectant le formalisme d'un CSP peuvent être proposées. **Proposez une modélisation du problème**.

========== Indices de correction ==========

Modélisation 1 :

Variables : S,E,N,D,M,O,R,Y

Domaines :

- DS = DM = {1,2,3,4,5,6,7,8,9}
- DE = DN = DD = DO = DR = DY = {0,1,2,3,4,5,6,7,8,9}

Contraintes :

- (D + E) = Y + Ret1 * 10, où Ret1 a pour domaine de valeur {0,1}
- (N + R + Ret1) = E + Ret2 * 10, où Ret2 a pour domaine de valeur {0,1}
- (E + O + Ret2) = N + Ret3 * 10, où Ret3 a pour domaine de valeur {0,1}
- (S + M + Ret3) = M * 10 + O
- Toutes les variables sont différentes

Modélisation 2 :

Variables : S,E,N,D,M,O,R,Y

Domaines :

- DS = DM = {1,2,3,4,5,6,7,8,9}
- DE = DN = DD = DO = DR = DY = {0,1,2,3,4,5,6,7,8,9}

Contraintes :

- 1000(S + M) + 100(E + O) + 10(N + R) + (D + E) = 10000M +1000O +100N +10E +Y
- Toutes les variables sont différentes

==========

METHODE « GENERER ET TESTER »

Si on utilise la méthode classique consistant à construire l'espace des états en générant toutes les solutions puis en les testant, **combien d'affectations différentes** doit-on tester pour trouver toutes les solutions ?

========== Indices de correction ==========

On teste toutes les affectations :

A = { (S, 1), (E, 0), (N, 0), (D, 0) , (M, 1), (O, 0), (R, 0), (Y,0)}

A = { (S, 1), (E, 0), (N, 0), (D, 0) , (M, 1), (O, 0), (R, 0), (Y,1)}

A = { (S, 1), (E, 0), (N, 0), (D, 0) , (M, 1), (O, 0), (R, 0), (Y,2)}

A = { (S, 1), (E, 0), (N, 0), (D, 0) , (M, 1), (O, 0), (R, 0), (Y,3)}

…

Taille de l'espace = taille de DS * taille de DM * taille de DM * … = 9*9*10*10*10*10*10*10 = 81 millions d'états.

==========

METHODE « RETOUR ARRIERE »

Dans cette méthode, on retourne en arrière quand, à l'évidence, il n'y a plus de solutions possibles. **Développez les affectations effectuées**.

========== Indices de correction ==========

Avec la modélisation 1 :

A = { (S, 1)} => consistant

A = { (S, 1), (E, 0)} => consistant

A = { (S, 1), (E, 0), (N, 0)} => inconsistant car 2 variables ont la même valeur

A = { (S, 1), (E, 0), (N, 1)} => inconsistant car 2 variables ont la même valeur

A = { (S, 1), (E, 0), (N, 2)} => consistant

A = { (S, 1), (E, 0), (N, 2), (D, 0)} => inconsistant car 2 variables ont la même valeur

A = { (S, 1), (E, 0), (N, 2), (D, 1)} => inconsistant car 2 variables ont la même valeur

A = { (S, 1), (E, 0), (N, 2), (D, 2)} => inconsistant car 2 variables ont la même valeur

A = { (S, 1), (E, 0), (N, 2), (D, 3)} => => consistant (l'algorithme ne voit pas que l'on doit avoir (D + E) = Ret1 * 10 +Y, or on a D+E = 3, il faudrait donc avoir Y=3, et donc avoir 2 variables avec la même valeur, il le verra seulement quand il arrivera à Y…

A = { (S, 1), (E, 0), (N, 2), (D, 3), (M, 1} => inconsistant

Donc on avance plus vite mais on fait encore plein de tests pour rien…

==========

METHODE « FILTRAGE »

Dans cette méthode, à chaque fois qu'une affectation est faite, on réduit le domaine des variables. Si on n'est pas arrivé au but et qu'un domaine de variable est vide alors on ne va pas plus loin dans l'affectation. **Développez les affectations effectuées**.

========== Indices de correction ==========

Avec la modélisation 1 :

A = { (S, 1)} => consistant, on enlève le 1 de tous les domaines de valeurs

A = { (S, 1), (E, 0)} => consistant, on enlève le 0 de tous les domaines de valeurs

A = { (S, 1), (E, 0), (N, 2)} => consistant, on enlève le 2 de tous les domaines de valeurs

A = { (S, 1), (E, 0), (N, 2), (D, 3)} => en enlevant les valeurs du domaine de Y, l'algo se rend compte que le domaine de Y devient vide et donc backtrack

A = { (S, 1), (E, 0), (N, 2), (D, 4)} => idem pour toutes les valeurs de D, mais on les teste toutes !

Donc on avance plus vite, on fait moins de tests mais on fait encore plein de tests pour rien… :-/

==========

HEURISTIQUE GENERALE

Proposer une heuristique générale, c'est-à-dire non spécifique aux problèmes des cryptarithmes, mais pouvant être utilisée pour n'importe quel autre problème de satisfaction de contraintes.

========== Indices de correction ==========

Toujours commencer par les variables les plus contraintes.

Si une variable n'a plus qu'une seule valeur de possible, on affecte cette variable.

==========

HEURISTIQUE SPECIFIQUE

Proposer une heuristique particulière aux problèmes de cryptarithmes donc pas forcément utilisable dans un autre problème de satisfaction de contraintes.

========== Indices de correction ==========

D'abord les unités, puis les dizaines, puis…

==========

PARTIE 2 – DECOMPOSITION DE PROBLEMES EN GRAPHE D'ETATS

Nous avons vu en cours que pour résoudre un problème, une des approches consiste à le décomposer en sous-problèmes « triviaux » que l'on pourra résoudre facilement. Nous allons, dans ce TD, étudier l'algorithme BSH qui permet de parcourir un graphe de décomposition de problèmes pour trouver une décomposition qui permet de résoudre le problème. Nous le mettrons en œuvre sur un problème « jouet » dont la décomposition du problème en graphe ET/OU est fournie. Nous travaillerons ensuite sur le problème du singe et des bananes pour voir comment construire ce type de graphes.

ALGORITHME BACKTRACK SEARCH DANS UN HYPERGRAPHE

La décomposition d'un problème en sous-problèmes plus simples est un principe applicable à des problèmes modélisables de manière récursive, comme celui des tours de Hanoï vu en cours, mais pas seulement ! Certains problèmes peuvent être décomposés selon une base d'opérateurs (des règles) de décomposition. Quel que soit le type de décomposition (récursive ou non), il est possible de construire un graphe ET/OU représentant la décomposition du problème. Il faut ensuite utiliser des algorithmes pour trouver un chemin permettant de résoudre le problème. Ce chemin contiendra la liste des règles de décomposition à appliquer pour obtenir des problèmes triviaux (problèmes dits terminaux par la suite).

L'algorithme BSH (Backtrack Search dans un Hypergraphe) est un algorithme de recherche aveugle permettant de faire une recherche dans un graphe ET/OU (hypergraphe particulier) **sans circuit** issu de la décomposition d'un problème.

Cet algorithme est fourni ci-dessous. Il n'exploite pas de fonction de coût. Pour borner l'espace exploré, il utilise un majorant sur le rang des états, noté rg(u). Si le rang des états explorés est supérieur à un Seuil, BSH retourne « Echec ».

Dans cet algorithme :

- RESOLUS est l'ensemble (1) des états terminaux, et (2) des états u tels qu'il existe un connecteur $S_i(u)$ dont tous les successeurs v sont dans RESOLUS ;
- INSOLUBLES est l'ensemble (1) des états non terminaux sans successeur, et (2) des états u tels que pour tout connecteur $S_i(u)$ il existe un successeur v qui est dans INSOLUBLE.

```
BSH(u) Backtrack Search dans un Hypergraphe
1.    Si u terminal Alors Retourner « Succès »
2.    Si aucune règle de décomposition n'est applicable en u ou si rg(u) > Seuil
      Alors Retourner « Echec »
3.    Itérer sur les règles de décomposition i applicables en u
3.1.    Flag ← vrai
3.2.    Tant que Flag, itérer sur les nœuds v, successeurs de u en lesquels la règle i décompose u
            Si v ∉ RESOLUS Alors faire :
                    Si v ∈ INSOLUBLES Alors Flag ← faux
                    Sinon faire :
                            rg(v) ← rg(u) + 1
                            Si BSH(v) = « Echec » Alors faire :
                                    Mettre v dans INSOLUBLES
                                    Flag ← faux
                            Sinon mettre v dans RESOLUS
            Fin Itération 3.2
3.3     Si Flag Alors faire :
            Mettre u dans RESOLUS
            règle(u) ← i
            Retourner « Succès »
        Fin Itération 3
4.    Mettre u dans INSOLUBLES
5.    Retourner « Echec »
```

Pour comprendre cet algorithme, nous allons l'appliquer à un problème qui serait décomposé selon les règles de décomposition suivantes :

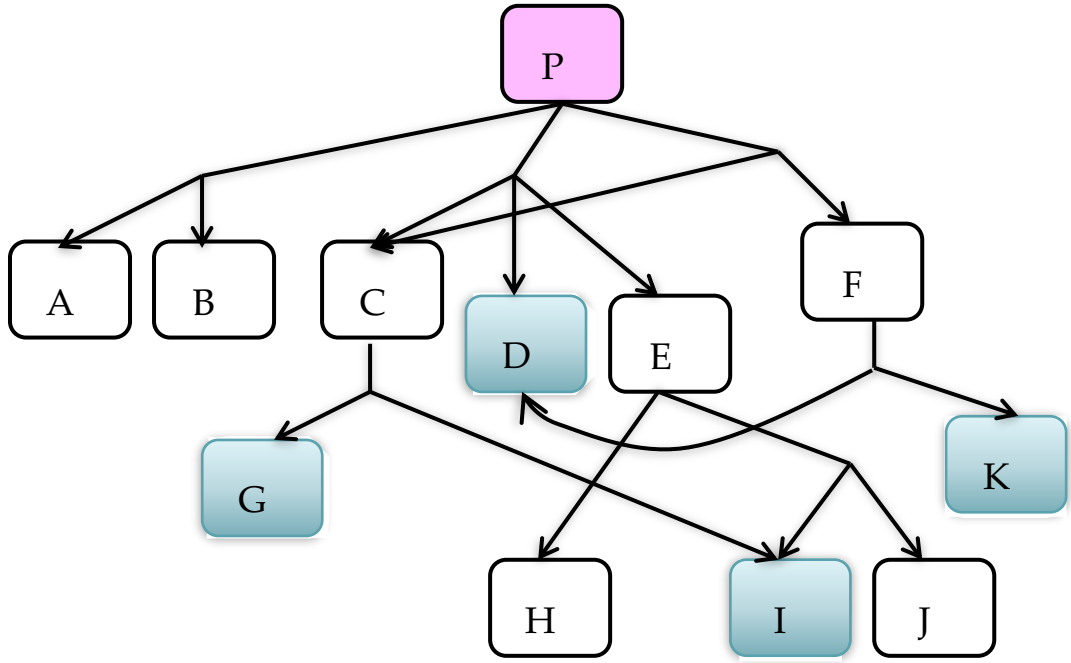
R1 : P → A, B R2 : P → C, D, E R3 : C → G, I R4 : E → H R5 : P → C, F R6 : E → I, J R7 : F → D, K	Les problèmes terminaux sont : G, D, I, K Le problème à résoudre est : P

Dessinez le graphe/arbre de décomposition en considérant les règles dans l'ordre croissant et les sous-problèmes de « gauche à droite ».

Puis **faites « tourner à la main » l'algorithme BSH** en traçant les structures et variables importantes.

========== Indices de correction ==========

Construire simplement l'arbre correspondant (cf. figure ci-dessous). Ensuite, on peut voir le déroulement de l'algorithme avec cet exemple.



Quand on fait tourner BSH(d) à la main :

On a les variables suivantes :

u = P (au début le problème à résoudre), RESOLU = {}, INSOLUBLE = {} , rg(u) = rg (P) = 0

Étape 3 : Les règles applicables en u : R1, R2, R5

On commence par R1 : les successeurs v sont A, B

Étape 3.2 : A n'est pas dans RESOLU, A n'est pas dans INSOLUBLE donc rg(A) = 1, et on lance BSH(A)

- BSH(A)
- u = A non terminal et sans règle de décomposition, donc échec

On met A dans INSOLUBLE = {A}, Flag <- faux

Retour à 3.2, le flag est faux donc on sort du tantque, on revient à l'étape 3, sur la règle R2

Les successeurs sont C, D, E et on remet le flag à vrai

Étape 3.2 : C n'est pas dans RESOLU, C n'est pas dans INSOLUBLE donc rg(C) = 1, et on lance BSH(C)

- BSH(C)
- u = C non terminal et décomposable avec R3 en G et I
- on prend v = G, G n'est pas dans RESOLU, G n'est pas dans INSOLUBLE donc rg(G) = 2, et on lance BSH(G)
    - BSH(G)
    - G est terminal donc succès
- on prend v = I, I n'est pas dans RESOLU, I n'est pas dans INSOLUBLE donc rg(I) = 2, et on lance BSH(I)

- BSH(I)
- I est terminal donc succès

- on a fini les successeurs et le flag est toujours à vrai, donc on met C dans RESOLU = {C}, on sauvegarde la règle R3 et on retourne succès

Retour à l'étape 3.2 avec le successeur D, qui est terminal donc on l'ajoute à RESOLU = {C, D}

Retour à l'étape 3.2 avec le successeur E qui va nous mener à un échec, à mettre E dans INSOLUBLE = {A, E} et à retourner échec pour la règle R2.

On recommence sur la règle R5 :

C est dans RESOLU et F sera résolu via D et K

donc l'algo retournera Succès avec R5, R3 et R7.

==========

PROBLEME DE PLANIFICATION : LE SINGE ET LES BANANES

Le problème du singe et des bananes fait intervenir un singe dans un laboratoire et des bananes qui pendent au plafond, hors de portée de l'animal, qui peut cependant grimper sur une caisse pour atteindre les fruits.

Au départ, le singe se trouve au point A, les bananes au point B et la caisse au point C. Le singe et la caisse ont une hauteur de 1 mètre. Le plafond est à 2 mètres de haut. Le singe peut aller d'un point à un autre, déplacer un objet d'un point à un autre, grimper sur un objet ou en descendre, saisir ou lâcher un objet. Pour saisir un objet, le singe doit être sur le même emplacement que l'objet et à la même hauteur.

Le singe veut tromper les chercheurs pendant qu'ils sont allés boire un café. Il veut saisir les bananes tout en laissant la caisse à l'emplacement initial.

MODELISATION DU PROBLEME

Nous allons dans un premier temps modéliser ce problème. **Proposez une représentation d'un état.**

Proposez des règles de décomposition pour passer d'un état à l'autre. Pour répondre à cette question, tentez informellement de faire les opérations nécessaires et de mettre au point le test de satisfaction de but atteint.

========== Indices de correction ==========

Pour modéliser ce problème, il faut faire comme pour les tours de Hanoï vu en cours.

On modélise le problème avec le canevas (Etat, Etat initial, Test de But, Opérateur) où un état représente la situation à un moment donné et les opérateurs représentent les actions.

Ensuite, on prend l'approche de résolution par décomposition et on change de point de vue : un état va rassembler l'ensemble du problème « Ei (une situation donnée), un opérateur (par exemple se déplacer), Ej (la situation résultante) ».

Modélisation (pas la seule possible) :

État :

Position(Singe,X) avec X dans {A,B,C}
Position(Bananes,X) avec X dans {A,B,C}
Position(Caisse,X) avec X dans {A,B,C}
Hauteur(Singe,H) avec H un entier positif
Hauteur(Bananes,H) avec H un entier positif
Hauteur(Caisse,H) avec H un entier positif
Possede(Singe,Banane) : vrai ou faux

État initial :

Position(Singe, A)
Position(Bananes, B)
Position(Caisse, C)
Hauteur(Singe, 1)
Hauteur(Bananes, 2)
Hauteur(Caisse, 1)
Possede(Singe,Banane) : faux

État final :

Possede(Singe,Banane)
Position(Singe,X) et Position(Bananes,X) avec X différent de C
Position(Caisse,C)

Opérateurs :

Op1 : Aller(P1,P2) pour « Le singe peut aller d'un point à un autre »

- Préconditions : Position(Singe,P1)
- Fonction successeur : modification de Position(Singe,P2)

Op2 : Deplacer(Caisse, P1,P2) pour « Le singe peut déplacer un objet d'un point à un autre »

- Préconditions : Position(Singe,P1) et Hauteur(Singe,1) et Position(Caisse,P1) et Hauteur(Caisse,1)
- Fonction successeur : modification de Position(Singe,P2) et Position(Caisse,P2)

Op3 : Saisir(Banane) pour « Le singe peut saisir un objet »

- Préconditions : Position(Singe,P) et Position(Banane,P) et Hauteur(Singe,H) et Hauteur(Banane,H)
- Fonction successeur : modification de Possede(Singe,Banane) = vrai

Op4 : Lâcher(O) pour « Le singe peut lâcher un objet »

- Préconditions : Possede(Singe, O)
- Fonction successeur : modification de Position(O,P) où P celui de Position(Singe,P) et de Hauteur(O,H) où H est celui de Hauteur(Singe,H)

Op5 : Grimper(Caisse) pour « Le singe peut grimper sur un objet »

- Préconditions : Position(Singe,P) et Position(Caisse,P) et Hauteur(Singe,H) et Hauteur(Caisse,H)
- Fonction successeur : modification de Hauteur(Singe, 2*H)

Op6 : Descendre(Caisse) pour « Le singe peut descendre d'un objet »

- Préconditions : Position(Singe,P) et Position(Caisse,P) et Hauteur(Singe,H1) et Hauteur(Caisse,H2)
- Fonction successeur : modification de Hauteur(Singe, H1-H2)

Le plan obtenu (pas demandé dans le sujet) lors de la résolution est :

[aller(a, b), aller(b, c), deplacer(c, b), grimper(b), saisir(b), descendre(b), deplacer(b, c), aller(c, a)]

==========

GRAPHE ET/OU DE RESOLUTION

Sur la base de la modélisation que vous venez de proposer, **construisez un graphe ET/ OU** de résolution correspondant (partiel).

========== Indices de correction ==========

Pour résoudre le problème, il faut résoudre les éléments de l'état but : Possede(Singe,Banane), puis Position(Caisse,C), puis Position(Singe,X) et Position(Bananes,X) avec X différent de C.

Ces sous-problèmes sont de mêmes types, seules les descriptions des états initial et final changent.

Etc.

Attention : nous avons les mêmes effets de bord que dans le monde des blocs vu en cours

==========