PROMOTION 2011
ANNÉE 3
PÉRIODE 2

INF580

PROGRAMMATION PAR CONTRAINTES ET PROGRAMMATION MATHÉMATIQUE

ECOLE POLYTECHNIQUE — CHRISTOPH DÜRR
FEBRUARY 13, 2015

Table des matières

1
Problèmes de satisfaction de contraintes

Merci à Grégoire Spiers pour avoir pris une première version de ces notes de cours.

La programmation par contraintes, est un modèle de problèmes très général, qui capte des problèmes NP-difficiles.

Une instance d'un CSP (problème de satisfaction de contraintes) consiste en

— un nombre fini de variables, chacune avec un domaine fini (en général). Disons $X_1, \ldots, X_n$ sont les variables et $D_1, \ldots, D_n$ les domaines respectifs.

— un nombre fini de contraintes, chacune est spécifiée par une portée S_i qui est une séquence de variables $X_{i_1}, \ldots, X_{i_r}$ et une relation $R_i \subseteq D_{i_1} \times \cdots \times D_{i_r}$. L'arité de la contrainte est r.

Le but est de trouver une assignation aux variables $\overline{a} \subseteq D_1 \times \cdots \times D_n$ qui satisfasse toutes les contraintes, donc pour tout i on ait $\overline{a}[S_i] \in R_i$, où $\overline{a}[S_i]$ est la projection de l'assignation aux variables spécifiées par S_i.

En pratique une relation pourrait soit être représentée par l'ensemble des tuples qu'elle contient, soit par une expression qui la décrit de manière compacte. Les contraintes unaires jouent un rôle un peu à part, car elles peuvent être codées directement dans les domaines de variables.

Exemples

Sudoku 81 variables de domaine $1, \ldots, 9$, des contraintes unaires de type $X_i = v$ pour les affectations initiales données, et des contraintes binaires $X_i \neq X_j$ pour tout couple de cases $i \neq j$, telles que i et j soient dans une même ligne, colonne ou un même bloc.

Mots croisés Une modélisation possible, associe une variable par case, le domaine est tout simplement l'alphabet, et il a une contrainte par segment (cases continues dans une même colonne ou même ligne), forçant le mot constitué par les valeurs de ses cases à être dans le dictionnaire.

Coloration de cartes Une variable par région, le domaine est l'ensemble des couleurs. Des contraintes binaires forcent les régions adjacents à avoir des couleurs distincts.

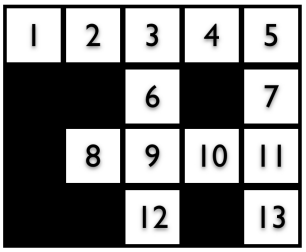


FIGURE 1.1: Un problème de mots croisés

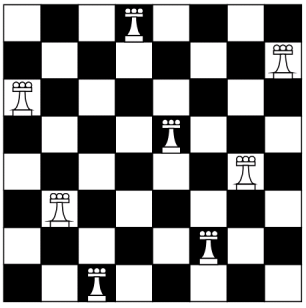


FIGURE 1.2: Une solution au problème de n reines ((c) algospot.com)

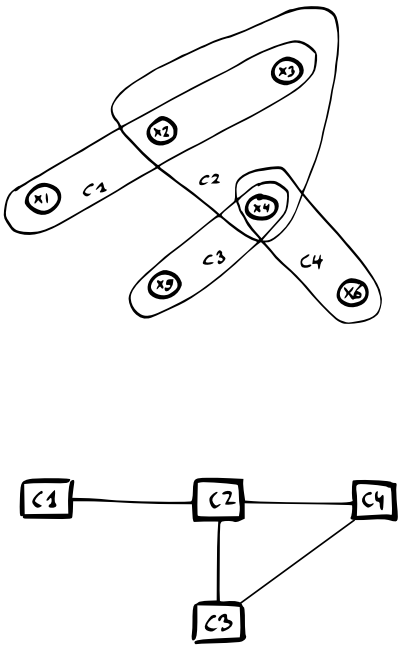


FIGURE 1.3: Le réseau associé à une instance de CSP est en général un hypergraphe. Mais le réseau associé à l'instance dual est un graphe, car toutes les contraintes sont binaires.

n reines Une variable par ligne, indiquant la colonne où se trouve la reine de cette ligne, puis pour chaque couple de variables distincts, une contrainte indiquant que les reines ne soient ni dans la même colonne, ni dans la même diagonale ni dans la même anti-diagonale. C'est un problème scolaire pour les CSP, mais il est facile à résoudre. Il existe une description explicite des solutions étant donnée n, et la recherche locale résout ce problème plus efficacement qu'une recherche avec backtracking. Ceci est dû aux contraintes de ce problème qui sont très permissives et autorisent un grand nombre de solutions.

Graphe

On associe un graphe primal à une instance de CSP. Les sommets sont les variables et les hyperarêtes sont les contraintes, reliant les variables de leur porté. Si les contraintes sont toutes binaires, on parle alors d'arêtes.

À ce graphe on peut associer un graphe dual, dont les sommets sont les contraintes, et il y a une arête entre deux contraintes quand leur portés partagent une variable. Ce graphe correspond à une formulation duale du CSP, voir figure 1.3.

variables duales sont les contraintes primales du CSP. Le domaine de la variable associée à la contrainte $\langle S_i, R_i \rangle$ est tout simplement R_i, vu comme ensemble de tuples.

contraintes duales relient deux variables duales associées à des contraintes $\langle S_i, R_i \rangle$ et $\langle S_j, R_j \rangle$ telles que $S_i \cap S_j \neq \emptyset$ et n'accepte que les affectations $a_i \in R_i, a_j \in R_j$, tel que $a_i[x] = a_j[x]$ pour $x = S_i \cap S_j$. — par abus de notation on a interprété les portés comme des ensembles de variables.

La formulation duale est équivalente à la formulation primale, dans le sens qu'il existe une bijection entre les solutions, et donc on peut transformer tout CSP en un CSP équivalent où toutes les contraintes sont binaires. Donc sans perte de généralité, on va se restreindre dans la suite à des contraintes unaires et binaires.

En particulier pour le problème des mots croisés, on a alors une formulation équivalente : une variable par segment, le domaine est le dictionnaire. Contraintes unaires, forçant les mots à avoir la bonne longueur, et des contraintes binaires pour chaque couple de segments qui partagent une case, forçant les mots à avoir la même lettre dans la case de l'intersection. Cette formulation est par contre moins efficace pour la résolution. Comme le domaine des variables est tellement grand, on aura un très grand arbre de recherche.

Recherche avec backtracking

L'idée est qu'on recherche une solution en explorant un arbre de recherche. À tout moment, il y a des variables instanciées à une valeur de leur domaine et des variables encore libres. On veut pré-

server l'invariant que les contraintes concernant que des variables instanciées soient satisfaits. Pour explorer l'arbre on choisit une variable libre, puis pour chaque valeur dans son domaine, on tente de l'instancier, si les contraintes concernées sont respectées. Dans ce cas on procède à un appel récursif. La procédure de recherche principale pourrait alors avoir la forme suivante (ici en Python).

```
def solve():
    if solved():
        return True
    i = selectVar()
    dom = var[i] #save domain....
    for v in dom:
        if consistant1(i,v):
            var[i] = [v]
            if solve():
                return True
    var[i] = dom #...restore domain
    return False #backtrack
```

Pour illustration, en figure 1.4 l'arbre des appels récursifs de cette procédure pour le problème des 5 reines.

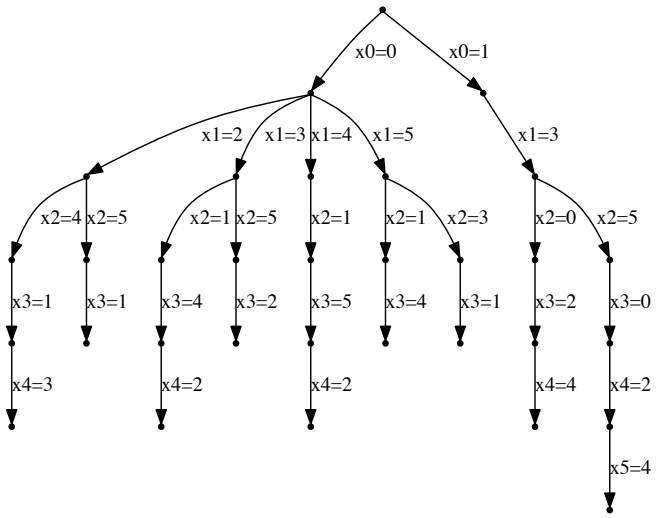


FIGURE 1.4: Un arbre de recherche pour le problème des 6 reines

Heuristiques

La procédure décrite ci-haut est sensible à deux choix d'implémentation. Par exemple le choix de la variable de branchement est crucial. La figure 1.4 devrait vous convaincre qu'une recherche passe la plupart de son temps dans des sous-arbres qui n'aboutissent pas à des solutions. Donc si on est dans un cul-de-sac, on aimerait s'en rendre compte le plus rapidement possible. Donc on

aimerait produire des petits arbres. Une possibilité est de choisir la variable dont le domaine est le plus petit. C'est clairement un bon choix quand ce domaine est de taille 0 ou 1. Une autre possibilité est de choisir la variable qui intervient dans le plus grand nombre de contraintes. Le but est toujours d'arriver le plus rapidement à une contradiction.

Mais la procédure de recherche est aussi sensible à l'ordre dans lequel les différentes valeurs du domaine de la variable sont essayés. Dans le cas où la branche ne va pas aboutir à une solution, ce choix est sans importance. Prenez une minute pour vous convaincre. Alors on aimerait tenter d'abord les valeurs qui semblent le moins contraindre l'instance. La taille du support de la valeur peut alors être un indicateur. Le support d'un couple variable, valeur est expliqué plus tard, dans le cadre de l'algorithme AC4.

Consistance de noeuds

Le domaine d'une variable est consistant s'il ne contient pas de valeur qui violerait une contrainte unaire. Une instance de CSP est noeud-consistant si le domaine de toutes les variables est consistant.

Clairement on peut rendre une instance CSP noeud-consistante *avant* d'effectuer la recherche de solution, pendant laquelle les contraintes unaires peuvent être ignorées.

Support

Soit une variable x et une valeur u de son domaine. Pour une contrainte binaire (x, y) on appèle le support pour l'affectation $x := u$, l'ensemble des valeurs v du domaine de y, tel que $x := u, y := v$ est accepté par la contrainte, donc $(u, v) \in R_{x,y}$.

Vérification en avant

Pour éviter d'effectuer des mêmes tests de consistance pendant la recherche par backtrack, on peut maintenir que le domaine des variables libres x soit restreint aux valeurs supportées par les affectations des variables instantiées. L'opération de maintenance s'appèle la *vérification en avant* (forward checking). Elle consiste pour chaque affectation $y := v$ à restreindre le domaine des variables libres x au support, quand x, y sont liées par une contrainte.

Exercice 1 *Trouvez un exemple de CSP où la vérification en avant améliore de manière dramatique le temps de résolution. Trouvez un exemple de CSP où la vérification en avant n'améliore pas sensiblement le temps de résolution. Est-ce que la vérification en avant est utile pour la résolution du problème de n reines ?*

Reconstitution de domaine

Lors d'une recherche avec la vérification en avant, au moment d'un backtrack, quand on annule l'affectation $y := v$, on doit reconstituer le domaine des variables à leur état d'avant l'opération de restriction du domaine. Plusieurs solutions techniques sont envisageables, mais dans tous les cas on doit stocker les variables et valeurs concernées.

Exercice 2 *Concevez une structure de données qui permet de manipuler facilement les domaines : parcourir les valeurs, enlever des valeurs, reconstituer le domaine.*

Arc-consistance

Une instance de CSP est consistante pour l'arc (x, y), si pour toute valeur u du domaine de x, il existe une valeur v du domaine de y, telle que l'affectation $x := u, y := v$ satisfait toutes les contraintes binaires sur (x, y). D'ailleurs, sans perte de généralité il n'y a au plus qu'une contrainte binaire par paire de variables (x, y).

Une instance de CSP est arc-consistante, si pour toute contrainte binaire C, l'instance est consistante pour les arcs (x, y) et (y, x) où x, y sont les variables sur lesquelles C porte. Voir figure 1.5.

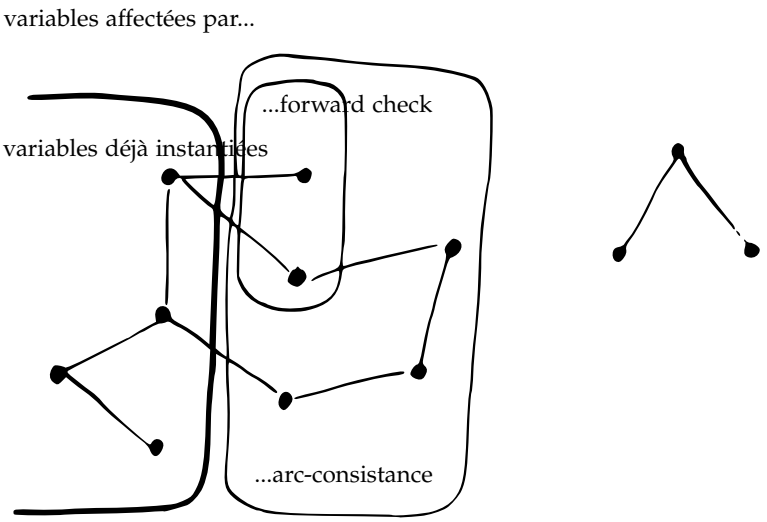


FIGURE 1.5: Les variables affectées par la vérification en avant et par la maintenance d'arc consistance

Exercice 3 *Dans quel cas une recherche avec arc-consistance pourrait être plus intéressante qu'une recherche avec vérification en avant seulement ?*

L'idée d'une recherche avec arc-consistance, est d'établir l'arc consistance *avant* de débuter la recherche, puis de *maintenir* l'arc consistance après chaque affectation.

Procédure Revise

L'ingrédient principal pour maintenir l'arc consistance — ou la vérification en avant — est la procédure revise(x, y), qui rend l'arc

(x, y) consistant, et qui retourne vrai si le domaine de x a été réduit. Elle réduit le domaine de x aux valeurs u, tel qu'il existe une valeur $v \in D_y$ avec $(u, v) \in R_{xy}$.

Algorithme AC1

Voici un premier algorithme pour établir l'arc consistance. Il appèle revise(x, y) pour toutes les paires de variables (x, y) liées par une contrainte binaire, et répète cette boucle tant qu'un des appels retourne vrai.

Algorithme AC3

Une amélioration possible pour la maintenance de l'arc consistance est d'appeler revise(x, y) seulement quand le domaine de y a diminué, en particulier lors d'une affectation de y, quand le domaine a été réduit à un singleton. Cet algorithme travaille avec un ensemble de variables dont le domaine a diminué.

AC3(y) maintient l'arc consistance après une affectation à y. Initialement $Q = \{y\}$. Puis tant que Q n'est pas vide : $y = Q$.dépile(). Pour toute variable x liée à y, appeler revise(x, y). Si l'appel retourne vrai, alors ajouter x à Q s'il n'y est déjà.

Algorithme AC4

Réponse cachée à l'exo 4 :

Plutôt que de vérifier régulièrement qu'une valeur $v \in D_y$ a encore un support dans D_x à chaque fois que D_x a diminué, on peut travailler avec une structure de données plus subtile, qui stocke pour chaque triplet (x, u, y), le support de $x := u$ dans le domaine de y, c'est-à-dire $\{v : v \in D_y, (u, v) \in R_{x,y}\}$. Notons support$[x, u, y]$ cette donnée.

Cet algorithme travaille alors avec un ensemble Q de couples variable-valeur. Initialement pour établir l'arc consistance Q contient tous les couples (x, u) tel qu'il existe une variable y avec support$[x, u, y] = \emptyset$. Alors que pour maintenir l'arc consistance après une affectation $x := u$, la valeur initiale pour Q sera

$$\{(x, w) : w \in D_x, w \neq u\},$$

où D_x est le domaine de x avant l'affectation.

Puis tant que Q n'est pas vide, on extrait un couple (x, u) de Q : Puis on supprime u du domaine de x, et pour chaque variable y et valeur $v \in$ support $[x, u, y]$, on enlève u de support$[y, v, x]$. Si jamais cet ensemble est devenu vide, alors on ajoute (y, v) à Q.

Exercice 4 *Soit t le nombre de contraintes, n le nombre de variables, et m une borne supérieur sur la taille des domaines. Analysez la complexité en pire des cas pour les procédures Revise, AC1, AC3 et AC4.*

Expériences pour arc-consistance, instances aléatoires

Nous allons comparer différentes méthodes de résolution de CSP sur des instances aléatoires, afin de comprendre la sensibilité de telle ou telle méthode à la densité du graphe et à la densité des contraintes. Les trois méthodes comparées sont forward checking, AC3 et AC4. Toutes les méthodes ont été combinées avec l'heuristique *minimum remaining value*.

Pour le modèle probabiliste nous avons fixés 100 variables, chacune avec un domaine de taille 10. Puis deux paramètres déterminent la génération des instances.

densité du graphe c détermine le nombre d'arêtes du graphe, chaque couple (x, y) est choisi avec probabilité c.

densité des contraintes t détermine le nombre de couples de valeurs dans une densité, chaque couple de valeurs (u, v) apparait avec probabilité t dans une relation R_{xy}.

Nous avons générés huit instances aléatoires, et nous nous sommes assurées que les trois méthodes différentes sont comparées sur les mêmes instances.

Résultats

Le tableau 1.1 indique le temps de résolution en milli-secondes arrondi à un chiffre. L'implémentation est en Java.

En général nos instances aléatoires étaient négatives si et seulement $t = 0.9$. Donc on observe bien ici, qu'avec le paramètre t grand, il faut plus de temps pour se rendre compte qu'il n'existe pas de solution. Les instances aux graphes éparses semblent plus difficile quand l'instance est positive et plus facile quand l'instance est négative. C'est surprenant, mais sur ces instances FC domine AC3 qui domine AC4. Peut-être que notre implémentation de la reconstitution de domaine n'est pas très efficace.

TABLE 1.1: Des temps de résolution pour différents solveurs et des instances aléatoires. Ici ∞ représente une valeur supérieure à 30 minutes.

	FC				AC3				AC4			
c\ t	0.1	0.3	0.6	0.9	0.1	0.3	0.6	0.9	0.1	0.3	0.6	0.9
0.1	5	20	90000	50	6	60	20000	80	100	100	400000	400
0.3	4	20	100	100	7	60	100	100	100	300	700	800
0.6	4	40	80	∞	40	60	100	∞	200	300	1000	∞
1.0	5	50	70	10000	50	60	100	100000	300	500	2000	∞

Consistance de chemin

Définitions Deux variables x, y sont chemin consistants avec z si pour toute affectation consistante $x = a$, $y = b$, il existe une valeur c pour z tel que $(x = a, z = c)$ est consistant et $(y = b, z = c)$ aussi. Dans ce cas on dit aussi que la contrainte R_{xy} est consistante par rapport à z. Voir figures 1.6 et 1.7.

Un réseau est *chemin consistant* si toute relation R_{xy} est chemin consistant par rapport à toute variable z.

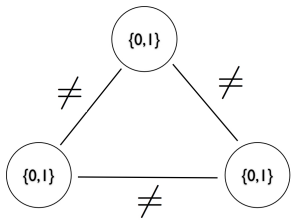


FIGURE 1.6: réseau arc-consistant mais sans solution.

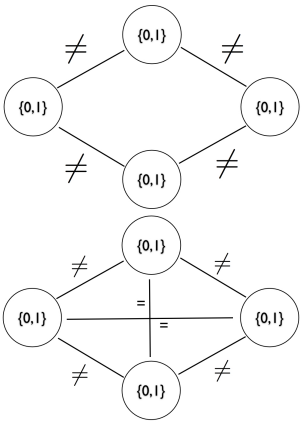


FIGURE 1.7: Le réseaux de droite est le réseau de gauche rendu chemin consistant.

Note Il existe des algorithmes de maintenance de chemin consistance qui sont similaires à AC3() et AC4().

Complexité : $O(ntm^3)$ où n est le nombre de variables, t ("tightness") est une borne supérieure sur le nombre de couples dans toute relation et m est une borne sur la taille des domaines. C'est optimal car il faut $O(ntm^3)$ pour vérifier la chemin consistance.

Définition équivalente R_{xy} est consistant par rapport à un chemin $(x = x_0, x_1, \ldots, x_{l-1}, x_l = y)$ si $\forall (u_0, u_l) \in R_{xy}$ il existe $u_1, \ldots, u_{l-1} \in D_{x_1} \times \cdots \times D_{x_{l-1}}$ tel que $\forall i = 1 \ldots l,\ (u_{i-1}, u_i) \in R_{x_{i-1}, x_i}$

Exercice 5 *Prouver l'équivalence par induction.*

i-consistance

Pour chaque ensemble S de $i-1$ variables et toute variable z, S est i-consistant par rapport à z si pour toute affectation partielle consistante aux variables dans S il existe une valeur $c \in D_z$ tel que affecter en plus $z = c$ satisfait toutes les contraintes sur ces i variables. Et S est fortement i-consistant si elle j-consistant $\forall j \in \{1, \ldots, i\}$.

— 2-consistance = arc consistance

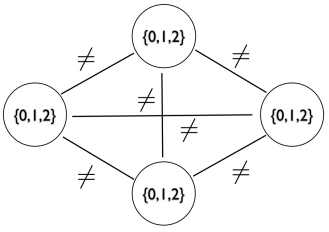
— 3-consistance = chemin consistance

— n-consistance = globalement consistant

Complexité pour maintenir i-consistance : $O((nm)^i)$

Exercice 6 *Montrer qu'un réseau sans cycle et arc consistant est globalement consistant.*

Exercice 7 *Est-ce que le réseau suivant est arc-consistant, chemin-consistant, 4-consistant ?*



2
Un exemple de résolution avec JaCoP

Il existe de nombreux solveurs d'instances CSP, à la fois commercial (IBM ILOG CPLEX CP Optimizer, ...) et académique (AIspace, Choco, JaCoP, ...). Pour ce cours on a choisit d'utiliser la bibliothèque JaCoP, qui a une API en Java, et une bonne documentation. Cette bibliothèque est assez petite (moins de 3 Megaoctets) et fourni les codes sources sous licence GNU, ce qui permet de vérifier comment certaines fonctionalités sont implémentées.

FIGURE 2.1: Le logo de JaCoP

Principe

Un programme qui utilise JaCoP, va d'abord créer un objet de type *Store*, qui va contenir le modèle. Puis on crée des variables et des contraintes les reliant. Finalement on appèle une méthode de résolution et on lit la solution, s'il y en a une.

Les variables sont toutes de domaine fini. Il existe des variables de type entier. Le domaine peut être spécifié lors de la création sous-forme d'intervalle $\{a, a+1, \ldots, b\}$ où a, b seront donnée en argument au constructeur. D'autres intervalles peuvent être ajoutés par la suite, de sorte qu'on peut spécifier tout domaine comme une union d'intervalles. Il existe également des variables de type booléens *BooleanVar* et de type ensemble *SetVar*.

Les contraintes ne sont pas spécifiées de manière explicite par l'ensemble des tuplets dans la relation, mais de manière implicite utilisant un vocabulaire proposé par JaCoP. Celui-ci comporte des opérations arithmétique basiques et des relations d'ordre et des contraintes logiques. Puis il existe des contraintes globales, qui portent sur un grand ensemble de variable, comme la contrainte *Alldifferent* par exemple, qui assure qu'un ensemble de variables ont toutes des valeurs distinctes.

Et finalement la méthode *labelling* de la classe *Store* permet de trouver une affectation consistante aux variables de l'instance générée. Pour cela on peut spécifier des règles précisant les variables choisies pour le branchement à chaque noeud de l'arbre de recherche et l'ordre dans lequel les valeurs de son domaine vont être explorées. JaCoP fourni aussi la possibilité d'énumérer toutes les solution, ou de trouver une solution minimisant une fonction de coût. Cette dernière fonctionnalité est obtenu par résolution succes-

sive avec la contrainte $objValue < treshold$, où treshold est la valeur obtenu par la dernière résolution.

Exemple : Eternity

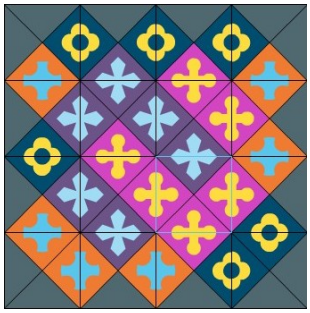


FIGURE 2.2: Une solution pour le jeu Eternity avec une grille 4×4

C'est un jeu de pavage. On dispose d'une grille rectangulaire de 36 cases, et d'un même nombre de tuiles carrées. Chacun des côtés a une couleur spécifique, et deux tuiles adjacencente doivent avoir des couleurs correspondantes sur les côtés adjacentes. Il existe une couleur spécifique qui ne doit apparaître que sur le bord de la grille.

Modélisation

Le principe de la modélisation est assez simple, la difficulté réside dans l'implémentation en JaCoP avec les outils disponible. Clairement on doit spécifier quelle tuile va être mise dans quelle case et avec quelle rotation. Puis les contraintes doivent assurer la correspondance des couleurs des côtés adjacentes. On doit d'abord faire un choix devant deux possibilités.

— Pour chaque tuile il y a des variable indiquant la case où elle est posée et sa rotation.

— Pour chaque case il y a des variables indiquant la tuile qui y est posée et sa rotation.

Comme on doit contraindre la correspondance des couleurs des cases adjacentes, dans la première possibilité on devrait tester pour chaque couple de tuiles s'ils sont adjacentes et dans ce cas si les couleurs correspondent. Ceci génère $O(N^4)$ contraintes où $N \times N$ est la dimension de la grille. Par contre pour la deuxième possiblité, il suffit de faire ce test pour les cases adjacentes, ce qui génère seulement $O(N^2)$ contraintes.

Ensuite il faut décider comment contraindre la correspondance des couleurs. La contrainte de JaCoP qui semble adaptée est *Element*. La contrainte *Element(x,t,y)* est satisfaite par une affectation $x := i, y := v$ si l'élément du tableau t à l'indice i a la valeur v. Ainsi la variable x pourrait représenter une tuile avec sa rotation, et le tableau t pourrait contenir les couleurs des côtés. Pour utiliser cette contrainte on devra alors introduire des variables supplémentaires h et v qui codent les couleurs des côtés des cases (respectivement horizontales et verticales). Les quatres côtés de la case (i, j) sont codés par

— la couleur du bord haut est $h[i, j]$,

— la couleur du bord gauche est $v[i, j]$,

— la couleur du bord bas est $h[i + 1, j]$,

— la couleur du bord droite est $v[i, j + 1]$.

Dans le jeu Eternity le bord de la grille doit être coloré avec la couleur grise (que nous avons codé par 0), et cette couleur ne peut

pas apparaître ailleurs. Cette règle peut être facilement codée dans le domaine des variables h, v, qui sera alors $\{0\}$ pour le bord de la grille et $\{1, \ldots, 7\}$ sinon.

Voyons à présent les détails de la contrainte *Element*. Considérons par exemple le bord haut de la case (i, j). On va créer une contrainte *Element(x,t,h[i,j])*, où x code à la fois la tuile dans la case (i, j) ainsi que sa rotation, et t est un tableau qui code les couleurs des bords des pièces. Une possibilité est d'utiliser les variables suivantes pour cette case

— la pièce sera codée en $p[i, j] \in \{0, \ldots, N^2 - 1\}$,

— la rotation de la pièce en $r[i, j] \in \{0, 1, 2, 3\}$,

— le codage conjoint pièce/rotation en $pr[i, j] \in \{0, \ldots, 4N^2 - 1\}$.

Il nous faut alors la contrainte *Element(pr[i,j],t,h[i,j])*, et les trois autres contraintes similaires pour les bords gauche, bas, et droit. Mais il ne peut pas s'agir du même tableau t pour chaque contrainte, nous avons besoin d'un tableau haut, gauche, bas et droit pour les différents côtés d'une pièce-rotation.

De plus il faut la contrainte $pr[i, j] = 4 * p[i, j] + r[i, j]$, pour forcer le codage des pièces-rotation. JaCoP ne propose que des opérations arithmétiques binaires. Alors il nous faut créer d'abord une variable $p4[i, j]$ qui est forcée à $4p[i, j]$ puis forcer $pr[i, j] = p4[i, j] + r[i, j]$.

Il existe par contre une autre possibilité que je trouve plus élégante, qui se passe de la variable r. Elle consiste en les variables $p[i, j], pr[i, j]$ et la contrainte $p[i, j] = \lfloor pr[i, j]/4 \rfloor$. Par contre on ne peut pas se passer des variables p, car pour s'assurer que chaque pièce est utlisée exactement une fois dans la solution, nous posons la contrainte *AllDifferent(p)*.

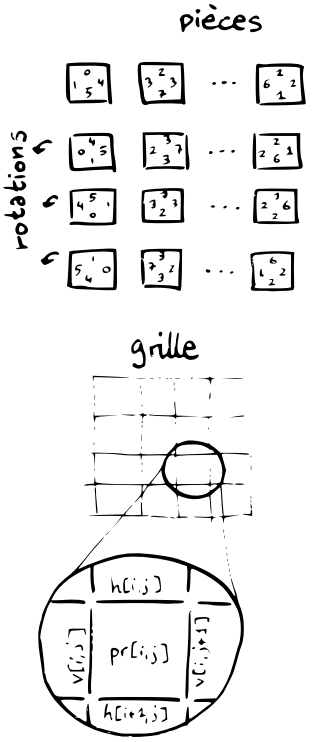


FIGURE 2.3: Notre modélisation. Une variable $pr[i, j]$ pour chaque case (i, j) de la grille code la pièce et sa rotation qui y est posée. Des variables additionnelles h, v codent les couleurs des bords des cases.

Détail

Pour JaCoP les indices des tableaux passés en argument à la contrainte *Element* commencent à 1, alors qu'en Java on aime bien commencer à 0. La documentation ne le dit pas, mais en regardant les sources de JaCoP on trouve qu'on peut passer -1 en 4ème argument pour spécifier un décalage et ainsi débuter les tableaux à l'indice 0.

Les variables h, v qui correspondent au bord de la grille ne sont pas nécessaire à la modélisation. En effet comme le domaine des autres variables de couleur interdit la couleur du bord 0, elle ne peut apparaître qu'au bord. Donc à priori ces variables semblent inutiles. Par contre leur présence dans le modèle, restreint d'avantage les pièces qui peuvent être posées sur le bord de la grille, et permettent ainsi une résolution plus efficace, concrètement générant 1826 noeuds au lieu de 67678.

Il est donc parfois intéressant d'avoir de la redondance dans une modélisation.

Résolution

Lors de la résolution par JaCoP il faut spécifier la liste des variables sur lesquels on veut brancher, la politique avec laquelle la

prochaine variable est choisie en chaque noeud de l'arbre d'exploration et l'ordre avec lequel les valeurs du domaine sont parcourues. Pour le dernier on choisit l'ordre naturel des entiers par simplicité.

Si on donne à JaCoP la liste des toutes les variables et on demande de brancher à chaque fois sur celle qui a le plus petit domaine, alors il branchera probablement sur les variables h et v. Et il semble que ce n'est pas trop efficace, car alors 1826 noeuds sont générés lors de la recherche.

Une autre possibilité est de fournir la liste des variables dans un certain ordre et de demander à JaCoP de choisir les variables dans cet ordre. Si alors on pose les variables pr et p en premier, on obtient une exploration avec seulement 261 noeuds.

Pour obtenir une exploration encore plus efficace on recourt à l'idée suivante. En expérimentant avec JaCoP nous nous sommes rendus compte que la méthode de résolution *labeling* cherche une affectation consistante des variables passées en argument, et s'assure seulement que le reste de l'instance est arc-consistante. Qu'est-ce qui se passerait si on donnait à JaCoP seulement les variables pr. Si les variables pr sont toutes affectées, donc leur domaines réduits à des singletons, alors l'arc-consistance va aussi réduire les domaines des variables p, puis h et v à des singletons. Donc il suffirait d'affecter les variables pr pour chercher une solution. Avec cette modification, et l'heuristique du choix de la variable au plus petit domaine, nous arrivons à une résolution avec 32 noeuds seulement.

3
Tout différent

Beaucoup de problème de satisfaction de contraintes, comme les problèmes scolaires des n-reines, ou Sudoku comportent des contraintes de types $x \neq y$, et plus généralement demandent qu'un ensemble de variables aient tous des valeurs distincts.

Nous avons vu que forcer l'arc consistance sur ces contraintes n'est pas très efficace. En effet tant que les domaines D_x, D_y contiennent tous au moins deux éléments, la contrainte $x \neq y$ *est* arc-consistante. Ceci motive de nouvelles notions de consistance locale.

Consistance locale

Étant donnée une contrainte C, on dit qu'une valeur $u \in D_x$ pour une variable x de la porté de C a un *support de domaine* dans C, s'il existe une affectation partielle $a \in C$, tel que $a[x] = u$, et pour tout autre variable y de la porté de C on ait $a[y] \in D_y$.

Pour le *support d'intervalle* on demande seulement que $\min D_y \leq a[y] \leq \max D_y$.

Avec ces notions on introduit trois types de consistances locale.

consistance de borne Pour chaque contrainte C, et toute variable x de la porté de C, chacune des valeurs $\min D_x, \max D_x$ doit avoir un support d'intervalle dans C.

consistance d'intervalle Pour chaque contrainte C, et toute variable x de la porté de C, chacune des valeurs $u \in D_x$ doit avoir un support d'intervalle dans C.

consistance de domaine Pour chaque contrainte C, et toute variable x de la porté de C, chacune des valeurs $u \in D_x$ doit avoir un support de domaine dans C.

Clairement ces contraintes sont dans l'ordre de la plus faible à la plus forte, et ceci se reflète aussi dans le coût pour établir la consistance. Voici les complexités des meilleurs algorithmes connus à jour à notre connaissance pour établir la consistance locale d'une contrainte *AllDifferent* qui porte sur n variables.

Dans le tableau 3.1 $\text{sort}(n)$ représente la complexité d'un tri des variables sur les bornes de leur domaines. Dans certains cas, on peut alors utiliser le tri par panier et obtenir une complexité linéaire.

TABLE 3.1: Résultats de complexité d'algorithme pour établir la consistance

consistance	complexité	référence
consistance de domaine	$O(n^{2.5})$	[Régin'1994]
consistance d'intervalle	$O(n^2)$	[Leconte'1996]
consistance de borne	$O(n \log n)$	[Puget'1998]
	$O(n + \text{sort}(n))$	[Mehlhorn,Thiel'2000] amélioration
	$O(n + \text{sort}(n))$	[Lopez-Ortiz,Quimper'2003] amélioration pratique

Exemple

Voici le domaine de 6 variables lié par une contrainte *AllDifferent*. En gras les valeurs qui survivent après avoir établi la consistance de borne.

$$
\begin{aligned}
x_1 &\in \{ \quad \quad \quad \mathbf{3}, \ \mathbf{4} \quad \quad \quad \} \\
x_2 &\in \{ \quad \mathbf{2}, \ 3, \ 4 \quad \quad \quad \} \\
x_3 &\in \{ \quad \quad \quad \mathbf{3}, \ \mathbf{4} \quad \quad \quad \} \\
x_4 &\in \{ \quad 2, \ 3, \ 4, \ \mathbf{5} \quad \quad \} \\
x_5 &\in \{ \quad \quad \quad 3, \ 4, \ 5, \ \mathbf{6} \ \} \\
x_6 &\in \{ \ \mathbf{1}, \ 2, \ 3, \ 4, \ 5, \ 6 \ \}
\end{aligned}
$$

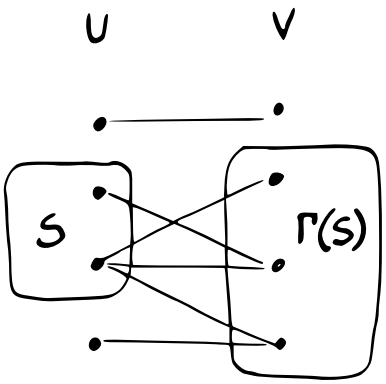


FIGURE 3.1: il existe un couplage parfait de U à V ssi pour tout ensemble $S \subseteq U$ l'ensemble des voisins $\Gamma(S)$ satisfait $|S| \leq |\Gamma(S)|$.

Couplage

L'idée qui se trouve derrière une contrainte $C =$*AllDifferent*$(x_1, \ldots, x_n)$, est qu'une affectation $a \in C$ décrit un couplage dans le graphe bi-parti suivant. D'un côté on a les variables $x_1, \ldots, x_n$ et de l'autre les valeurs $\bigcup D_{x_i}$, avec une arête entre chaque variable x_i et toutes les valeurs de D_{x_i}. Le théorème suivant est fondamental pour les couplages dans les graphes bi-partis.

Théorème 1 (Hall'1935) *Dans un graphe biparti $G(U, V, E)$ il existe un coupage parfait de U vers V si et seulement si*

$$\forall S \subseteq U : |S| \leq |\Gamma(S)|\,, \tag{3.1}$$

où $\Gamma(S)$ note l'ensemble des voisins de S.

Il est clair que cette condition est nécessaire à l'existence d'un couplage. C'est un très joli exercice de montrer que cette condition est aussi suffisante, voir figure 3.1.

Preuve : La preuve est par induction sur $|U|$. Supposons que la condition (3.1) est satisfaite. Quand U est un singleton u, par la condition (3.1) il existe au moins une arête (u, v), qui forme alors un couplage parfait. Nous allons montrer qu'il existe un couplage parfait pour $G(U, V, E)$, sous l'hypothèse que le graphe restreint à U' pour $U' \subsetneq U$ admet un couplage parfait.

Soit $u \in U$ un sommet arbitraire. Deux cas de figures sont à considérer. Si $\forall S \subseteq U \backslash \{u\} : |S| < |\Gamma(S)|$, alors soit $v \in V$ un voisin arbitraire de u. Le graphe amputé de u, v satisfait (3.1), car la partie droite est au pire décrémenté de 1 par la suppression de v, et il y avait de la marge. L'hypothèse d'induction garantie un couplage parfait pour le graphe restreint, auquel il suffit d'ajouter l'arête (u, v) pour obtenir un couplage parfait pour le graphe initial.

Dans le cas contraire il existe un ensemble $S \subseteq U \backslash \{u\}$ avec $|S| = |\Gamma(S)|$. Soit S un tel ensemble de cardinalité minimale. Par hypothèse d'induction il existe un couplage parfait dans le graphe restreint à S. Il suffit de montrer que c'est aussi le cas pour le graphe amputé de S et de $\Gamma(S)$ – noté G' – car l'union de ces couplages serait un couplage parfait pour le graphe initial. Soit S' un sous-ensemble de $U \backslash S$. Puisque $S \cup S'$ satisfait la condition de Hall, le voisinage de S' a au moins $|S'|$ éléments en dehors de $\Gamma(S)$. Ceci montre que G' satisfait également (3.1) et on peut appliquer l'hypothèse d'induction pour conclure la preuve. □

Ensembles de Hall

On appèle les ensembles S où (3.1) est une égalité, les *ensembles de Hall*. Les graphes que nous considérons ont la propriété que le voisinage d'une variable x est un intervalle de valeurs. On parle alors de graphe *d'intervalle*, et chaque ensemble de Hall correspond en fait à un intervalle I, tel que le nombre de variables x avec $D_x \subseteq I$ soit égal à la cardinalité de I. Le principe de tout algorithme de consistance pour les contraintes *AllDifferent* est de détecter les ensembles de Hall S, et de restreindre le domaine des variables y avec $D_y \not\subseteq S$ à $D_y \backslash S$. Ceci est une opération valide, car dans une affectation consistante les valeurs dans S seront tous utilisées par les variables x avec $D_x \subseteq S$. Ainsi les variables y qui n'en font pas partie ne peuvent pas être affectées à des valeurs dans S.

Donc notre objectif maintenant est d'identifier rapidement des intervalles de Hall.

Algorithme de Puget

Comment détecter des ensembles de Hall efficacement ?

Première observation : pour établir la consistance de borne il suffit de restreindre la condition (3.1) à des ensembles S qui soient des intervalles. Et de plus on peut se restreindre à des intervalles de la forme $[\min_i, \max_j]$ pour des variables i, j. Ici, on dénote $\min_i := \min D_{x_i}$ et $\max_i := \max D_{x_i}$.

Comprenez vous pourquoi ?

L'algorithme suivant maintient des compteurs C_k^i qui comptent le nombre de variables $j \leq i$, dont le domaine est inclus dans $[k, \max_i]$. Pour chaque nouvelle variable i, on a alors $C_k^i = C_k^{i-1} + 1$

si $k \leq \min_i$ et $C_k^i = C_k^{i-1}$ sinon.

Data: n variables, avec les bornes de leur domaine $\min_i, \max_i$
Result: détecte les intervalles de Hall
trier les variables, tel que $i < j \Rightarrow \max_i \leq \max_j$;
Poser $\mathcal{T} := \emptyset$;
for *toutes les variables* x_i *dans cet ordre* **do**
  ajouter $\min_i$ à $\mathcal{T}$;
  Mettre à jour les compteurs... ;
  invariant $C_k^i := |\{j \leq i : \min_j \geq k\}|$ pour tout $k \in \mathcal{T}$;
  if $C_k^i = |[k, \max_i]|$ *pour un* $k \in \mathcal{T}$ **then**
    l'intervalle $[k, \max_i]$ est un intervalle de Hall
  end
end

Algorithm 1: Algorithme de Leconte

Puget apporte une amélioration à l'algorithme 1. Il représente ces compteurs par un arbre binaire de recherche. Mais parlons plutôt de l'algorithme d'Ortiz-Lopez. Il repose sur plusieurs observations.

Marges

Soit $K = \{\min_1, \ldots, \min_n\} \cup \{\max_1 + 1, \ldots, \max_n + 1\}$ l'ensemble de valeurs intéressants. Les intervalles de Hall seront toujours de la forme $[j, k)$ pour $j, k \in K$.

L'idée est d'associer à chaque $[j, k)$ une marge, qui initialement est $k - j$. Pour pour chaque $i = 1, \ldots, n$, on décrémente de 1 les marges pour les intervalles $[j, k) \ni \min_i$. À ce moment là, si la marge d'un intervalle $[j, \max_i + 1)$ devient zéro, alors il s'agit d'un intervalle de Hall.

Représentation par un graphe

Pour faciliter ces opérations de décrémentation, nous introduisons un graphe, dont les sommets sont les éléments de K, et chaque sommet – à l'exception du premier – est relié à son prédécesseur direct dans K. Initialement les arcs de la forme $j' \longleftarrow k'$ sont étiquettés par $k' - j'$, de telle sorte que la marge d'un intervalle $[j, k)$ soit la longueur du chemin de k à j.

Avec cette représentation, pour chaque i, il existe un seul arc $j \longleftarrow k$ avec $j \leq \min_i < k$, et il suffit de décrémenter l'étiquette associée. Nous allons maintenir l'invariant que les étiquettes des arcs soient non-négatifs. L'opération clé est la suivante. Quand l'étiquette de l'arc $j \longleftarrow k$ est devenue zéro, si jamais $k = \max_i + 1$, alors $[j, k)$ est un intervalle de Hall. Et dans tous les cas, le sommet k est tout simplement supprimé du graphe. Le sommet ℓ qui pointait sur k, pointe maintenant sur j, et l'étiquette de l'arc reste inchangée.

Dominance

La suppression du sommet k est justifiée par la notion de dominance suivante. Soit $v_k^i = |[k, \max_i + 1)| - C_k^i$, qui représentent la *capacité* de l'intervalle $[k, \max_i + 1)$, qui est 0 si et seulement si l'intervalle est un intervalle de Hall.

Lemme 1 (domination) *Soient $k < \ell$. Si $v_k^i \leq v_\ell^i$ alors $v_k^j \leq v_\ell^j$ pour tout $j > i$.*

Un sommet $k \in K$ est supprimé du graphe lors du traitement de l'intervalle $[\min_i, \max_i + 1)$ si et seulement si v_k^i est dominé.

Chaînage

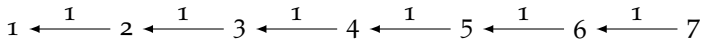
Pour déterminer rapidement l'arc $j \longleftarrow k$ avec $j \leq \min_i < k$, nous ne supprimons pas véritablement les sommet du graphe, mais maintenons une structure de données à la manière de *union-find*, pour relier tous les sommets k supprimés au prochain sommet $\ell \in K$ encore dans le graphe. Ce chaînage est réalisé par un tableau de pointeurs h, qui relie les sommets dominés sous forme d'un arbre au prochain élément de K non-dominé, alors racine de l'arbre.

Au moment de la mise à jour il suffit de décrémenter l'étiquette de l'arc $j \longleftarrow k$. Pour cela on utilise un autre tableau d, tel que $d[k]$ soit l'étiquette de l'arc partant de k.

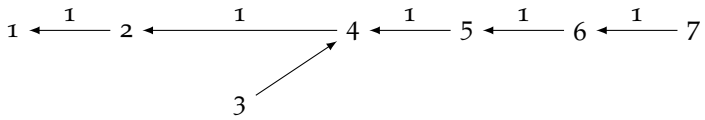
Quand cette étiquette devient zéro, il y a un traitement particulier à faire. D'abord si $k = \max_i + 1$ alors on a identifié $[j, \max_i]$ comme un intervalle de Hall. Donc on peut faire le traitement qui convient à ces intervalles. Puis on va enlever j du chaînage des indices non-dominés, puis fusionner l'arbre sous j avec l'arbre sous k. Ceci se fait par les opérations $h[k] = h[j]; d[k] = d[j]; h[j] = k$.

Exemple

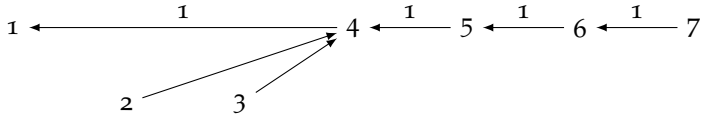
On reprend l'exemple donné en début de ce chapitre. Les sommets de ce graphe sont les éléments de $\mathcal{K}$, les arcs sont les pointeurs définis par h, et les étiquettes les valeurs du tableau d.



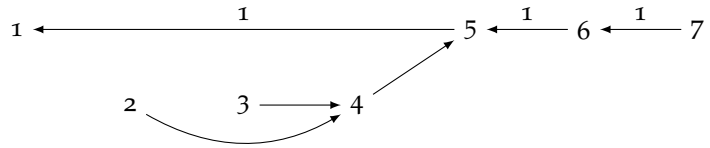
Après avoir traité la variable x_1 avec $\min_1 = 3, \max_1 = 4$, l'arc $(4, 3)$ devient zéro, et 3 est supprimé du graphe. La structure de données devient :



Après avoir traité la variable x_2 avec $\min_2 = 2, \max_2 = 4$, l'arc $(4, 2)$ devient zéro à son tour et la structure de données devient :



Après avoir traité la variable x_3 avec $\min_3 = 3, \max_3 = 4$ la structure de données devient :



L'intervalle $[3, 5)$ est identifié comme un intervalle de Hall.

Détails d'implémentation

On a triché un peu dans la description ci-haut. Bien sûr on ne veut pas que les tableaux h et d soient indicés directement par les valeurs dans $\mathcal{K}$, car sinon la taille des tableau pourrait ne pas être polynomial en n. On va plutôt identifier les éléments de $\mathcal{K}$ par leur rang, et utiliser le rang comme indice dans ces tableaux.

4
Liens dansants

Problème de la couverture exacte

La Rolls-Royce des algorithmes avec backtracking est l'algorithme des liens dansants, qui résout le problème général de couverture exacte. Beaucoup de problèmes se réduisent vers le problème de couverture exacte, et du coup, si vous maîtrisez cet algorithme, vous pourrez résoudre une large classe de problèmes.

Le problème de couverture exacte, consiste en un ensemble de points U, appelé l'univers, et d'une collection de sous-ensembles de U, $\mathcal{S} \subseteq 2^U$. On dit qu'un ensemble $A \subseteq U$ couvre $x \in U$ si $x \in A$. Le but est de trouver une sélection des ensembles de $\mathcal{S}$, c'est-à-dire une collection $\mathcal{S}^* \subseteq \mathcal{S}$ qui couvre chaque élément de l'univers exactement une fois, c'est-à-dire $\forall x \in U, \exists! A \subseteq \mathcal{S}^* : x \in A$.

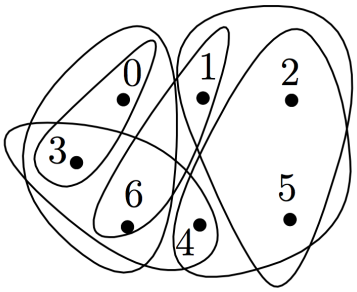


FIGURE 4.1: Une instance du problème de couverture exacte

Concrètement, on peut voir le problème comme cela. En entrée on reçoit une matrice binaire M, dont les colonnes représentent les éléments de l'univers, et les lignes les ensembles de la collection $\mathcal{S}$. Il y a un 1 dans la matrice en colonne x et ligne A si $x \in A$. En sortie il faut produire un ensemble de lignes $\mathcal{S}^*$, tel que la matrice restreinte à $\mathcal{S}^*$ contienne exactement un 1 dans chaque colonne.

Exemple Sudoku

Comment modéliser le problème de Sudoku comme un problème de couverture exacte ? Il y a 4 types de contraintes. Chaque cellule doit avoir une valeur. Dans chaque ligne de la grille du Sudoku, chaque valeur doit apparaître exactement une fois. Idem pour les colonnes et les blocs. Il y a donc quatre types d'éléments dans l'univers : les couples ligne-colonne, les couples ligne-valeur, les couples colonne-valeur et les couples bloc-valeur. Ces éléments constituent les colonnes de la matrice M du problème de couverture exact. Je sais, il faut faire attention, à ne pas confondre les lignes, colonnes de la grille Sudoku avec les lignes, colonnes de M.

Maintenant les lignes de M vont constituer les affectations. Ce sont donc des triplets ligne-colonne-valeur. Chaque affectation couvre exactement quatre éléments de l'univers.

La matrice sera donc assez creuse et on voit qu'il faut la représenter de manière compacte, car sa taille est $16 \cdot 16 \cdot 16 \cdot 4 \cdot 16 \cdot 16$,

de l'ordre du million, ce qui est un peu trop grand pour une implémentation efficace.

Comment coder dans M les valeurs déjà fixées par l'instance du Sudoku donnée ? Une possibilité est d'ajouter une nouvelle ligne i, et une nouvelle colonne j, et de poser $M_{ij} = 1$, ainsi que $M_{ik} = 1$ pour tous les éléments de l'univers couverts par l'instance. Ainsi comme i est la seule ligne qui couvre j, elle devra faire partie de toutes les solutions au problème de couverture exacte, ce qui assure que les solutions seront des extensions de l'instance donnée. Une autre possibilité est d'omettre dans la matrice initiale les colonnes posées par l'instance donnée.

Ce qui est agréable avec cette représentation c'est qu'on fait complètement abstraction des notions de lignes, colonnes, blocs ou valeurs, et on se retrouve devant un problème qui semble plus facile à manipuler.

Détails d'implémentations

Cette section est basée sur une note de Donald Knuth, http ://arxiv.org/abs/cs/0011047

La structure général de l'algorithme est la suivante. Si la matrice M est vide, alors il faut retourner l'ensemble vide, qui est une solution. Sinon, chercher une colonne c de M, avec le moins de 1 possibles. Boucler sur toutes les lignes r qui pourraient couvrir c, c'est-à-dire $M_{rc} = 1$. Pour chaque ligne r effectuer les étapes suivantes. Enlever de M la ligne r ainsi que toutes les colonnes c' couvertes par r ($M_{rc'} = 1$). Si la matrice ainsi obtenue admet une solution S alors retourner $S \cup \{r\}$. Sinon restituer M.

D'abord pour représenter de manière compacte la matrice éparse M, on ne stocke que les cellules M contenant un 1, et on les relie par double chaînage verticalement et horizontalement. Comme ça on peut parcourir très facilement tous les r tel que $M_{rc} = 1$ en utilisant les liens verticaux et tous les c' tel que $M_{rc'} = 1$ en utilisant les liens horizontaux. Chaque cellule dispose donc de 4 champs L, R, U, D pour coder ces double chaînages.

En plus pour chaque colonne on disposera d'une cellule d'entête de cellule, qui fera partie du chaînage vertical, pour avoir un accès à la colonne. Ces entêtes seront stockées dans un tableau *col* indicé par les numéros de colonnes.

Et enfin on disposera d'une cellule particulière `h`, qui fera partie du chaînage horizontal des entêtes de colonnes pour pouvoir les accéder. Cette cellule n'utilisera pas les champs `U`,`D`.

Chaque cellule aura aussi de deux champ supplémentaires `S`,`C`, qui auront une signification différente pour les différentes cellules. Pour les cellules de matrices, ils contiendront les numéros de la ligne `S` et colonne `C` auxquels est associée la cellule. Pour les cellules d'entête de colonne, `S` contiendra le nombre de 1 dans la colonne, et le champ `C` sera ignoré. La cellule `h` ignorera les deux champs.

```
class Cell {
    Cell L,R,U,D; // double chainage vertical U,D, et horizontal L,R
    int S,C; // deux informations, en general indices ligne, colonne

    /* cree une cellule et l'insere evt dans une liste verticale et
       horizontale la cellule sera mise a gauche de left. Si left est
       la cellule la plus a gauche, la cellule sera insere a
       l'extremite droite. Idem pour below.
    */
    Cell(Cell left, Cell below, int _S, int _C) {
        S=_S;
        C=_C;
        if (left==null) {L=R=this;}
        else {
            L = left.L;
            R = left;
            L.R = this;
            R.L = this;
        }
        if (below==null) {U=D=this;}
        else {
            U = below.U;
            D = below;
            U.D = this;
            D.U = this;
        }
    }
```

Les liens

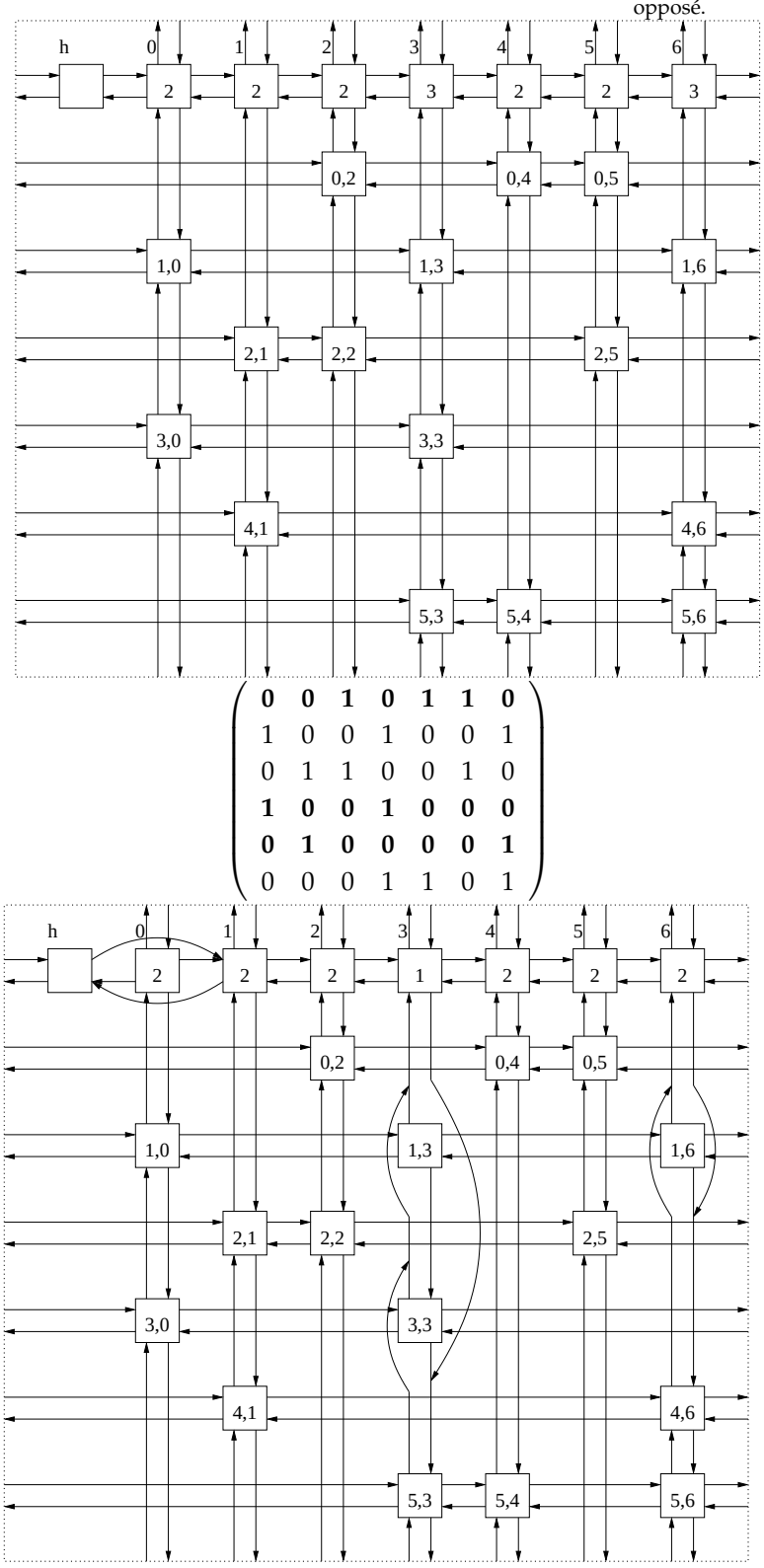
L'idée de l'algorithme des liens dansants, vient d'une observation faite par Hitotumatu et Noshita en 1979 et décrite par Donald Knuth en 2000. Pour enlever un élément d'une liste doublement chaînée, il suffit de faire

```
    void hideVert() {
        U.D = D;
        D.U = U;
    }
```

mais l'élément j est toujours en vie et reste relié à la liste par les liens L,R, il est en fait juste *couvert*. Pour le remettre dans la liste il suffit de faire

```
    void unhideVert() {
        D.U = this;
        U.D = this;
    }
```

FIGURE 4.2: Une matrice binaire M, en haut son codage et en bas le résultat de la couverture de la colonne 0. Les listes sont circulaires, les liens qui quittent un bord du dessin entrent par le bord opposé.



$$\begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

On utilise cette observation pour facilement enlever et remettre une colonne dans la matrice. L'opération de *couvrir* une colonne c, consiste à enlever c de la liste horizontale des entêtes de colonnes et pour chaque ligne r avec $M_{rc} = 1$, d'enlever tous les entrées de la forme $M_{rc'} = 1$. Dans ce cas il faut maintenir à jour le compteur S de l'entête de cellule c'.

```
void cover(int ic) { // i doit etre une entete de colonne
    Cell c=col[ic];
    c.hideHorz();
    for (Cell i=c.D; i!=c; i=i.D)
        for (Cell j=i.R; j!=i; j=j.R) { // -- boucer dans la ligne
            j.hideVert();
            col[j.C].S--; // -- une entree de moins dans cette colonne
        }
}

void uncover(int ic) {
    Cell c=col[ic];
    for (Cell i=c.U; i!=c; i=i.U)
        for (Cell j=i.L; j!=i; j=j.L) { // -- boucer dans la ligne
            col[j.C].S++; // -- une entree de plus de nouveau
            j.unhideVert();
        }
    c.unhideHorz();
}
```

La recherche

Dans la procédure de recherche, on parcourt tout simplement toutes les colonnes pour trouver la colonne c qui minimise $S[c]$. On pourrait peut-être accélérer les choses en utilisant une file de priorité pour les colonnes, mais alors la procédure de couverture de colonne sera plus coûteuse. La fonction suivante écrit la solution dans un tableau `sel` dont les entrées de `0` à `len-1` contiendront la solution, si la fonction retourne vrai.

```
boolean solve() {
    if (h.R==h) // matrice vide, on a trouve une solution
        return true;

    // choisir colonne c de plus petite taille
    int s = Integer.MAX_VALUE;
    Cell c=null;
    for (Cell j=h.R; j!=h; j=j.R)
        if (j.S<s) {
            c = j;
            s = j.S;
        }
```

```
        cover(c.C);
        // essayer chaque ligne
        for (Cell r=c.D; r!=c; r=r.D) {
            sel.push(r.S); // -- selectionner r
            for (Cell j=r.R; j!=r; j=j.R)
                cover(j.C);
            if (solve())
                return true;
            for (Cell j=r.L; j!=r; j=j.L)
                uncover(j.C);
            sel.pop();
        }
        uncover(c.C);

        return false;
    }
```

5
Programmation linéaire

Beaucoup de problèmes combinatoires peuvent se modéliser par un programme linéaire. En opposition à la programmation par contraintes, le domaine des variables est continue et rationnel voir réel, et les contraintes sont toutes des contraintes linéaires. De plus on cherche en général une solution qui maximise ou minimise une fonction linéaire sur les variables.

Résoudre un programme linéaire se fait en temps polynomial. Beaucoup plus de problèmes se modélisent comme un programme linéaire à variables entières (MIP pour *mixed integer linear program*), par contre la résolution est alors NP-dur en général.

Exemple

Cette partie vient du livre *Essentials of Linear Programming*, Jit Chandan, Mehendra Kawatra, Ki Ho Kim

On veut produire deux modèles d'ordinateurs portables. Les variables x_1, x_2 représentent la quantité produite par semaine. Chaque modèle a un profit distinct, et nécessite un temps d'assemblage, un temps de contrôle, et un espace de stockage. Les contraintes sont données par le temps total d'assemblage disponible, le temps total de contrôle disponible et l'espace de stockage disponible. On veut maximiser le profit, sans dépasser les capacités données.

$$\begin{aligned} \max\ & 80x_1 + 40x_2 \\ s.t.\ & 4x_1 + 6x_2 \leq 300 \\ & 2x_1 + 2x_2 \leq 100 \\ & 6x_1 + 4x_2 \leq 280 \\ & x_1, x_2 \geq 0 \end{aligned}$$

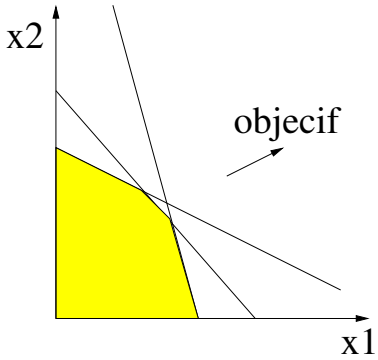


FIGURE 5.1: Un polyèdre

En général un programme linéaire est composé d'un vecteur de n variables x, d'un vecteur objectif c, et d'une matrice A de dimension $m \times n$ et d'un vecteur b de taille m. Et les solutions optimales sont

$$\max\{c^T x : Ax \leq b, x \geq 0\}.$$

Un tel programme linéaire peut

— ne pas avoir de solution,

— avoir des solutions non-bornées,

— avoir des solutions optimaux.

Représentation graphique

Chaque contrainte définit un demi-espace. L'espace des solutions $\{x : Ax \leq b\}$ est l'intersection de ces demi-espaces. C'est un polyèdre. S'il est borné, on l'appèle un polytope. Il est convexe. Son bord est composé de faces, qui sont définies par un ensemble de contraintes, et composées de tous les points qui saturent exactement ces contraintes. Les faces minimaux, sont ceux qui saturent un ensemble maximal (dans le sens de l'inclusion) de contraintes. On les appèle les sommets.

Définition 1 *Une solution x est un sommet du polyèdre P, si $x \in P$ et qu'il n'existe pas de $z \neq 0$ tel que $x + z \in P$ et $x - z \in P$.*

Complexité de la résolution

On connaissait une méthode qui fonctionnait bien en pratique, mais il était ouvert pendant longtemps s'il existe un algorithme de résolution polynomial.

Algorithme du Simplexe

En 1951 Dantzig a trouvé cet algorithme dit du simplexe. Il prend en entrée un sommet du polyèdre. Puis il fait une recherche locale, en effectuant une marche sur le graphe des sommets, améliorant à chaque fois la solution. Il trouve alors une solution optimale, ou affiche que le PL est non-borné. Puis Klee, Minty (1972) et Avi, Chvátal (1978) ont montré que cet algorithme est de complexité exponentielle dans le pire des cas. Borgwardt (1982) a montré que l'algorithme est polynomial en moyenne sur des instances aléatoires. Daniel Spielman et Shang-Hua Teng (2005) ont montré qu'une petite perturbation aléatoire des contraintes, transforme toute instance en une instance sur laquelle l'algorithme du simplexe est de complexité polynomiale.

Méthode ellipsoïde

Une autre méthode alternative a été trouvé par Judin, Nemirovski (1976) et Shor (1977). Elle a été prouvé de complexité polynomiale en 1979 par Khachiyan. Puis les personnes suivantes ont montré comment adapter cette méthode à un oracle de séparation : Grötschel, Lovász, Schrijver (1981), Karp, Papadimitriou (1982) et Padberg, Rao (1981). L'idée est qu'on peut résoudre alors en temps polynomial des programmes linéaires qui ont un nombre exponentiel de contraintes, pourvu qu'on puisse concevoir un oracle de séparation : il prend en entrée un point x, et soit répond en temps polynomial que x satisfait toutes les contraintes (est dans le po-

lyèdre), soit retourne une contrainte du PL violée par x (un plan séparant x du polyèdre).

Méthode du point intérieure

La méthode ellipsoïde est certes prouvé polynomial, mais pas efficace en pratique. Beaucoup d'effort a été fait pour développer la méthode de point intérieur pour résoudre les PL, qui a culminé en 1984 en l'algorithme de Karmarkar.

Dualité

cette section est prise du livre *Approximation Algorithms* par Vijay Vazirani

Considérons le programme linéaire suivant.

$$\begin{aligned} \min\, & 7x_1 + x_2 + 5x_3 \\ s.t.\ & x_1 - x_2 + 3x_3 \geq 10 \\ & 5x_1 + 2x_2 - x_3 \geq 6 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Notons $\mathbf{z}^*$ la solution optimale. Comment se convaincre que $\mathbf{z}^* \leq 30$? Il suffit d'exhiber une solution, par exemple $(x_1, x_2, x_3) = (2, 1, 3)$ et vérifier qu'elle satisfait bien toutes les contraintes. Comme sa valeur est 30, on a montré que la solution optimale est au plus 30.

Bien, mais comment donner des bornes inférieures sur $\mathbf{z}^*$? Par exemple on voit que les coefficients de chaque variable dans la fonction objective dominent les coefficients de la première contrainte. Donc comme les variables sont non-négatives on a

$$7x_1 + x_2 + 5x_3 \geq x_1 - x_2 + 3x_3$$

et donc $\mathbf{z}^* \geq 10$, la partie de droite de cette contrainte. On arrive a une meilleure borne inférieure si on somme ces deux contraintes :

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 16.$$

Et en général on cherche deux coefficients non-négatifs y_1, y_2 pour pondérer ces contraintes, tel que pour chaque variable le coefficient de la fonction objective domine le coefficient de la somme pondérée des contraintes. Trouver le meilleur choix des contraintes revient à résoudre le PL suivant

$$\begin{aligned} \max\, & 10y_1 + 6y_2 \\ s.t.\ & y_1 + 5y \leq 7 \\ & -y_1 + 2y_2 \leq 1 \\ & 3y_1 - y_2 \leq 5 \\ & y_1, y_2 \geq 0 \end{aligned}$$

En général du dual d'un PL primal

$$\min\{c^T x : Ax \geq b, x \geq 0\}$$

est le PL

$$\max\{b^T y : A^T y \leq c, y \geq 0\}.$$

On a les propriétés suivants.

Le dual du dual est de nouveau le programme linéaire primal.

Théorème 2 (dualité) *Le PL primal est sans solution fini ssi le PL dual n'a pas de solution bornée. Le PL primal a un optimum fini ssi le PL dual a un optimum fini. Dans ce cas*

$$\min\{c^T x : Ax \geq b, x \geq 0\} = \max\{b^T y : A^T y \leq c, y \geq 0\}.$$

La version faible du théorème de dualité ne demande que l'inégalité $\min\{\ldots\} \geq \max\{\ldots\}$.

Théorème 3 (complementary slackness) *Soient* $\boldsymbol{x}$,$\boldsymbol{y}$ *des solutions au PL primal et dual respectifs. Alors ils sont optimaux si et seulement si les conditions suivantes sont satisfaites.*

primal complementary slackness condition *Pour tout i, soit la i-ème variable dans* $\boldsymbol{x}$ *est* 0*, soit la i-ème contrainte du PL dual est saturée.*

dual complementary slackness condition *Pour tout j, soit la j-ème variable dans* $\boldsymbol{y}$ *est* 0*, soit la j-ème contrainte du PL primal est saturée.*

Une interprétation physique

Cette interprétation m'a été enseigné par Nikhil Bansal en 2012

Le dual de $\min\{c^T x : Ax \geq b\}$ est $\max\{b^T y : Ay = c, y \geq 0\}$. Considérez l'espace primal, et une particule à une position x dans le polyèdre. On veut minimiser $c^T x$, donc on imagine une force de gravité qui agit sur la particule dans la direction $-c$. Maintenant la physique fait que la particule bouge dans cette direction, peut-être touche le bord du polyèdre, glisse le long du bord, jusqu'à atteindre une position stable où elle ne peut plus tomber plus.

Mais la physique dit que si la particule est stable, alors il y a des forces qui compensent la gravité. D'où viennent ces forces ? Chaque contrainte de la forme $a_i^T x \geq b_i$ (la i-ème ligne de $Ax \geq b$) peut agir dans la direction a_i sur la particule en réponse de la gravité exercé sur elle. Donc il existe des coefficients non-négatifs y_i tel que $-c + \sum y_i a_i = 0$, donc tel que les forces agissant sur x s'annulent.

Le point important est que seules les contraintes saturées par x peuvent agir sur x, car elle correspondent à des faces du polyèdre touché par x. On retrouve ainsi les conditions du *dual complementary slackness*.

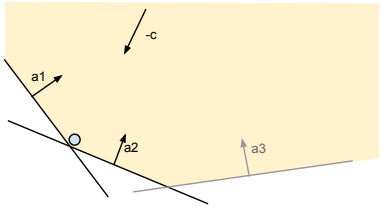


FIGURE 5.2: Une interprétation physique des conditions d'optimalité

Solveurs en pratique

Il existe de nombreux logiciels pour résoudre des programmes linéaires. Par exemple :

— CPLEX de la société ILOG racheté depuis quelques années par IBM. Libre pour le monde académique.

— Gurobi, un concurrent.

— lp_solve, un petit programme sous licence GNU.

— GLPK, une bibliothèque avec une API et un solveur *lpsolve* qui lit des programmes linéaires écrit avec le langage *MathProg*, aussi appelé GMPL et qui est une version GNU du langage commercial AMPL.

Coder des graphes en GMPL

En général on utilise n pour le nombre de sommets, et on identifie les sommets avec les entiers de 1 à n.

par liste d'adjacence

```
param n, integer, > 0;
/* number of vertices */

set V, default 1..n;
/* set of vertices */

set E, within V cross V;
/* set of edges */

set E :=
   (1,*) 3 6 2
   (2,*) 5 4
   (3,*)
   (4,*) 5
   (5,*) ;
```

définir les sommets comme extrémités des arêtes

```
set E, dimen 2;
/* set of edges */

set V := (setof{(i,j) in E} i) union (setof{(i,j) in E} j);
/* set of nodes */

set E := a1 b1, b1 c1, a1 b2, b2 c2, a2 b3, b3 c3, a2 b4, b4 c4, a3 b5,
         b5 c5, a3 b6, b6 c6, a4 b1, a4 b2, a4 b3, a5 b4, a5 b5, a5 b6,
         a6 b1, a6 b2, a6 b3, a6 b4, a7 b2, a7 b3, a7 b4, a7 b5, a7 b6;
```

avec pondération sur les arêtes

```
param n, integer, >= 1;
/* number of nodes */

set V, default {1..n};
/* set of nodes */

set E, within V cross V;
/* set of arcs */

param a{(i,j) in E}, > 0;
/* a[i,j] is capacity of arc (i,j) */

param n := 5;

param : E : a :=
        1 2 14
        1 4 23
        2 3 10
        2 4 9
        3 5 12;
```

Exercices

Exercice 8 *Transformez les programmes linéaires sous la forme* $\max\{c'^T x' : A'x' \leq b', x' \geq 0\}$ *:*

— $\max\{c^T x : Ax \leq b\}$

— $\max\{c^T x : Ax = b\}$

Exercice 9 *Prouvez la version faible du théorème de dualité :*

$$\min\{c^T x : Ax \geq b, x \geq 0\} \geq \max\{b^T y : A^T y \leq c, y \geq 0\}.$$

Exercice 10 *Modélisez le problème de plus court chemin comme un programme linéaire. Ici on nous donne un graphe dirigé* $G(V, A)$*, pondéré sur les arcs* $w : A \to \mathbb{Q}$ *et une source* $s \in V$*. Le poids d'un chemin est la somme du poids des arcs, et pour tout sommet* $v \in V$*, on cherche le chemin de* s *à* v *de plus petit poids. Écrivez le dual de ce programme linéaire.*

Exercice 11 *Soit un graphe dirigé* $G(V, A)$*, avec des capacités sur les arcs* $c : A \to \mathbb{Q}^+$ *et deux sommets* $s, t \in V$*. On veut trouver un flux maximal de* s *à* t *: les flots les long des arcs ne doivent pas dépasser la capacité tout ce qui entre en un sommet* $v \in V \backslash \{s, t\}$ *doit être égal à tout ce qui sort.*

Écrivez le programme linéaire correspondant et son dual.

Linéarisation

Dans certains cas on doit modéliser des expressions non-linéaires. Ces expressions peuvent être la valeur objective ou la partie gauche

d'une contrainte. Les exemples sont multiples. Par exemple les transporteurs proposent des frais de transports dégressifs en fonction du volume à transporter. De nombreux musées font des prix de groupe. La résistance de l'air est quadratique en la vitesse du véhicule.

Une approche couramment utilisée est de représenter l'expression comme une fonction linéaire par morceaux. Soit g une telle fonction, composée de k morceau, tel que pour $i = 1, \ldots, k$, la fonction s'évalue à $c_i x + a_i$ dans l'intervale $[b_{i-1}, b_i)$.

Plusieurs manières ont été proposées dans la littérature pour modéliser une telle fonction.

La méthode de choix multiple

On introduit une variable booléenne y_i par segment $i = 1, \ldots, k$, avec la contrainte $\sum y_i = 1$, pour sélectionner exactement un segment, le segment contenant x. Puis on introduit des variables z, où z_i vaut x pour le segment i contenant x et 0 ailleurs.

Cette modélisation n'est pas très efficace pour la résolution par branchement, car les deux sous-arbres $y_i = 1$ et $y_i = 0$ ne sont pas équilibrés.

La méthode incrementale

Cette fois-ci, la variable z_i indique l'intersection entre le segment $[b_{i-1}, b_i)$ et l'intervale $(-\infty, x)$. Si x est contenu dans le segment i, alors $z_i = x - b_{i-1}$, et les valeurs z des segments précédents sont égal à la taille de leur segment alors que les valeurs z des segments suivants sont à 0. Le rôle des variables y est de sélectionner les segments i qui ont une valeur z_i non-nulle. Donc y soit former une séquence décroissante. Le passage de 1 à 0 indique le segment contenant x.

La méthode par combinaison convexe

Cette méthode utilise le fait qu'à la fois la valeur x mais aussi la valeur $g(x)$ est une combinaison linéaire convexe des extrémité du segment. À la place de z on introduit alors des variables μ, λ par segment, qui sont non-nuls seulement pour le segment contenant x et somment à 1.

La méthode special ordered set

Puisque la méthode de choix multiple est utilisé souvent, certains solveurs ont introduit un mot clé dans le langage de modélisation `SOS`, qui permet d'indiquer un ensemble de variables entières dont la somme ne doit pas dépasser une constante, en général 1 ou 2. Ensuite le solveur applique des méthodes de résolution adapté sur ces variables. Par contre le solveur GLPK n'a pas cette fonction.

6
CSP booléens

Dans cette section on considère des problèmes de satisfaction de contraintes dont le domaine de toutes les variables est {Vrai,Faux}.

Dans ce contexte, on utilise le vocabulaire suivant.

Littéral = soit une variable (x) (littéral positif) soit la négation d'une variable ($\neg x$) (littéral négatif).

Clause = ensemble de littéraux. La clause est satisfaite si au moins un des littéraux s'évalue à vrai.

Instance = ensemble de clauses. L'instance est satisfaite si toutes les clauses sont satisfaites. Est aussi appelé formule.

Exemple :

$$(a \vee \neg b \vee c) \wedge (a \vee b \vee \neg d) \wedge b$$

Types de formules

— 3-SAT : chaque type de clause a au plus 3 littéraux.

— 2-SAT : chaque clause a au plus 2 littéraux.

— HornSAT : chaque clause a au plus un littéral positif.

— HornSAT renommable : il existe un sous ensemble de variables R tel que si on remplace tout $x \in R$ par $\neg x$ et $\neg x$ par x alors le résultat est HornSAT.

— XOR-SAT : chaque clause est de la forme $(l_1 \oplus l_2 \oplus \cdots \oplus l_k)$ pour des littéraux l_i où $\oplus$ est l'opération "ou exclusif".

3-SAT est NP-dur les autres sont polynomiaux.

Exercice 12 *Réduisez k-SAT en temps polynomial à 3-SAT. C'est-à-dire pour une formule F où chaque clause a au plus k littéraux, pour un $k > 3$, construisez une formule 3-SAT F' équivalente de taille polynomiale en la taille de F. L'équivalence est dans le sens qu'il existe une surjection des solutions pour F' vers les solutions pour F.*

Résoudre une formule XOR-SAT

. . . en temps polynomial.

On considère la formule comme un système d'équation linéaire dans $\mathbb{Z}_2$

$$(l_1 \oplus l_2 \oplus \cdots \oplus l_k) = Vrai$$

et on sait résoudre des systèmes d'équation linéaire par exemple par l'élimination de Gauss-Jordan.

Exercice 13 *Quelle est la complexité d'un telle résolution ? Attention la formule donnée en entrée décrit de manière compacte le système.*

Résoudre une formule 2-SAT

. . . en temps linéaire par un algorithme de Tarjan.
Toutes les clauses sont de la forme $(a \vee b) = \neg a \Rightarrow b = \neg b \Rightarrow a$
Le graphe d'implications : $G(V, A)$
V ensemble des littéraux
A ensemble des implications

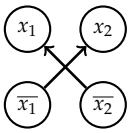
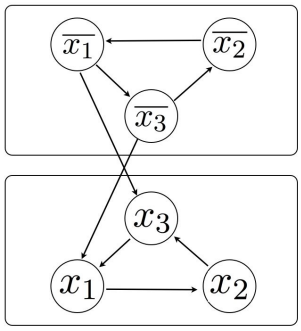


FIGURE 6.1: Arcs générés par une clause $x_1 \vee x_2$

Exemple

$$(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (x_1 \vee x_3)$$



Le graphe se décompose en composantes fortement connexes.
Définition : $S \subset V$ est fortement connexe si $\forall u, v \in S$, $\exists$ chemin dirigé de u à v (et donc aussi de v à u).
Deux observations clé :

1. Tous les littéraux d'une même composante doivent avoir la même valeur.
2. Deux sommets qui font partie d'un même cycle doivent faire partie d'une même composante fortement connexe.
3. Les composantes peuvent être reliées entre eux par des arcs, mais ne peuvent pas former de cycle.
4. Il existe donc toujours une composante sans arc sortant et toujours une composante sans arc entrant.

Théorème 4 *S'il existe une composante C et une variable x tel que $x \in C$, $\neg x \in C$ alors la formule n'est pas satisfiable. Sinon la formule est satisfiable.*

Preuve : Il existe toujours une composante C de degré entrant zéro (aucun arc n'entre en C). Il existe une composante associé $\overline{C}$ qui contient tous les négations des littéraux en C. Et $\overline{C}$ a un degré sortant nul (aucun arc ne quitte $\overline{C}$).

On peut affecter tous les littéraux $l \in C$ à faux et donc $\bar{l} \in \overline{C}$ à vrai. Ceci préserve les clauses (implications) adjacentes à $C, \overline{C}$.

On peut alors enlever (ou ignorer) C et $\overline{C}$ du graphe et recommencer. Pour cela on utilise le lemme suivant. Cette procédure ainsi que le calcul des composantes fortement connexes peut se faire en temps linéaire. □

Résoudre une formule HornSAT

Pour le même prix on va calculer une assignation minimale (nombre de variables posées à vrai) qui satisfait toutes les clauses.

L'algorithme suivant porte le nom *d'unification*. Tant qu'il existe une clause contenant un seul littéral l positif alors

— poser $l = vrai$

— enlever toutes les clauses C avec $l \in C$

— enlever $\neg l$ de toutes les clauses C avec $\neg l \in C$

On distingue 3 types de clauses de Horn

1. y
2. $x_1 \wedge x_2 \wedge \cdots \wedge x_k \Rightarrow y$
3. $\overline{x_1} \vee \overline{x_2} \vee \cdots \vee \overline{x_k}$

Si toutes les clauses sont de la forme 2 ou 3 alors affecter toutes les variables à faux rend la formule satisfiable.

Implémentation en temps linéaire : Classer les clauses par le nombre de littéraux positifs et le nombre de littéraux négatifs, ce qui est un raffinement des trois types ci-haut. L'opération de suppression d'un littéral négatif d'une clause la fait changer de catégorie. La boucle principale est facile à réaliser car elle ne concerne qu'une catégorie. On a aussi besoin d'une structure associant à un littéral toutes les clauses qui le contiennent.

Résoudre une formule HornSAT renommable

Exercice 14 *Réduire HornSAT renommable à 2-SAT.*

Il existe un algorithme en temps linéaire pour résoudre une formule HornSAT renommable, mais il est trop compliqué à expliquer ici.

exemple Voici une instance de HornSAT.

— pluie $\wedge$ dehors $\Rightarrow$ parapluie

— novembre $\Rightarrow$ pluie

— anniversaire $\Rightarrow$ novembre

— anniversaire ⇒ dehors

— tsunami ⇒ pluie

— anniversaire

Si on applique l'algorithme d'unification, alors on se rend compte qu'il faut poser à Vrai *anniversaire, dehors, novembre, pluie, parapluie* (dans cet ordre) et qu'on peut poser à Faux *tsunami*.

7
Résoudre des formules SAT

Les algorithmes exponentiels

Nous allons voir des algorithmes pour résoudre des formules SAT. Ils seront toujours des algorithmes exponentiels, mais l'amélioration peut être importante. Par exemple pour $n = 80$, un algorithme qui effectue 2^n opérations à raison de 10^8 opérations par seconde, mettrait 390 millions d'années, alors qu'un algorithme de complexité $(4/3)^n$ mettrait seulement une minute et demi.

Simplification

Un ingrédient important aux algorithmes de cette section est la simplification d'une formule. Elle consiste simplement en l'application répétée des règles suivants — qui sont plutôt de bon sens.

1. Si une clause contient une variable et à la fois sa négation, alors remplacer cette clause par *Vrai*. Si une clause contient deux fois le même littéral, alors remplacer les deux par un seul.
2. Si les littéraux d'une clause C_1 sont un sous-ensemble des littéraux d'une clause C_2 alors supprimer C_2.
3. Si une clause contient la constante *Faux*, alors supprimer la constante de la clause. Si la clause devient alors vide, alors remplacer la formule entière par *Faux*.
4. Si une clause est un unique littéral x ($\overline{x}$), alors remplacer x par *Vrai* (*Faux*) partout dans la formule.
5. Si la formule contient un littéral mais pas sa négation, alors le replacer par *Vrai* partout dans la formule.

La propriété importante de la procédure de simplification est qu'elle préserve la faisabilité de la formule et qu'elle réduit le nombre de clauses, sinon le nombre de variables. Elle peut être implémentée en temps polynomial en la taille de la formule.

Davis-Putnam

Dans l'algorithme suivant $F|_{x=c}$ représente le résultat du remplacement de chaque occurrence de x par la constante $c \in \{0, 1\}$ dans F. Ici Simplification$_{1-5}(F)$ fait référence aux 5 règles ci-haut.

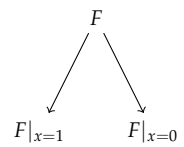


FIGURE 7.1: L'arbre de récursion de l'algorithme de Davis-Putnam

Data: Une formule F en forme normale conjonctive
Result: teste si F est satisfiable
F := Simplification$_{1-5}(F)$;
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
Soit x une variable de F;
if *DavisPutnam*$(F|_{x=1})$ **then**
  retourner Vrai;
else
  retourner DavisPutnam$(F|_{x=0})$
end

Algorithm 2: DavisPutnam

Notation

n nombre de variables

m nombre de clauses

nm borne sur la taille du codage de la formule

Cet algorithme a une complexité de l'ordre de $O(2^n \text{poly}(nm))$. Pour obtenir un meilleur algorithme en terme de n, il faudrait restreindre la taille des clauses, ou d'exprimer la complexité en terme de m, le nombre de clauses. Un ingrédient important est la procédure de simplification que nous décrivons maintenant.

Exercice 15 *Montrer que la complexité de Davis-Putnam est $O(2^m poly(nm))$.*

Résolution

On va ajouter une règle supplémentaire à la procédure de simplification.

6. Soit x une variable apparaissant p fois positif dans F et q fois négatif dans F, avec $p + q \geq pq$. On distingue dans F les clauses contenant x, les clauses contenant $\overline{x}$ et ceux ne contenant ni x ni $\overline{x}$. Alors F a la forme

$$\begin{aligned} &(x \vee C_1) \wedge \ldots \wedge (x \vee C_p) \\ \wedge\ &(\overline{x} \vee D_1) \wedge \ldots \wedge (\overline{x} \vee D_q) \\ \wedge\ &U, \end{aligned}$$

et on la remplace par

$$\begin{aligned} &(C_1 \vee D_1) \wedge \ldots \wedge (C_1 \vee D_q) \\ \wedge\ &\vdots \\ \wedge\ &(C_p \vee D_1) \wedge \ldots \wedge (C_p \vee D_q) \\ \wedge\ &U. \end{aligned}$$

Exercice 16 *La règle de résolution préserve la faisabilité de la formule.*

Exercice 17 *La procédure de simplification avec les règles 1–6, fonctionne en temps polynomial en n, m, nm.*

Théorème 5 *La procédure Davis-Putnam avec les règles de simplification 1–6, a la complexité $O(1,325^m poly(nm))$.*

Preuve : Après simplification, chaque variable de la formule x qui apparait p fois positivement et q fois négativement satisfait $\min\{p,q\} \geq 2$ et $\max\{p,q\} \geq 3$. Donc sur les formules $F|_{x=0}$ et $F|_{x=1}$, une a au plus $m-2$ clauses et l'autre au plus $m-3$ clauses. La complexité satisfait donc la récursion

$$T(m,n) \leq T(m-2,n) + T(m-3,n) + \text{poly}(nm),$$

pour $m > 0$ et $T(0,n) = O(\text{poly}(nm))$ sinon. La preuve est complétée par l'analyse standard des récursions linéaires. □

Récursions linéaires

Voici un outil important pour l'analyse de la complexité des algorithmes de cette section.

Théorème 6 *Si*

$$f(n) \leq a_1 f(n-1) + \ldots + a_k f(n-k),$$

pour tout $n \geq k$ et des constantes réels non-négatives $a_1, \ldots, a_k$, alors $f(n) = O(v^n)$ pour v l'unique racine réelle positive de

$$g(t) = 1 - \frac{a_1}{t} - \ldots - \frac{a_k}{t^k}.$$

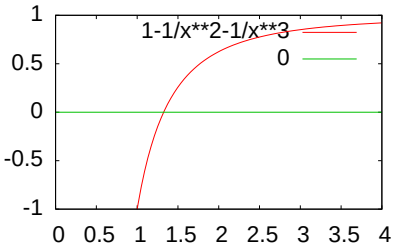


FIGURE 7.2: La racine de $t \mapsto 1 - 1/t^2 - 1/t^3$ est 1,3247 ce qui permet d'affirmer le Théorème 5.

Preuve : La fonction g est monotone croissante, et tend vers $-\infty$ quand t tend vers o et tend vers 1 quand t tend vers $+\infty$. Elle a donc une unique racine positive.

On montre par récurrence que $f(n) \leq cv^n$ pour une constante c. Pour le cas de base on choisit c suffisamment grand afin que ceci soit vrai pour tout $0 \leq n \leq k$.

Maintenant $n > k$ et et on fait l'hypothèse d'induction que $f(i) \leq cv^i$ pour tout $0 \leq i < n$. Alors

$$\begin{aligned} f(n) &\leq a_1 f(n-1) + \ldots + a_k f(n-k) \\ &\leq a_1 cv^{n-1} + \ldots + a_k cv^{n-k} \\ &= cv^n (a_1 v^{-1} + \ldots + a_k v^{-k}) \\ &= cv^n (1 - g(v)) \\ &= cv^n. \end{aligned}$$

□

Backtracking de Monien, Speckenmeyer

Pour améliorer la procédure Davis-Putnam, on introduit une opération supplémentaire. Une affectation associe à chaque variable

de la formule une valeur booléenne. L'affectation *majoritaire* associe à une variable x la valeur *Vrai* si x apparaît plus souvent dans la formule que $\overline{x}$, et associe la valeur *Faux* dans le cas contraire. Quelles sont les clauses qui ne sont *pas* validées par l'affectation majoritaires ? Mais tout simplement les clauses dont tous les littéraux sont dans leur forme minoritaire. Ces clauses vont jouer un rôle important dans la suite.

$$\begin{aligned} F =& (x \vee y \vee \overline{z}) \\ \wedge& (x \vee \overline{y} \vee z) \\ \wedge& (\overline{x} \vee y \vee z) \\ \wedge& (\overline{x} \vee \overline{y} \vee z) \\ \wedge& (\overline{x} \vee \overline{y} \vee \overline{z}) \end{aligned}$$

FIGURE 7.3: L'affectation majoritaire est $x = 0, y = 0, z = 1$ qui ne satisfait pas la première clause. Les variables de cette clause sont permises. La formule $F|_{x=T}$ n'a que 2 clauses de moins que F, mais heureusement la règle 6 (résolution) appliquée à la variable y va diminuer strictement le nombre de clauses, car elle n'apparaît qu'une fois positivement dans $F|_{x=1}$ Elle n'est pas bien faite la vie ?

La procédure de Monien Speckmeyer commence par tester si l'affectation majoritaire valide la formule. Mais elle fait plus encore. Elle contrôle aussi le choix de la variable choisie pour brancher. Le but est de choisir une variable qui permet lors d'une récursion de diminuer le nombre de clauses d'au moins 3.

Data: Une formule SAT F sur n variables
Result: teste si F est satisfiable
F := Simplification$_{1-6}(F)$;
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
if *l'affectation majoritaire valide F* **then**
  annoncer la valeur *Vrai*;
end
S'il existe une variable x qui apparaît au moins 3 fois positivement et au moins 3 fois négativement, alors choisir cette variable. Dans le cas échéant toutes les variables apparaissent 2 fois dans une forme et au moins 3 fois dans la forme opposée. Comme l'affectation majoritaire n'a pas validé toutes les clauses, il existe des clauses contentant seulement des variables dans leur forme minoritaire. Soit x une telle variable.;
if *MonienSpeckenmeyer*$(F|_{x=1})$ **then**
  retourner Vrai;
else
  retourner MonienSpeckenmeyer$(F|_{x=0})$
end

Algorithm 3: Monien-Speckenmeyer

Théorème 7 *La complexité de l'algorithme de Monien-Speckenmeyer est* $O(2^{m/3}poly(nm)) = O(1.26^m poly(nm))$.

Preuve : On va tout simplement monter que la complexité satisfait la récurrence

$$T(m, n) \leq T(m-3, n) + T(m-3, n) + \text{poly}(nm).$$

Le seul cas qui pose problème est quand toutes les variables apparaissent 2 fois dans une forme et au moins 3 fois dans la forme opposé. Soit x la variable choisie, et C une clause contenant x dans sa forme minoritaire, disons $\bar{x}$ (le cas $x \in C$ est symétrique). Alors par la règle 4, C contient aussi une autre variable y, et dans sa forme minoritaire. On pourrait penser que lors du branchement sur $F|_{x=0}$, seul les deux clauses contenant $\bar{x}$ disparaissent. Mais la cette formule y apparaît désormais 0 ou 1 fois dans sa forme minoritaire et

la règle 5 ou 6 diminue strictement le nombre de clauses. En résultat les simplifications des formules $F|_{x=0}$ et $F|_{x=1}$ contiennent au moins 3 clauses de moins que F. □

8
Résoudre des formules 3-SAT

On a vu la semaine dernière que pour espérer des algorithmes qui résolvent SAT dans un temps meilleur que $\Omega(2^n)$, il faudrait restreindre la taille des clauses. C'est ce que nous faisons dans ce cours, qui présente des algorithmes pour 3-SAT, les formules donc chaque clauses contient au plus 3 littéraux.

Le tableau 8.1 résume la complexité de certains algorithmes de résolution de 3-SAT, illustrant ainsi la course que se font les chercheurs depuis des décennies sur ce problème. Ici on note $g(n) \in O^*(f(n))$ s'il existe une constante c tel que $g(n) \in O(f(n)\log^c f(n))$. Ainsi dans l'étoile on cache des facteurs polynomiaux de la complexité exponentielle.

Algorithmes déterministes	
$O^*(2^n)$	recherche exhaustive
$O^*(1.618^n)$	Monien,Speckenmeyer (1985)
$O^*(1.505^n)$	Kullmann (1999)
$O^*(1.481^n)$	Dantsin, Goerdt, Hirsch, Schöning (2000)
$O^*(1.439^n)$	Kutzkov, Scheder (2010)
$O^*(1.333^n)$	Moser, Scheder (2011)
Algorithmes probabilistes	
$O^*(1.587^n)$	Paturi, Pudlák, Zane (1997)
$O^*(1.5^n)$	Schöning (1998)
$O^*(1.447^n)$	Paturi, Pudlák, Saks, Zane (1998)
$O^*(1.333^n)$	Schöning (1999)
$O^*(1.32113^n)$	Iwama, Seto, Takai, Tamaki (2010)
$O^*(1.32065^n)$	Hertli, Moser, Scheder (2011)

TABLE 8.1: Complexité de certains algorithmes pour 3-SAT.

Algorithmes probabilistes

Pour ce cours on distingue les algorithmes déterministes pour 3-SAT, des algorithmes probabilistes. Les algorithmes probabilistes de ce cours, tournent en temps polynomial en n et soit affichent *échec*, soit trouvent une assignation satisfaisant la formule. L'analyse de l'algorithme va montrer que dans le cas où la formule est satisfiable, l'algorithme succède avec une petite probabilité p au moins. Pour l'amplifier, on peut alors répéter l'algorithme par exemple

$20/p$ fois. En effet la probabilité que l'algorithme échoue à chaque fois est au plus

$$(1-p)^{20/p} \leq e^{-20} = 0,000000002.$$

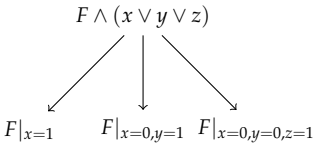


FIGURE 8.1: L'arbre de récursion de l'algorithme de Monien-Speckmeyer

Monien, Speckenmeyer pour 3-SAT

Data: Une formule 3-SAT F sur n variables
Result: teste si F est satisfiable
if *F n'a pas de variables* **then**
  annoncer la valeur de F;
end
Soit C une clause arbitraire de F avec un nombre minimal de littéraux, disons $C = x \vee y \vee z$;
if *backtracking*$(F|_{x=1})$ **then**
  annoncer Vrai;
end
if *backtracking*$(F|_{x=0,y=1})$ **then**
  annoncer Vrai;
end
if *backtracking*$(F|_{x=0,y=0,z=1})$ **then**
  annoncer Vrai;
end
annoncer Faux;

Algorithm 4: Monien-Speckenmeyer-3

La complexité de cette procédure est la solution à la récursion

$$T(m,n) \leq T(m,n-1) + T(m,n-2) + T(m,n-3) + \text{poly}(nm),$$

qui est bornée par $T(m,n) = O(1.84^n \text{poly}(nm))$.

Exercice 18 *Considérez le problème NP-complet de 3-colorabilité d'un graphe de n sommets. Modélisez par une formule 3-SAT et montrez que l'algorithme Monien-Speckenmeyer-3 a une complexité $O(1.618^n poly(nm))$, où 1.618... est le nombre d'or.*

Recherche locale dans un voisinage de rayon k

Soient a, b deux affectations. La *distance de Hamming* entre a et b est définie comme le nombre de variables auxquelles a et b donnent des valeurs différentes. Voici un algorithme qui cherche une solution parmi toutes les affectations à distance au plus k d'une affectation donnée.

Ici a_x représente l'affectation qui diffère de a seulement dans la

variable correspondant au littéral x.

Data: Une formule 3-SAT F sur n variables, une affectation a et un entier $k \geq 0$
Result: cherche s'il existe une assignation validant F et qui soit de distance au plus k de a
if $F(a)$ *est vrai* **then**
  retourner *Vrai*;
end
if $k = 0$ **then**
  retourner *Faux*;
end
Soit $C = x \vee y \vee z$ une clause de F qui n'est pas satisfaite par a;
if *RechercheBoule*$(F, a_x, k-1)$ **then**
  retourner *Vrai*;
end
if *RechercheBoule* $(F, a_y, k-1)$ **then**
  retourner *Vrai*;
end
if *RechercheBoule* $(F, a_z, k-1)$ **then**
  retourner *Vrai*;
end
retourner *Faux*;

Algorithm 5: RechercheBoule

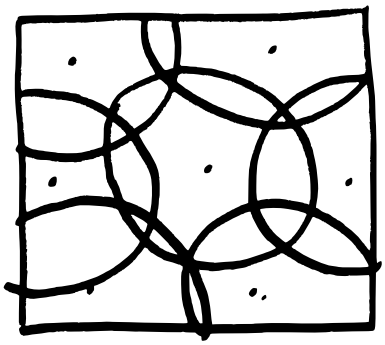


FIGURE 8.2: Couvrir l'espace de recherche avec des boules dans lesquelles on cherche une solution

La complexité de cet algorithme est clairement en $O^*(3^k)$, ignorant les facteurs polynomiaux. Alors un appel à *RechercheBoule*$(F, 0^n, n)$ succède car l'algorithme explore toutes les affectations possibles. Et ceci en temps $O^*(3^n)$, ce qui n'est pas terrible, avouez.

Une petite amélioration est d'appeler plutôt *RechercheBoule*$(F, 0^n, n/2)$ puis *RechercheBoule*$(F, 1^n, n/2)$ qui succède en temps $O(3^{n/2}) = O(1.732^n)$.

On a alors envie d'initier plusieurs recherches qui couvrirons tout l'espace de recherche pour obtenir une complexité encore plus petite. Vous n'avez pas envie ?

Ceci a été fait en analysant la probabilité de succès de *RechercheBoule*$(F, a, n/4)$ pour une affectation uniforme aléatoire a, ce qui donnait un algorithme dont la complexité en espérance est $O^*(1.5^n)$. Plus tard cet algorithme a été dérandomisé, c'est à dire qu'on a su remplacer le choix de l'affectation aléatoire par une série d'affectations dont le voisinage couvre toutes les affectations.

Codes couvrants

Ce qu'on cherche, c'est un ensemble $C \subseteq \{0,1\}^n$ — appelé *code* — tel que chaque mot $a \in \{0,1\}^n$ est à distance au plus d d'un mot du code $b \in C$. Et de plus on veut qu'on puisse générer en temps $O^*(|C|)$ tous les mots du code. On appèle alors C un *code couvrant de rayon d*.

Un tel code donnerait un algorithme de résolution 3-SAT en

temps

$$O^*(|C|3^d).$$

Data: Une formule 3-SAT F sur n variables
Result: cherche s'il existe une assignation validant F
Soit C un code couvrant de rayon d;
for *tout mot a du code* C **do**
  if *RechercheBoule*(F, a, d) **then**
    retourner *Vrai*;
  end
end
retourner *Faux*;

Algorithm 6: RechercheCodeCouvrant

On a de la chance, car il existe de bons codes : pour chaque $0 < \rho, \epsilon < 1/2$ et un n assez grand, il existe un code couvrant de rayon $(\rho + \epsilon)n$ et de taille $2^{(1-h(\rho)+\epsilon)n}$. Ici $h : [0,1] \to [0,1]$ est la fonction d'*entropie*, définie par

$$h(\rho) = -\rho \log_2(\rho) - (1-\rho)\log_2(1-\rho).$$

Théorème 8 *Pour chaque* $0 < \epsilon < 0.1$ *fixé il existe un algorithme de résolution pour 3-SAT de complexité* $O^*((1.5 + 3\epsilon)^n)$.

Preuve : On va utiliser un code couvrant C de rayon $(1/4 + \epsilon)n$ et de taille au plus

$$2^{(1-h(1/4)+\epsilon)n} = (2(1/4)^{1/4}(3/4)^{3/4}2^{\epsilon})^n.$$

La complexité de l'algorithme 6 est alors de l'ordre

$$\begin{aligned} |C|3^d &\leq (2(1/4)^{1/4}(3/4)^{3/4}2^{\epsilon})^n 3^{n/4} \\ &= (2(1/4)^{1/4}(3/4)^{3/4}3^{1/4}6^{\epsilon})^n \\ &= (2(3/4)6^{\epsilon})^n \\ &\leq (1.5 + 3\epsilon)^n \end{aligned}$$

où la dernière inégalité est valide pour $\epsilon < 0.1$. □

Recherche locale d'Uwe Schöning

L'algorithme suivant, remplace la recherche systématique d'un voisinage par une procédure aléatoire.

Data: Une formule 3-SAT F sur n variables, un entier positif d
Result: produit une assignation a qui valide la formule, s'il en existe une
for *ever* **do**
  Soit a une assignation arbitraire;
  for $3n$ *fois au plus* **do**
    if $F(a) =$*Vrai* **then**
      retourner *Vrai*;
    end
    Soit C une clause *arbitraire* de F, tel que $C(a) =$Faux;
    Soit x un littéral *aléatoire* de C;
    Changer la valeur $a[x]$ affectée à x
  end
end

Algorithm 7: Recherche locale de Uwe Schöning

Quand on change la valeur du littéral x, on valide cette clause, mais au risque d'invalider d'autres. Comment s'assurer qu'il y aura du progrès, au moins en espérance ? Cet algorithme ne s'arête jamais si la formule n'est pas satisfiable, mais sinon il trouve une assignation valide a^* au bout d'un certain temps, et on va analyser l'espérance de temps.

Pour cela on fixe une assignation valide arbitraire a^* et on mesure la *distance de Hamming* avec a, c'est-à-dire le nombre de variables où a et a^* diffèrent.

Lemme 2 *À chaque étape, la distance de Hamming $d(a, a^*)$ diminue de* 1 *avec probabilité au moins* $1/3$ *et augmente de* 1 *avec une probabilité au plus* $2/3$.

Preuve : Soit k le nombre de différences entre a et a^* sur les trois littéraux de la clause C choisie dans l'étape.

cas $k = 0$ impossible, car C est valide dans a^* et l'algorithme a choisit une clause qui n'est pas valide dans a.

cas $k = 1$ alors avec probabilité $1/3$ l'algorithme inverse l'unique variable où a et a^* différent dans C, et la distance diminue. Dans l'autre cas, donc avec probabilité $2/3$ la distance augmente.

cas $k = 2$ *ou* $k = 3$ alors la probabilité que la distance diminue est encore plus grande.

□

Le comportement de l'algorithme décrit une marche dans le graphe de la figure 8.3. L'étiquette dans les sommets représente la distance de Hamming avec l'assignation valide a^*.

Pour une analyse pessimiste on peut associer une probabilité $1/3$ aux arcs vers la gauche et $2/3$ aux arcs vers la droite.

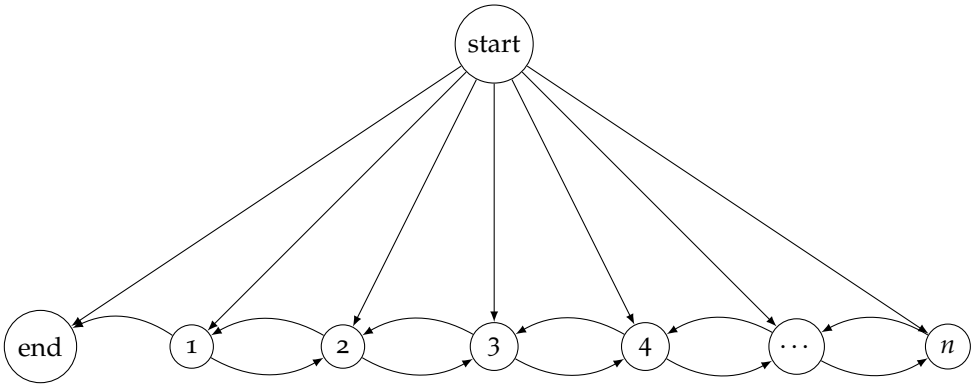


FIGURE 8.3: Le comportement de l'algorithme correspond à une marche aléatoire dans ce graphe.

On va analyser une marche aléatoire sur le graphe ci-haut, mais de manière pessimiste. Pour rendre les choses plus difficiles à l'algorithme, on considère le graphe dont les sommets sont les entiers $\mathbb{Z}$, et on cherche à déterminer la probabilité que l'algorithme atteint le sommet o du sommet j au bout de $3j$ étapes. Pire que ça, on veut déterminer la probabilité d'un tel chemin qui utilise exactement j pas du type $i \to i+1$ et $2j$ pas de type $j \to j-1$. Cette probabilité est au moins

$$q_j \geq \binom{3j}{j} \left(\frac{2}{3}\right)^j \left(\frac{1}{3}\right)^{2j}.$$

Utilisant la formule d'approximation de Stirling $n! = \Theta(\sqrt{n} \cdot (n/e)^n)$, on obtient

$$\binom{3j}{j} = \Theta\left(\frac{1}{\sqrt{j}} \frac{3^{3j}}{2^{2j}}\right).$$

Et ainsi la probabilité de succès de la boucle intérieure de l'algorithme de Schöning est pour une constante c au moins

$$\begin{aligned} & c \cdot \sum_{j=0}^{n} \frac{1}{\sqrt{j}} \cdot \frac{1}{2^j} \binom{n}{j} \cdot \frac{1}{2^n} \\ \geq & \frac{c}{\sqrt{n}} \sum_{j=0}^{n} \binom{n}{j} \frac{1}{2^{n+j}} \\ = & \frac{c}{\sqrt{n}} \left(\frac{3}{4}\right)^n, \end{aligned}$$

où la dernière égalité utilise l'expansion binomiale de $(1/2 + 1/4)^n$.

Pour conclure on utilise l'observation de la section 8 pour montrer que la complexité de l'algorithme de Schöning est $O^*((4/3)^n)$ et qu'il succède avec une grande probabilité.

9
Recherche locale

Dans cette section on va formaliser et généraliser le principe la recherche locale. Dans beaucoup de problèmes on cherche un objet combinatoire qui minimise une certaine fonction de coût, parmi un voisinage donnée.

Les motivations de ce problème sont multiple. Pour certains problèmes NP-durs, on aimerait déjà se contenter d'un optimum local. En théorie de jeux algorithmique, les équilibres de Nash sont des optima locaux. En physique, les optima locaux sont des configurations stable de moindre énergie.

Il existe un concours de résolution de problème Challenge ROADEF/EURO. Les dernières années les équipes gagnantes ont remporté le prix en utilisant de la recherche locale.

Donc on veut trouver un objet combinatoire qui minimise une fonction de coût f. Ces objets définissent les sommets d'un graphe V. On va définir une notion de voisinage utilisant les arrêtes E. Typiquement

— le nombre de sommets est exponentiel en la taille de l'entrée.
Donc le graphe est donnée sous forme implicite.

— et la taille du voisinage est polynomiale.

Dans ce contexte un minimum local est un sommet $x \in V$ si $\forall y,\ (x,y) \in E,\ f(x) \leq f(y)$.

Exemple

Problème des n-reines. $V = [n]^n$ avec $[n] := \{1, \ldots, n\}$ $(x,y) \in E$ si $\exists!\ i\ x_i \neq y_i$.

Modélisation alternative : $V = ([n]^2)^n$ avec le même voisinage : $(x,y) \in E$ si $\exists!\ i\ x_i \neq y_i$.

La fonction de coût $f(x)$ compte le nombre de conflits entre reines, donc le nombre de couples (i,j) tels que les reines i et j sont en conflit, donc tel que $x_i - x_j \in \{i-j, 0, j-i\}$.

Exemple

Trouver le minimum sur une grille $m \times m$ où les cellules voisines sont les au plus 4 cellules adjacentes.

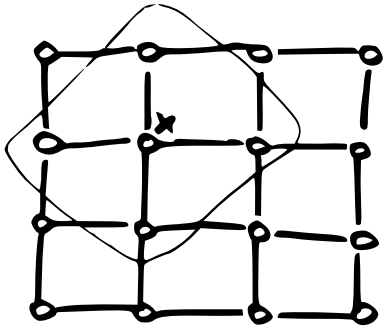


FIGURE 9.1: Un voisinage de taille 4 dans une grille à 2 dimensions

1	2	3	4	5	6
100	100	100	100	100	7
13	12	11	10	9	8
14	100	100	100	100	100
15	16	17	18	19	20
100	100	100	100	100	21
27	26	25	24	23	22

FIGURE 9.2: L'exemple qui tue

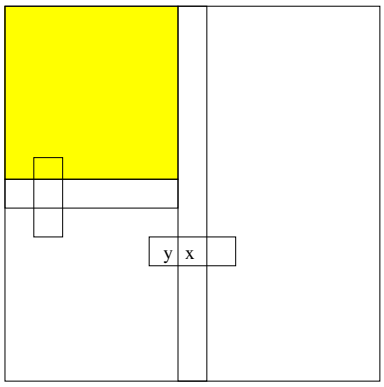
Descente locale

Commencer en une cellule x. Tant qu'il y a un voisin y avec $f(y) < f(x)$ poser $x := y$.

La complexité de cet algorithme naïf est $\Theta(m^2)$ dans le pire des cas sur la grille $m \times m$. La borne inférieure est donnée par une grille qui dispose d'un unique minimum local et d'un chemin descendant de longueur $\Omega(m^2)$, voir figure 9.2.

Diviser pour conquérir

On commence par évaluer f sur toutes les cellules de la colonne du milieu. On identifie ainsi un sommet x avec $f(x)$ minimum. Puis on teste ses deux voisins pour déterminer s'il s'agit d'un minimum local. Si ce n'est pas le cas, par exemple si le voisin de gauche y a une valeur $f(y) < f(x)$ alors on sait que la partie de gauche doit contenir un minimum local. Pourquoi ? Tout simplement parce qu'une descente locale qui débuterait en y ne peut plus dépasser la colonne. Il suffit ensuite d'itérer récursivement sur la sous-grille ainsi identifié.



Complexité

$C(m) = \frac{3}{2}m + 4 + C(\frac{m}{2})$ La complexité est $C(m) = O(m)$ ce qui est une amélioration quadratique par rapport à la descente locale.

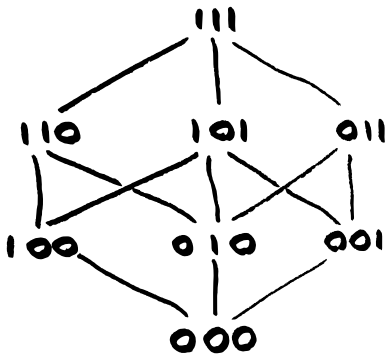


FIGURE 9.3: Un hypercube organisé en couches

Le cas de l'hypercube

Soit l'hypercube $\{0,1\}^n$.

Organiser l'hypercube en couches (ici distance de Hamming). La couche k consiste en les sommets $\{x | \sum_i x_i = k\}$ et sa taille est donc $\binom{m}{k}$.

On va faire une recherche dichotomique par couches :

On sait qu'un minimum local se trouve entre les couches k et l.

Alors on évalue tous les sommets de la couche $p = \lfloor (k+l)/2 \rfloor$

On choisit un minimum x dans cette couche

On regarde tous les voisins de x :

— soit on a déterminé que x est un minimum local

— soit on sait qu'un minimum local se trouve dans l'intervalle de couches [k,p-1] ou [p+1,l].

Complexité

Chaque itération coûte au plus $\binom{n}{\lfloor n/2 \rfloor} + n$ appels à f et il faut au plus $\lfloor \log_2 n \rfloor + 1$ itérations.

Ce qui donne

$$\left(\sqrt{\frac{2}{\pi}} + o(n)\right)(\log n)\frac{2^n}{\sqrt{n}}$$

par la formule de Stirling.

Aléa

Est ce que l'aléa peut aider à la recherche d'un minimum local ?

Voici une variante très simple de la descente locale. Soit un graphe avec N sommets. Choisir un ensemble S de $\sqrt{N}$ sommets uniformément au hasard. Soit x un des sommets qui minimise $f(x)$. Faire une descente locale à partir de x. Si au bout de $\sqrt{N}$ pas on n'a pas trouvé de minimum local, recommencer.

Quel est le nombre de pas espérés par cet algorithme ?

La fonction f définit un ordre total sur les sommets (raffiner si nécessaire). Ceci définit un rang sur les sommets : le sommet de rang 1 est le minimum global par exemple. Quelle est l'espérance du rang de x ?

Exercice 19 *Notez A_y l'évènement que le rang de y est au plus $\sqrt{N}$. Supposez que N est un nombre carré.*

— *Calculez $E[A_y]$.*

— *Minorez $E[\cup_{y \in S} A_y]$ pour S un ensemble choisit uniformément parmi les N sommets, avec $|S| = \sqrt{N}$.*

— *Majorez la complexité espérée de l'algorithme.*

Comment sortir des minimum locaux ?

— Pour éviter les plateaux, autoriser des pas horizontaux (aller à un voisin de même valeur).

— Si on reste trop longtemps sans améliorer, recommencer la recherche à partir d'une position choisie uniformément aléatoire ou alors perturber la solution courante.

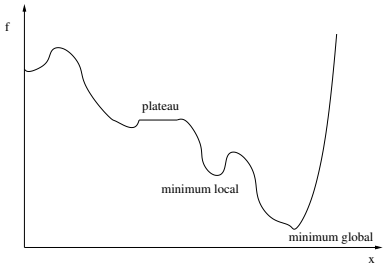


FIGURE 9.4: On veut éviter que la recherche locale reste bloqué dans des minimaux locaux

Ceci ouvre à des métaheuristiques, qui sont plus une science expérimentale que des théories fondés. Il sont basés sur deux principes.

intensifier restreindre le voisinage, faire une descente locale,

diversifier agrandir le voisinage, sortir des minimaux locaux.

Recuit simulé

On effectue une marche aléatoire, avec les probabilités suivantes. On est dans un sommet x et on génère uniformément un voisin y de x. Puis

$$P[\text{aller de x à y}] = \begin{cases} 1 & \text{si } f(y) < f(x) \\ \exp\left(\frac{f(x)-f(y)}{c_t}\right) & \text{si } f(y) \geq f(x) \end{cases}$$

Ici c_t est un paramètre qui contrôle comment s'autorisent des sauts non améliorant. Une politique de refroidissement indique comment c_t va changer pour approcher o.

D'autres métaheuristiques

Recherche tabou Garde en mémoire un historique (borné) des configurations explorées en s'interdisant d'y repasser.

Algorithme génétiques A partir de plusieurs minima locaux on forme de nouvelles configurations point de départ pour de nouvelles descentes locales. Christos Papadimitriou critique cette technique. Il dit que la nature optimise la robustesse pas une fonction objective précise. Et il ne connait pas de problème qui peut être résolu efficacement seulement par un algorithme génétique.

Ingrédients pour la recherche locale

Définir une représentation concise du problème.
Définir un voisinage compact.
Définir une fonction d'évaluation la plus fine possible.
Trouver une méthode efficace pour choisir un voisin.
Pour les métaheuristiques définir une politique de diversification (perturbation) et de concentration.