Cours *Optimisation*
Partie *Optimisation Combinatoire*
3ième année ISAE
Année scolaire 2008-2009

Gérard Verfaillie
ONERA/DCSD/CD, Toulouse
Gerard.Verfaillie@onera.fr

Septembre 2008

Résumé

Ce document couvre la partie *Optimisation Combinatoire* du cours *Optimisation* dispensé en 3ième année ISAE.

On y traite essentiellement d'optimisation sur des variables à domaines de valeur *discrets*, souvent *finis*. Les cadres de modélisation et de résolution présentés sont essentiellement le *problème de satisfiabilité d'une expression booléenne* (SAT), le *problème de satisfaction de contraintes* (CSP) et la *programmation linéaire en nombres entiers* (PLNE).

Ce document ne couvre pas l'optimisation sur des variables à domaines de valeur *continus*, c'est à dire la *programmation linéaire* (PL) et la *programmation non linéaire* (PNL) (voir pour cela [Cav06a]). Il ne couvre pas non plus la modélisation et la résolution de problèmes à base de *graphes* (voir pour cela [Cav06b]).

Table des matières

Chapitre 1

Exemples de problèmes d'optimisation combinatoire

1.1 Recherche d'un itinéraire de moindre coût dans un réseau routier

Soit un réseau routier constitué d'un ensemble de villes et d'un ensemble de routes permettant d'aller d'une ville à une autre, avec au plus une route entre deux villes. Soit un coût associé à chaque route, représentant sa longueur, ou le temps, ou encore la consommation nécessaire à son parcours, supposé indépendant du sens de ce parcours (de A vers B ou de B vers A) et supposé additif (le coût d'un itinéraire est égal à la somme des coûts des routes qu'il utilise). Voir la figure 1.1 pour un exemple de réseau.

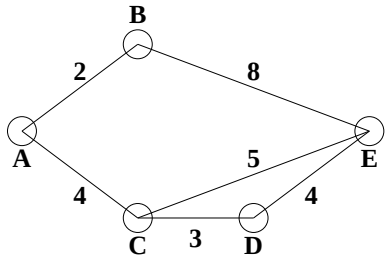

FIGURE 1.1 – Exemple de réseau routier.

Soit le problème suivant : étant donné deux villes, déterminer un itinéraire de moindre coût de la première à la seconde. Voir la figure 1.2 pour un

exemple d'un tel itinéraire de coût 9 entre les villes A et E dans le réseau de la figure 1.1.

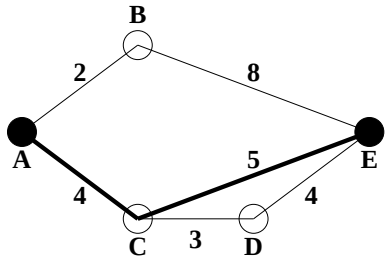

FIGURE 1.2 – Exemple d'itinéraire de moindre coût de A à E.

Ce problème se modélise naturellement comme un problème de recherche d'un plus court chemin dans un graphe non orienté pondéré (*Shortest Path Problem* ; voir [Cav06b]). Ce dernier est certainement l'un des problèmes d'optimisation combinatoire les plus connus, essentiellement du fait de ses très nombreuses applications et des algorithmes efficaces permettant de le résoudre (voir aussi [Cav06b]).

1.2 Organisation de la tournée d'un cadre commercial

Soit un réseau routier similaire au précédent.

Soit le problème suivant : déterminer un itinéraire de moindre coût permettant à un cadre commercial de visiter chaque ville une fois et une seule et de revenir à son point de départ. Voir la figure 1.4 pour un exemple d'un tel itinéraire de coût 20 dans le réseau de la figure 1.3.

Ce problème, connu sous le nom de *Problème de Voyageur de Commerce* (*Traveling Salesman Problem*), se modélise naturellement comme un problème de recherche d'un plus court circuit hamiltonien dans un graphe non orienté pondéré [1] (voir [Cav06b]). Malgré sa proximité avec le problème précédent (*Shortest Path Problem*), ce dernier constitue l'un des problèmes classiques les plus difficiles en optimisation combinatoire. Il n'existe pas d'algorithme permettant de le résoudre efficacement de façon optimale (avec garantie d'optimalité) et même un problème apparemment plus simple (déterminer

1. Dans un graphe non orienté, un circuit est un chemin dont les points de départ et d'arrivée coïncident et un circuit hamiltonien est un circuit passant par tous les sommets du graphe une fois et une seule.

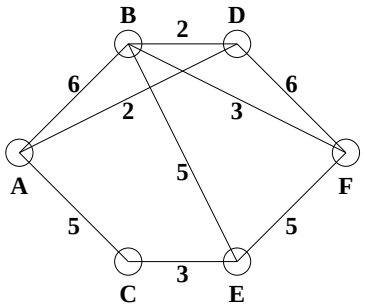

FIGURE 1.3 – Autre exemple de réseau routier.

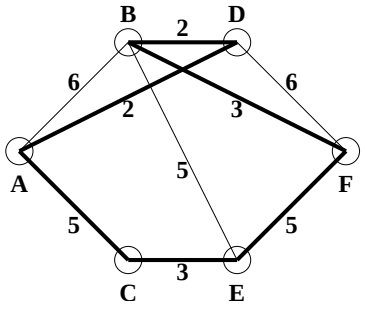

FIGURE 1.4 – Exemple de tournée de moindre coût.

s'il existe un circuit hamiltonien dans un graphe non orienté) est aussi difficile.

1.3 Organisation de la tournée d'une flotte de véhicules de livraison

Il s'agit d'une variante du problème précédent où les villes sont des magasins et une ville particulière est le dépôt en charge de l'approvisionnement des magasins. On suppose disposer d'un ensemble de véhicules de livraison.

Le problème est de déterminer l'itinéraire de chaque véhicule (partant du dépôt et y revenant) de telle façon que l'ensemble des magasins soient approvisionnés par un véhicule et un seul et qu'un critère qui peut être la somme ou le maximum sur l'ensemble des véhicules des coûts des itinéraires soit minimum. Voir la figure 1.6 pour un exemple d'un tel itinéraire à deux véhicules de coût 19 (somme des coûts des deux itinéraires) dans le réseau

de la figure 1.5. Des variantes de ce problème peuvent prendre en compte le volume des commandes des magasins, leurs horaires d'ouverture et les capacités des véhicules.

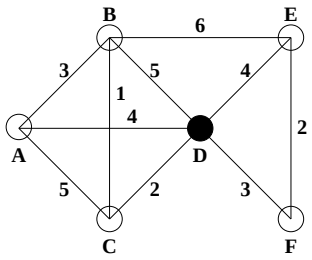

FIGURE 1.5 – Autre exemple de réseau routier avec D comme dépôt.

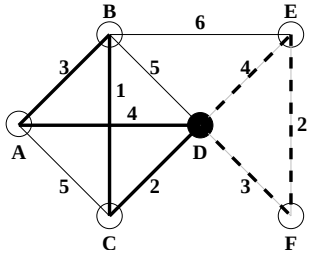

FIGURE 1.6 – Exemple de tournée de moindre coût à deux véhicules.

Connu sous le nom de *Vehicle Routing Problem* et au moins aussi difficile que le problème précédent, ce problème a fait et fait encore l'objet de nombreuses études du fait de son intérêt pratique dans le domaine de la logistique.

1.4 Affectation de tâches à des personnes

Soit un ensemble de n tâches à réaliser. Soit un ensemble de n personnes aptes à réaliser ces tâches, avec pour chaque paire tâche-personne un coût associé à l'affectation de cette tâche à cette personne. Ce coût peut représenter n'importe quelle quantité comme le temps nécessaire à la réalisation de cette tâche par cette personne ou le prix à payer pour cette affectation. Il est supposé additif (le coût d'une affectation globale est la somme des coûts des

affectations individuelles). Voir la figure 1.7 pour une représentation graphique d'un problème d'affectation à 3 tâches et 3 personnes.

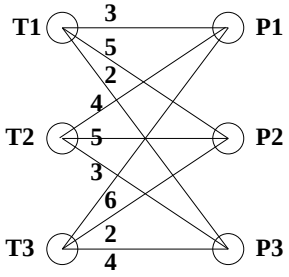

FIGURE 1.7 – Exemple de problème d'affectation.

Soit le problème suivant : déterminer une affectation de chaque tâche à une personne et une seule qui soit de moindre coût. Voir la figure 1.8 pour un exemple d'une telle affectation de coût 8 pour le problème de la figure 1.7.

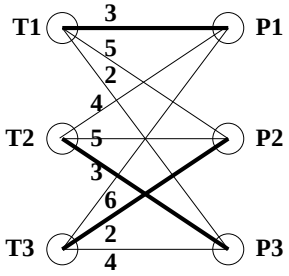

FIGURE 1.8 – Exemple d'affectation de moindre de coût.

Connu sous le nom de *Assignment Problem*, ce problème se modélise comme un problème de couplage optimum dans un graphe biparti pondéré [2] (*Optimum Matching* ; voir [Cav06b]) pour lequel des algorithmes efficaces sont disponibles (voir aussi [Cav06b]). À noter qu'il peut aussi se modéliser comme un problème de programmation linéaire en nombres entiers (PLNE ; voir la section 5.2.1).

2. Un graphe biparti est un graphe dont l'ensemble S des sommets est partitionné en deux sous ensembles S_1 et S_2 tels que toute arête connecte un sommet de S_1 avec un sommet de S_2. Dans un graphe non orienté dont l'ensemble des sommets est S et l'ensemble des arêtes A, un couplage est un sous-ensemble A' de A tel que tout sommet de S soit incident à au plus une arête de A'.

1.5 Gestion d'un chantier de construction

Soit un ensemble de tâches nécessaires à la réalisation d'une construction (immeuble, pont, ...) ou plus généralement d'un projet quelconque. On suppose qu'une durée est associée à chaque tâche. On suppose aussi que certaines précédences sont imposées entre tâches (la viabilisation du terrain doit par exemple précéder la réalisation des fondations de l'immeuble), mais qu'une réalisation en parallèle est possible pour tout sous-ensemble de tâches libre de contraintes de précédence. Voir la figure 1.9 pour une représentation graphique d'un problème de gestion de chantier à 4 tâches.

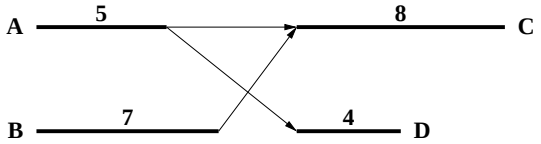

FIGURE 1.9 – Exemple de problème de gestion de chantier.

Soit le problème suivant : déterminer pour chaque tâche sa date de début et sa date de fin au plus tôt et pour l'ensemble du chantier sa date de réalisation au plus tôt, c'est-à-dire le maximum sur l'ensemble des tâches de leur date de fin au plus tôt. Voir la figure 1.10 pour une indication des dates de début et de fin au plus tôt de chaque tâche sur l'exemple de la figure 1.9. La date de réalisation au plus tôt du chantier y est égale à 15.

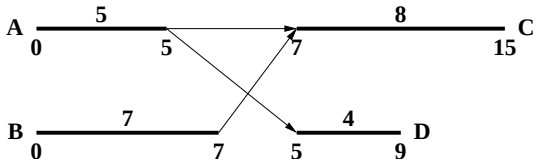

FIGURE 1.10 – Dates de début et de fin au plus tôt de chaque tâche.

Connu sous le nom de *Project Management Problem*, ce problème se modélise curieusement comme un problème de recherche d'un plus long chemin dans un graphe orienté pondéré, variante du problème de plus court chemin évoqué en section 1.1 (voir [Cav06b]). Il bénéficie de ce fait des mêmes algorithmes efficaces (voir aussi [Cav06b]). À noter qu'il peut aussi se modéliser comme un problème de programmation linéaire simple (PL ; voir la section 5.2.5).

1.6 Organisation de tâches de production

Soit un ensemble de tâches. Comme pour le problème précédent, on suppose qu'une durée est associée à chaque tâche et que certaines précédences sont imposées entre tâches. Mais on suppose en plus qu'une date de début au plus tôt et une date de fin au plus tard sont associées à chaque tâche (représentant pour la première la date au plus tôt de disponibilité des éléments nécessaires à la réalisation de cette tâche et pour la seconde la date au plus tard de mise à disponibilité des produits de cette tâche). On suppose aussi que certaines tâches partagent une même ressource dite non partageable (une machine, une personne de qualification particulière, ...) et ne peuvent donc pas être réalisées en parallèle. Voir la figure 1.11 pour une représentation graphique d'un problème d'ordonnancement à 4 tâches.

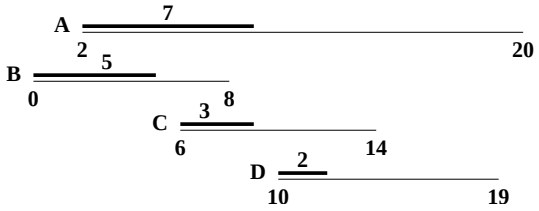

FIGURE 1.11 – Exemple de problème d'ordonnancement.

Soit le problème suivant : sur chaque ressource non partageable, déterminer un ordre de réalisation des tâches qui requièrent cette ressource tel que les dates de début au plus tôt et de fin au plus tard de chaque tâche soient respectées et éventuellement que la date de réalisation au plus tôt de l'ensemble des tâches soit minimum. Voir la figure 1.12 pour un exemple d'un tel ordonnancement sur le problème de la figure 1.11, avec l'hypothèse que les 4 tâches requièrent une même ressource non partageable. La date de réalisation au plus tôt de l'ensemble des tâches y est égale à 18.

Ce problème est connu sous le nom de *Job-Shop Scheduling Problem*. Malgré sa proximité avec le problème précédent, il constitue l'un des problèmes classiques les plus difficiles en optimisation combinatoire et c'est essentiellement la prise en compte de la capacité limitée des ressources qui crée cette difficulté (voir la section 5.2.5 pour une modélisation comme un problème de programmation linéaire en nombres entiers (PLNE) et la section 5.3.5 pour une modélisation comme un problème de satisfaction de contraintes (CSP)). Il a fait et fait encore l'objet de nombreuses études du fait de son intérêt pratique dans le domaine de la gestion de projet ou de production,

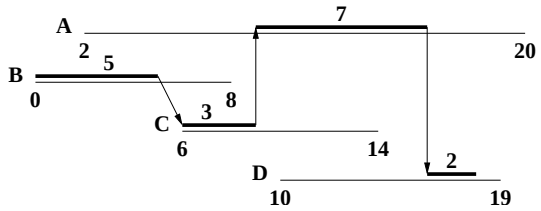

FIGURE 1.12 – Exemple d'ordonnancement optimal.

ainsi que dans bien d'autres.

1.7 Construction d'un emploi du temps scolaire

Soit un ensemble de cours à dispenser. Soient un ensemble de créneaux d'enseignements disponibles, un ensemble de salles et un ensemble de professeurs, avec des contraintes qui limitent pour chaque cours les créneaux, les salles et les professeurs disponibles.

Le problème de base, connu sous le nom de *Timetabling Problem*, consiste à affecter à chaque cours un créneau, une salle et un professeur. Il connaît de nombreuses variantes suivant les contraintes et préférences prises en compte et fait, malgré son caractère difficile, l'objet de nombreux travaux du fait de son intérêt pratique indéniable.

1.8 Affectation de fréquences à des liaisons radio

Soient un ensemble de sites et un ensemble de liaisons radio orientées à assurer de site à site (voir la figure 1.13). Soit un ensemble de fréquences disponibles. Deux types de contrainte pèsent sur les fréquences utilisées par les liaisons radio : contraintes liant la fréquence utilisée par une liaison et celle utilisée par la liaison inverse et contraintes de non interférence liant les fréquences utilisées par les liaisons arrivant sur un même site ou sur des sites voisins.

Le problème de base, connu sous le nom de *Frequency Assignment Problem*, consiste à associer à chaque liaison une fréquence telle que ces contraintes soient respectées et éventuellement que le spectre de fréquences utilisé soit le plus étroit possible (voir la section 5.3.4 pour une modélisation comme un problème de satisfaction de contraintes (CSP)).

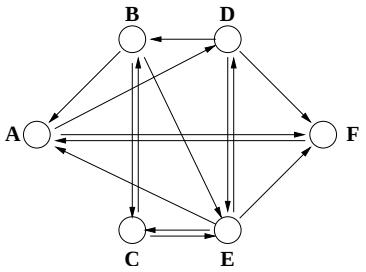

FIGURE 1.13 – Exemple de liaisons radio à assurer.

Malgré sa difficulté, le problème d'affectation de fréquences fait actuellement l'objet de recherches actives du fait de l'augmentation incessante des besoins en communication radio et de la limitation du spectre de fréquences disponible.

1.9 Surveillance d'une zone urbaine

Soit un ensemble de rues, supposées rectilignes, connectées par des carrefours (voir la figure 1.14). On suppose disposer de systèmes de surveillance humains ou matériels et on suppose qu'un système de surveillance placé à un carrefour assure la surveillance de l'ensemble des rues arrivant à ce carrefour.

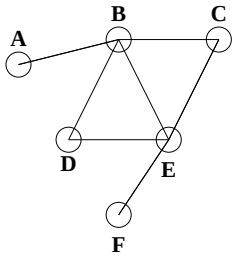

FIGURE 1.14 – Exemple de problème de surveillance.

Soit le problème suivant : déterminer le nombre minimum de systèmes de surveillance permettant d'assurer la surveillance de l'ensemble des rues. Voir la figure 1.15 pour une solution à deux systèmes de surveillance sur l'exemple de la figure 1.14.

Ce problème se modélise naturellement comme un problème de couver-

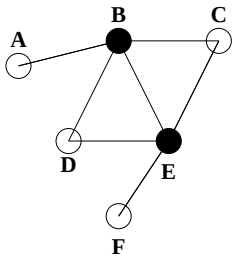

FIGURE 1.15 – Exemple de solution à deux systèmes de surveillance.

ture d'arête de taille minimum dans un graphe non orienté [3](*Minimum Vertex Covering Problem* ; voir [Cav06b]). Il s'agit là encore d'un problème difficile.

1.10 Choix d'instruments à embarquer sur un engin spatial

Soit un ensemble d'instruments d'observation scientifique que l'on souhaite embarquer sur un engin spatial. On suppose que chaque instrument se caractérise par un poids, un volume, une consommation électrique, un volume de calcul nécessaire au traitement des données, un espace mémoire nécessaire à leur stockage, ...mais que le poids, le volume, la consommation électrique, ...de l'ensemble des instruments au sein de l'engin sont limités. On suppose de plus qu'un nombre peut être objectivement associé à chaque instrument représentant son utilité et que ces utilités sont additives (l'utilité d'un ensemble d'instruments est égale à la somme des utilités des instruments qui le composent). Voir la figure 1.16 pour une représentation graphique d'un problème à 4 instruments et à une seule dimension, par exemple le poids.

Soit le problème suivant : déterminer un sous-ensemble d'instruments qui soit à la fois embarquable et d'utilité maximum. Voir la figure 1.17 pour un exemple d'un tel sous-ensemble d'utilité 18 sur l'exemple de la figure 1.16.

Ce problème est connu sous le nom de sac à dos multi-dimensionnel (*Multi Dimensional Knapsack Problem* ; sac à dos en référence au problème du randonneur qui doit choisir ce qu'il met dans son sac à dos de volume et

3. Dans un graphe non orienté dont l'ensemble des sommets est S et l'ensemble des arêtes A, une couverture d'arête est un sous-ensemble S' de S tel que toute arête de A contienne au moins un sommet de S'.

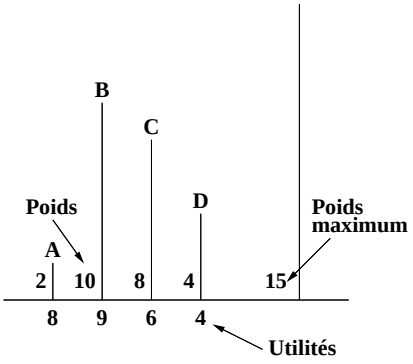

FIGURE 1.16 – Exemple de problème d'embarquabilité.

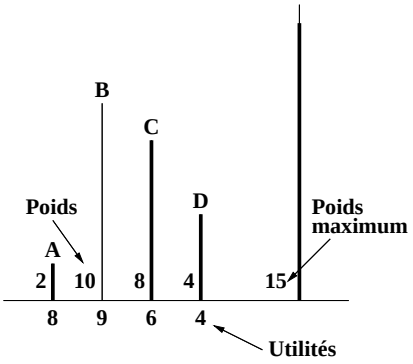

FIGURE 1.17 – Exemple de solution d'utilité maximum.

de poids limité ; multi-dimensionnel en référence aux multiples dimensions du problème : poids, volume, ... ; voir la section 5.2.4 pour une modélisation comme un problème de programmation linéaire en nombres entiers (PLNE)). Il s'agit encore une fois d'un problème difficile.

1.11 Connection entre composants d'un système avionique

Soit un ensemble de composants constituant un système avionique. Soit un ensemble de liaisons possibles entre ces composants (voir la figure 1.18). On suppose qu'un coût est associé à chacune de ces liaisons et que ce coût

est additif (le coût d'un ensemble de liaisons est égal à la somme des coûts des liaisons qui le constituent).

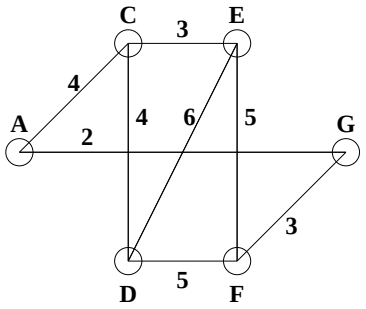

FIGURE 1.18 – Exemple de problème de connections entre composants.

Soit le problème suivant : déterminer un sous-ensemble des liaisons possibles qui soit de coût minimum et qui assure que tout composant soit directement ou indirectement connecté à tout autre. Voir la figure 1.18 pour un exemple d'un tel sous-ensemble de coût 16 sur le problème de la figure 1.18.

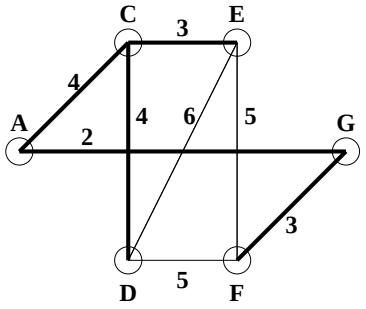

FIGURE 1.19 – Exemple de problème de solution de coût minimum.

Ce problème se modélise comme un problème de recherche d'un arbre couvrant de coût minimum dans un graphe non orienté pondéré [4] (*Minimum Spanning Tree Problem* ; voir [Cav06b]). Bien que cela puisse paraître surprenant, il s'agit d'un problème très facile pour lequel des algorithmes très simples et efficaces existent (voir la section 8.1 ; voir aussi [Cav06b]).

4. Un arbre est un graphe connexe et sans cycle. Dans un graphe non orienté dont l'ensemble des sommets est S et l'ensemble des arêtes A, un arbre couvrant est un sous-ensemble A' de A tel que que S et A' forment un arbre.

1.12 Conception d'un système à base de composants sur étagère

Soit un système quelconque (ordinateur, voiture, réseau local de communication ...) à constituer à partir de composants disponibles sur étagère (dans le commerce). Ce système comporte un ensemble de sous-systèmes. Pour chacun de ces sous-systèmes, il existe un certain nombre de composants appropriés, mais des contraintes technologiques interdisent certaines combinaisons de composants. Le problème de base, connu sous le nom de *Configuration Problem*, consiste à associer à chaque sous-système un composant approprié de façon à respecter les contraintes et éventuellement à minimiser un critère tel que le coût global du système. Ce problème, qui admet de nombreuses variantes suivant la complexité de la modélisation et des contraintes et critères à prendre en compte, fait aujourd'hui l'objet de nombreuses études du fait de son impact potentiel sur des systèmes industriels qui cherchent de plus en plus à répondre à des besoins spécifiques des clients par combinaison de composants standards.

Chapitre 2

Ce que ces problèmes ont en commun

Avant de se pencher sur ce que ces problèmes ont en commun, on peut noter la très grande diversité des problèmes évoqués qui vont de problèmes de transport (*Shortest Path*, *Traveling Salesman*, *Vehicle Routing*) à des problèmes de conception (*Configuration Problem*) en passant par des problèmes d'organisation de tâches (*Assignment*, *Project Management*, *Job-Shop Scheduling*). Les problèmes d'optimisation combinatoire dépassent très largement les quelques exemples présentés ici et concernent de nombreux domaines comme la conduite de robot, la gestion de grands systèmes, l'organisation de la production, la gestion de projets, la conception et le dimensionnement de systèmes, la gestion financière, la gestion du temps et de l'espace, la biologie moléculaire ...

2.1 Alternatives

La première caractéristique commune à tous ces problèmes est l'existence d'un ensemble *discret* d'*alternatives*. Dans le problème de plus court chemin, il s'agit de l'ensemble des itinéraires. Dans le problème d'organisation de tâches de production, il s'agit de l'ensemble des ordres possibles entre tâches partageant une même ressource non partageable. Bien qu'il existe des situations où cet ensemble d'alternatives est infini, il est très souvent *fini*. C'est l'hypothèse que nous ferons par défaut pour l'ensemble de ce cours.

2.2 Contraintes

La seconde caractéristique est l'existence d'un ensemble de propriétés ou de *contraintes* à satisfaire. Toutes les alternatives ne sont pas acceptables. Dans le problème d'organisation de tournée, on exige justement que les itinéraires soient des tournées. Dans le problème d'affectation, on exige que toute tâche soit affectée à une personne et une seule. Ces propriétés ou contraintes expriment que certaines alternatives sont soit, physiquement impossibles (voir par exemple les contraintes technologiques d'incompatibilité entre composants dans le problème de configuration ou les contraintes de capacité maximum dans le problème d'embarquabilité), soit inacceptables du point de vue de l'utilisateur. c'est-à-dire non conformes à ces exigences (voir par exemple les dates au plus tard de réalisation des tâches dans le problème d'organisation de tâches de production ou les contraintes de non interférence dans le problème d'affectation de fréquences).

2.3 Critères

La troisième caractéristique est l'existence de *critères* que l'on cherche à optimiser (la longueur de l'itinéraire dans le problème de plus court chemin, la date de réalisation de l'ensemble des tâches dans le problème d'organisation de tâches de production). Ces critères peuvent prendre des formes diverses : une valeur numérique associée à chacune des alternatives ou simplement un ordre total ou partiel entre alternatives [1]. Bien qu'un seul critère ait été mis en avant dans les exemples évoqués, de nombreuses situations réelles font intervenir plusieurs critères éventuellement conflictuels (par exemple, le coût et la fiabilité). C'est souvent l'existence de plusieurs critères qui crée des situations d'incomparabilité : alternative meilleure du point de vue d'un critère, mais moins bonne du point de vue d'un autre. Le fait que plusieurs critères persistent sans qu'ils aient été agrégés en un critère global pose des problèmes en termes d'optimisation : l'objectif n'est pas clairement identifié. C'est pourquoi nous ferons dans ce cours l'hypothèse d'un critère unique, numérique ou symbolique, résultant éventuellement de l'agrégation de plusieurs critères et induisant un ordre total sur l'ensemble des alternatives.

1. Dans le cas d'un ordre total, ou bien deux alternatives sont équivalentes, ou bien l'une est strictement supérieure à l'autre. Dans le cas d'un ordre partiel, ou bien elles sont équivalentes, ou bien elles sont incomparables, ou bien l'une est strictement supérieure à l'autre

2.4 Synthèse

Du point de vue terminologie, s'il existe simplement des contraintes et pas de critère, on parle de problème de *satisfaction pure*. S'il n'existe pas de contraintes et simplement un critère, on parle de problème d'*optimisation pure*. La plupart des problèmes réels combinent contraintes et critères. À noter que, quand les contraintes deviennent souples, peuvent être plus ou moins violées, comme par exemple des dates de réalisation au plus tard dans le problème d'organisation de tâches de production, la frontière entre contraintes et critères devient floue. Nous ferons dans ce cours l'hypothèse de contraintes dures à satisfaire absolument.

Pour résumer, un problème d'optimisation combinatoire peut être défini par un triplet $< Alt, Cont, Crit >$ où Alt est un ensemble fini d'alternatives, $Cont$ est une fonction de Alt dans $\{0, 1\}$ qui définit les alternatives admissibles et $Crit$ une fonction de Alt dans un ensemble totalement ordonné qui permet d'ordonner les alternatives admissibles.

2.5 Difficulté

Posé de cette façon extrêmement générale, le problème d'optimisation combinatoire peut apparaître d'une très grande trivialité. Puisque l'ensemble des alternatives est fini et totalement ordonné par le critère, il suffit de le parcourir à la recherche d'une alternative qui respecte les contraintes et optimise le critère. La difficulté tient au fait qu'un tel parcours est pratiquement impossible dans la plupart des situations réelles du fait du nombre plus qu'astronomique d'alternatives à envisager. C'est vrai pour un humain qui ploie rapidement face au nombre de combinaisons à considérer. Voir par exemple le problème d'organisation de tournées pour un peu plus de villes et de routes. De façon surprenante, c'est aussi vrai pour les ordinateurs les plus modernes qu'on imagine pourtant d'une telle puissance que les problèmes les plus complexes ne peuvent leur résister. À titre d'exemple, prenons le problème d'organisation de tournée. Une tournée est simplement un ordre sur l'ensemble des villes. Le nombre de tournées envisageables est donc égal au nombre d'ordres possibles : $n!$ si n est le nombre de villes. Supposons qu'une pico-seconde (10^{-12}s) soit nécessaire pour générer et évaluer un ordre (tester la satisfaction des contraintes et évaluer le critère ; hypothèse très largement optimiste). Si $n = 10$, le temps nécessaire à l'énumération de l'ensemble des tournées est simplement de $3,63 \cdot 10^{-6}$s (soit $3,63$ millisecondes). Si $n = 15$, il n'est encore que $1,31$s. Mais si $n = 20$, il passe à

2432902s (soit 0.77 année) et, si $n = 30$, il atteint le nombre astronomique de 2652528598121910586 36s (soit 84111130077 millénaires, soit plus de 500 fois le temps écoulé depuis la naissance de l'univers !).

D'où la préoccupation constante des méthodes d'optimisation combinatoire : produire des solutions optimales ou quasi-optimales sans passer par une énumération de l'ensemble des alternatives.

Chapitre 3

Notions de base en complexité

Ce chapitre peut être vu comme une annexe du cours. Il vise à fournir des résultats de la *théorie de la complexité* qui aident à comprendre un certain nombre de phénomènes rencontrés en optimisation combinatoire et en particulier la frontière qui sépare problèmes faciles et difficiles. Pour plus de détails, voir [GJ79, Pap94].

3.1 Problèmes de décision

Tout d'abord, la théorie de la complexité distingue *problèmes* et *instances*. Une instance est un problème entièrement spécifié. Un problème est un ensemble d'instances partageant la même structure. Un ensemble de paramètres est en général associé à cette structure. On passe de problème à instance en instanciant l'ensemble des paramètres du problème. Exemple de problème : le problème de plus court chemin dans un graphe pondéré. Exemple d'instance : l'exemple de la figure 1.1. Autre exemple de problème, sous-problème du précédent : le problème de plus court chemin dans un graphe pondéré et sans cycle.

Les problèmes que nous considérons ici sont uniquement des problèmes dits de *décision*, c'est-à-dire des problèmes pour lesquels la réponse est soit positive, soit négative : par exemple, le problème de l'existence d'un circuit hamiltonien dans un graphe ou le problème de l'existence d'un circuit hamiltonien de coût inférieur à un paramètre k dans un graphe pondéré. On appelle *certificat* ce qui justifie une réponse positive ou négative.

Les problèmes de *satisfaction* introduits dans le chapitre précédent sont

des problèmes de décision et une alternative satisfaisant les contraintes (on utilise souvent le terme de *solution*) est un certificat de réponse positive.

À tout problème d'*optimisation* P, on peut associer le problème de satisfaction $P(k)$ consistant à déterminer s'il existe une alternative pour laquelle la valeur du critère est strictement meilleure qu'une valeur k donnée. Toute instance I d'un problème d'optimisation P peut alors être résolue en résolvant une séquence finie d'instances $I(k)$ du problème de satisfaction $P(k)$. La première instance utilise une valeur de k pire que toutes les valeurs possibles. Chaque instance suivante utilise comme valeur de k la valeur du critère associée à la solution obtenue sur le problème précédent. Si l'instance $I(k)$ n'admet aucune solution, k est l'optimum du problème et une solution optimale est celle qui a été obtenue sur le problème précédent. La séquence est finie car on a fait l'hypothèse d'un nombre fini d'alternatives et donc d'un nombre fini de valeurs possibles du critère.

3.2 Terminaison, correction, complétude et complexité

Quand on analyse des algorithmes traitant de problèmes de décision, on s'intéresse en général à 5 caractéristiques : la *terminaison*, la *correction*, la *complétude*, la *complexité en temps* et la *complexité en espace*.

Un algorithme termine s'il s'arrête en un temps fini. Il est correct si, quand il fournit une réponse, cette réponse est correcte : positive quand c'est effectivement positif, négative quand c'est effectivement négatif. Il est complet s'il fournit toujours une réponse. À noter qu'un algorithme peut être correct sans être complet. Voir le chapitre 8 pour des exemples d'algorithmes corrects, mais incomplets.

Les complexités en temps et en espace mesurent respectivement le nombre d'instructions élémentaires et le nombre d'unités mémoire élémentaires requis par l'algorithme. Comme ces nombres dépendent de chaque instance, on s'intéresse généralement à leur maximum sur l'ensemble des instances d'une certaine taille, considérée comme le paramètre significatif. La *taille* d'une instance est elle-même définie comme le nombre d'unités mémoire élémentaires nécessaires à sa définition. Mais quand la taille n'est pas un paramètre suffisamment significatif, on peut s'intéresser à la complexité maximum sur l'ensemble des instances partageant une même combinaison de valeurs d'autres paramètres, considérés comme significatifs. Dans l'exemple du problème de plus court chemin, ces paramètres sont par exemple le nombre s de sommets et le nombre a d'arêtes du graphe. La complexité sera alors exprimée

en fonction de s et a.

Quand on étudie ces complexités, on ne s'intéresse pas aux valeurs exactes des maximums, mais plutôt à la façon dont elles évoluent en fonction des paramètres retenus. On veut pas exemple savoir si cette évolution est logarithmique, linéaire, quadratique, cubique, exponentielle ...C'est pourquoi on utilise une notation classique dite *asymptotique* : soit une fonction g d'un paramètre n, $O(g(n))$ est défini comme l'ensemble des fonctions f du même paramètre telles que $\exists c, \exists n_0$ positifs tels que $\forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n)$. Voir la figure 3.1 pour une illustration. $c \cdot g(n)$ constitue un majorant asymptotique de $f(n)$. Soit par exemple $f(n) = a \cdot n^2 + b \cdot n + c$ avec $a > 0$. Il est facile de montrer que $f(n) \in O(n^2)$. Asymptotiquement, c'est l'aspect quadratique qui l'emporte.

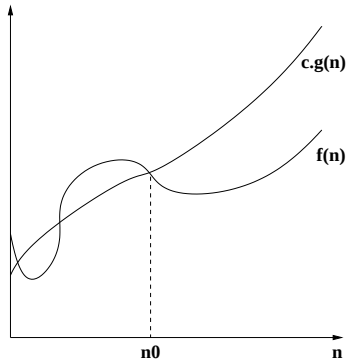

FIGURE 3.1 – $f(n) \in O(g(n))$.

On dit qu'un *algorithme* A est *polynômial* s'il existe un nombre k tel que la complexité en temps de A appartient à $O(n^k)$, où n désigne le paramètre taille. On dit qu'il est *exponentiel* si sa complexité en temps appartient à $O(e^n)$

On dit qu'une fonction f transforme un problème P_1 en un problème P_2 si elle associe à toute instance I_1 de P_1 une instance I_2 de P_2 pour laquelle la réponse est identique : positive pour I_2 si et seulement si elle est positive pour I_1. La modélisation d'un problème de couplage optimum dans un graphe biparti pondéré (voir la section 1.4) comme un problème de programmation linéaire en nombres entiers (PLNE ; voir la section 5.2.1) est un exemple de transformation. On dit qu'une *transformation* est *polynômiale* si l'algorithme qui l'assure est polynômial.

3.3 Classes de problèmes

Ces préliminaires nous permettent de définir différentes classes de problèmes : P, NP et NPC. Intuitivement, on peut dire que la classe P est la classe des problèmes pour lesquels trouver une solution ou prouver qu'il n'en existe pas est facile. La classe NP est la classe des problèmes pour lesquels vérifier une solution est facile. La classe NPC est la classe des problèmes les plus difficiles de NP.

Plus formellement, un problème de décision P_1 appartient à la classe P (P pour polynômial) s'il existe un algorithme polynômial permettant de le résoudre (sous-entendu de façon correcte et complète). Il appartient à la classe NP (NP pour non déterministe polynômial [1]) si le problème de vérification d'un certificat de réponse positive appartient à la classe P. Il appartient à la la classe NPC (NPC pour NP-complet) s'il appartient à NP et si, pour tout problème P_2 de NP, il existe une transformation polynômiale de P_2 vers P_1. Voir la figure 3.2.

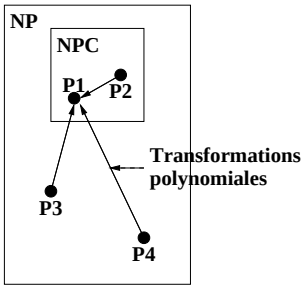

FIGURE 3.2 – Appartenance à NPC.

Nous verrons par exemple que le problème de l'existence d'un chemin de coût inférieur à k dans un graphe pondéré appartient à P. (voir [Cav06b] pour un exemple d'algorithme polynômial, correct et complet). Il est par ailleurs facile de voir que pratiquement tous les problèmes évoqués dans le chapitre 1 appartiennent à NP : vérifier qu'une alternative est conforme du point de vue des contraintes et du critère est polynômial. Nous verrons aussi que certains problèmes comme celui de l'existence d'un circuit hamiltonien de coût inférieur à k dans un graphe pondéré appartiennent à NPC. En fait,

1. Ce terme *non déterministe* vient de la façon dont les classes P, NP et NPC ont été pour la première fois présentées, en utilisant des notions de machine déterministe ou non déterministe. La présentation adoptée ici est équivalente et plus intuitive.

de très nombreux problèmes d'optimisation combinatoire d'intérêt pratique évident appartiennent à NPC.

Par définition, $NPC \subseteq NP$. Il est par ailleurs facile de montrer que $P \subseteq NP$. Les relations entre P et NPC à l'intérieur de NP sont plus problématiques. Il est facile de montrer que si $P \cap NPC \neq \emptyset$ (au moins un problème de NPC est dans P), alors $P = NP$ (tous les problèmes de NP sont dans P) : en effet, soit P_1 un problème appartenant à la fois à P et à NPC : par définition de NPC, pour tout problème P_2 de NP, il existe une transformation polynômiale de P_2 vers P_1 ; il suffit d'appliquer cette transformation, puis de résoudre P_1 pour résoudre P_2 de façon polynômiale ; P_2 appartient donc à P.

Malheureusement, personne n'a jusqu'à présent établi, ni que $P \cap NPC \neq \emptyset$, ni que $P \cap NPC = \emptyset$. De forts arguments plaident cependant en faveur de la seconde option : tout d'abord le fait que trouver une solution est a priori plus difficile que la vérifier ; puis le fait que les problèmes de NPC sont extrêmement nombreux et variés et qu'ils ont été étudiés depuis de longues années par des générations de chercheurs de tous les pays sans qu'aucun d'entre eux n'ait pu produire un algorithme polynômial capable de résoudre un problème de NPC.

D'où la conjecture couramment admise concernant P, NP et NPC : $P \cap NPC = \emptyset$ et $P \neq NP$. Voir la figure 3.3.

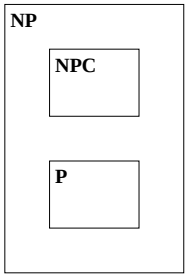

FIGURE 3.3 – Conjecture couramment admise concernant les relations entre P, NP et NPC.

Pour prouver qu'un problème P_1 appartient à P, il suffit d'exhiber un algorithme polynômial qui le résolve. Pour prouver qu'il appartient à NPC, il faut d'abord établir qu'il appartient à NP. C'est en général immédiat. Ensuite deux démarches sont possibles : ou bien une preuve directe, c'est ce qui a été fait par exemple pour le problème SAT (problème de satisfiabilité d'une expression booléenne ; voir la section 4.1), ou bien une preuve indirecte

qui utilise le fait que l'appartenance de certains problèmes à NPC a déjà été établie. On peut par exemple montrer que P_1 admet comme sous-problème un problème P_2 qui appartient à NPC. On peut aussi mettre en évidence une transformation polynômiale d'un problème P_2 qui appartient à NPC vers P_1. On utilise alors le fait que la combinaison de deux transformations polynômiales est aussi une transformation polynômiale. À noter l'existence de problèmes de NP dont l'appartenance, ni à P, ni à NPC, n'a pu être établie.

3.4 Conséquences pratiques

D'un point de vue pratique, l'appartenance à P est plutôt une bonne nouvelle. On parle de problème *facile*. Les anglais utilisent le terme *tractable*. D'un autre côté, l'appartenance à NPC est une franchement mauvaise nouvelle. On parle de problème *difficile*. Les anglais utilisent le terme un peu abusif *untractable*[2]. Mais bonnes et mauvaises nouvelles sont à moduler.

Si un problème appartient à P, il existe un algorithme polynômial permettant de le résoudre de façon correcte et complète. Mais il faut regarder le coefficient c caché derrière la notation asymptotique qui peut se révéler extrêmement élevé. Il faut surtout se préoccuper du degré du polynôme qui peut lui aussi être élevé : un degré de 1, 2, 3, voir 4 peut être acceptable ; un degré de 7 ou 12 est sérieusement problématique. Heureusement, des degrés élevés (7, 12, ...) sont plutôt rares. En tout cas, gagner un degré en retravaillant un algorithme peut avoir un impact non négligeable.

Si un problème appartient à NPC, il existe des algorithmes exponentiels permettant de le résoudre de façon correcte et complète, mais il y a peu de chances (voir la conjecture couramment admise) qu'il existe un algorithme polynômial permettant de le faire. Mais tout dépend de deux facteurs : d'une part, le temps et la puissance de calcul disponibles ; d'autre part, la nature exacte des instances que l'on aura à résoudre.

Si le temps et la puissance de calcul disponibles sont suffisants et si les instances à résoudre sont toutes de taille limitée, un algorithme correct et complet peut être éventuellement utilisé en dépit de sa complexité exponentielle.

Il faut aussi se rappeler que les complexités considérées sont des maximums sur toute les instances d'une taille donnée. Rien ne dit que les instances à résoudre sont parmi celles qui engendrent une complexité maximale. Des études essentiellement expérimentales sur les problèmes de sa-

2. Abusif car il laisse à penser que l'on ne peut rien faire sur ces problèmes.

tisfaction de contraintes ont par exemple fait apparaître que le degré de contrainte de l'instance traitée (le fait qu'elle soit plus ou moins contrainte) peut avoir un impact bien plus important que sa taille sur la complexité de sa résolution. Typiquement, les instances peu contraintes se résolvent facilement car presque toutes les alternatives sont solutions et il est aisé d'en trouver une. Les instances très contraintes se résolvent aussi facilement car il est aisé de prouver l'absence de solution. Les instances les plus difficiles sont les instances intermédiaires pour lesquelles trouver une solution équivaut à la recherche d'une aiguille dans un botte de foin ou pour lesquelles prouver l'absence de solution passe par un nombre astronomique d'étapes intermédiaires. Plus curieusement, les expérimentations ont montré l'existence d'un pic brutal de complexité au passage de la frontière entre problèmes cohérents (pour lesquels il existe une solution) et problèmes incohérents (pour lesquels il n'existe pas de solution). Voir la figure 3.4. Si les instances à résoudre se situent toutes loin de ce pic, un algorithme correct et complet peut aussi être éventuellement utilisé. Comme une bonne nouvelle est souvent suivie d'une mauvaise, on notera que quand on résout une instance de problème d'optimisation par une séquence d'instances de problème de satisfaction (voir la section 3.1), les dernières instances de la séquence se situent dans la zone du pic.

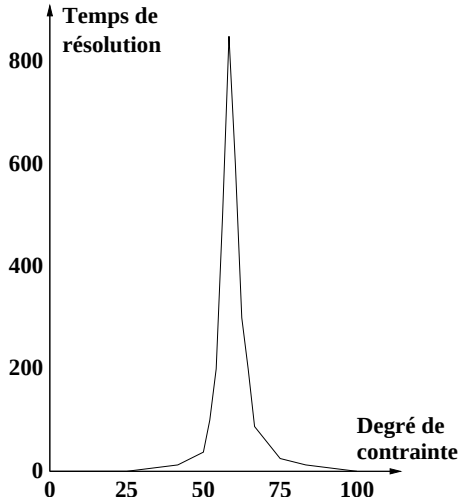

FIGURE 3.4 – Pic de complexité classique sur des problèmes de satisfaction.

Il faut aussi se rappeler que ce n'est pas parce qu'un problème appartient à NPC que tous ses sous-problèmes y appartiennent aussi. Il existe

de nombreux exemples de sous-problèmes polynômiaux de problèmes NP-complets. On verra par exemple dans la section 8.1 que, si le problème dit de satisfaction de contraintes (CSP) est NP-complet, le même problème restreint aux instances binaires dont le graphe de contraintes est acyclique est lui polynômial.

Quand les caractéristiques des instances à résoudre ne font pas espérer une résolution facile via un algorithme correct et complet, rien n'est encore tout à fait perdu. Pour des problèmes de satisfaction , la solution consiste à relâcher l'exigence de complétude. Relâcher l'exigence de correction semble en effet difficile. Un algorithme qui répondrait n'importe quoi semble difficilement acceptable. Relâcher l'exigence de complétude implique simplement que l'algorithme répondra parfois (souvent) qu'il ne sait pas si la réponse est positive ou négative (voir le chapitre 8 pour des exemples de tels algorithmes). En ce qui concerne les problèmes d'optimisation, l'équivalent du relâchement de l'exigence de complétude consiste à relâcher l'exigence d'optimalité. On admettra que l'algorithme produise des solutions non optimales, avec ou sans garantie de distance maximum à l'optimum (voir aussi le chapitre 8).

Il reste que si le problème que vous avez à résoudre appartient à NPC, il est plutôt illusoire d'espérer résoudre dans un temps raisonnable n'importe quelle instance de n'importe quelle taille et de n'importe quelle nature de façon complète dans le cas d'un problème de satisfaction ou de façon optimale dans le cas d'un problème d'optimisation.

Et si quelqu'un vous demande cela, par exemple dans le document de spécification d'un système informatique de décision ou d'aide à la décision , vous pouvez lui donner à lire des introductions à la théorie de la complexité pour lui faire comprendre que cet objectif est inatteignable, à moins de croire dur comme ferme que $P = NP$ et qu'on réussira à l'établir.

Chapitre 4

Cadres de modélisation à base de variables et de contraintes

Dans ce chapitre, nous présentons les cadres de modélisation à base de variables, de contraintes (éventuellement de critère) qui sont les plus classiques et les plus largement utilisés pour modéliser des problèmes d'optimisation combinatoire : *SAT*, *PLNE* et *CSP*. Ces trois cadres ont en commun :

- le fait que les *variables* et les *domaines* de valeur qui leur sont associés sont utilisés pour représenter les alternatives possibles : une alternative est une *affectation* à chaque variable d'une valeur de son domaine ;
- le fait que chaque *contrainte* peut être vue comme une fonction d'un sous-ensemble des variables dans $\{vrai, faux\}$: $vrai$ pour les combinaisons de valeurs qui satisfont la contrainte, $faux$ pour celles qui la violent ;
- le fait que l'éventuel *critère* soit une fonction d'un sous-ensemble des variables dans un ensemble totalement ordonné.

4.1 Satisfiabilité d'une expression booléenne

Le premier cadre, peut-être plus connu dans le domaine du raisonnement logique que dans celui de l'optimisation combinatoire, est celui de la *Satisfiabilité d'une expression booléenne* (*SAT*, *Boolean Satisfiability* en anglais).

Une instance de *SAT* est définie par un ensemble de n variables booléennes et un ensemble de m clauses pesant sur ces variables.

Une *variable booléenne* a deux valeurs possibles : *true* (t) ou *false* (f). Un *littéral* est une variable booléenne ou sa négation. Exemples de littéraux : x, $\neg y$ [1]. Une *clause* est une *disjonction* de littéraux. Exemple de clause c : $x \vee \neg y \vee z$.

Une clause est *satisfaite* si au moins un des ses littéraux est vrai. Par exemple, les affectations $\{(x = t), (y = t), (z = f)\}$, $\{(x = f), (y = f), (z = f)\}$, $\{(x = t), (y = f), (z = f)\}$, ...satisfont la clause c. Elle est insatisfaite si tous ses littéraux sont faux. Par exemple, l'affectation $\{(x = f), (y = t), (z = f)\}$ ne satisfait pas la clause c [2].

Une *instance* de SAT est une *conjonction* de clauses. Exemple d'instance $I_1 : (x \vee \neg y \vee z) \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z)$. On la présente souvent sous la forme suivante, avec des conjonctions implicites entre clauses :

$$x \vee \neg y \vee z$$
$$\neg x \vee y$$
$$\neg y \vee \neg z$$

Une instance est *satisfaite* si toutes ses clauses le sont. Par exemple, l'affectation $\{(x = t), (y = t), (z = f)\}$ satisfait l'instance I_1.

Une instance est dite *satisfiable* s'il existe au moins une affectation de ses variables qui la satisfait. Exemple d'instance satisfiable : l'instance I_1. Exemple d'instance insatisfiable $I_2 : (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (x \vee y) \wedge (\neg y \vee \neg z)$.

Le problème SAT consiste à déterminer si une instance est satisfiable. Malgré son apparente simplicité, il appartient à la classe NPC (voir le chapitre 3). C'est même le standard des problèmes NP-complets, le premier pour lequel l'appartenance à NPC a été démontrée. Dans sa version de base, c'est un problème de satisfaction pure : pas de critère à optimiser.

À noter que toute expression booléenne peut s'écrire de façon équivalente sous la forme d'une conjonction de clauses, c'est-à-dire sous la forme d'une instance SAT. Par exemple, l'expression $(x \vee y) \rightarrow (\neg z \wedge t)$ peut s'écrire de façon équivalente $(\neg(x \vee y)) \vee (\neg z \wedge t)$, soit $(\neg x \wedge \neg y) \vee (\neg z \wedge t)$ et soit encore $(\neg x \vee \neg z) \wedge (\neg x \vee t) \wedge (\neg y \vee \neg z) \wedge (\neg y \vee t)$. On dit que l'expression a été mise sous *forme normale conjonctive* (*Conjonctive Normal Form*, *CNF*). La forme normale conjonctive, c'est-à-dire le cadre SAT, peut donc être vue comme la forme standard de toute expression booléenne, ce qui justifie qu'on parle pour SAT de *Satisfiabilité d'une expression booléenne.*

1. $\neg$ = non, $\wedge$ = et, $\vee$ = ou, $\rightarrow$= implique et $\leftrightarrow$= équivalence.
2. En fait toute clause interdit une affectation et une seule de ses variables

4.2 Programmation linéaire en nombres entiers

Le second cadre, largement utilisé depuis de nombreuses années, est celui de la *Programmation Linéaire en Nombres Entiers* (*PLNE*, *Integer Linear Programming*, *ILP* en anglais). Il s'agit tout simplement du cadre de la programmation linéaire (*PL*, *LP* en anglais), vue dans le cours d'optimisation sur variables à domaines continus (voir [Cav06a]), avec la restriction que les variables doivent prendre des valeurs *entières*. On parle de *contrainte d'intégrité*.

Alors qu'une instance de *PL* peut s'écrire :

$$\begin{aligned} &\min \sum_{i=1}^{n} c_i \cdot x_i \\ &\forall j,\ 1 \leq j \leq m,\ \sum_{i=1}^{n} a_{ij} \cdot x_i \leq b_j \\ &\forall i,\ 1 \leq i \leq n,\ x_i \in R^+ \end{aligned} \tag{4.1}$$

une instance de *PLNE* s'écrit :

$$\begin{aligned} &\min \sum_{i=1}^{n} c_i \cdot x_i \\ &\forall j,\ 1 \leq j \leq m,\ \sum_{i=1}^{n} a_{ij} \cdot x_i \leq b_j \\ &\forall i,\ 1 \leq i \leq n,\ x_i \in N \end{aligned} \tag{4.2}$$

À titre d'exemple ;

$$\begin{aligned} &\max (2x + 3.5y) \\ &x + 4y \leq 10 \\ &3x + 5y \leq 15 \\ &x, y \in N \end{aligned}$$

qui s'écrit aussi :

$$\min (-2x - 3.5y)$$

$$x + 4y \leq 10$$
$$3x + 5y \leq 15$$
$$x, y \in N$$

est une instance de *PLNE* à deux variables, deux contraintes et un critère, que nous utiliserons dans la section 7.1. Elle admet une seule solution optimale de valeur 10 : $x = 5,\ y = 0$

À noter que la présentation standard des problèmes PL et $PLNE$ ne doit pas faire croire que le critère et toutes les contraintes pèsent sur toutes les variables. Dans la plupart des problèmes réels, le critère, ainsi que chaque contrainte, pèse sur un sous-ensemble particulier des variables du problème, ce qui peut induire de nombreux coefficients nuls dans la représentation matricielle standard.

À noter aussi le cadre plus général de la *Programmation Linéaire Mixte* (PLM) où seulement certaines variables sont contraintes à prendre des valeurs entières et le cadre plus restrictif de la *Programmation Linéaire en Variables Bivalentes* (*PL0/1*) où les variables ont deux valeurs possibles : 0 ou 1.

Alors qu'un néophyte pourrait penser que le passage d'un espace de recherche infini à un espace fini simplifie le problème, il n'est rien, au contraire : alors que le problème PL appartient à la classe P [3], le problème $PLNE$ appartient à la classe NPC (voir le chapitre 3). Alors que le premier est *facile*, le second est *difficile*. Une preuve de cette appartenance est qu'il existe une tansformation immédiate (de complexité linéaire) de toute instance SAT en une instance *PL0/1* (et donc $PLNE$) équivalente. Il suffit en effet de remplacer dans l'instance SAT t par 1, f par 0, $\neg x$ par $(1 - x)$ et $\vee$ par $+$. La clause $x \vee \neg y \vee z$ se transforme par exemple en la contrainte linéaire suivante : $x + (1 - y) + z \geq 1$.

4.3 Problèmes de satisfaction de contraintes sur domaines finis

Le troisième cadre, plus récent, mais de plus en plus utilisé, est celui des *Problèmes de Satisfaction de Contraintes* sur domaines finis (*Constraint Satisfaction Problems*, *CSP* en anglais). En deux mots, les différences essentielles vis-à-vis du cadre *PLNE* sont une restriction et une relaxation :

3. Même si l'algorithme du *simplexe*, encore le plus utilisé, est de complexité exponentielle, des algorithmes polynômiaux tels que l'algorithme dit du *point intérieur* sont disponibles.

alors que, dans le cadre *PLNE*, les domaines des variables peuvent être non bornés, ils sont obligatoirement finis dans le cadre *CSP* ; alors que, dans le cadre *PLNE*, les contraintes et le critère sont obligatoirement linéaires, ils sont quelconques dans le cadre *CSP*.

4.3.1 Composants de base

Les composants de base du cadre *CSP* sont au nombre de quatre : *variables*, *domaines*, *contraintes* et *relations*. Une instance de *CSP* peut être défini par un quadruplet (V, D, C, R) où :

- V est une séquence de n *variables* ;
- D est une séquence de n *domaines* ; à chaque variable $v_i \in V,\ 1 \leq i \leq n$, est associé son domaine $D_i \in D$ qui représente l'ensemble des valeurs possibles pour v_i ; ces domaines sont obligatoirement finis, mais de nature quelconque, symbolique ou numérique ;
- C est une séquence de e *contraintes* ; à chaque contrainte $c_j \in C,\ 1 \leq j \leq e$, est associée une sous-séquence V_j de V qui représente l'ensemble des variables sur lesquelles pèse c_j ;
- R est une séquence de e *relations* ; à chaque contrainte $c_j \in C,\ 1 \leq j \leq e$, est associée une relation $R_j \in R$ portant sur les domaines des variables de V_j, c'est-à-dire un sous-ensemble du produit cartésien des domaines des variables de V_j ; ce sous-ensemble (ou cette relation) définit l'ensemble des combinaisons de valeurs des variables de V_j autorisées par c_j ; ces relations sont absolument quelconques ; elles peuvent être définies, soit en *extension* par la liste des combinaisons de valeurs autorisées ou interdites, soit en *intension* sous la forme d'une équation liant les variables de V_j ou plus généralement sous la forme d'une fonction booléenne (ou *fonction caractéristique*) f_j (qui peut consister en un code informatique), acceptant en entrée une combinaison de valeurs des variables de V_j et renvoyant *vrai* ou *faux* suivant que la contrainte c_j est satisfaite ou non par cette combinaison.

4.3.2 Structure

V et C représentent la partie dite *structurelle* du *CSP*. Elle peut être représentée sous la forme d'un (hyper)graphe non orienté dont les sommets représentent les variables de V et les arêtes représentent les contraintes de C. On parle souvent de *graphe de contraintes* ou *macrostructure* pour désigner cet (hyper)graphe. La figure 4.1 montre un exemple de macrostuc-

ture associée à une instance de *CSP*. Cette instance implique 4 variables et 4 contraintes : 3 contraintes binaires (c_1 entre v_1 et v_2, c_2 entre v_2 et v_4 et c_3 entre v_3 et v_4) et une contrainte ternaire (c_4 entre v_1, v_2 et v_3). On définit l'*arité* a_j d'une contrainte c_j comme le nombre de variables qu'elle lie, soit $a_j = |V_j|$. On définit le *degré* d_i d'une variable v_i comme le nombre de contraintes dans lesquelles elle intervient, soit $d_i = \sum_{c_j \in C}(v_i \in V_j)$. Dans l'exemple de la figure 4.1, la contrainte c_4 est d'arité 3 et la variable v_1 de degré 2.

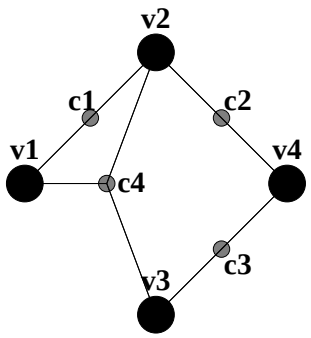

FIGURE 4.1 – Exemple de macrostructure.

4.3.3 Sémantique

D et R représentent la partie dite *sémantique* du *CSP*. Elle peut de façon analogue être représentée sous la forme d'un (hyper)graphe non orienté multiparti [4] dont les sommets représentent les valeurs des domaines de D, les parties sont les domaines de D et les arêtes représentent les combinaisons de valeurs des relations de R. On parle souvent de de *graphe de cohérence* ou *microstructure* pour désigner cet (hyper)graphe. La figure 4.2 montre un exemple de microstucture associée à une instance de *CSP* ayant la macrostructure de la figure 4.1. Pour des raisons de lisibilité, on omet généralement de représenter les combinaisons de valeurs autorisées du fait de l'absence de contrainte : comme il n'existe pas de contrainte entre v_1 et v_3, toutes les combinaisons de valeurs sont implicitement autorisées, mais elles ne sont pas représentées. À titre d'exemple, le domaine associé à v_4 est $\{1, 2, 3\}$. La

4. Un (hyper)graphe non orienté est multiparti s'il existe une partition de ses sommets tel que, pour toute (hyper)arête, les sommets qu'elle connecte appartiennent chacun à des éléments distincts de la partition.

relation associée à la contrainte c_2 entre v_2 et v_4 est définie en intension par l'inéquation $v_2 < v_4$ qui autorise les couples de valeurs $(1, 2)$, $(1, 3)$ et $(2, 3)$. La relation associée à la contrainte c_4 entre v_1, v_2 et v_3 est définie en intension par l'inéquation $v_1 + v_2 + v_3 \leq 3$ qui autorise le seul triplet $(1, 1, 1)$.

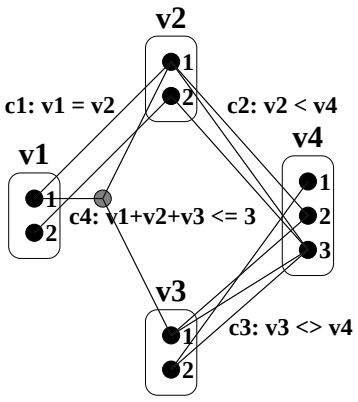

FIGURE 4.2 – Exemple de microstructure.

4.3.4 Définition des domaines et des contraintes

Il faut insister sur le fait que les *domaines* sont de nature quelconque. Un domaine peut être défini en *extension* par un ensemble de valeurs, comme dans l'exemple de la figure 4.2. Il peut aussi être défini en *intension* par un type de valeur et une contrainte unaire, à la seule condition que type et contrainte définissent un domaine fini. Exemples : l'ensemble des entiers compris entre 3 et 9, l'ensemble des entiers compris entre 2 et 5 ou entre 7 et 12 ou encore l'ensemble des entiers multiples de 3 compris entre 1 et 20. Il est aussi possible de définir des domaines de valeur dont les éléments sont des symboles, des vecteurs, des structures, des objets, des ensembles ...

De la même façon, les *contraintes* sont de nature quelconque. On a vu dans l'exemple de la figure 4.2 des contraintes numériques linéaires ($v_1 + v_2 + v_3 \leq 3$) et non linéaires ($v_3 \neq v_4$). Mais bien d'autres types de contraintes non linéaires sont possibles. Exemples : $c_1 \vee c_2$ où c_1 et c_2 sont deux contraintes linéaires, $x \cdot y \leq z$ ou encore $x^y \geq z$. Sur des domaines de valeur quelconques, numériques ou non, on peut imaginer de nombreux types de contraintes. Exemples : la contrainte *tousdifférents* qui pèse sur un ensemble de variables et stipule que ces variables doivent prendre des valeurs toutes différentes ou encore la contrainte *élément* qui prend en paramètres un ensemble ordonné

de variables V, un indice variable i et une variable v et stipule que le $i$ème élément de V est égal à v.

4.3.5 Affectations

Affecter une variable équivaut à lui associer une valeur de son domaine. Soit A l'*affectation* d'un sous-ensemble $V(A) \subseteq V$ des variables. A est dite *complète* si toutes les variables sont affectées ($V(A) = V$). Elle est dite *partielle* sinon. Soit $c_j \in C$ une contrainte dont toutes les variables sont affectées par A ($V_j \subseteq V(A)$). c_j est dite *satisfaite* par A si la restriction de A aux variables de V_j est un élément de la relation R_j ($A[V_j] \in R_j$) [5]. Une affectation A est dite *cohérente* si toutes les contraintes dont toutes les variables sont affectées par A sont satisfaites par A ($\forall c_j \in C / V_j \subseteq V(A), A[V_j] \in R_j$). Elle est dite *incohérente* sinon. Une *solution* est une affectation complète cohérente ($[V(A) = V] \wedge [\forall c_j \in C, A[V_j] \in R_j]$), en d'autres termes une affectation de toutes les variables qui satisfait toutes les contraintes. Une instance de *CSP* est dite *cohérente* s'il existe une solution. Elle est dite *incohérente* sinon.

Dans l'exemple de la figure 4.2, l'affectation partielle $\{(v_1 = 1), (v_2 = 1), (v_4 = 1)\}$ est incohérente car elle ne satisfait pas la contrainte c_2 dont toutes les variables sont affectées. L'affectation partielle $\{(v_1 = 1), (v_2 = 1), (v_4 = 2)\}$ est par contre cohérente car c_2, la seule contrainte dont toutes les variables sont affectées, est satisfaite. L'affectation complète $\{(v_1 = 1), (v_2 = 1), (v_3 = 1), (v_4 = 2)\}$ est aussi cohérente et constitue l'une des deux solutions de cette instance, qui est donc elle même cohérente.

4.3.6 Requêtes

De nombreuses *requêtes* peuvent être exprimées relativement à une instance de *CSP*. On peut citer entre autres :

1. déterminer si elle est ou non cohérente ;
2. produire une solution en cas de cohérence ;
3. produire toutes les solutions ;
4. déterminer le nombre de solutions ;
5. déterminer si une valeur d'une variable participe ou non à une solution ;
6. déterminer si une combinaison de valeurs de certaines variables participe ou non à une solution ;

5. Si A est une affectation et V' un ensemble de variables tel que $V' \subseteq V(A)$, $A[V']$ désigne la projection de A sur V'.

7. éliminer de chaque domaine les valeurs qui ne participent à aucune solution ;
8. éliminer de chaque relation les combinaisons de valeurs qui ne participent à aucune solution ;

La seconde de ces requêtes est de loin la plus courante. Dans l'exemple de la figure 4.2, les réponses aux différentes requêtes sont les suivantes :

1. *oui*, cette instance est cohérente ;
2. $\{(v_1 = 1), (v_2 = 1), (v_3 = 1), (v_4 = 2)\}$ est une solution ;
3. $\{\{(v_1 = 1), (v_2 = 1), (v_3 = 1), (v_4 = 2)\}, \{(v_1 = 1), (v_2 = 1), (v_3 = 1), (v_4 = 3)\}\}$ est l'ensemble des solutions ;
4. il existe 2 solutions ;
5. *oui*, par exemple, $(v_1 = 1)$ participe à une solution, mais *non*, $(v_1 = 2)$ ne participe à aucune solution ;
6. *oui*, par exemple, $\{(v_1 = 1), (v_3 = 1)\}$ participe à une solution, mais *non*, $\{(v_1 = 1), (v_3 = 2)\}$ ne participe à aucune solution ;
7. élimination de la valeur 2 des domaines de v_1, v_2 et v_3 et de la valeur 1 du domaine de v_4 ;
8. élimination de la combinaison $\{(v_1 = 2), (v_2 = 2)\}$ de la relation associée à la contrainte c_1, de la combinaison $\{(v_2 = 2), (v_4 = 3)\}$ de la relation associée à la contrainte c_2 et des combinaisons $\{(v_3 = 2), (v_4 = 1)\}$ et $\{(v_3 = 2), (v_4 = 3)\}$ de la relation associée à la contrainte c_3 ;

La figure 4.3 montre la microstructure de l'instance de la figure 4.2 après réponse aux requêtes 7 et 8.

4.3.7 Optimisation

Dans sa version de base, le problème *CSP* n'est pas un problème d'*optimisation*, mais un problème de *satisfaction* pure. On notera cependant que si un critère d'évaluation existe et est fonction de la valeur prise par un sous ensemble $V_c \subseteq V$ des variables de l'instance considérée, la recherche d'une solution pour laquelle le critère prend une valeur meilleure qu'une certaine valeur c est un problème de satisfaction de contraintes, après ajout aux variables initiales d'une variable v_c représentant la valeur prise par le critère et ajout de deux contraintes : une contrainte liant v_c aux variables de V_c et spécifiant le critère et une contrainte unaire sur v_c de type $v_c \prec c$ (si $x \prec y$ indique que la valeur x du critère est meilleure que la valeur y).

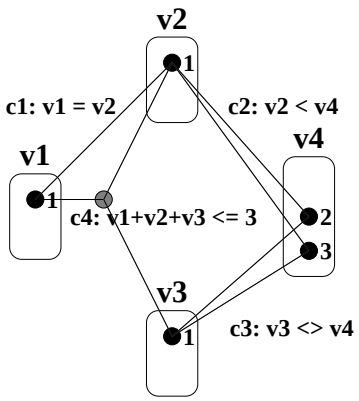

FIGURE 4.3 – Microstructure après élimination des valeurs et combinaisons de valeur ne participant à aucune solution.

4.3.8 Complexité

Du point de vue complexité, le problème CSP appartient à la classe NPC, dans la mesure où il est une généralisation du problème SAT, dont l'appartenance à NPC a été établie (voir le chapitre 3).

On utilisera souvent les notations suivantes : n pour le nombre de variables d'une instance, e pour le nombre de contraintes et d pour la taille maximale des domaines. Avec ces notations, la taille de l'espace de recherche (l'ensemble des affectations possibles) appartient à $O(d^n)$.

4.4 Points communs et différences

Ces différents cadres se différencient essentiellement par la nature des domaines, des contraintes et des critères qu'ils permettent de représenter.

Du point de vue des *domaines* de valeur des variables :

— dans le cadre SAT, les domaines sont tous réduits à $\{t, f\}$;

— dans le cadre $PLNE$, les domaines sont tous constitués de l'ensemble N des *entiers naturels* ;

— dans le cadre CSP, ils sont *quelconques*, symboliques ou numériques, mais obligatoirement *finis*.

Du point de vue des *contraintes* entre variables :

— dans le cadre SAT, les contraintes ont toutes la forme de *clauses*, c'est-à-dire de disjonction de littéraux ;

— dans le cadre $PLNE$, les contraintes sont toutes *linéaires* ;

— dans le cadre *CSP*, elles sont absolument *quelconques*.

Du point de vue enfin du *critère* :

— il est *absent* du cadre SAT ;

— il est *linéaire* dans le cadre $PLNE$;

— il est *quelconque* dans le cadre *CSP*.

On notera que les trois problèmes de décision associés (Existe-t-il une affectation des variables qui satisfasse les contraintes et telle que la valeur du critère soit meilleure qu'un valeur donnée ?) appartiennent à NPC. Si on limite le cadre $PLNE$ de telle façon que toutes les variables aient un domaine fini et si on oublie le critère, il existe une transformation immédiate de chaque instance de l'un de ces trois problèmes en une instance équivalente de l'un des deux autres.

L'existence de ces transformations n'implique pas que les représentations d'un problème réel dans chacun de ces trois cadres soient toutes équivalentes du point de vue de leur compacité, de leur lisibilité, de leur structure et surtout de leur complexité de résolution via les méthodes existantes dans chacun des trois cadres (voir le chapitre suivant pour quelques exemples).

Chapitre 5

Modélisation de problèmes d'optimisation combinatoire à base de variables et de contraintes

5.1 Exemples de modélisation en SAT

5.1.1 Modélisation du problème de diagnostic de pannes sur un circuit logique

Considérons le micro-circuit logique de la figure 5.1, résultat de la combinaison d'une porte *et* et d'une porte *ou*.

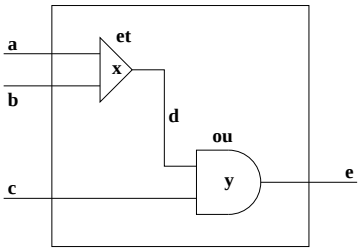

FIGURE 5.1 – Exemple de micro-circuit logique.

a, b, c, d et e représentent les valeurs courantes associées aux différentes connexions internes ou externes : 0 ou 1 (f ou t). x et y représentent l'état

courant des portes *et* et *ou* : t si la porte est en état de bon fonctionnement, f sinon.

On suppose avoir seulement accès aux valeurs des connexions externes, en entrée et en sortie du circuit (a, b, c et e). On suppose donc n'avoir accès, ni aux valeurs des connexions internes (d), ni à l'état interne des composants (x et y).

On voudrait pouvoir, sur la base de l'observation des valeurs des connexions externes, détecter et identifier des pannes au niveau des composants. On parle de *détection* et de *diagnostic*. L'approche que nous allons utiliser pour ce faire est dite *à base de modèle* : elle utilise un modèle de bon fonctionnement du système ; pour détecter l'occurence de pannes, elle va détecter l'incohérence entre ce modèle, une hypothèse de bon fonctionnement de l'ensemble des composants et les observations ; pour faire ensuite un diagnostic, elle va rechercher les hypothèses sur l'état des différents composants qui permettent de rétablir la cohérence entre modèle et observations.

Le modèle de bon fonctionnement du circuit que nous allons utiliser dit seulement que si la porte *et* (respectivement *ou*) est en état de bon fonctionnement, elle se comporte comme une porte *et* (respectivement *ou*). On obtient les deux équations logiques suivantes :

$$\begin{aligned} x &\rightarrow (d \leftrightarrow (a \wedge b)) \\ y &\rightarrow (e \leftrightarrow (c \vee d)) \end{aligned} \tag{5.1}$$

Ces deux équations peuvent s'écrire sous une forme normale conjonctive équivalente, c'est-à-dire sous la forme d'une instance SAT :

$$\begin{aligned} &\neg x \vee \neg d \vee a \\ &\neg x \vee \neg d \vee b \\ &\neg x \vee \neg a \vee \neg b \vee d \\ &\neg y \vee \neg e \vee c \vee d \\ &\neg y \vee \neg c \vee e \\ &\neg y \vee \neg d \vee e \end{aligned} \tag{5.2}$$

Supposons qu'on observe que $\{(a = t), (b = t), (c = f), (e = t)\}$. Si on fait l'hypothèse que $\{(x = t), (y = t)\}$, on constate que l'instance 5.3 est satisfiable : il existe une affectation de la seule variable non affectée ($(d = t)$) telle que toutes les clauses sont satisfaites. L'hypothèse de bon fonctionnement de l'ensemble des composants est donc raisonnable. En toute rigueur,

il est aussi possible que x soit en panne, que y soit en panne et même que les deux le soient. Mais on utilise le fait que le bon fonctionnement est largement plus probable que le mauvais (parti pris optimiste).

Supposons maintenant qu'on observe que $\{(a = t), (b = t), (c = f), (e = f)\}$. Si on fait la même hypothèse que $\{(x = t), (y = t)\}$, on constate que l'instance 5.3 est insatisfiable : il n'existe pas d'affectation de la variable non affectée (d) telle que toutes les clauses sont satisfaites. L'hypothèse de bon fonctionnement de l'ensemble des composants n'est plus valide. Si on relâche maintenant cette hypothèse, on obtient 4 combinaisons possibles des variables non affectées : $\{(x = t), (y = f), (d = t)\}$, $\{(x = f), (y = t), (d = f)\}$, $\{(x = f), (y = f), (d = t)\}$ et $\{(x = f), (y = f), (d = f)\}$. Si on ne s'intéresse qu'à x et y et si on utilise le fait qu'une panne est largement plus probable que deux simultanées (toujours le même parti pris optimiste), il reste deux hypothèses raisonnables : soit x est en panne, soit y est en panne. Pour aller plus loin, il faudrait pouvoir faire un test permettant d'observer d. Supposons qu'on observe que $(d = t)$. Il ne reste que 2 combinaisons possibles des variables non affectées : $\{(x = t), (y = f)\}$, $\{(x = f), (y = f)\}$ et plus qu'une hypothèse raisonnable : y est en panne.

Remarquons que dans certaines situations, il est possible de conclure sans test sur le composant en panne. Supposons par exemple qu'on observe que $\{(a = t), (b = t), (c = t), (e = f)\}$. Comme précédemment, l'hypothèse de bon fonctionnement de l'ensemble des composants n'est pas valide. Si on relâche cette hypothèse, on obtient 3 combinaisons possibles des variables non affectées : $\{(x = t), (y = f), (d = t)\}$, $\{(x = f), (y = f), (d = t)\}$ et $\{(x = f), (y = f), (d = f)\}$ et une seule hypothèse raisonnable : y est en panne.

5.2 Exemples de modélisation en PLNE

5.2.1 Modélisation du problème d'affectation de tâches à des personnes

Considérons le problème d'affectation de tâches à des personnes présenté dans la section 1.4. Soit n le nombre de tâches et de personnes et C_{ij} le coût de l'affectation de la tâche i à la personne j.

Associons une variable x_{ij} au fait que la tâche i est affectée ou non à la personne j : 1 si elle est affectée à cette personne, 0 sinon.

Nous devons exprimer qu'une tâche doit être affectée à une personne et une seule, qu'inversement une personne doit se voir affectée une tâche et une

seule et que le critère à minimiser est la somme des coûts des affectations. Il en résulte le problème *PLNE* (plus précisément *PL0/1*) suivant :

$$\begin{aligned} &\min \Big(\sum_{i,j=1}^{n} C_{ij} \cdot x_{ij} \Big) \\ &\forall i,\ 1 \leq i \leq n : \ \sum_{j} x_{ij} = 1 \\ &\forall j,\ 1 \leq j \leq n : \ \sum_{i} x_{ij} = 1 \end{aligned} \tag{5.3}$$

À noter que ce problème peut aussi se modéliser comme un problème de couplage optimum dans un graphe biparti pondéré (voir la section 1.4 ; voir aussi [Cav06b]), ce qui illustre le fait que le même problème puisse parfois être modélisé dans des cadres extrêmement différents.

5.2.2 Modélisation du problème des n reines

Le problème dit des n reines est un problème jouet qui s'exprime de la façon suivante : soit un échiquier $n \times n$ et n reines ; placer les n reines sur l'échiquier de telle façon qu'aucune ne soit en position de prendre une autre, c'est-à-dire de telle façon que deux reines ne se trouvent pas sur une même ligne, une même colonne ou une même diagonale.

Associons une variable x_{ij} à chaque case indicée par sa ligne i et sa colonne j : 1 si la case est occupée par une reine, 0 sinon.

Nous devons exprimer que, sur chaque ligne, sur chaque colonne et sur chaque diagonale, au plus une case est occupée. Il en résulte le problème *PLNE* (plus précisément *PL0/1*) suivant :

$$\begin{aligned} &\forall i,\ 1 \leq i \leq n : \ \sum_{j} x_{ij} = 1 \\ &\forall j,\ 1 \leq j \leq n : \ \sum_{i} x_{ij} = 1 \\ &\forall k,\ 2 \leq k \leq 2n : \sum_{i,j \,|\, i+j=k} x_{ij} \leq 1 \\ &\forall k,\ -(n-1) \leq k \leq (n-1) : \sum_{i,j \,|\, i-j=k} x_{ij} \leq 1 \end{aligned} \tag{5.4}$$

À noter qu'il s'agit d'un problème de satisfaction pure, sans critère à optimiser. Le problème a une solution évidente pour $n = 1$, pas de solution pour $n = 2$ et $n = 3$ et au moins une solution pour tout $n \geq 4$. La figure 5.2 montre un exemple de solution pour $n = 4$.

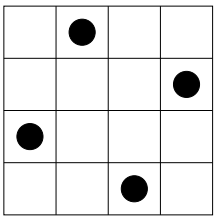

FIGURE 5.2 – Exemple de solution du problème des 4 reines.

5.2.3 Modélisation du problème des mariages stables

Considérons le problème (irréaliste) suivant, impliquant n hommes et n femmes. On suppose que chaque homme i est capable de classer les n femmes par ordre de préférence, de la plus à la moins préférée. Réciproquement, on suppose que chaque femme j est capable de classer les n hommes par ordre de préférence, du plus au moins préféré. On veut réaliser les n mariages tout en garantissant leur stabilité. Un mariage entre un homme i et une femme j est considéré comme stable ssi, si i préfère une autre femme j' à sa propre femme j, alors j' préfère son propre mari i' à i et réciproquement, si j préfère un autre homme i' à son propre mari i, alors i' préfère sa propre femme j' à j. Autrement dit, si un membre du couple est tenté d'aller voir ailleurs, cet ailleurs n'est pas partant.

Soit PH_{ij} le rang de la femme j dans l'ordre de préférence exprimé par l'homme i. Réciproquement, soit PF_{ji} le rang de l'homme i dans l'ordre de préférence exprimé par la femme j. Comme dans le problème d'affectation présenté dans la section 5.2.1, on associe à toute paire constituée d'un homme i et d'une femme j, une variable x_{ij} qui vaut 1 si i et j sont mariés et 0 sinon. Avec ces variables, les contraintes sont les suivantes :

$$\begin{aligned} &\forall i,\ 1 \leq i \leq n : \sum_j x_{ij} = 1 \\ &\forall j,\ 1 \leq j \leq n : \sum_i x_{ij} = 1 \end{aligned} \tag{5.5}$$

$$\forall i,j,i',j',\ 1 \leq i,j,i',j' \leq n,\ ((i \neq i') \wedge (j \neq j') \wedge$$
$$(((PH_{ij'} < PH_{ij}) \wedge (PF_{j'i} < PF_{j'i'})) \vee$$
$$((PF_{ji'} < PF_{ji}) \wedge (PH_{i'j} < PH_{i'j'}))))\ :$$
$$x_{ij} + x_{i'j'} \leq 1 \neq 1)$$

Les contraintes des deux premiers types imposent la monogamie : un homme est marié à une femme et une seule et réciproquement une femme est mariée à un homme et un seul. Celles du troisième type garantissent la stabilité : si deux mariages potentiels présentent des risques d'instabilité mutuelle, au moins l'un des deux n'est pas réalisé. Comme pour le problème des n reines, il s'agit d'un problème de satisfaction pure, sans critère à optimiser.

5.2.4 Modélisation du problème de choix d'instruments à embarquer sur un engin spatial

Considérons le problème de choix d'instruments à embarquer sur un engin spatial présenté dans la section 1.10. Soit n le nombre d'instruments candidats et m le nombre de dimensions à considérer. Soient C_{ij} la quantité requise par l'instrument i suivant la dimension j, CA_j la capacité de l'engin suivant la dimension j et U_i l'utilité de l'instrument i.

Associons une variable x_i au fait que l'instrument i est embarqué ou non : 1 s'il est embarqué, 0 sinon.

Nous devons simplement exprimer que les capacités de l'engin doivent être respectées suivant toutes les dimensions et que le critère à maximiser est la somme des utilités. Il en résulte le problème *PLNE* (plus précisément *PL0/1*) suivant :

$$\max\left(\sum_{i=1}^{n} U_i \cdot x_i\right) \tag{5.6}$$
$$\forall j,\ 1 \leq j \leq m :\ \sum_i C_{ij} \cdot x_i \leq CA_j$$

Pour ce type de problème, on préferera souvent raisonner en termes de priorité. On n'a plus une utilité U_i associée à chaque instrument i, mais une priorité P_i. L'objectif n'est plus de maximiser la somme des utilités associées aux instruments embarqués, mais de minimiser la priorité maximale associée aux instruments non embarqués. Le problème se formule alors de la façon suivante :

$$\min\left(\max_{i=1}^{n} P_i \cdot (1 - x_i)\right) \tag{5.7}$$
$$\forall j,\ 1 \leq j \leq m :\ \sum_i C_{ij} \cdot x_i \leq CA_j$$

Malheureusement, le critère (un *max*) n'est plus linéaire. Il est cependant facile de revenir à un critère linéaire en introduisant une variable p qui représente la priorité maximale associée aux instruments non embarqués et en exprimant que toutes les priorités associées aux instruments non embarqués sont bien inférieures à p. Il en résulte le problème *PLNE* (plus précisément *PL0/1*) suivant :

$$\min p \tag{5.8}$$
$$\forall i,\ 1 \leq i \leq n :\ P_i \cdot (1 - x_i) \leq p$$
$$\forall j,\ 1 \leq j \leq m :\ \sum_i C_{ij} \cdot x_i \leq CA_j$$

5.2.5 Modélisation du problème d'organisation de tâches de production

Considérons le problème d'organisation de tâches de production présenté dans la section 1.6. Soit n le nombre de tâches à organiser. Pour chaque tâche i, soient DU_i sa durée, TO_i sa date de début au plus tôt et TA_i sa date de fin au plus tard. Pour chaque paire de tâches i, j, soit PR_{ij} une constante représentant le fait qu'il existe ou non une contrainte de précédence entre i et j : 1 si i doit précéder j, 0 sinon. Pour chaque paire de tâches $i, j,\ i < j$, soit RP_{ij} une constante représentant le fait que i et j requièrent toutes deux l'usage d'une même ressource non partageable : 1 si c'est le cas, 0 sinon.

Première modélisation Associons une variable d_i réelle positive ou nulle à la date de début de la tâche i.

Nous devons exprimer que les dates de début au plus tôt et de fin au plus tard doivent être respectées, que les précédences entre tâches doivent aussi être respectées, qu'en cas de ressource partageable commune à deux tâches i et j, ou bien i doit précéder j, ou bien j doit précéder i. Nous devons enfin exprimer que le critère à minimiser est la date de fin maximum sur l'ensemble des tâches. Il en résulte le problème suivant :

$$\min\left(\max_{i=1}^{n}\left(d_i + DU_i\right)\right) \tag{5.9}$$
$$\forall i,\ 1 \leq i \leq n:\ TO_i \leq d_i \leq TA_i - DU_i$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ (PR_{ij} = 1):\ (d_i + DU_i \leq d_j)$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ i < j,\ (RP_{ij} = 1):\ ((d_i + DU_i \leq d_j) \vee (d_j + DU_j \leq d_i))$$

La difficulté est qu'un tel problème n'est pas linéaire et ce pour deux raisons : le critère (un *max*) n'est pas linéaire et les contraintes du dernier type, résultant du caractère non partageable des ressources, ne sont pas linéaires (ce sont des disjonctions de contraintes linéaires).

La première difficulté peut être réglée facilement de la même façon que dans la section précédente en introduisant une variable f qui représente la date de fin maximum sur l'ensemble des tâches et en exprimant que cette date est supérieure ou égale aux dates de fin de toutes les tâches. Il en résulte le problème suivant :

$$\min f \tag{5.10}$$
$$\forall i,\ 1 \leq i \leq n:\ d_i + DU_i \leq f$$
$$\forall i,\ 1 \leq i \leq n:\ TO_i \leq d_i \leq TA_i - DU_i$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ (PR_{ij} = 1):\ (d_i + DU_i \leq d_j)$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ i < j,\ (RP_{ij} = 1):\ ((d_i + DU_i \leq d_j) \vee (d_j + DU_j \leq d_i))$$

La seconde difficulté est plus délicate. Une façon classique de traiter une disjonction consiste à introduire une variable booléenne qui la représente explicitement. Pour chaque paire de tâches $i, j,\ i < j$ requérant la même ressource ($RP_{ij} = 1$), soit x_{ij} une variable booléenne qui vaut 1 si la tâche i précède la tâche j et 0 sinon. La contrainte $(d_i + DU_i \leq d_j) \vee (d_j + DU_j \leq d_i)$ peut alors s'écrire $x_{ij} \cdot (d_j - d_i - DU_i) + (1 - x_{ij}) \cdot (d_i - d_j - DU_j) \geq 0$, ce qui n'est toujours pas une contrainte linéaire. Une façon classique de la linéariser consiste à introduire une grande constante C négative. Il est facile de vérifier que la contrainte peut alors s'écrire sous la forme d'une conjonction de deux contraintes linéaires : $d_j - d_i - DU_i \geq (1 - x_{ij}) \cdot C$ et $d_i - d_j - DU_j \geq x_{ij} \cdot C$. Il en résulte le problème suivant qui est maintenant linéaire :

$$\min f \tag{5.11}$$
$$\forall i,\ 1 \leq i \leq n:\ d_i + DU_i \leq f$$

$$\forall i,\ 1 \leq i \leq n :\ TO_i \leq d_i \leq TA_i - DU_i$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ (PR_{ij} = 1) :\ (d_i + DU_i \leq d_j)$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ i < j,\ (RP_{ij} = 1) :\ (d_j - d_i - DU_i \geq (1 - x_{ij}) \cdot C)$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ i < j,\ (RP_{ij} = 1) :\ (d_i - d_j - DU_j \geq x_{ij} \cdot C)$$

Il s'agit plus précisément d'un problème de programmation linéaire mixte (PLM), puisque certaines variables sont réelles (d_i, f) et d'autres entières (x_{ij}). Il aussi facile de vérifier qu'il suffit de prendre pour C une valeur strictement inférieure au minimum sur toutes les tâches i et j partageant une ressource commune de la quantité $TO_i - TA_j - DU_j$.

Seconde modélisation Il existe d'autres façons de transformer ce problème en un problème linéaire mixte. Supposons que toutes les tâches requièrent une même ressource non partageable. Dans ces conditions, une solution prend forcément la forme d'une séquence de tâches.

Soit p_{ij} une variable booléenne qui représente le fait que la tâche i est en position j dans la séquence solution. Pour toute position j, soit d_j le décalage forcément positif ou nul entre la date de fin de la tâche située en position $j-1$ et la date de début de la tâche située en position j dans la séquence solution. d_1 est simplement la date de début de la tâche située en première position. On obtient le problème de programmation linéaire mixte (PLM) suivant :

$$\min \left(\sum_{j=1}^{n} d_j\right) \tag{5.12}$$
$$\forall i,\ 1 \leq i \leq n :\ \sum_{j=1}^{n} x_{ij} = 1$$
$$\forall j,\ 1 \leq i \leq n :\ \sum_{i=1}^{n} x_{ij} = 1$$
$$\forall i, i',\ 1 \leq i, i' \leq n,\ (PR_{ii'} = 1) :\ (\forall j,\ 1 \leq j \leq n :\ \sum_{j'=1}^{j} x_{ij'} \geq \sum_{j'=1}^{j} x_{i'j'})$$
$$\forall j,\ 1 \leq i \leq n :\ \sum_{j'=1}^{j} d_{j'} + \sum_{j'=1}^{j-1} \sum_{i=1}^{n} x_{ij'} \cdot DU_i \geq \sum_{i=1}^{n} x_{ij} \cdot TO_i$$

$$\forall j,\ 1 \leq i \leq n :\ \sum_{j'=1}^{j} d_{j'} + \sum_{j'=1}^{j} \sum_{i=1}^{n} x_{ij'} \cdot DU_i \leq \sum_{i=1}^{n} x_{ij} \cdot TA_i$$

Le premier type de contrainte exprime qu'une tâche est à une position et une seule dans la séquence. Le second exprime qu'une position dans la séquence est occupée par une tâche et une seule. Le troisième exprime les contraintes de précédence. Le quatrième exprime que les dates au plus tôt sont respectées et le cinquième que les dates au plus tard sont respectées [1].

Ceci illustre qu'il n'existe pas, pour un problème, de modélisation unique, même dans un cadre de modélisation fixé, comme celui de la programmation linéaire. Du point de vue de la lisibilité, certaines modélisations sont meilleures que d'autres, mais il s'agit là d'un critère extrêmement subjectif. Du point de vue de la complexité de résolution, certaines sont aussi meilleures que d'autres, mais il est très difficile de déterminer a priori lesquelles.

On notera que s'il n'y a pas de conflit d'accès à des ressources non partageables et donc pas de décision de séquencement à prendre, le problème peut se modéliser comme un problème de programmation linéaire (PL) simple :

$$\begin{aligned} &\min f \\ &\forall i,\ 1 \leq i \leq n :\ d_i + DU_i \leq f \\ &\forall i,\ 1 \leq i \leq n :\ TO_i \leq d_i \leq TA_i - DU_i \\ &\forall i, j,\ 1 \leq i, j \leq n,\ (PR_{ij} = 1) :\ (d_i + DU_i \leq d_j) \end{aligned} \tag{5.13}$$

Mais il peut aussi se modéliser comme un problème de plus long chemin dans un graphe orienté pondéré (voir la section 1.5 ; voir aussi [Cav06b]), ce qui illustre encore une fois le fait que le même problème puisse parfois être modélisé dans des cadres a priori extrêmement différents.

5.3 Exemples de modélisation en CSP

5.3.1 Modélisation du problème des n reines

Reprenons le problème des n reines présenté dans la section 5.2.2.

1. Pour toute tâche i et toute position j, $\sum_{j'=1}^{j} x_{ij'}$ représente le fait que la tâche i se trouve avant la position j ou à la position j dans la séquence. Pour toute position j, $\sum_{i=1}^{n} x_{ij} \cdot PA_i$ est la valeur du paramètre PA (DU, TO ou TA) pour la tâche située à la position j dans la séquence.

Pour le modéliser dans le cadre CSP, on est tenté d'associer à chaque reine i une ligne (y_i) et une colonne (x_i). Mais une simple réflexion permet de diminuer par deux le nombre de variables. En effet, comme deux reines ne peuvent pas se trouver sur une même ligne et qu'il y a autant de lignes que de reines, il y a forcément une reine par ligne. Le problème est de déterminer pour chaque ligne la colonne sur laquelle se trouve la reine de cette ligne, ce qui conduit à la modélisation suivante.

Pour chaque ligne $i,\ 1 \leq i \leq n$, soit x_i la colonne sur laquelle est placée la reine de cette ligne. On a : $\forall i,\ 1 \leq i \leq n :\ 1 \leq x_i \leq n$.

Nous devons exprimer que deux reines ne peuvent pas se trouver sur une même colonne ou une même diagonale, ce qui donne les contraintes suivantes :

$$\begin{aligned} &\forall i, j,\ 1 \leq i, j \leq n,\ i < j :\ x_i \neq x_j \\ &\forall i, j,\ 1 \leq i, j \leq n,\ i < j :\ |x_i - x_j| \neq (j - i) \end{aligned} \tag{5.14}$$

À noter que ces contraintes sont non linéaires. On aurait pu remplacer la contrainte $|x_i - x_j| \neq j - i$ par une conjonction de deux contraintes : $(x_i - x_j) \neq (j - i)$ et $(x_j - x_i) \neq (j - i)$. Mais une contrainte de différence n'est pas linéaire. C'est une disjonction de deux contraintes linéaires.

5.3.2 Modélisation du problème du cavalier d'Euler

Le problème dit du cavalier d'Euler est un autre problème jouet qui s'exprime de la façon suivante : soit un échiquier $n \times n$ et 1 cavalier placé sur une case quelconque ; déplacer le cavalier en respectant les règles de mouvement d'un cavalier et de telle façon qu'il parcourt toutes les cases de l'échiquier une fois et une seule et revienne à son point de départ. Comme il s'agit de parcourir toutes les cases et de revenir au point de départ, on peut sans perte de généralité supposer que le cavalier se trouve au départ sur la case $(1, 1)$.

Première modélisation Dans une première modélisation, nous associons une variable (en fait deux) à chaque instant, représentant la position du cavalier à cet instant. Comme le cavalier doit parcourir n^2 cases et revenir à son point de départ, nous considérons $n^2 + 1$ instants que nous numérotons de 0 à n^2. Pour chaque instant $i,\ 0 \leq i \leq n^2$, soient x_i et y_i les positions en x et en y du cavalier à cet instant. On a : $\forall i,\ 0 \leq i \leq n^2 :\ 1 \leq x_i, y_i \leq n$.

Nous devons exprimer que le cavalier se trouve sur la case $(1, 1)$ à l'instant 0 et à l'instant n^2, que, sauf en ce qui concerne les instants 0 et n^2, il ne se

trouve pas à deux instants sur la même case et enfin qu'il effectue bien des mouvements de cavalier, ce qui donne les contraintes suivantes :

$$
\begin{aligned}
& x_0 = y_0 = x_{n^2} = y_{n^2} = 1 \\
& \forall i, j,\ 0 \leq i, j \leq n^2,\ (i < j) \wedge \neg(i = 0 \wedge j = n^2) :\ ((x_i \neq x_j) \vee (y_i \neq y_j)) \\
& \forall i,\ 0 \leq i \leq (n^2 - 1) :\ |x_i - x_{i+1}| \cdot |y_i - y_{i+1}| = 2
\end{aligned}
\tag{5.15}
$$

Seconde modélisation Dans une seconde modélisation (duale) nous associons une variable à chaque case, représentant l'instant auquel le cavalier passe par cette case. Comme il y a n^2 cases, nous considérons n^2 instants que nous numérotons de 1 à n^2. Pour chaque case $i, j,\ 1 \leq i, j \leq n$, soit t_{ij} l'instant auquel le cavalier passe dans cette case. On a : $\forall i, j,\ 1 \leq i, j \leq n$: $1 \leq t_{ij} \leq n^2$.

Nous devons exprimer que le cavalier se trouve sur la case $(1, 1)$ à l'instant 1 et sur une case à partir de laquelle il peut atteindre la case $(1, 1)$ à l'instant n^2, qu'il ne se trouve pas sur deux cases au même instant et enfin qu'il effectue bien des mouvements de cavalier, ce qui donne les contraintes suivantes :

$$
\begin{aligned}
& t_{11} = 1 \\
& \forall i, j,\ 1 \leq i, j \leq n,\ (i-1) \cdot (j-1) \neq 2 :\ t_{ij} \neq n^2 \\
& \forall i, j, i', j',\ 1 \leq i, j, i', j' \leq n,\ ((i < i') \vee (j < j')) :\ t_{ij} \neq t_{i'j'} \\
& \forall i, j, i', j',\ 1 \leq i, j, i', j' \leq n,\ ((i < i') \vee (j < j')) \wedge (|i' - i| \cdot |j' - j| \neq 2) : |t_{i'j'} - t_{ij}| \neq 1
\end{aligned}
\tag{5.16}
$$

Troisième modélisation Dans une troisième modélisation nous associons de nouveau une variable (en fait deux) à chaque instant, mais cette variable représente le mouvement du cavalier à cet instant. Comme il y a n^2 mouvements à réaliser, nous considérons n^2 instants que nous numérotons de 1 à n^2. Pour chaque instant $i,\ 1 \leq i \leq n^2$, soient dx_i et dy_i les mouvements en x et en y du cavalier à cet instant. On a : $\forall i,\ 1 \leq i \leq n^2 :\ dx_i, dy_i \in \{-2, -1, 1, 2\}$.

Nous devons exprimer que le cavalier revient au bout de n^2 mouvements à son point de départ, qu'il ne passe pas deux fois sur la même case, qu'il ne sort pas de l'échiquier et qu'il effectue bien des mouvements de cavalier, ce qui donne les contraintes suivantes :

$$\sum_{i=1}^{n^2} dx_i = \sum_{i=1}^{n^2} dy_i = 0 \tag{5.17}$$

$$\forall i,j,\ 1 \leq i,j \leq n^2,\ (i<j):\ (\sum_{k=i}^{j-1} dx_k \neq 0) \vee (\sum_{k=i}^{j-1} dy_k \neq 0)$$

$$\forall i,\ 1 \leq i \leq n^2:\ (0 \leq \sum_{j=1}^{i-1} dx_j \leq (n-1)) \wedge (0 \leq \sum_{j=1}^{i-1} dy_j \leq (n-1))$$

$$\forall i,\ 1 \leq i \leq n^2:\ |dx_i| \cdot |dy_i| = 2$$

Des trois modélisations, il est difficile de dire quelle est la plus lisible. Des expérimentations seraient nécessaires pour déterminer celle qui facilite le plus la résolution. Mais on peut remarquer que la première implique $2{\cdot}(n^2{+}1)$ variables de domaines de taille n, que la seconde implique n^2 variables de domaines de taille n^2 et que la troisième implique n^2 variables de domaines de taille 4, ce qui penche en faveur de cette dernière (domaines de taille beaucoup plus réduite).

6	**16**	**29**	**14**	**25**	**18**	**27**
5	**3**	**24**	**17**	**28**	**13**	**10**
4	**30**	**15**	**4**	**11**	**26**	**19**
3	**5**	**2**	**23**	**34**	**9**	**12**
2	**22**	**31**	**36**	**7**	**20**	**33**
1	**1**	**6**	**21**	**32**	**35**	**8**
	1	**2**	**3**	**4**	**5**	**6**

FIGURE 5.3 – Exemple de solution du problème du cavalier d'Euler pour $n = 6$.

À noter que ce problème peut aussi se modéliser comme un problème de recherche d'un circuit hamiltonien dans le graphe dont les sommets représentent les cases de l'échiquier et une arête existe entre deux sommets si et seulement si un cavalier peut passer directement de la case associée au premier sommet à la case associée à la seconde (voir la section 1.2 ; voir

aussi [Cav06b]), ce qui illustre encore une fois le fait que le même problème puisse parfois être modélisé dans des cadres extrêmement différents.

Le problème a une solution évidente pour $n = 1$ et pas de solution pour $2 \leq n \leq 5$. La figure 5.3 montre un exemple de solution pour $n = 6$.

5.3.3 Modélisation d'un problème de planification d'actions

Considérons le problème informellement décrit de la façon suivante : *T*he U2 group has a concert that starts in 17 minutes and they must all cross a bridge to get there. All four men begin on the same side of the bridge. You must help them across to the other side. It is night. There is one flashlight. A maximum of two people can cross at one time. Any party who crosses, either 1 or 2 people, must have the flashlight with them. The flashlight must be walked back and forth, it cannot be thrown, etc. Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace : 1 minute to cross for Bono, 2 minutes for Edge, 5 minutes for Adam, and 10 minutes for Larry. For example : if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.

La première difficulté tient au fait que nous ne connaissons pas a priori le nombre de traversées nécessaires à la traversée de l'ensemble du groupe. C'est une constante en planification où nous ne connaissons pas a priori le nombre d'actions nécessaire à la réalisation d'un but. Une façon de contourner cette difficulté consiste à considérer le problème plus simple de recherche d'un plan de longueur k fixée, à commencer avec $k = 1$ et à incrémenter k chaque fois que l'absence de solution a été établie, jusqu'à trouver une solution qui est alors forcément optimale en termes de nombre d'actions. Pour le problème de U2, considérons donc le problème de trouver un plan en k traversées.

S'il y a k traversées, il y a $k+1$ états successifs que nous numérotons de 0 à k. Nous modélisons un état du groupe grâce à 5 variables qui représentent la position de chacun des membres du groupe et de la lampe à un instant donné : 0 pour la rive de départ, 1 pour la rive d'arrivée. Pour tout $p \in \{b, e, a, l, f\}$ [2] et pour tout instant i, $0 \leq i \leq k$, soit $x_{p,i}$ la position de p à l'instant i. Pour tout instant i, $0 \leq i \leq k$, soit t_i le temps réel écoulé depuis l'instant 0. On a : $\forall i,\ 0 \leq i \leq k,\ \forall p \in \{b, e, a, l, f\} :\ x_{p,i} \in \{0, 1\}$ et $\forall i,\ 0 \leq i \leq k :\ 0 \leq t_i \leq 17$. Pour tout $p \in \{b, e, a, l\}$, soit T_p le temps de traversée de p.

2. b pour Bono, e pour Edge, a pour Adam, l pour Larry et f pour la lampe.

Nous obtenons les contraintes suivantes :

$$
\begin{aligned}
&t_0 = 0 \\
&\forall p\ p \in \{b,e,a,l,f\} :\ x_{p,0} = 0 \\
&\forall p\ p \in \{b,e,a,l,f\} :\ x_{p,k} = 1 \\
&\forall i,\ 1 \le i \le k :\ 1 \le (\sum_{p \in \{b,e,a,l\}} |x_{p,i} - x_{p,(i-1)}|) \le 2 \\
&\forall i,\ 1 \le i \le k,\ \forall p \in \{b,e,a,l\} :\ (x_{p,i} \neq x_{p,(i-1)}) \rightarrow ((x_{f,i} - x_{f,(i-1)}) = (x_{p,i} - x_{p,(i-1)})) \\
&\forall i,\ 1 \le i \le k,\ \forall p \in \{b,e,a,l\} :\ (x_{p,i} \neq x_{p,(i-1)}) \rightarrow ((t_i - t_{i-1}) \ge T_p)
\end{aligned}
\tag{5.18}
$$

La première contrainte exprime que le temps réel est nul à l'instant 0. Les contraintes du second type expriment que les membres du groupe et la lampe se trouvent sur la rive de départ à l'instant 0 et celles du troisième type qu'ils se trouvent sur la rive d'arrivée à l'instant k. Les contraintes du quatrième type expriment qu'une ou deux personnes traversent à chaque fois [3]. Celles du cinquième type expriment que personne ne traverse sans la lampe. Les dernières expriment qu'une traversée prend un temps supérieur ou égal au temps requis par chacune des personnes qui traversent.

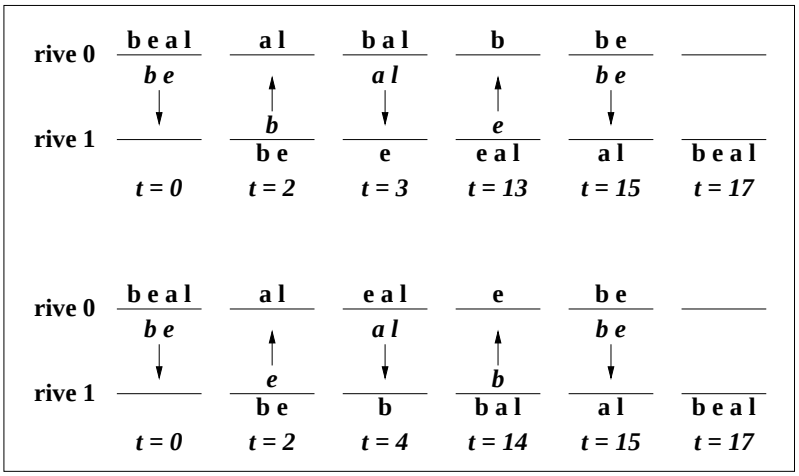

FIGURE 5.4 – Les deux solutions du problème de U2, chacune à 5 traversées.

3. Le fait d'imposer qu'une personne traverse à chaque fois évite les transitions inutiles ou rien ne change.

Les deux solutions possibles sont représentées sur la figure 5.4. Elles impliquent toutes deux 5 traversées. À chaque instant, la flèche indique le sens de la traversée et les lettres en italique les personnes impliquées dans la traversée.

5.3.4 Modélisation du problème d'affectation de fréquences à des liaisons radio

Considérons le problème d'affectation de fréquences à des liaisons radio présenté dans la section 1.8. Soit n le nombre de sites. Pour chaque paire de sites (i, j), soit L_{ij} une constante représentant le fait qu'une liaison doit être assurée de i vers j : 1 si c'est le cas, 0 sinon. Soit k la distance exacte qui doit être respectée entre une liaison (de i vers j) et sa liaison inverse (de j vers i). Pour chaque paire de sites (i, j), soit k_{ij} la distance minimum qui doit être respectée entre une liaison arrivant sur i et une autre arrivant sur j (dépendante de la distance entre les sites i et j). Soit $(FMin, FMax)$ la spectre de fréquences utilisable. On le suppose discrétisé.

Associons une variable f_{ij} à chaque paire de sites telle que $L_{ij} = 1$. On a : $\forall i, j \ L_{ij} = 1 : \ FMin \leq f_{ij} \leq FMax$.

Si on cherche à minimiser le spectre de fréquences utilisé, on obtient le critère et les contraintes suivants :

$$\min_{i,j,i',j',\ L_{ij}=L_{i'j'}=1} (|f_{ij} - f_{i'j'}|) \tag{5.19}$$
$$\forall i, j, \ 1 \leq i, j \leq n, \ L_{ij} = L_{ji} = 1 : \ |f_{ij} - f_{ji}| = k$$
$$\forall i, j, i', j', \ 1 \leq i, j, i', j', \leq n, \ L_{ij} = L_{i'j'} = 1 : \ |f_{ij} - f_{i'j'}| \geq k_{jj'}$$

5.3.5 Modélisation du problème d'organisation de tâches de production

Considérons le problème d'organisation de tâches de production présenté dans la section 1.6. Nous avons présenté en section 5.2.5 différentes façons de le modéliser dans le cadre de la programmation linéaire mixte (PLM), mais nous avons constaté que cette modélisation n'était pas immédiate, essentiellement du fait du caractère non linéaire des contraintes résultant du caractère non partageable de certaines ressources.

Cet obstacle disparaît dans le cadre *CSP* où les contraintes sont quelconques. La contrepartie est que les variables représentant les dates de début des tâches ne peuvent plus être réelles. Elles doivent être entières, ce qui im-

pose de discrétiser le temps, avec le caractère toujours arbitraire d'une telle démarche.

Supposons que cela soit fait et associons une variable d_i entière à la date de début de chaque tâche i. On a : $\forall i,\ 1 \leq i \leq n :\ TO_i \leq d_i \leq TA_i - DU_i$. On obtient le critère et les contraintes suivants :

$$\min\left(\max_{i=1}^{n}\left(d_i + DU_i\right)\right) \tag{5.20}$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ (PR_{ij} = 1) :\ (d_i + DU_i \leq d_j)$$
$$\forall i, j,\ 1 \leq i, j \leq n,\ i < j,\ (RP_{ij} = 1) :\ ((d_i + DU_i \leq d_j) \vee (d_j + DU_j \leq d_i))$$

Le fait que le critère et les contraintes du dernier type ne soient pas linéaires ne posent pas problème, au moins en termes de modélisation.

Chapitre 6

Méthodes de raisonnement

Dans ce chapitre nous présentons un ensemble de méthodes dont l'objectif premier n'est pas de produire des solutions, mais de mieux cerner l'espace des solutions en déduisant, à partir des contraintes initiales du problème (contraintes explicites), de nouvelles contraintes, conséquences logiques des précédentes (contraintes implicites).

Nous présentons ces méthodes principalement dans le cadre *CSP* au sein duquel elles ont été largement développées. Mais nous montrons à la fin de ce chapitre sous quelle forme elles existent aussi dans les cadres *SAT* et *PLNE*.

6.1 Méthodes de raisonnement dans le cadre CSP

Nous avons remarqué qu'un problème de satisfaction de contraintes résulte de la conjonction d'un grand nombre de contraintes qui sont à la fois locales et interdépendantes du fait des variables qu'elles partagent.

L'intuition présente derrière les mécanismes de raisonnement proposés est que raisonner directement sur le problème global est bien trop complexe, mais que raisonner sur des sous-problèmes locaux peut permettre de produire des informations qui seront utiles pour le raisonnement global.

Plus précisément, considérons une instance P, impliquant un ensemble de variables V et un ensemble de contraintes C, et une sous-instance P', impliquant un sous-ensemble des contraintes $C' \subset C$ et l'ensemble des variables sur lesquelles elles pèsent $V' \subset V$. Considérons un sous-ensemble de variables $V'' \subseteq V'$ et supposons qu'on puisse établir qu'une certaine affectation A des variables de V'' ne participe à aucune solution de P' (voir la figure 6.1). On peut en déduire que A ne participe à aucune solution de P et ajouter

explicitement à la définition de P le fait que la combinaison de valeurs A est interdite. C'est l'opération de base réalisée par les mécanismes de raisonnement les plus utilisés dans le cadre *CSP*. Ces mécanismes peuvent donc être vus comme des mécanismes de *déduction* qui permettent d'expliciter des contraintes qui étaient implicites dans la définition de l'instance. Comme ils éliminent des combinaisons de valeurs des relations existantes ou des valeurs des domaines existants, ils peuvent aussi être vus comme des mécanismes de *simplification* ou de *filtrage*. Le cas extrême de simplification se produit quand une relation ou un domaine devient vide suite à des éliminations successives. Dans ce cas, aucune solution ne peut exister et l'incohérence de P a été établie. Ils peuvent donc être vus comme des mécanismes de *détection d'incohérence*. Ajoutons que l'élimination d'une combinaison de valeurs A peut conduire à l'élimination d'une autre combinaison de valeurs A' qui n'avait aucune raison d'être éliminée jusqu'alors, mais qui se révèle ne participer à aucune solution d'une sous-instance P' du fait de l'élimination de A. C'est cet effet de propagation possible dans le réseau de contraintes qui a conduit à présenter ces mécanismes comme des mécanismes de *propagation de contraintes*.

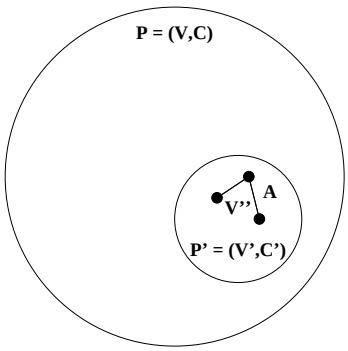

FIGURE 6.1 – Raisonnement sur un problème local.

6.1.1 Cohérence locale

Pour réaliser un certain type de déduction, on s'appuie généralement sur une propriété dite de *cohérence locale*, notée π. Soit une instance P. Cette propriété porte sur des combinaisons de valeurs de variables de P, c'est-à-dire sur sur des affectations partielles. Elle peut être vérifiée localement, en ne considérant que des ensembles limités de variables et de contraintes de

P. Son non-respect par une combinaison de valeurs A implique que A ne participe à aucune solution de P. Éliminer A de P (interdire A dans P) ne modifie pas l'ensemble des solutions de P. L'instance résultante est dite équivalente à P. Dans ces conditions, partant d'une instance P, on élimine toutes les combinaisons de valeurs qui ne vérifient pas la propriété π, avec deux résultats possibles :

1. une relation ou un domaine devient vide, ce qui prouve l'incohérence de P ;
2. l'instance résultant des éliminations successives vérifie la propriété π ; le résultat noté $\pi(P)$ est une instance simplifiée et équivalente à P.

Il est important de souligner que, puisque π est une propriété de cohérence locale qui ne considère pas une instance de façon globale, rien ne garantit dans la seconde situation que $\pi(P)$ soit cohérent (voir l'exemple de la figure 6.6). On peut résumer la situation en disant que l'incohérence locale implique l'incohérence (situation 1), mais que la cohérence locale n'implique pas la cohérence (situation 2). On la résume aussi en disant que les mécanismes qui établissent la cohérence locale sont *incomplets* : ils peuvent ne pas détecter l'incohérence.

6.1.2 Arc-cohérence

À titre d'illustration, prenons l'exemple d'un niveau de cohérence locale, parmi les plus simples et les plus utilisés, à savoir l'*arc-cohérence* (*arc-consistency*) dans le cadre des *CSP* binaires.

Une valeur val d'une variable v est dite arc-cohérente selon une contrainte c qui la lie à une autre variable v', s'il existe dans le domaine de v' au moins une valeur val' cohérente avec val, c'est-à-dire telle que l'affectation $\{(v = val), (v' = val')\}$ satisfasse la contrainte c (voir la figure 6.2). Elle est dite arc-cohérente si elle est arc-cohérente selon toutes les contraintes qui pèsent sur elle. Une instance est dite arc-cohérente si toutes les valeurs des domaines de toutes les variables le sont.

L'algorithme le plus naïf permettant de filtrer une instance par arc-cohérence consiste à parcourir toutes les valeurs de tous les domaines, à éliminer chaque valeur qui ne vérifie pas la propriété d'arc-cohérence et à refaire ce parcours tant que le parcours précédent a éliminé au moins une valeur. En effet une valeur val d'une variable v qui était arc-cohérente selon une contrainte c, la liant à une autre variable v', du fait de l'existence dans le domaine de v' d'une valeur val' cohérente avec val, peut se retrouver arc-incohérente si val' était la seule valeur cohérente avec val dans le domaine

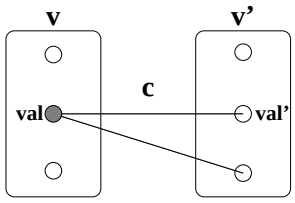

FIGURE 6.2 – Sous-instance considérée par l'arc-cohérence.

de v' et si elle en a été éliminée (propagation des éliminations dans le réseau de contraintes).

La figure 6.3 montre le résultat de ce filtrage sur une instance impliquant 3 variables v_1, v_2 et v_3, ayant chacune deux valeurs possibles 1 et 2 et liées par deux contraintes c_1 et c_2, c_1 spécifiant que $v_1 = v_2$ et c_2 que $v_2 > v_3$. Les valeurs éliminées sont indiqués en noir. Si v_1, v_2 et v_3 sont traitées dans l'ordre, un premier parcours permet d'éliminer la valeur 1 de v_2 car il n'existe dans le domaine de v_3 aucune valeur compatible avec elle et la valeur 2 de v_3 car il n'existe dans le domaine de v_2 aucune valeur compatible avec elle. Comme des valeurs ont été éliminées, un second parcours est nécessaire. Il permet d'éliminer la valeur 1 de v_1 car il n'existe plus aucune valeur compatible avec elle dans le domaine de v_2. Un troisième parcours est nécessaire, mais ne produit aucune élimination. L'instance résultante est arc-cohérente et équivalente à l'instance initiale.

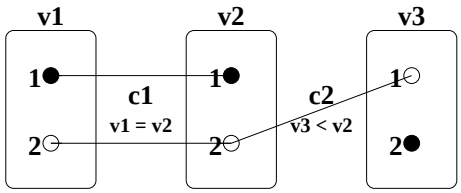

FIGURE 6.3 – Arc-cohérence : exemple de simplification.

La figure 6.4 montre le résultat de ce filtrage sur une instance identique à la précédente, sauf que la contrainte c_2 ne spécifie plus que $v_2 > v_3$, mais que $v_2 \neq v_3$. Aucune valeur n'est éliminée car l'instance initiale est déjà arc-cohérente.

La figure 6.5 montre le résultat de ce filtrage sur une instance proche des deux précédentes, où la contrainte c_1 autorise la seule paire $(1, 1)$ et la contrainte c_2 la seule paire $(2, 2)$. Le premier parcours élimine la valeur 2 de

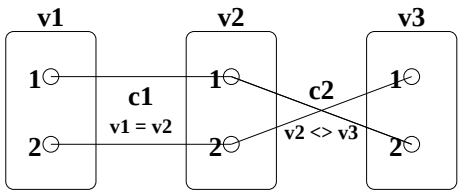

FIGURE 6.4 – Arc-cohérence : exemple de non simplification.

v_1 et les valeurs 1 et 2 de v_2. À ce stade, le domaine de v_2 est vide. Aller plus loin est inutile, car l'incohérence de l'instance a été démontrée.

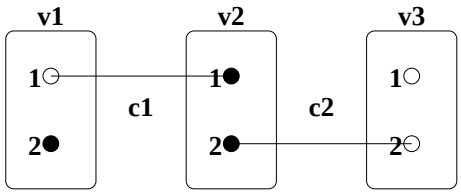

FIGURE 6.5 – Arc-cohérence : exemple de détection d'incohérence.

La figure 6.6 montre le résultat de ce filtrage sur une instance proche des trois précédentes, mais avec trois contraintes c_1, c_2 et c_3, toutes trois contraintes de différence. De la même façon que sur l'exemple de la figure 6.4, aucune valeur n'est éliminée car l'instance initiale est déjà arc-cohérente. Elle est pourtant clairement incohérente, ce qui montre bien que cohérence locale n'implique pas cohérence globale.

De nombreux algorithmes, moins naïfs que celui qui a été présenté plus haut, ont été proposés pour filtrer une instance par arc-cohérence. Celui qui est encore le plus utilisé est dénommé *AC3*. Il consiste à gérer un ensemble Q de paires variable-contrainte à visiter. Q est initialisé avec toutes les paires (v, c) telles que v est une variable et c une contrainte pesant sur elle. Chaque fois qu'une paire $(v, c) \in Q$ est sélectionnée, elle est retirée de Q, le domaine de v est réduit en éliminant toutes les valeurs qui sont arc-incohérentes selon c et, s'il a été effectivement réduit, toutes les paires (v', c') telles que $v' \neq v$ et c' pèse sur v sont ajoutées à Q (propagation). L'algorithme termine quand un domaine est vide (situation 1) ou quand Q est vide (situation 2). Sa complexité temporelle est en $O(e \cdot d^3)$. Bien que d'autres algorithmes de complexité temporelle plus faible et optimale, en $O(e \cdot d^2)$, aient été proposés depuis, la grande facilité d'implémentation d'*AC3* et sa capacité

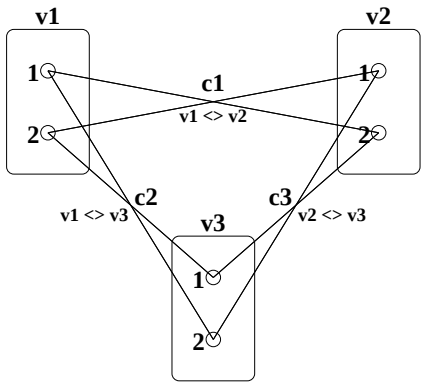

FIGURE 6.6 – Arc-cohérence : exemple de non détection d'incohérence.

d'adaptation à des cadres plus larges que le cadre *CSP* classique (domaines numériques continus (voir la section 6.1.4) ou contraintes faisant appel à des méthodes de propagation particulières (voir ci-dessous)) font qu'il est encore maintenant souvent préféré à ses concurrents.

L'arc-cohérence a été généralisée du cas des contraintes binaires à celui de contraintes *non binaires*. Elle a aussi été spécialisée pour traiter plus efficacement des contraintes ayant des propriétés particulières comme la *monotonie*, qui permet de remplacer un raisonnement sur les valeurs par un raisonnement sur les bornes des domaines. Les contraintes d'inégalité sont un exemple de contrainte monotone. Supposons deux variables x et y de domaines respectifs $[a, b]$ et $[c, d]$ reliées par la contrainte $x \leq y$. Cette contrainte permet de diminuer la borne supérieure du domaine de x et d'augmenter la borne inférieure du domaine de y. Plus précisément, les domaines résultants de x et y sont $[a, min(b, d)]$ et $[max(a, c), d]$ (voir la section 6.1.4 pour une généralisation à des domaines continus).

6.1.3 Au delà de l'arc-cohérence

On peut considérer que raisonner sur une contrainte et les deux variables sur lesquelles elle pèse, comme le fait l'arc-cohérence sur des contraintes binaires, est un raisonnement trop local pour produire des informations intéressantes (voir l'exemple de la figure 6.6). D'où la proposition de propriétés de cohérence locale plus fortes.

L'exemple le plus connu est celui de la *chemin-cohérence* (*path-consistency*). Considérons trois variables v, v' et v'' et les contraintes qui les lient (c entre

v et v', c' entre v et v'' et c'' entre v' et v''). Une paire (val, val') constituée d'une valeur $val \in v$ et d'une valeur $val' \in v'$, satisfaisant c, est dite chemin-cohérente selon v'' s'il existe dans le domaine de v'' au moins une valeur val'' telle que l'affectation $\{(v = val), (v' = val'), (v'' = val'')\}$ satisfasse aussi c' et c'' (voir la figure 6.7). Elle est dite chemin-cohérente si elle est chemin-cohérente selon toutes les autres variables. Une instance est dite chemin-cohérente si toutes les paires de valeurs autorisées par les contraintes le sont.

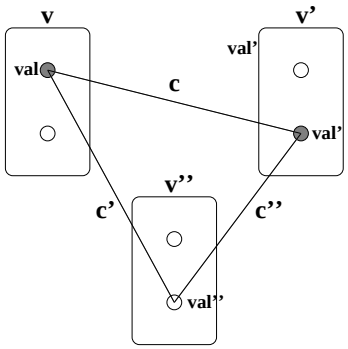

FIGURE 6.7 – Sous-instance considérée par la chemin-cohérence.

Comme pour l'arc-cohérence, l'algorithme le plus naïf permettant de filtrer une instance par chemin-cohérence consiste à parcourir toutes les paires de valeurs autorisées par les contraintes (même celles entre deux variables non liées par une contrainte), à éliminer chaque paire qui ne vérifie pas la propriété de chemin-cohérence et à refaire ce parcours tant que le parcours précédent a éliminé au moins une paire.

La figure 6.8 montre le résultat de ce filtrage sur une instance impliquant 3 variables v_1, v_2 et v_3, ayant chacune deux valeurs possibles 1 et 2 et liées par deux contraintes c_1 et c_2, c_1 spécifiant que $v_1 \neq v_3$ et c_2 que $v_2 \neq v_3$. Comme aucune contrainte ne lie v_1 et v_2, toutes les paires de valeurs sont implicitement autorisées entre ces deux variables. Les paires de valeurs éliminées sont indiquées en gras. Par exemple, la paire $\{(v_1 = 1), (v_2 = 2)\}$ est éliminée car il n'existe dans le domaine de v_3 aucune valeur qui soit compatible à la fois avec $(v_1 = 1)$ et $(v_2 = 2)$. L'instance résultante est chemin-cohérente et équivalente à l'instance initiale.

La figure 6.9 montre le résultat de ce filtrage sur l'instance de la figure 6.6. Comme précédemment les paires de valeurs $\{(v_1 = 1), (v_2 = 2)\}$ et $\{(v_1 = 2), (v_2 = 1)\}$ sont éliminées, ce qui produit une relation vide. Le filtrage

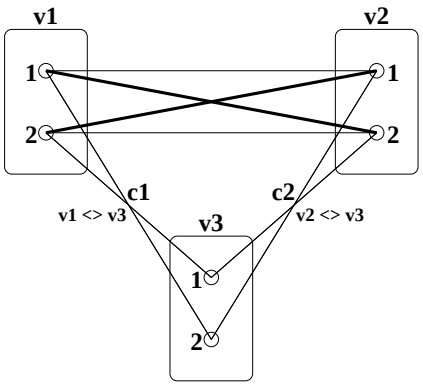

FIGURE 6.8 – Chemin-cohérence : exemple de simplification.

par chemin-cohérence prouve donc l'incohérence de cette instance, ce que ne fait pas le filtrage par arc-cohérence. Ceci n'implique évidemment pas que le filtrage par chemin-cohérence soit complet : il peut lui aussi sur d'autres instances ne pas détecter l'incohérence.

À noter que l'élimination de paires de valeurs par chemin-cohérence peut entraîner qu'une valeur précédemment arc-cohérente ne l'est plus. D'où la possibilité d'activation du filtrage par arc-cohérence par le filtrage par chemin-cohérence et inversement.

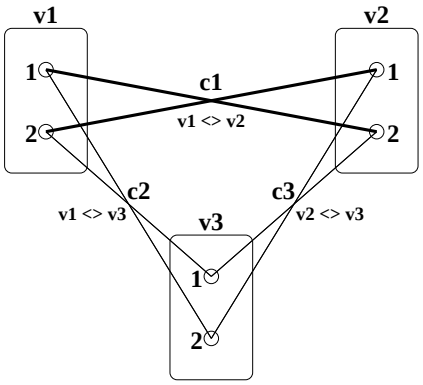

FIGURE 6.9 – Chemin-cohérence : exemple de détection d'incohérence.

Alors que la complexité temporelle du filtrage par arc-cohérence est en $O(e \cdot d^2)$, avec $e < n^2$, celle du filtrage par chemin-cohérence est en $O(n^3 \cdot d^3)$.

La notion de (i, j)-*cohérence* permet de généraliser arc et chemin-cohérence. Informellement, une instance est dite (i, j)-cohérente si toute affectation cohérente de i variables peut se prolonger de façon cohérente sur tout ensemble de j variables distinctes des i premières. Dans le cas binaire, arc-cohérence équivaut à $(1, 1)$-cohérence et chemin-cohérence à $(2, 1)$-cohérence.

6.1.4 Extension au cadre des CSP continus

En fait, les mécanismes de raisonnement à base d'arc-cohérence peuvent être généralisés au cadre des *CSP continus*, c'est à dire aux *CSP* dans lesquels les variables ou au moins certaines d'entre elles ont des domaines de valeur numériques continus, représentés par des intervalles. À titre d'exemple considérons l'instance suivante impliquant 4 variables réelles x, y, z et t et les contraintes suivantes : $x \leq 10$, $4 \leq y \leq 5$, $1 \leq z \leq 3$, $0 \leq t \leq 1.5$, $x = y \cdot z$ et $z \leq t$. Les 4 premières contraintes unaires délimitent les domaines des variables. La cinquième est ternaire et la sixième binaire. On remarquera que la contrainte ternaire est non linéaire.

Le calcul sur les intervalles permet d'étendre le raisonnement local à une contrainte au cadre des *CSP* continus. Soit une contrainte c impliquant un ensemble de variables V' et soit $v \in V'$. Ce raisonnement local utilise la contrainte c et les domaines courants des variables de $V' - \{v\}$ pour réduire le domaine de v. Supposons par exemple la contrainte $x = y + z$ avec $x \in [a_x, b_x]$, $y \in [a_y, b_y]$ et $z \in [a_z, b_z]$. Cette contrainte peut être utilisée pour réduire le domaine de x. En effet, de la contrainte et des domaines de y et z, on peut déduire que $x \in [a_y + a_z, b_y + b_z]$ et donc que $x \in [a_x, b_x] \cap [a_y + a_z, b_y + b_z]$. Mais elle peut aussi être utilisée pour réduire le domaine de y. En effet, de la contrainte et des domaines de x et z, on peut déduire que $y \in [a_x - b_z, b_x - a_z]$ et donc que $y \in [a_y, b_y] \cap [a_x - b_z, b_x - a_z]$. On retiendra que le calcul sur les intervalles n'implique que des opérations sur les bornes, qu'il est correct, mais pas toujours exact : l'intervalle obtenu pour v contient toutes les valeurs de v compatibles avec la contrainte c et les domaines des variables de $V' - \{v\}$, mais éventuellement aussi des valeurs incompatibles.

Le tableau de la figure 6.10 montre les réductions de domaine successives réalisées par un algorithme de filtrage par arc-cohérence sur l'instance utilisée comme exemple. Chaque ligne correspond à une tentative de réduction du domaine d'une variable par une contrainte. Par exemple, la troisième ligne du tableau montre la réduction du domaine de x du fait de la contrainte $x = y \cdot z$ et des domaines de y et z. La quatrième montre la réduction du domaine de y du fait de la même contrainte et des domaines de x et z. Il n'y

a en fait dans ce cas aucune réduction. On notera la différence entre l'état initial et l'état final des domaines.

Variables	x	y	z	t
Domaines initiaux	$[-\infty, 10]$	$[4, 5]$	$[1, 3]$	$[0, 1.5]$
$x = y \cdot z$	$[4, 10]$	-	-	-
$y = x/z$	-	$[4, 5]$	-	-
$z = x/y$	-	-	$[1, 2.5]$	-
$z \leq t$	-	-	$[1, 1.5]$	-
$t \geq z$	-	-	-	$[1, 1.5]$
$x = y \cdot z$	$[4, 7.5]$	-	-	-
$y = x/z$	-	$[4, 5]$	-	-
$z = x/y$	-	-	$[1, 1.5]$	-
Domaines finaux	$[4, 7.5]$	$[4, 5]$	$[1, 1.5]$	$[1, 1.5]$

FIGURE 6.10 – Exemple de filtrage par arc-cohérence d'une instance de CSP continu.

6.2 Méthodes de raisonnement dans le cadre SAT

Les méthodes de raisonnement développées dans le cadre *SAT* visent essentiellement à déduire de nouvelles clauses à partir des clauses existantes, avec l'espoir que les nouvelles soient *sympathiques*, c'est-à-dire d'arité la plus faible possible [1]. Une clause d'arité 1 permet en effet de conclure sur la valeur de la variable booléenne qu'elle implique [2] et une clause d'arité 0 permet de conclure à l'incohérence de l'instance considérée.

Le mécanisme de déduction utilisé le plus simple est dit de *résolution*. Il permet de produire une nouvelle clause à partir de deux clauses qui partagent la même variable, mais positivement dans l'une et négativement dans l'autre. Soient en effet deux clauses $x \vee C$ et $\neg x \vee C'$. Comme x ne peut pas être à la fois vrai et faux, il est nécessaire que $C \vee C'$. Ainsi, à partir des clauses $x \vee y \vee \neg z$ et $\neg x \vee \neg z \vee t$, il est possible de produire la clause $y \vee \neg z \vee \neg z \vee t$, qui se réécrit $y \vee \neg z \vee t$.

Si l'une des deux clauses est unaire, l'application du mécanisme de résolution est particulièrement intéressante puisqu'elle réduit d'une unité

1. L'arité d'une clause est, de la même façon que pour une contrainte quelconque, égale au nombre de variables booléennes qu'elle implique.

2. x implique que $x = t$ et $\neg x$ implique que $x = f$.

l'arité de l'autre clause. Par exemple, à partir des clauses $\neg y$ et $y \vee \neg z \vee t$ (d'arité 3), on produit la clause $\neg z \vee t$ (d'arité 2).

La figure 6.11 montre les clauses produites par application systématique du mécanisme de résolution sur l'instance insatisfiable de la section 4.1. Les 5 premières lignes correspondent aux 5 clauses initiales de l'instance, numérotées de 1 à 5. Chaque ligne suivante correspond à une application du mécanisme de résolution. La première colonne de chaque ligne indique les deux clauses utilisées, la seconde colonne indique la clause résultante et la troisième le numéro qui lui est attribué. La stratégie ici utilisée consiste à utiliser au maximum une clause unaire dès qu'elle est produite. À la dernière ligne, la clause résultante est vide, ce qui permet de conclure à l'incohérence de l'instance.

Origine	Résultat	Numéro
	$x \vee \neg y$	1
	$y \vee \neg z$	2
	$\neg x \vee z$	3
	$x \vee y$	4
	$\neg y \vee \neg z$	5
1,2	$x \vee \neg z$	6
1,3	$\neg y \vee z$	7
1,4	x	8
3,8	z	9
2,9	y	10
5,9	$\neg y$	11
10,11	$\emptyset$	

FIGURE 6.11 – Exemple d'application systématique du mécanisme de résolution sur une instance SAT.

Il est établi que l'application systématique du mécanisme de résolution permet de décider de la satisfiabilité d'une instance SAT (terminaison, correction et complétude de l'algorithme). Malheureusement, le nombre et la taille des clauses générées peut rapidement devenir prohibitif. C'est pourquoi le mécanisme de résolution est souvent utilisé uniquement sous sa forme restreinte de propagation des clauses unaires. Dans ce cas, il n'est évidemment plus complet : l'insatisfiabilité d'une instance sera parfois détectée, parfois non.

6.3 Méthodes de raisonnement dans le cadre PLNE

Les méthodes de raisonnement les plus utilisées dans le cadre *PLNE* utilisent le lien qui existe entre une instance *PLNE* et ce qu'on appelle sa *relaxation linéaire*, c'est à dire l'instance *PL* qui résulte de la relaxation de la contrainte dite d'*intégrité* (contrainte spécifiant que les variables doivent prendre des valeurs entières).

Considérons une instance P impliquant n variables et m contraintes et sa relaxation linéaire P'. Chaque contrainte de P définit un sous-ensemble de R^n délimité par un hyper-plan. L'ensemble des contraintes définit un polyèdre convexe PC' délimité par les hyper-plans associés aux m contraintes. La figure 6.12 montre par exemple le polyèdre PC'_0 associé à une instance P_0 à deux variables x et y entières positives et 4 contraintes : $-3x + 4y \leq 7$, $2y \leq 5$, $6x + 4y \leq 25$ et $2x - y \leq 6$.

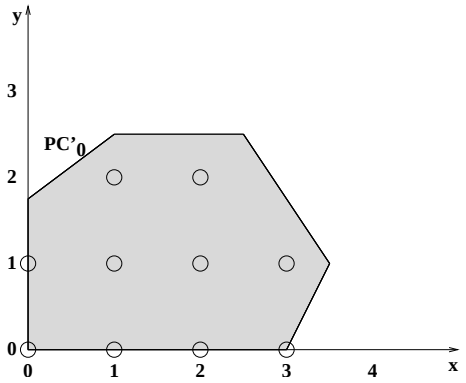

FIGURE 6.12 – Polyèdre convexe associé à la relaxation linéaire d'une instance *PLNE*.

L'ensemble des solutions de P' est l'ensemble des points de PC' et l'ensemble des solutions de P est l'ensemble des points de PC' de coordonnées entières. On sait qu'au moins un des sommets du polyèdre PC' correspond à une solution optimale de P'. Elle peut facilement être produite par un algorithme de programmation linéaire tel que le *Simplexe*. Soit *opt* l'optimum de P et *opt*$'$ l'optimum de P' (produit par l'algorithme de programmation linéaire).

Plaçons nous dans un contexte de maximisation. La première déduction qui peut être faite est que $opt \leq opt'$, tout simplement parce que l'ensemble des solutions entières est un sous-ensemble de l'ensemble des solu-

tions réelles. Sur l'instance précédente, si l'on cherche à maximiser le critère $x + 2y$, on obtient que $opt' = 7.5$ et donc que $opt \leq 7.5$.

La seconde déduction qui peut être faite est que, si une solution entière a une valeur du critère val, alors $val \leq opt$, tout simplement parce que toute solution d'un problème de maximisation a par définition une valeur inférieure ou égale à l'optimum.

La troisième déduction est que, si la solution optimale de P' (produite par l'algorithme de programmation linéaire) est entière (a toutes ses coordonnées entières), elle est aussi solution optimale de P. En effet, soit val sa valeur. On a : $val \leq opt \leq opt'$. Comme $val = opt'$, $val = opt$. Mais ce n'est pas le cas dans notre exemple.

Mais il est possible d'aller plus loin. On peut en effet montrer qu'il existe un polyèdre convexe minimal PC, dont tous les sommets ont des coordonnées entières, incluant l'ensemble des points de PC' de coordonnée entière. La figure 6.13 montre les polyèdres PC_0 et PC'_0 associés à l'instance utilisée comme exemple.

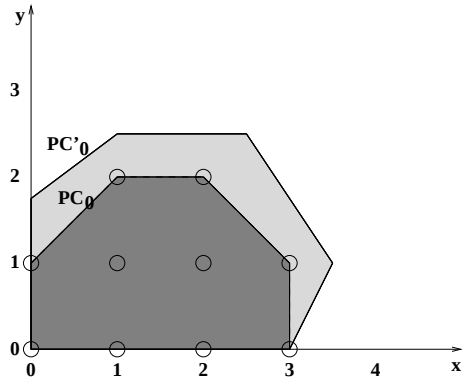

FIGURE 6.13 – Polyèdres convexes associés à une instance *PLNE* et à sa relaxation linéaire.

Si l'on connaissait les contraintes définissant le polyèdre PC, utiliser un algorithme de programmation linéaire sur ces contraintes permettrait de trouver la solution optimale dans le domaine des réels, qui serait dans ce cas obligatoirement entière (puisque tous les sommets de PC ont toutes leurs coordonnées entières) et serait donc la solution optimale de P. Malheureusement, ces contraintes ne sont pas connues a priori. Elles résultent de la conjonction des contraintes linéaires de P et de la contrainte d'intégrité et leur production peut se révéler extrêmement coûteuse. Cependant toute

contrainte permettant de rapprocher PC' de PC est a priori intéressante, car elle permet éventuellement d'aboutir à la situation où la solution optimale de P' est entière et donc solution optimale de P. C'est ce que réalisent les coupes dites de *Gomory* qui permettent de générer une nouvelle contrainte à partir d'une contrainte linéaire à coefficients non entiers et de la contrainte d'intégrité.

Dans notre exemple, on a vu que $opt \leq 7.5$. On peut donc ajouter à l'instance P_0 la contrainte $x + 2y \leq 7.5$. Mais comme x et y sont entiers on peut en déduire que $x + 2y \leq 7$, ce qui constitue une coupe dans PC'_0 qui a l'avantage de rapprocher PC'_0 de PC_0 et d'éliminer la solution optimale précédente de P'_0 (voir la figure 6.14).

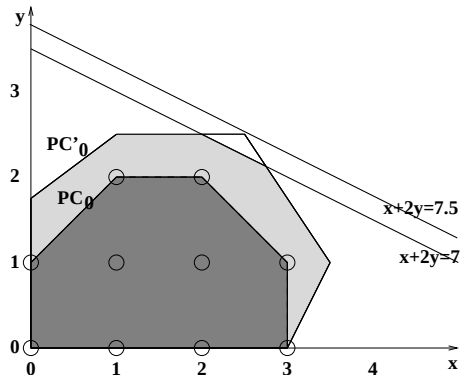

FIGURE 6.14 – Application d'une coupe dans la relaxation linéaire d'une instance *PLNE*.

On peut aussi utiliser un algorithme de programmation linéaire pour calculer les valeurs minimales et maximales de chaque variable dans le domaine des réels. Dans notre exemple, on obtient que $0 \leq x \leq 3.5$ et que $0 \leq y \leq 2.5$. Comme x et y sont entiers, on en déduit que $0 \leq x \leq 3$ et que $0 \leq y \leq 2$, ce qui constitue des coupes supplémentaires dans PC'_0 (voir la figure 6.15).

On peut ensuite utiliser un algorithme de programmation linéaire pour calculer l'optimum dans le polyèdre ainsi réduit, toujours dans le domaine des réels. On obtient $opt' = 41/6$. D'où la contrainte $x + 2y \leq 41/6$, qui donne $x + 2y \leq 6$ du fait que x et y sont entiers, ce qui constitue une nouvelle coupe dans PC'_0 (voir la figure 6.16).

On peut alors observer que la solution optimale dans le polyèdre ainsi réduit et dans le domaine des réels est entière ($x = 2$ et $y = 2$). Comme

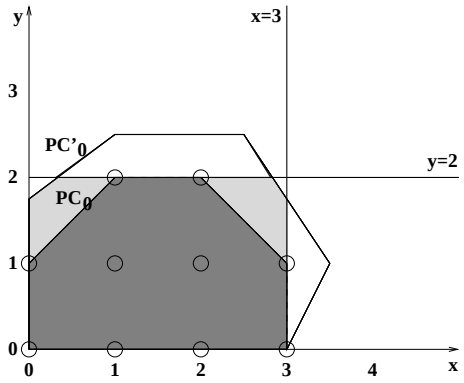

FIGURE 6.15 – Application de deux autres coupes dans la relaxation linéaire d'une instance *PLNE*.

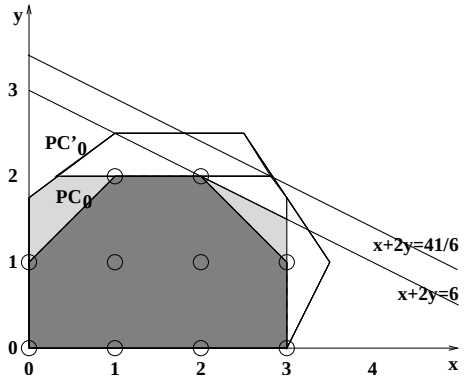

FIGURE 6.16 – Application d'une autre coupe dans la relaxation linéaire d'une instance *PLNE*.

démontré précédemment, c'est forcément la solution optimale dans le domaine des entiers et la solution optimale du problème initial, dont l'optimum est donc 6.

Ceci est évidemment un cas heureux où la mise en œuvre de raisonnements simples a permis de produire la solution optimale du problème initial. Toutes les situations ne sont pas aussi heureuses, mais des raisonnements de ce type peuvent grandement aider à la résolution de problèmes complexes. On remarquera cependant qu'il n'est pas nécessaire d'avoir entièrement explicité le polyèdre PC pour produire une solution optimale. Il suffit en fait que les polyèdres PC et PC' aient le même optimum, ce qui est le cas dans la figure 6.16.

Chapitre 7

Méthodes complètes de recherche de solutions

Dans ce chapitre, nous présentons des méthodes de recherche de solutions qui sont dites complètes ou exactes. Le terme *complet* est plutôt utilisé pour des problèmes de *satisfaction*. Une méthode est complète si elle produit une solution pour toute instance satisfiable (cohérente) et prouve l'absence de solution pour toute instance insatisfiable (incohérente). Le terme *exact* est lui plutôt utilisé pour des problèmes d'*optimisation*. Une méthode est exacte si elle produit pour toute instance une solution optimale, dont l'optimalité est établie.

Les méthodes que nous présentons partagent un même schéma de *recherche arborescente*. Après avoir présenté ce schéma, nous montrons comment il s'applique aux cadres *CSP*, *SAT* et *PLNE*.

7.1 Recherche arborescente

L'idée de base de la recherche arborescente est qui si une instance est trop complexe pour être abordée dans sa globalité, il peut être intéressant de la décomposer en deux (ou plus généralement n) sous-instances disjointes a priori plus abordables. Comme il est possible que les sous-instances générées ne soient en fait pas plus abordables, la décomposition peut continuer de façon récursive jusqu'à aboutir à des sous-instances abordables (voir la figure 7.1).

Considérons une instance P décomposée en deux sous-instances disjointes P_1 et P_2. Si P est un problème de satisfaction, une solution est trouvée pour P si et seulement une solution est trouvée pour P_1 ou pour P_2.

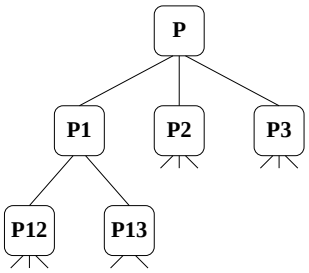

FIGURE 7.1 – Recherche arborescente.

En fait, si une solution est trouvée pour P_1, aucune recherche n'est effectuée sur P_2. Si P est un problème d'optimisation, l'optimum de P est le minimum des optimums de P_1 et P_2 en cas de minimisation (le maximum en cas de maximisation).

Les méthodes que nous allons présenter dans les cadres *CSP*, *SAT* et *PLNE* ont toutes pour base ce principe de décomposition récursive, mais différent sur la façon de *décomposer* une instance en sous-instances, sur la façon de *parcourir* l'arborescence résultant de la décomposition récursive et sur la façon de *couper* cette arborescence.

Elles ont par contre toutes en commun de présenter une complexité temporelle dans le pire cas qui est une fonction exponentielle du nombre de variables impliquées, ce qui peut empêcher leur utilisation même sur des instances de taille moyenne (quelques centaines, parfois quelques dizaines, de variables).

7.1.1 Recherche arborescente dans le cadre CSP

Dans le cadre *CSP*, les variables ont des domaines de valeur finis et une façon naturelle de décomposer une instance en sous-instances disjointes consiste à considérer une variable v et tous les sous-instances correspondant à l'affectation à v d'une valeur de son domaine [1] (voir la figure 7.2). Cela peut être vu comme une forme de raisonnement hypothétique : que se passe-t-il si je fais l'hypothèse que $v = a$?

Cette décomposition peut se poursuivre tant qu'il existe des variables non affectées. Quand toutes les variables sont affectées, le problème à résoudre est

1. Dans le cas de *CSP* continus, une méthode de décomposition plus adaptée consiste à considérer une variable v, à diviser l'intervalle correspondant à son domaine en deux sous-intervalles disjoints et à considérer les deux sous-instances correspondant à la restriction de la valeur de v à un des deux sous-intervalles.

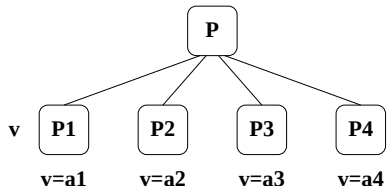

FIGURE 7.2 – Décomposition d'une instance en sous-instances dans le cadre *CSP*.

un simple test de satisfaction de contraintes : est-ce que toutes les contraintes sont satisfaites ? La profondeur de l'arborescence est donc limitée par le nombre n de variables de l'instance considérée. Sa taille peut par contre devenir rapidement astronomique. Soit D_i le domaine associé à une variable v_i et $d = max_{i=1}^{n}|D_i|$. La taille maximale de l'arborescence générée est $\sum_{i=0}^{n}(\prod_{j=1}^{i}|D_i|) \leq \sum_{i=0}^{n} d^i \in O(d^n)$.

La méthode de parcours de cette arborescence la plus utilisée est dite en *profondeur d'abord*. Elle consiste à choisir une variable, à lui affecter une valeur et à choisir immédiatement une autre variable pour lui affecter aussi une valeur, et ainsi de suite jusqu'à ce qu'une incohérence soit détectée ou que toutes les variables soient affectées sans détection d'incohérence. Dans le premier cas (détection d'incohérence), une autre valeur est choisie pour la dernière variable affectée. C'est ce qu'on appelle un *backtrack* ou encore une *coupe* de l'arborescence. Si aucune autre valeur n'existe pour la dernière variable affectée, une autre valeur est choisie pour l'avant-dernière variable affectée (backtrack au niveau supérieur). Dans le second cas (affectation de toutes les variables sans détection d'incohérence), une solution a été trouvée et la recherche s'arrête. L'incohérence globale de l'instance est établie si toute l'arborescence est parcourue sans qu'aucune solution soit trouvée, ce qui correspond à un backtrack au niveau 0. Le terme *backtrack* est utilisé pour décrire le comportement de l'algorithme en cas de détection d'incohérence, mais aussi souvent pour décrire l'algorithme dans son ensemble.

Le mécanisme de détection d'incohérence le plus simple à mettre en œuvre dans le cadre *CSP* consiste à vérifier, après chaque affectation d'une variable v, les contraintes dont toutes les variables sont affectées suite à l'affectation de v. Ce mécanisme porte le nom de *backward-checking*. Si l'une de ces contraintes est insatisfaite, il est certain que la sous-instance associée à ce nœud de l'arborescence est incohérente et qu'un backtrack est nécessaire.

La figure 7.3 montre l'arborescence développée par un algorithme de type

backtrack, utilisant un backward-checking comme mécanisme de détection d'incohérence, sur le problème dit des n reines avec $n = 3$ (voir la section 5.3.1). Une variable est associée à chaque ligne. Elle indique la colonne sur laquelle se trouve la reine de cette ligne. À noter que le problème des n reines est incohérent avec $n = 3$. Les variables sont affectées dans l'ordre des lignes de haut en bas. Pour chaque variable, les valeurs sont choisies dans l'ordre des colonnes de gauche à droite. Les affectations sont indiquées par une croix dans la case correspondante. Les détections d'incohérence sont indiquées par une croix en gras en dessous du nœud associé. Les nombres à droite de chaque nœud représentent l'ordre de génération et d'exploration de ces nœuds. La troisième branche de l'arborescence n'est pas représentée car elle est symétrique de la première.

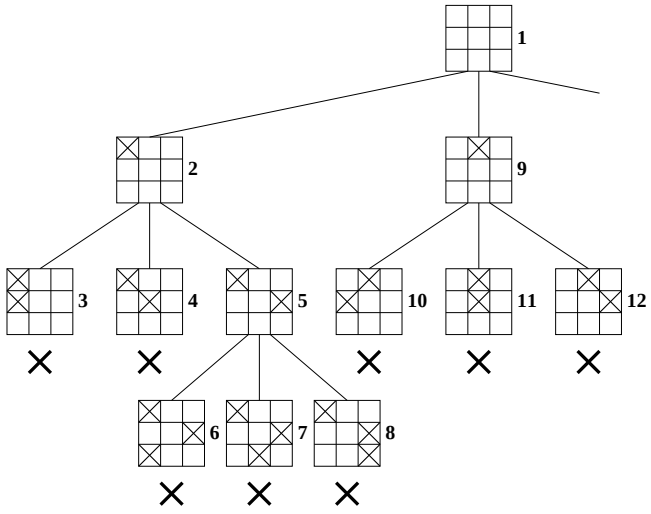

FIGURE 7.3 – Backtrack + Backward-checking sur le problème des 3 reines.

Un mécanisme de détection d'incohérence un peu plus sophistiqué consiste, chaque fois qu'une nouvelle variable v est affectée, à éliminer des domaines des variables non encore affectées les valeurs qui sont incohérentes avec l'affectation de v. Ce mécanisme porte le nom de *forward-checking*. Si l'un des domaines des variables non encore affectées devient vide, il est certain que la sous-instance associée à ce nœud de l'arborescence est incohérente et qu'un backtrack est nécessaire.

La figure 7.4 montre l'arborescence développée par un algorithme de type backtrack, utilisant un forward-checking comme mécanisme de détection

d'incohérence, sur le même problème des 3 reines. Les valeurs filtrées par forward-checking sont indiquées en gris.

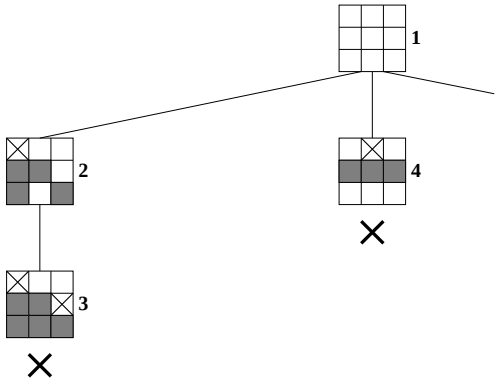

FIGURE 7.4 – Backtrack + Forward-checking sur le problème des 3 reines.

Un mécanisme encore plus sophistiqué consiste, chaque fois qu'une nouvelle variable v est affectée, à réaliser un forward-checking sur les variables non encore affectées, puis un filtrage par *arc-cohérence* (voir la section 6.1.2) du problème restreint aux variables non encore affectées. Si l'un des domaines de ces variables devient vide, il est certain que la sous-instance associée à ce nœud de l'arborescence est incohérente et qu'un backtrack est nécessaire.

La figure 7.5 montre l'arborescence développée par un algorithme de type backtrack, utilisant un forward-checking et un filtrage par arc-cohérence comme mécanismes de détection d'incohérence, sur le même problème des 3 reines. Les valeurs filtrées par forward-checking sont indiquées en gris et celles filtrées par arc-cohérence en gris foncé. Cet arborescence est, dans ce cas particulier, réduite à son nœud racine, car le filtrage par arc-cohérence détecte l'incohérence de l'instance initiale. La figure 7.6 montre par contre l'arborescence développée par le même algorithme sur le problème des 4 reines. À noter que le problème des n reines est incohérent avec $n = 3$, mais cohérent avec $n = 4$. Les détections d'incohérence (respectivement cohérence) sont indiquées par une croix (respectivement un cercle) en gras en dessous du nœud associé. Dans ce cas, le filtrage par arc-cohérence ne simplifie pas le problème initial qui est déjà arc-cohérent. Par contre, dès le placement de la première reine sur la première colonne, l'incohérence est détectée (domaine associé à la troisième reine vide) et, dès le placement de la première reine sur la seconde colonne, la cohérence est établie (domaines des variables non affectées tous réduits à un singleton). Les deux autres branches

de l'arborescence ne sont pas représentées car elles sont symétriques des deux premières.

FIGURE 7.5 – Backtrack + Forward-checking + Arc-cohérence sur le problème des 3 reines.

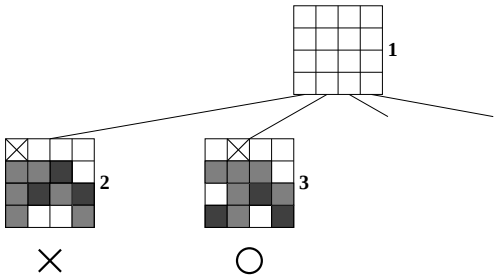

FIGURE 7.6 – Backtrack + Forward-checking + Arc-cohérence sur le problème des 4 reines.

Plus généralement, n'importe quel mécanisme de filtrage par cohérence locale, tel que ceux présentés dans la section 6.1, peut être utilisé pour détecter l'incohérence à chaque nœud de l'arborescence. Plus ce mécanisme est puissant, plus l'incohérence est détectée haut dans l'arborescence, donc plus l'arborescence effectivement développée est réduite (voir les figures 7.3, 7.4 et 7.5), mais plus le travail réalisé à chaque nœud est important. Un compromis doit donc être trouvé. La pratique tend à indiquer que le filtrage par arc-cohérence, voire par des propriétés de cohérence locale intermédiaires entre arc-cohérence et chemin-cohérence, sont des compromis raisonnables.

Deux autre paramètres importants d'une recherche arborescente dans le cadre *CSP* sont l'ordre dans lequel les variables sont affectées et l'ordre dans lequel les valeurs des domaines sont explorées. En ce qui concerne l'ordre sur les variables (ordre vertical), deux principes sont utilisés : (1) affecter les variables de plus petit domaine en premier, car cela diminue la taille maximale de l'arborescence générée (voir la formule utilisée plus haut pour calculer cette taille), (2) affecter les variables les plus contraintes en premier, car leur affectation permet a priori de détecter les incohérences et

donc de couper l'arborescence au plus tôt (on peut par exemple considérer qu'une variable est d'autant plus contrainte que son degré dans le graphe de contraintes est plus élevé : plus de contraintes pesant sur elle). En ce qui concerne l'ordre sur les valeurs (ordre horizontal), le principe est de choisir en premier les valeurs qui ont a priori le plus de chances de conduire à une solution. À noter que l'ordre sur les valeurs n'a aucune importance en cas d'incohérence. On utilise souvent le terme d'*heuristique* pour désigner le critère qui permet de déterminer un ordre sur les variables ou les valeurs.

À titre d'illustration, les figures 7.8, 7.9 et 7.10 montrent l'arborescence développée sur l'instance incohérente de la figure 7.7 par un algorithme de type backtrack, utilisant un simple backward-checking comme mécanisme de détection d'incohérence, associé aux heuristiques suivantes d'ordonnancement des variables : (1) aucune heuristique, soit l'ordre naturel des variables pour la figure 7.8, (2) l'heuristique du plus petit domaine en premier pour la figure 7.9 et (3) l'heuristique du plus grand degré en premier pour la figure 7.10.

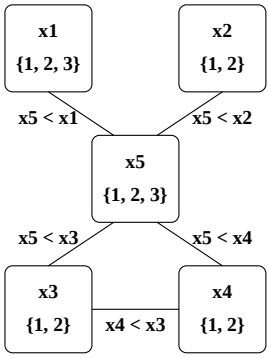

FIGURE 7.7 – Instance CSP exemple (incohérente).

Les différences en termes de taille de l'arborescence générée sont spectaculaires, même pour une instance de si petite taille. Pour cette instance, l'heuristique consistant à ordonner les variables par degré décroissant est la plus performante. Il n'existe cependant malheureusement pas d'heuristique systématiquement supérieure aux autres et il est très difficile de déterminer a priori, au vu des caractéristiques globales d'une instance, quelle heuristique sera la meilleure pour cette instance. En général, seule l'expérimentation apporte une réponse définitive. Des heuristiques combinant la taille du domaine courant et le degré sont cependant les heuristiques par défaut les plus

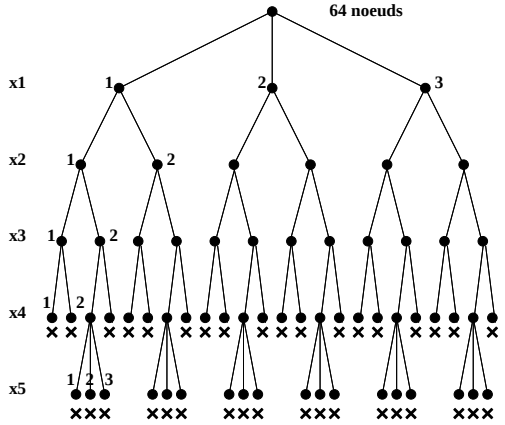

FIGURE 7.8 – Backtrack + backward-checking, sans heuristique sur l'instance CSP de la figure 7.7.

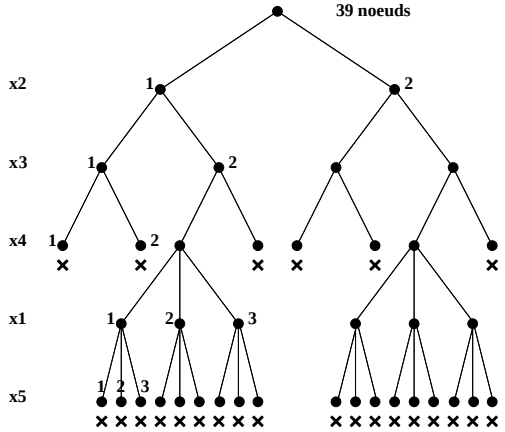

FIGURE 7.9 – Backtrack + backward-checking, avec l'heuristique du plus petit domaine en premier sur l'instance CSP de la figure 7.7.

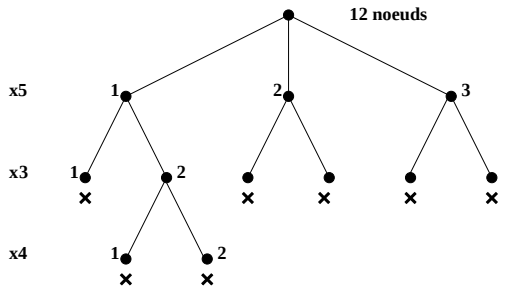

FIGURE 7.10 – Backtrack + backward-checking, avec l'heuristique du plus grand degré en premier sur l'instance CSP de la figure 7.7.

couramment utilisées pour l'ordre sur les variables (par exemple, choisir en premier la variable dont le ratio taille du domaine courant sur degré est le plus faible, en prenant en compte pour le degré uniquement les contraintes liant la variable aux autres variables non affectées). Pour l'ordre sur les valeurs, il n'existe par contre pas d'heuristique générique vraiment efficace et des heuristiques liées au type de problème traité sont généralement utilisées.

7.1.2 Recherche arborescente dans le cadre SAT

Les méthodes de recherche arborescente développées dans le cadre *SAT* sont très proches de celles développées dans le cadre *CSP*. Les seules différences introduites par le cadre *SAT* sont en effet que les variables sont booléennes et que les contraintes sont des clauses.

La méthode la plus utilisée est connue sous le terme *DPLL*, du nom de ses auteurs : Davis, Putnam, Logemann et Loveland.

Comme dans le cadre *CSP*, une instance est décomposée en deux sous-instances disjointes en considérant une variable v et les deux sous-instances correspondant, d'un côté à l'affectation $v = t$, de l'autre à l'affectation $v = f$. La profondeur de l'arborescence est limitée par le nombre n de variables de l'instance considérée et sa taille appartient à $O(2^n)$. Comme dans le cadre *CSP*, la méthode de parcours de l'arborescence est en *profondeur d'abord*. Les mécanismes de détection d'incohérence sont par contre différents. Il s'agit des mécanismes de résolution présentés dans la section 6.2, limités à la propagation des clauses unaires : dès qu'une clause unaire est générée, en particulier par la décomposition, toute clause qui la contient comme sous-clause peut être éliminée (par exemple, si $\neg x$ est générée, la clause $\neg x \vee y \vee \neg z$ peut être ignorée) et toute clause qui contient sa négation peut être simplifiée

(toujours, si $\neg x$ est générée, la clause $x \vee \neg y \vee t$ peut être simplifiée en $\neg y \vee t$ par application du mécanisme de résolution). Comme la prise en compte d'une clause unaire peut rendre unaire des clauses binaires (par exemple, $\neg x$ et $x \vee y$ génèrent y), on parle de propagation de clauses unaires, comme on parle de propagation de contraintes dans le cadre *CSP*.

Sur l'instance insatisfiable de la section 4.1, on peut vérifier que ce mécanisme de propagation des clauses unaires permet de détecter l'incohérence dès que x est fixé à t d'un côté et à f de l'autre. D'où une arborescence limitée dans ce cas à son premier niveau (voir la figure 7.11).

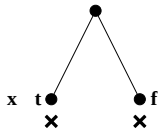

FIGURE 7.11 – Arborescence générée par l'algorithme DPLL sur l'instance SAT insatisfiable de la section 4.1.

7.1.3 Recherche arborescente dans le cadre PLNE

Les méthodes de recherche arborescente développées dans le cadre *PLNE* sont par contre sensiblement différentes de celles développées dans les cadres *CSP* et *SAT*. Cela tient essentiellement au fait que les domaines ne sont pas a priori finis, que les contraintes et le critère sont linéaires (possibilité de raisonnement sur la relaxation linéaire) et qu'il s'agit d'un problème d'optimisation.

Considérons, comme dans la section 6.3, une instance P et sa relaxation linéaire P'. P' peut être résolu par un algorithme de programmation linéaire. Si P' est incohérent (polyèdre PC' vide), P est forcément lui aussi incohérent. La résolution de P est terminée. Si, par contre, P' est cohérent, sa résolution par un algorithme de programmation linéaire fournit une solution optimale s' et un optimum associé opt'. Supposons un contexte de maximisation. Nous avons vu en section 6.3 que forcément $opt \leq opt'$. Supposons par ailleurs que seules nous intéressent des solutions entières de valeur strictement supérieure à val. Si $opt' \leq val$, forcément $opt \leq val$: il n'existe aucune solution entière de valeur strictement supérieure à val ; la résolution de P peut être abandonnée car sans intérêt. Si, par contre, $opt' > val$, la résolution de P est potentiellement intéressante. Supposons maintenant que toutes les valeurs de s' soient entières. Nous avons vu en section 6.3 que,

dans ce cas, s' est forcément solution optimale de P. La résolution de P est terminée. Si, par contre, au moins une des valeurs de s' est non entière, il est nécessaire de décomposer P pour aller plus loin.

La façon de décomposer P la plus utilisée consiste à choisir une variable v correspondant à une valeur f non entière de s. Soit a (respectivement b) le plus grand (respectivement le plus petit) entier inférieur (respectivement supérieur) à f. On considère alors les deux sous-instances P_1 et P_2 de P résultant de l'ajout à P d'un côté de la contrainte $v \leq a$ et de l'autre de la contrainte $v \geq b$. Notons que P_1 et P_2 sont disjointes et que les solutions de P sont l'union des solutions de P_1 et de P_2, puisque seules des solutions non entières ont été éliminées. Notons aussi que s' est exclue des relaxations linéaires associées à P_1 et P_2.

La figure 7.12 est une représentation graphique de l'instance de la section 4.2. La solution optimale de sa relaxation linéaire correspond à $x = 10/7$ et $y = 15/7$ et l'optimum associé vaut $145/14$. Dans la solution optimale, les deux variables x et y sont non entières. Supposons que nous choisissions x comme base de la décomposition. On génère alors deux sous-instances, l'une résultant de l'ajout de la contrainte $x \leq 1$, l'autre de l'ajout de la contrainte $x \geq 2$ (voir la figure 7.13).

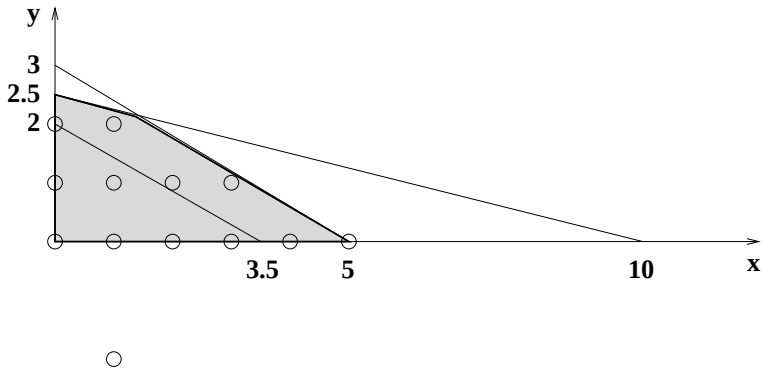

FIGURE 7.12 – Représentation graphique de l'instance PLNE de la section 4.2.

La méthode la plus utilisée pour parcourir l'arborescence résultant des décompositions successives est de type *meilleur d'abord*. Cet algorithme maintient la meilleure solution entière s trouvée jusqu'alors et sa valeur val, initialisée à $-\infty$ en cas de maximisation. Il maintient aussi une liste ordonnée LN de nœuds candidats à la décomposition. Le critère utilisé pour

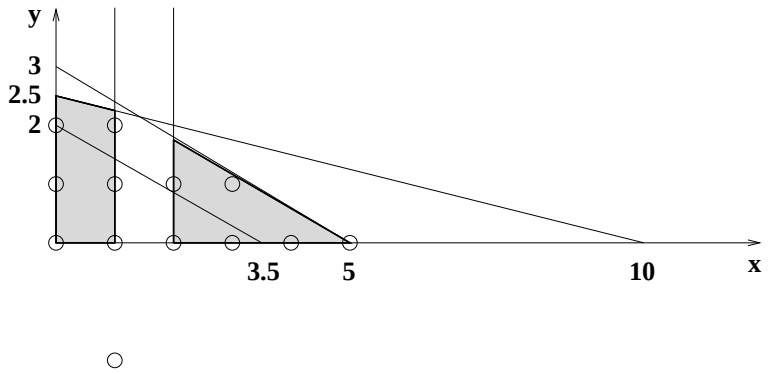

FIGURE 7.13 – Décomposition en deux sous-instances de l'instance PLNE de la section 4.2.

ordonner cette liste est en général la valeur de l'optimum de la relaxation linéaire associée. L'idée est que plus cet optimum est élevé, plus le nœud est un candidat potentiellement intéressant et donc à explorer en priorité. Cette liste est initialisée avec le nœud racine, correspondant à l'instance à résoudre. À chaque étape, l'algorithme sélectionne le premier nœud n de LN, le retire de LN et génère par décomposition deux nœuds n_1 et n_2 qui sont ajoutés à LN, sauf si une des conditions suivantes est satisfaite : (1) la relaxation linéaire associée est incohérente, (2) son optimum est inférieur ou égal à val, (3) sa solution optimale est entière. Dans ce dernier cas, une meilleure solution entière à été trouvée et s, val et LN sont mis à jour en conséquence. De LN, sont retirés tous les nœuds tels que l'optimum de la relaxation linéaire associée est inférieur ou égal à la nouvelle valeur de val. L'algorithme termine quand $LN = \emptyset$. La solution optimale et l'optimum sont mémorisés via s et val.

Cet algorithme est une application au cadre *PLNE* d'un algorithme plus général dénommé *Branch-and-Bound*, *Branch* en référence à la décomposition et *Bound* en référence au calcul d'un minorant de l'optimum en cas de minimisation ou d'un majorant en cas de maximisation, calcul effectué dans le cadre *PLNE* via la résolution de la relaxation linéaire.

La figure 7.14 montre l'arborescence générée par cet algorithme sur l'instance de la section 4.2. À chaque nœud, apparaît la contrainte ajoutée lors de la création de ce nœud : aucune pour le nœud racine. Attention : les contraintes s'accumulent le long des branches. Par exemple, l'instance associée au nœud 4 est le résultat de l'ajout des contraintes $x \geq 2$ et $y \leq 1$

à l'instance initiale. À chaque nœud, apparaissent aussi, sauf en cas d'incohérence, les coordonnées de la solution optimale de la relaxation linéaire et la valeur de l'optimum associé. La solution optimale de l'instance initiale apparaît lors du traitement du nœud 8.

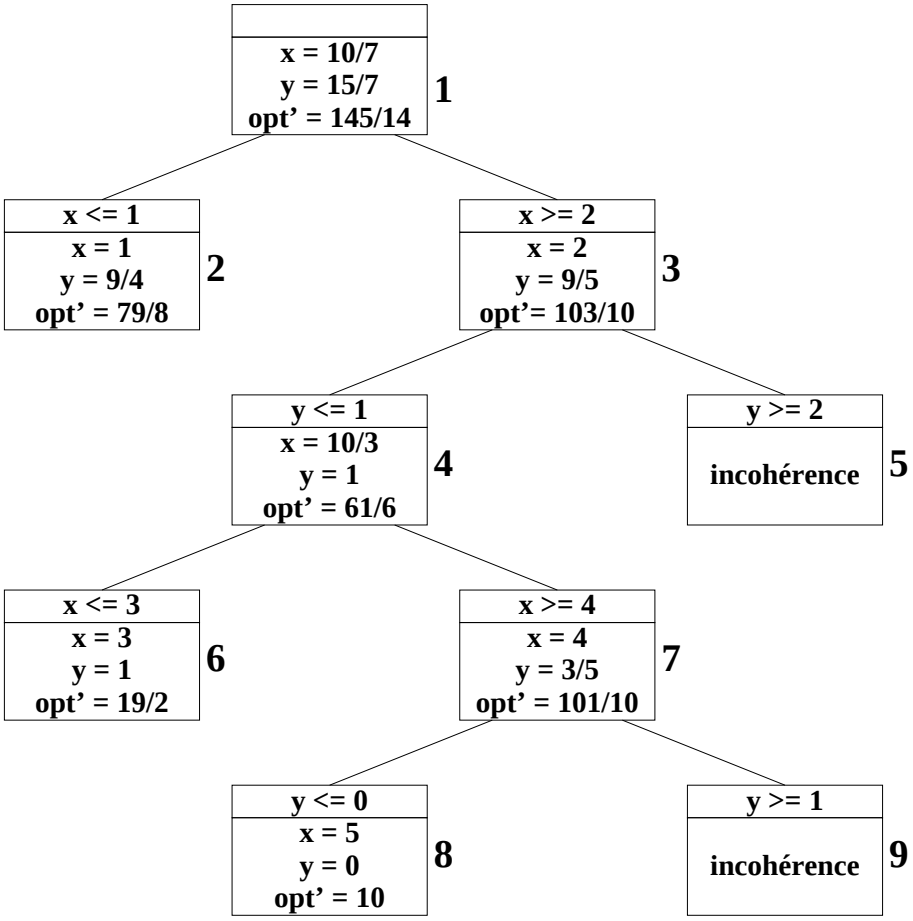

FIGURE 7.14 – Arborescence générée par l'algorithme de type *Branch-and-Bound* sur l'instance PLNE de la section 4.2.

Le tableau 7.15 montre l'état courant de s, val et LN suite à la création de chaque nœud.

	s	val	LN
1	$\emptyset$	$-\infty$	$\{1\}$
2	$\emptyset$	$-\infty$	$\{2\}$
3	$\emptyset$	$-\infty$	$\{3, 2\}$
4	$\emptyset$	$-\infty$	$\{4, 2\}$
5	$\emptyset$	$-\infty$	$\{4, 2\}$
6	$\{x = 3, y = 1\}$	$19/2$	$\{2\}$
7	$\{x = 3, y = 1\}$	$19/2$	$\{7, 2\}$
8	$\{x = 5, y = 0\}$	10	$\emptyset$
9	$\{x = 5, y = 0\}$	10	$\emptyset$

FIGURE 7.15 – Suivi de l'algorithme de type *Branch-and-Bound* sur l'instance PLNE de la section 4.2.

Chapitre 8

Méthodes incomplètes de recherche de solution

Dans ce chapitre, nous présentons des méthodes de recherche de solutions qui sont dites incomplètes ou inexactes. Rappelons que le terme *incomplet* est plutôt utilisé pour des problèmes de *satisfaction*. Une méthode est incomplète si elle ne garantit pas de prouver la satisfiabilité d'une instance satisfiable, ni de prouver l'insatisfiabilité d'une instance insatisfiable. En fait les méthodes que nous allons présenter prouvent éventuellement la satisfiabilité d'une instance satisfiable (en exhibant une solution), mais sont incapables de prouver l'insatisfiabilité d'une instance insatisfiable. Autrement dit, sur une instance satisfiable, ou bien elles produisent une solution, ou bien elles ne concluent pas. Sur une instance insatisfiable, elles sont incapables de conclure. Le terme *exact* est plutôt utilisé pour des problèmes d'*optimisation*. Une méthode est inexacte si elle ne garantit pas de produire une solution optimale.

Nous présentons les deux grandes classes de méthode incomplète : à base de *recherche gloutonne* et à base de *recherche locale*. Nous le faisons dans le cadre le plus général des problèmes d'optimisation sous contraintes.

8.1 Recherche gloutonne

Considérons un problème d'optimisation sous contraintes impliquant un ensemble de variables ayant chacune un domaine de valeur. Une recherche gloutonne est une systématisation de la méthode la plus naïve qu'utiliserait un être humain pour tenter de le résoudre. Elle part d'une affectation vide, choisit une variable à affecter, choisit une valeur pour cette variable, puis

choisit une nouvelle variable à affecter et une valeur pour cette variable, et ainsi de suite jusqu'à ce que toutes les variables soient affectées. On peut la voir comme la première branche d'une recherche arborescente en profondeur d'abord (voir la section 7.1) qui n'effectuerait aucun backtrack.

Il est évident que son efficacité dépend fortement de l'ordre suivant lequel les variables sont affectées, ainsi que des valeurs choisies pour les variables. L'ordre d'affectation des variables peut être déterminé, soit avant la recherche, soit pendant la recherche en fonction de l'affectation courante. Quant au choix de valeur pour chaque variable, il est en général déterminé pendant la recherche en fonction de l'affectation courante, c'est-à-dire des choix effectués pour les variables précédemment affectées. On parle d'heuristique de choix de variable pour désigner la procédure qui permet de déterminer l'ordre d'affectation des variables et d'heuristique de choix de valeur pour désigner celle qui permet de déterminer les valeurs affectées aux variables.

Il est évident que cette méthode n'offre, dans le cas général, aucune garantie sur la qualité du résultat, ni en termes de satisfaction des contraintes, ni en termes d'optimalité. C'est pourquoi, face à des problèmes de satisfaction, son utilisation est peu recommandée sauf si un faible degré de contrainte (peu de contraintes ou des contraintes peu contraignantes) rend quasi-certaine la production d'une solution. Face à des problèmes d'optimisation, son utilisation n'est recommandée que si l'exigence en termes d'optimalité est faible.

Pour pallier à ces faiblesses, une variante a été proposée sous le terme de *heuristic stochastic sampling*. Dans cette variante, les heuristiques de choix de variable et de valeur sont bruitées (choix aléatoires biaisés par les heuristiques) et des recherches gloutonnes successives sont effectuées, toutes partant d'une affectation vide. En satisfaction, on récupère éventuellement la première solution trouvée. En optimisation, on récupère la meilleure solution.

Il reste que l'avantage essentiel d'une recherche gloutonne est sa très faible complexité : fonction linéaire du nombre de variables et de la taille des domaines.

Il existe des situations particulières ou une recherche gloutonne garantit la production d'une solution ou d'une solution optimale. Nous en donnerons seulement deux exemples : le premier en optimisation (la recherche d'un arbre couvrant de coût minimum sur un graphe non orienté et pondéré ; voir la section 1.11 ; voir aussi [Cav06b]), le second en satisfaction (la recherche d'une solution d'un *CSP* binaire dont le graphe de contraintes est acyclique).

Recherche d'un arbre couvrant de coût minimum Considérons un graphe non orienté $G =< S, A, w >$ (S ensemble des sommets, A ensemble des arêtes, w fonction de A dans R). $G' =< S, A', w >$ est un graphe *partiel* de G si $A' \subseteq A$ (l'ensemble des sommets est conservé, mais seule une partie des arêtes l'est). Supposons G connexe. Un *arbre couvrant* G' est un graphe partiel de G dont la structure est un arbre, c'est-à-dire un graphe connexe acyclique. Informellement, un arbre couvrant correspond à une façon mimimale de maintenir la connexité entre les sommets de G tout en utilisant uniquement des arêtes de G (enlever une seule arête d'un arbre couvrant détruit la connexité). La figure 8.1 montre un exemple d'arbre couvrant un graphe. Le coût d'un graphe partiel est supposé être la somme des coûts de ses arêtes. La figure 8.2 montre un exemple d'arbre couvrant de coût minimum.

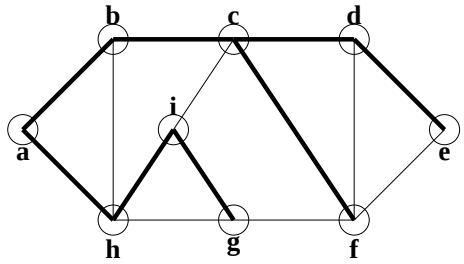

FIGURE 8.1 – Exemple d'arbre couvrant.

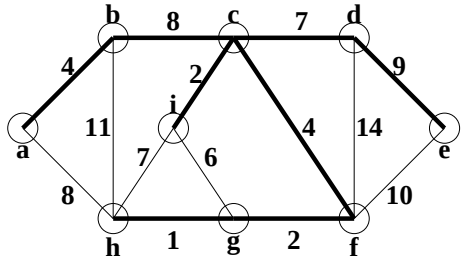

FIGURE 8.2 – Exemple d'arbre couvrant de coût minimum.

Considérons l'algorithme suivant qui utilise deux types de donnée : un arbre AC (arbre en construction) et un ensemble de sommets SC (sommets restant à considérer). Au départ, l'algorithme choisit un sommet $s \in S$ quelconque, $AC =< \{s\}, \emptyset >$ et $SC = S - \{s\}$. À chaque étape, l'algorithme

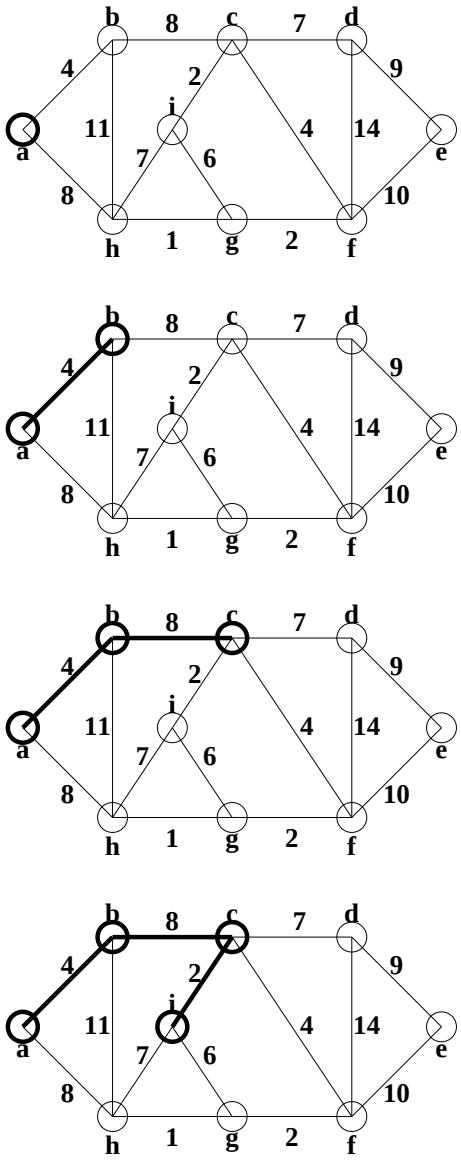

FIGURE 8.3 – Les 4 premières étapes de l'algorithme glouton sur l'exemple de la figure 8.2.

ajoute à l'arbre AC une arête $a \in A$ qui relie un sommet de AC à un sommet $s' \in SC$ et qui soit de coût minimum parmi toutes les arêtes qui relient un sommet de AC à un sommet de SC. Il ajoute s' à AC et le retire de SC. Il stoppe quand $SC = \emptyset$. La figure 8.3 montre les 4 premières étapes de cet algorithme sur l'exemple de la figure 8.2. On peut montrer que le résultat de cet algorithme glouton (choix à chaque étape de l'arête à ajouter à l'arbre AC) est un arbre couvrant de coût minimum.

CSP binaire à graphe de contraintes acyclique Considérons une instance *CSP* binaire dont le graphe de contraintes est acyclique. Si ce graphe n'est pas connexe, chaque composante connexe peut être considérée indépendamment des autres. Considérons donc une composante connexe et réalisons un filtrage par arc-cohérence de l'instance associée (voir la section 6.1.2). Si ce filtrage élimine toutes les valeurs du domaine d'une variable, l'incohérence de l'instance est établie. Sinon, l'algorithme glouton suivant garantit la production d'une solution et donc la cohérence de l'instance. Cet algorithme choisit une variable quelconque v et une valeur quelconque val dans le domaine de v (après filtrage par arc-cohérence). Ensuite, pour toutes les variables v' reliées à v par une contrainte, il choisit une valeur val' dans le domaine de v' (après filtrage par arc-cohérence) qui soit cohérente avec la valeur val de v (l'existence d'une telle valeur est garantie par la propriété d'arc-cohérence). Il procède ensuite de façon récursive à partir de toutes les variables v', en considérant pour chacune d'elles toutes les variables $v'' \neq v$ reliées à v' par une contrainte. Le fait que le graphe de contraintes soit un arbre garantit que toutes les variables soient parcourues une fois et une seule et qu'une solution soit ainsi trouvée. La figure 8.4 montre une séquence possible d'affectations à partir de la variable b. Comme le filtrage par arc-cohérence est une opération polynômiale, l'ensemble constitue un algorithme polynômial de résolution de toute instance *CSP* binaire dont le graphe de contraintes est acyclique. Le problème *CSP* restreint aux instances binaires dont le graphe de contraintes est acyclique constitue donc un exemple de sous-problème polynômial du problème *CSP*.

8.2 Recherche locale

À la différence d'une recherche gloutonne qui part d'une affectation vide, une recherche locale part d'une affectation complète (une valeur est associée à chaque variable du problème). Cette affectation n'est en général ni cohérente, ni optimale. La recherche consiste à appliquer de façon itérative

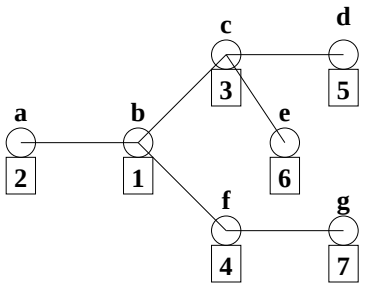

FIGURE 8.4 – Séquence possible d'affectations sur un *CSP* binaire à graphe de contraintes acyclique.

des modifications locales sur cette affectation de façon à obtenir, du moins l'espère-t-on, une affectation complète cohérente et de qualité aussi bonne que possible. Plus précisément, une recherche locale suit le schéma algorithmique général suivant : soit A_0 une affectation complète ; initialiser l'affectation complète courante A avec A_0 ; initialiser de la même façon la meilleure affectation complète trouvée A_m avec A_0 ; puis, tant que des conditions d'arrêt ne sont pas satisfaites, remplacer A_m par A si A est strictement meilleure que A_m et choisir comme nouvelle affectation courante une affectation A' dans le voisinage de A. Un tel schéma est illustré par la figure 8.5 : les A_i correspondent aux affectations complètes successives et les V_i à leurs voisinages.

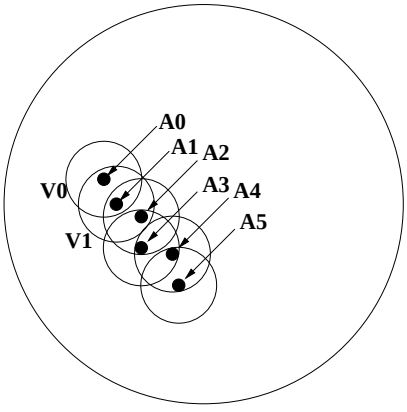

FIGURE 8.5 – Illustration d'une recherche locale dans l'espace des affectations complètes.

Les paramètres essentiels qui interviennent dans une recherche locale sont au nombre de quatre :

1. le mécanisme de *génération* d'une affectation complète initiale ;
2. la définition du *voisinage* d'une affectation complète ;
3. le mécanisme de *choix* d'une affectation complète dans le voisinage d'une autre :
4. les *conditions d'arrêt*.

Concernant la génération d'une affectation complète initiale, elle peut être effectuée de façon aléatoire ou de façon gloutonne (voir la section précédente). La première méthode permet de varier les affectations initiales et est particulièrement utile au cas où plusieurs recherches locales successives sont lancées à partir de plusieurs affectations initiales. La seconde méthode permet de lancer la recherche locale à partir d'affectations qui sont a priori de bonne qualité. Une méthode de type heuristic stochastic sampling (voir aussi la section précédente) combine les avantages des deux méthodes précédentes : variété et qualité des choix initiaux.

Concernant la définition du voisinage d'une affectation complète, elle dépend fortement de la structure du problème. Pour les problèmes *SAT* ou *CSP*, structurés en variables, le voisinage le plus classique d'une affectation complète est défini comme l'ensemble des affectations complètes résultant du changement de valeur d'une variable et d'une seule. Ce type de voisinage peut être généralisé en considérant par exemple l'ensemble des affectations complètes résultant du changement de valeur de k variables, k fixé. À noter cependant que la taille du voisinage est une fonction exponentielle de k. Pour des problèmes où les solutions sont des permutations, comme les n reines (section 5.3.1), le cavalier d'Euler (section 5.3.2) ou le voyageur de commerce (section **??**), le voisinage le plus classique d'une permutation est défini comme l'ensemble des permutations résultant de la permutation de deux éléments.

Concernant le choix d'une affectation complète dans le voisinage d'une autre, différentes options sont disponibles. Il peut être aléatoire. Pour un problème d'optimisation et plus particulièrement de minimisation, si c mesure la qualité de l'affectation courante A, il peut consister à choisir une affectation A' de qualité c' telle que $c' < c$ (strictement meilleure), $c' \leq c$ (meilleure ou égale) ou $c' \leq c + s,\ s > 0$ (meilleure, égale ou pire, mais dans la limite d'un certain seuil). Il peut aussi consister à choisir la meilleure ou une des meilleures affectations du voisinage.

Concernant les conditions d'arrêt, elles sont de deux types :

— pour un problème de satisfaction, une affectation cohérente (une solution) a été produite ; sur un problème d'optimisation, une affectation de qualité c a été produite et, ou bien c est considéré comme suffisant ou raisonnable, ou bien on a par ailleurs la garantie qu'il n'existe pas d'affectation de qualité strictement meilleure que c (en minimisation par exemple, c est un minorant de l'optimum) ;
— le temps maximum ou le nombre maximum d'itérations accordés à l'algorithme sont épuisés ; sur un problème d'optimisation, un temps maximum ou un nombre maximum d'itérations se sont écoulés sans qu'aucune solution de meilleure qualité que la meilleure trouvée jusqu'à présent n'ait été trouvée.

L'hypothèse (discutable) sur laquelle repose le paradigme de recherche locale pour des problèmes d'optimisation est qu'il existe une certaine continuité au niveau du critère : pas de différences énormes en termes de qualité dans le voisinage d'une affectation, proximité entre bonnes et très bonnes affectations. Il est évidemment possible d'exhiber des exemples où cette hypothèse n'est pas vérifiée : unique solution optimale accessible uniquement via des très mauvaises affectations.

Pour traiter des problèmes de satisfaction (*SAT*, *CSP*), il suffit de considérer comme critère à minimiser le nombre de clauses ou de contraintes insatisfaites. Une solution est trouvée quand le critère vaut 0. Pour traiter des problèmes d'optimisation sous contraintes faisant intervenir un critère c, une démarche souvent adoptée consiste à transformer, comme pour les problèmes de satisfaction, la satisfaction des contraintes en critère (c_c) et à optimiser un critère résultant de la combinaison de c et de c_c. La façon précise de réaliser cette combinaison peut cependant se révéler problématique : comment pondérer l'importance accordée à la satisfaction des contraintes et celle accordée à l'optimisation du critère c ?

Une recherche locale peut être vue comme le prolongement d'une recherche gloutonne. Mais, comme la recherche gloutonne, elle n'offre, dans le cas général, aucune garantie sur la qualité du résultat, ni en termes de satisfaction des contraintes, ni en termes d'optimalité. Correctement paramétrée, elle peut cependant être étonnamment puissante, aussi bien sur des problèmes de satisfaction que sur des problèmes d'optimisation, étonnamment plus puissante qu'une recherche arborescente, surtout par sa capacité à produire des solutions de bonne qualité en temps limité. Sa complexité est fonction du critère d'arrêt retenu. Sa faiblesse essentielle tient cependant au fait qu'elle est incapable de prouver l'insatisfiabilité d'une instance insatisfiable ou l'optimalité d'une solution.

Une instanciation particulière des quatre paramètres de base donne nais-

sance à certains algorithmes remarquables.

Le plus simple est désigné sous le terme de *iterative improvement*. Partant d'une affectation quelconque, il consiste à choisir à chaque étape une affectation strictement meilleure dans le voisinage de l'affectation courante si une telle affectation existe, et à stopper la recherche si elle n'existe pas. Si l'ensemble des affectations possibles est fini, cet algorithme termine sur une affectation localement optimale (optimale dans son voisinage), mais pas forcément globalement optimale (voir l'exemple de la figure 8.6 qui montre un graphe de voisinage entre affectations et un optimum local A_3 de valeur 3 qui ne permet pas d'atteindre l'optimum global A_4 de valeur 2). À noter qu'une affectation peut être optimale dans un voisinage et ne pas l'être dans un autre plus large. Il reste que la préoccupation essentielle en recherche locale est de concevoir des mécanismes permettant à la fois d'atteindre des minimums locaux (intensification) et de s'en échapper (diversification).

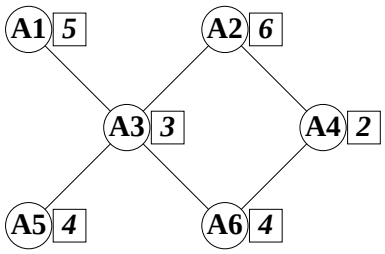

FIGURE 8.6 – Graphe de voisinage et optimum local.

Parmi ces mécanismes, le premier consiste à accepter de retenir comme nouvelle affectation courante une affectation de qualité identique, ce qui permet de parcourir d'éventuels plateaux de qualité à la recherche d'affectations meilleures. L'algorithme résultant n'offre plus de garantie de terminaison (possibilité de bouclage sur un plateau) et il est nécessaire de lui adjoindre un critère d'arrêt.

Le second consiste à aller plus loin en acceptant de retenir comme nouvelle affectation courante une affectation de moins bonne qualité à condition que la dégradation en termes de qualité ne dépasse pas un seuil fixé.

Le troisième consiste à faire un choix aléatoire dans le voisinage, ou bien à chaque étape avec une certaine probabilité, ou bien à chaque fois qu'un optimum local est atteint.

Un des algorithmes de recherche locale les plus utilisés porte le nom énigmatique de *recuit simulé* (*simulated annealing*). Avec cet algorithme,

l'affectation initiale est aléatoire. À chaque étape, une affectation est choisie de façon aléatoire dans le voisinage de l'affectation courante. Elle est acceptée si elle est de qualité meilleure ou identique à celle de l'affectation courante. Si ce n'est pas le cas, elle est acceptée avec une certaine probabilité qui est une fonction décroissante de la dégradation en termes de qualité et croissante d'un facteur dénommé *température* (par analogie avec le recuit physique dont le recuit simulé est inspiré). Plus précisément, si Δc mesure la dégradation en qualité (positive) et T la température, la probabilité d'acceptation de la nouvelle affectation est égale à $e^{-\Delta c/T}$. La température diminue au fur et à mesure du déroulement de l'algorithme. Au début, elle est fixée à un niveau élevé, ce qui revient à admettre pratiquement tous les changements, même ceux qui sont les plus pénalisants en termes de qualité. Puis, elle décroît palier par palier, jusqu'à devenir presque nulle, ce qui revient à n'admettre pratiquement que des changements qui améliorent ou maintiennent la qualité. Malgré sa simplicité, un tel algorithme, correctement paramétré (nature du voisinage, température initiale, longueur des paliers, baisse de température entre paliers successifs ...) a montré sa capacité à produire des affectations de qualité sur des problèmes très divers.

Un autre algorithme, lui aussi très utilisé, porte le nom aussi énigmatique de *recherche tabou* (*tabu search*). Avec cet algorithme, l'affectation initiale est généralement produite par une recherche gloutonne. À chaque étape, une affectation est choisie parmi les meilleures dans le voisinage de l'affectation courante (affectation courante exclue). Quand l'affectation courante est un optimum local, l'affectation choisie est de qualité moindre ou identique à celle de l'affectation courante. Une liste dite *tabou* mémorise les k derniers mouvements (k fixé) et interdit tout mouvement inverse, sauf s'il mène à une affectation strictement meilleure que la meilleure affectation produite jusqu'alors. L'ajout de cette liste tabou au mécanisme de base évite tout cycle de longueur inférieure à k dans l'espace de recherche. De même que pour le recuit simulé et malgré son encore plus grande simplicité, un tel algorithme, correctement paramétré (nature du voisinage, nature et longueur de la liste tabou ...) peut se révéler très efficace sur des problèmes très divers.

D'autres algorithmes ont été proposés sous la dénomination d'*algorithmes génétiques*. Inspirés des phénomènes de sélection observés dans la nature, ils maintiennent non pas une affectation courante, mais un ensemble d'affectations courantes (une *population*) qu'ils font évoluer via des opérations de *sélection* (sélection des meilleurs), *mutation* (modification locale aléatoire) et *croisement* (combinaison de deux affectations pour en produire une troisième).

Dans le cadre particulier des problèmes de satisfaction de contraintes (*CSP*), un algorithme de recherche locale a été proposé sous le terme de

min-conflicts. Il part d'une affectation initiale aléatoire. À chaque étape il détermine l'ensemble des variables dites *en conflit*, c'est-à-dire l'ensemble des variables qui interviennent dans au moins une contrainte insatisfaite. Il choisit aléatoirement une de ces variables (v). Pour chacune des valeurs (val) de son domaine, il détermine le nombre de *conflits*, c'est le nombre de contraintes insatisfaites si v prenait la valeur val. Comme nouvelle valeur de v, il choisit une des valeurs qui minimisent le nombre de conflits. Il itére jusqu'à trouver une solution ou qu'un nombre maximum d'itérations soit atteint. Auquel cas, il repart d'une nouvelle affectation initiale aléatoire. Il le fait jusqu'à trouver une solution ou qu'un temps maximum ou un nombre maximum d'essais soit atteint. La figure 8.7 montre un exemple d'exécution de cet algorithme sur le problème des 4 reines. L'affectation initiale apparaît à l'étape 0. Toutes les variables sont en conflit. Supposons que l'algorithme choisisse la 3ième reine. Le calcul des conflits fait apparaître que la position courante (4) est la meilleure (1 conflit). Aucun changement n'est effectué et toutes les variables restent en conflit (étape 1). Supposons que l'algorithme choisisse maintenant la 2ième reine. Le calcul des conflits fait apparaître que les positions 1 et 2 sont les meilleures (1 conflit chacune). Supposons que l'algorithme choisisse la position 1 pour cette reine. Le changement effectué ne modifie pas l'ensemble des variables en conflit (étape 2). Supposons que, par chance, l'algorithme choisisse la 4ième reine. Le calcul des conflits fait apparaître que la position 2 est la meilleure (0 conflit). Le changement effectué fait disparaître tous les conflits et produit une solution (étape 3).

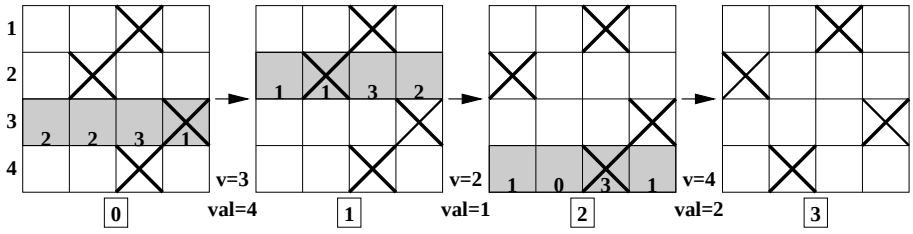

FIGURE 8.7 – Exemple d'exécution de l'algorithme *min-conflicts* sur le problème des 4 reines.

On notera que l'introduction de choix aléatoires dans la plupart des méthodes de recherche locale fait que deux exécutions peuvent donner des résultats très différents : solutions différentes, temps d'exécution différents, éventuellement production d'une solution dans un cas et pas dans l'autre. Pour pallier à ce défaut, une recherche locale est souvent lancée un certain

nombre de fois sur la même instance. Puis, la solution ou la meilleure solution est récupérée.

On notera enfin qu'il existe des situations particulières où une recherche locale, même très simple, de type *iterative improvement*, garantit la production d'une solution optimale. C'est le cas de la programmation linéaire (PL) ou l'algorithme du *Simplexe* peut être vu comme une recherche locale dans l'espace des sommets du polyèdre délimité par les contraintes : le voisinage d'un sommet est l'ensemble de ses sommets voisins dans le polyèdre ; on passe d'un sommet à un sommet voisin strictement meilleur du point de vue du critère, jusqu'à atteindre un sommet dont aucun voisin n'est strictement meilleur (optimum local) ; dans le cas particulier de la programmation linéaire, on montre que cet optimum local est aussi global. La complexité de l'algorithme du *Simplexe* (exponentielle) ne vient pas de la complexité de l'algorithme lui-même (recherche locale), mais de la taille exponentielle de l'espace de recherche (l'ensemble des sommets du polyèdre).

Chapitre 9

Éléments de méthodologie face à un problème d'optimisation combinatoire

Dans ce chapitre, nous visons à fournir quelques éléments de méthodologie qui peuvent être utiles à celui(celle) qui a à traiter un problème d'optimisation combinatoire. Pour la clarté de la présentation, nous faisons l'hypothèse de problèmes liés à la conduite de haut niveau d'un système physique quelconque, pouvant inclure des hommes et/ou des machines (avion, satellite, robot, système de production, de distribution, de transport, de communication ...).

Le premier conseil que nous voudrions donner, en fait le conseil essentiel, est de ne pas se précipiter vers telle ou telle façon de résoudre le problème et de passer un temps suffisant à son *analyse*. Cette analyse doit concerner le problème lui-même :

- Quelles sont les *décisions* possibles, c'est-à-dire les degrés de liberté du système physique sur lesquels il est possible de jouer ?
- Quelles sont les *contraintes* qui pèsent sur ces décisions ? Contraintes issues du monde physique ou contraintes correspondant à des situations jugées inacceptables.
- Quels sont les *critères* qui permettent d'évaluer une décision ou de comparer deux décisions ? S'il existe plusieurs critères, est-il possible de les combiner pour aboutir à un critère unique ? S'il n'est pas possible de les combiner, existe-t-il une hiérarchie entre eux ? Attention à la frontière entre contraintes et critères. N'exprimer comme contrainte que ce qui est absolument requis. Exprimer le reste sous

forme de critère.

— Toutes les données du problème sont-elles effectivement disponibles ? Pour certaines, avec quel niveau de précision ou de certitude ?

L'analyse doit aussi concerner l'*environnement* du problème et la façon dont il s'insère dans cet environnement :

— Quelle est la place du problème à résoudre dans l'ensemble du système de contrôle du système physique ? Le problème doit-il être résolu une seule fois, avant le démarrage du système physique (résolution *hors-ligne*) ? Doit-il être résolu plusieurs fois après le démarrage (au cours de la vie) du système physique (résolution *en ligne*) ? Dans ce cas, à quel rythme ? À intervalle réguliers ? Suite à certains événements ? Avec quelles contraintes temporelles sur la résolution ?

— Les données et la résolution sont-elles centralisées, décentralisées ou partiellement décentralisées ? En cas de décentralisation de la décision, quelles communications existent entre les différents centres de décision ? Les décisions sont-elles synchronisées ? Sont-elles coordonnées ?

Suite à cette phase d'analyse, une autre phase essentielle est celle qui consiste à construire un *modèle* du problème à résoudre. Pour construire un modèle, il faut d'abord de choisir un *cadre* de modélisation. Plusieurs peuvent éventuellement retenus (*CSP*, *PLNE*, graphes ...). Une fois un ou plusieurs cadres retenus, la modélisation doit être réalisée avec la plus grande rigueur possible. Ne pas laisser tel ou tel aspect dans l'ombre. Faire le point de ce qui peut être exprimé dans le modèle et de ce qui ne peut pas l'être. Une fois un ou plusieurs modèles construits, il est possible de les analyser et de les comparer :

— Taille sur des instances standards ? Nombre de variables, taille des domaines, nombre de contraintes ...

— Nature des contraintes et du critère ? Dureté des contraintes ?

— Classe de complexité ? P, NP, NPC ...

— Lien avec des problèmes classiques ?

— Possibilités de simplification, permettant éventuellement de se ramener à des problèmes classiques (par relaxation ou restriction des contraintes ou par modification du critère) ?

— Possibilités de décomposition en sous-problèmes indépendants ?

— Existence de symétries exploitables ?

C'est seulement ensuite qu'il est temps de se préoccuper de la façon de résoudre le problème. Comme pour la modélisation, plusieurs méthodes peuvent être retenues et comparées. Essentiellement, trois options sont alors disponibles :

— Récupérer ou réimplémenter un algorithme classique ;

— Faire appel à un outil de modélisation et de résolution industriel ou universitaire, commercial ou libre ;
— Concevoir et implémenter une méthode de résolution ad-hoc.

Il est nécessaire de comparer ces options en termes d'effort, de coût, de qualité, de pérennité et d'évolutivité du résultat.

Le choix entre grandes classes de méthodes de résolution (recherche arborescente, programmation dynamique, recherche locale, recherche gloutonne ...) doit se faire en fonction des résultats de l'analyse du ou des modèles retenus (taille des instances, nature des contraintes et du critère, structure du problème ...) mais aussi en fonction des résultats de l'analyse de l'environnement du problème (temps disponible pour la résolution, exigences en termes de qualité ...).

Ne pas forcément chercher à modéliser et à résoudre le problème dans toute sa complexité. Commencer avec ce qui est essentiel et ce qui est simple. Sophistiquer au fur et à mesure. À chaque étape, réaliser des expérimentations sur des instances jouets, puis sur des instances de taille standard. Analyser les résultats. Les utiliser pour améliorer les méthodes de résolution et régler leurs divers paramètres.

Quelques conseils en vrac pour terminer :

— Ne pas sauter les phases plus rébarbatives d'analyse du problème, de modélisation et d'analyse des modèles ;
— Ne pas réinventer la brouette ; se documenter via des livres de base, le web ou des spécialistes du domaine ;
— Ne pas se polariser sur tel aspect du problème difficile, mais marginal ; ne pas être obsédé par le gain de quelques points en termes d'optimisation si l'impact réel en termes de conduite du système est négligeable ; relever de temps en temps la tête pour se demander quels sont les véritables besoins des utilisateurs ?
— Garder à l'esprit que, pour de nombreux problèmes, l'algorithme magique n'existe pas ou pas encore ...

Chapitre 10

Outils logiciels existants

10.1 Outils de programmation linéaire et de programmation linéaire en nombres entiers

Certainement le plus utilisé : *C*PLEX [Ilob], maintenu et commercialisé par la société *I*log.

Une alternative : *X*Press-MP [Opt], développé et commercialisé par la société *D*ash Optimization.

10.2 Outils de programmation par contraintes

Le plus connu : *C*P Optimizer [Iloa], développé et commercialisé par la société *I*log.

Des alternatives : *C*HIP [Cos], développé et commercialisé par la société *C*osytec, et *S*ICStus Prolog [SIC], développé et commercialisé par la société *S*ICStus.

À la limite du monde industriel et du monde universitaire : *E*CLiPSe [Ecl], développé à *I*C PARC.

Un outil libre : *C*HOCO [Lab].

D'autres outils libres : *GNU Prolog* [Dia], *Gecode* [SS] et *FaCiLe* [Bri].

Un autre outil libre centré sur les méthodes de recherche locale : *C*OMET [HM, HM05].

10.3 Outils de modélisation : l'exemple d'OPL

10.3.1 Présentation générale

OPL est un outil de modélisation de problèmes d'optimisation combinatoire conçu par Pascal Van Hentenryck [Hen99], aujourd'hui développé et commercialisé par la société *Ilog* [Iloc].

Il permet de modéliser des problèmes de programmation linéaire en variables réelles (*PL*), entières (*PLNE*) ou mixtes (*PLM*), ainsi que des problèmes de satisfaction ou d'optimisation sous contraintes sur des variables à domaines finis (*CSP*).

Il permet donc d'exprimer des problèmes où les variables sont réelles (flottantes) ou entières, bornées ou non, et où les contraintes et le critère qui les lient sont linéaires (*PL*, *PLNE* ou *PLM*). Quand contraintes ou critère ne sont pas linéaires (exemple : une disjonction de contraintes linéaires), un pré-processing automatique et transparent peut permettre de les linéariser (transformation en un problème linéaire). De plus, l'outil permet d'exprimer des problèmes où les variables ont un domaine de valeur fini, numérique ou symbolique, et où les contraintes et le critère qui les lient sont quelconques (*CSP*).

Attention ! Il ne permet pas d'exprimer tous les problèmes où les variables sont réelles ou entières non bornées et où les contraintes et le critère sont quelconques. Exemples : `dvar float x ; constraints{x * x = 4 ;}` ou simplement `dvar float x; constraints{x < 2 ;}` (pas de linéarisation possible). L'outil ne permet donc pas de modéliser n'importe quel problème d'optimisation non linéaire (*PNL*).

Si vous voulez travailler dans le cadre *PL*, il vous faut donc veiller à ce que les contraintes et le critère soient linéaires ou linéarisables. Par contre, si vous voulez travailler dans le cadre *CSP*, il vous faut veiller à ce que les domaines soient finis.

Pour la résolution, *OPL* s'appuie, soit sur l'outil de programmation linéaire *CPLEX* [Ilob], soit sur l'outil de programmation par contraintes *CP Optimizer* [Iloa]. Par défaut, c'est *CPLEX* qui est appelé. Si vous voulez appeler *CP Optimizer*, vous devez écrire en début de modèle : `using CP ;`. Les méthodes de résolution utilisées par ces deux outils sont du type de celles qui ont été présentées dans les chapitres 6 et 7 (raisonnement et recherche arborescente).

10.3.2 Exemple idiot

On observe 20 têtes de lapins ou de faisans sans pouvoir différencier têtes de lapin et têtes de faisan. On observe de même 56 pattes de lapins ou de faisans sans pouvoir différencier pattes de lapin et pattes de faisan. Sachant qu'un faisan a 2 pattes et un lapin 4, on voudrait déterminer le nombre de lapins et de faisans qui ont produit ces têtes et pattes. Le modèle *OPL* correspondant à ce problème est le suivant :

```
dvar int nblapins ;
dvar int nbfaisans ;

constraints {
   nblapins + nbfaisans == 20 ; //tetes
   4*nblapins + 2*nbfaisans == 56 ; //pattes
}
```

Le problème, qui est une instance *PLNE*, est résolu via *CPLEX*, qui fournit son unique solution : `nblapins = 8, nbfaisans = 12`.

10.3.3 Mise en garde

OPL est un langage de modélisation dit *déclaratif*. Ce qu'on y exprime sont des équations, des relations entre variables. Toute tentative d'utilisation comme un langage de programmation *impératif*, de type *C* ou *C++* est vouée à l'échec. Par exemple, le modèle suivant, qui correspondrait à une tentative d'initialisation et d'incrémentation d'une variable, est clairement incohérent (seconde contrainte insatisfiable) :

```
dvar int x ;

constraints {
   x == 0 ;
   x == x + 1 ;
} ;
```

Autrement dit, *OPL* est un langage de *modélisation* de problème, pas de *programmation* de la façon de le résoudre, sauf via des commandes spéciales de contrôle de la recherche (voir section suivante).

10.3.4 Structure d'un modéle

Un modéle OPL comprend en général 5 parties :

1. des *données* ;
2. des *variables* ;
3. un *critère* ;
4. des *contraintes* ;
5. des commandes de *pré* ou *post-processing* et de *contrôle de la recherche* (optionnelles).

Dans le cas d'un problème de satisfaction pure (pas de critère), la structure est :

```
<Données>
<Variables>
constraints {<Contraintes>} ;
```

Dans le cas d'un problème d'optimisation, la structure est :

```
<Données>
<Variables>
minimize (ou maximize) <Critère> ;
subject to {<Contraintes>} ;
```

En l'absence de commandes de contrôle de la recherche, c'est la stratégie de recherche standard de l'outil qui est utilisée (transparente à l'utilisateur). Il est donc possible, au moins dans un premier temps, de se consacrer uniquement à la modélisation du problème.

10.3.5 Exemple un peu plus sérieux

Une société produit du gaz ammoniaque ($NH3$) et du chlorure d'ammonium ($NH4Cl$). Elle a en stock 50 unités d'azote (N), 180 unités d'hydrogène (H) et 40 unités de chlore (Cl). Le profit escompté est de 40 keuros par unité de $NH3$ et de 50 keuros par unité de $NH4Cl$. La question de déterminer comment utiliser le stock (quantités respectives de $NH3$ et de $NH4Cl$ à produire) de façon à maximiser le profit escompté. En voici une première modélisation :

```
dvar float+ gaz ; // production de NH3
dvar float+ chlorure ; // roduction de NH4Cl
```

```
maximize 40 * gaz + 50 * chlorure ; // profit escompté

subject to{
   gaz + chlorure <= 50 ; // stock de N
   3 * gaz + 4 * chlorure <= 180 ; // stock de H
   chlorure <= 40 ; // stock de Cl
}
```

Ce problème, qui est une instance *PL*, est résolu via *CPLEX*, qui fournit la solution optimale suivante : `gaz = 20, chlorure = 30`. La valeur optimale du critère associée est de `2300`.

Mais on peut observer que ce modèle est très spécialisé et vouloir l'étendre à la prise en compte de n'importe quel ensemble de produits. D'où cette seconde modélisation qui utilise un ensemble `Produits` de noms de produits et un tableau de variables `production` indexé par cet ensemble.

```
{string} Produits = {"gaz", "chlorure"} ; // produits

dvar float+ production[Produits] ;
  // production de chaque produit

maximize 40 * production["gaz"] + 50 * production["chlorure"] ;
  // profit escompté

subject to{
   production["gaz"] + production["chlorure"] <= 50 ;
    // stock de N
   3 * production["gaz"] + 4 * production["chlorure"] <= 180 ;
    // stock de H
   production["chlorure"] <= 40 ;
    // stock de Cl
}
```

On peut vouloir aller plus loin et séparer les données des contraintes et du critère. D'où cette seconde modélisation qui utilise deux ensembles `Produits` et `Composants` de noms de produits et de composants, des tableaux de données et de variables indexés par ces ensembles, l'opérateur d'agrégation `sum` et le quantificateur `forall`.

```
{string} Produits = {"gaz", "chlorure"} ; // produits
```

```
{string} Composants = {"azote", "hydrogene", "chlore"} ; // composants

float Demande[Produits][Composants] = [[1,3,0],[1,4,1]] ;
  // demande en composant de chaque produit
float Profit[Produits] = [40,50] ;
  // profit escompté par produit
float Stock[Composants] = [50,180,40] ;
  // stock par composant

dvar float+ production[Produits] ;
  // production de chaque produit

maximize sum(p in Produits) Profit[p] * production[p] ;
    // profit global escompté

subject to{
   forall(c in Composants)
     sum(p in Produits) Demande[p][c] * production[p] <= Stock[c] ;
      // respect du stock de chaque composant
}
```

On peut vouloir pousser encore plus loin la séparation entre les données d'une part et les contraintes et le critère d'autre part. *OPL* permet d'isoler les données dans un fichier différent. Le fichier contenant le modèle suffixé par `.mod` et le fichier contenant les données suffixé par `.dat` sont tous deux regroupés dans un projet suffixé par `.prj`. D'où cette troisième modélisation avec le fichier `.mod`:

```
{string} Produits = ... ; // produits
{string} Composants = ... ; // composants

float Demande[Produits][Composants] = ... ;
  // demande en composant de chaque produit
float Profit[Produits] = ... ;
  // profit escompté par produit
float Stock[Composants] = ... ;
  // stock par composant

dvar float+ production[Produits] ;
  // production de chaque produit
```

```
maximize sum(p in Produits) Profit[p] * production[p] ;
    // profit global escompté

subject to{
   forall(c in Composants)
     sum(p in Produits) Demande[p][c] * production[p] <= Stock[c] ;
      // respect du stock de chaque composant
}
```

puis le fichier `.dat` :

```
Produits = {"gaz", "chlorure"} ; // produits
Composants = {"azote", "hydrogene", "chlore"} ; // composants

Demande = [[1,3,0],[1,4,1]] ;
  // demande en composant de chaque produit
Profit = [40,50] ;
  // profit escompté par produit
Stock = [50,180,40] ;
  // stock par composant
```

On peut observer que le fichier `.mod` est maintenant complétement indépendant des ensembles de produits et de composants considérés, ainsi que des données numériques associées. C'est un modèle extrêmement général utilisable pour tout problème de gestion de production à base de composants.

10.3.6 Autres exemples

Problème des reines

Voir la section 5.3.1. Dans le cadre *CSP* :

```
using CP ;

range Position = 1..8 ;

dvar int reine[Position] in Position ;
  // une reine par ligne qui prend sa valeur dans une colonne

constraints{
```

```
   forall(ordered i,j in Position){
     reine[i] != reine[j] ;
      // pas sur la même colonne
     abs(reine[j] - reine[i]) != (j - i) ;
      // pas sur une même diagonale
     }
}
```

et en utilisant en plus la contrainte globale `allDifferent` qui spécifie que tous les éléments d'un tableau doivent être différents :

```
using CP ;

range Position = 1..8 ;

dvar int reine[Position] in Position ;
  // une reine par ligne qui prend sa valeur dans une colonne

constraints{
   allDifferent(reine) ;
      // pas sur la même colonne
   forall(ordered i,j in Position)
     abs(reine[j] - reine[i]) != (j - i) ;
      // pas sur une même diagonale
}
```

Problème de la série magique

Trouver une séquence de n entiers $S = \{s_0, s_1, \ldots, s_{n-1}\}$ telle que, pour tout i, $0 \leq i \leq n-1$, s_i soit le nombre d'occurrences de i dans S. Par exemple $\{1, 2, 1, 0\}$ est une série magique pour $n = 4$, puisqu'elle contient bien 1 zéro, 2 un, 1 deux et 0 trois.

```
using CP ;

int n = ...;
range Indices = 0..n-1 ;
range Nombres = 0..n ;

dvar int nombre[Indices] in Nombres ;
```

```
constraints{
   forall(i in Indices)
     nombre[i] == sum(j in Indices) (nombre[j] == i)) ;
}
```

À noter que `s[j] = i` est une contrainte dont le résultat est un booléen : 1 si elle est satisfaite et 0 sinon. D'une façon générale, ceci permet de compter le nombre d'éléments d'un ensemble qui satisfont une contrainte. Ici, on peut aussi utiliser la fonction `count` qui renvoie le nombre d'occurrences d'une valeur dans un tableau :

```
using CP ;

int n = ... ;
range Indices = 0..n-1 ;
range Nombres = 0..n ;

dvar int nombre[Indices] in Nombres ;

constraints{
   forall(i in Indices)
     nombre[i] == count(nombre, i) ;
}
```

Problème d'organisation de tâches de production

Voir la section 5.3.5. Dans le cadre *CSP* :

```
using CP ;

int n = ... ; // nombre de tâches
range T = 1..n ; // tâches

int DU[T] = ... ;  // durées des tâches
int TO[T] = ... ; // dates au plus tôt
int TA[T] = ... ; // dates au plus tard
int Hmin = min(t in T) TO[t] ;
int Hmax = max(t in T) (TA[t]-DU[t]) ;
range H = Hmin..Hmax ; // horizon temporel

int PR[T,T] = ... ; // matrice de précédence
```

```
int RP[T,T] = ... ; // matrice de partage de ressource

dvar int d[T] in H ; // date de début de chaque tâche

minimize max(t in T)(d[t] + DU[t]) ;
    // date de fin de l'ensemble des tâches

subject to{
   forall(t in T)
         TO[t] <= d[t] <= TA[t] - DU[t] ;
      // respect des dates au plus tôt et au plus tard
   forall(t1,t2 in T : PR[t1,t2] == 1)
         d[t1] + DU[t1] <= d[t2] ;
      // respect des contraintes de précédence
   forall(ordered t1,t2 in T : RP[t1,t2] == 1)
         (d[t1] + DU[t1] <= d[t2]) || (d[t2] + DU[t2] <= d[t1]) ;
      // respect des contraintes de partage des ressources
}
```

On notera que toutes les variables ont des domaines finis. Maintenant, dans le cadre PL :

```
int n = ... ; // nombre de tâches
range T = 1..n ; // tâches

float DU[T] = ... ;  // durées des tâches
float TO[T] = ... ; // dates au plus tôt
float TA[T] = ... ; // dates au plus tard

int PR[T,T] = ... ; // matrice de précédence
int RP[T,T] = ... ; // matrice de partage de ressource

dvar float d[T] ; // date de début de chaque tâche

minimize max(t in T)(d[t] + DU[t]) ;
    // date de fin de l'ensemble des tâches

subject to{
   forall(t in T)
         TO[t] <= d[t] <= TA[t] - DU[t] ;
```

```
      // respect des dates au plus tôt et au plus tard
   forall(t1,t2 in T : PR[t1,t2] == 1)
         d[t1] + DU[t1] <= d[t2] ;
      // respect des contraintes de précédence
   forall(ordered t1,t2 in T : RP[t1,t2] == 1)
         (d[t1] + DU[t1] <= d[t2]) || (d[t2] + DU[t2] <= d[t1]) ;
      // respect des contraintes de partage des ressources
}
```

On notera que ces dernières contraintes sont non linéaires (disjonction de contraintes linéaires) : *OPL* les linéarise automatiquement en pré-processing.

10.3.7 Survol des types de donnée en OPL

Entiers

```
int A = 5 ;
```

```
int A = -3 ;
```

```
int A = 5 ;
int B = A * A ;
```

Flottants

```
float A = -3.2 ;
```

```
float A = 2.5e-3 ;
```

Chaines de caractères

```
string A = ‘‘gaz’’ ;
```

Intervalles d'entiers

```
range Indices = 1..10 ;
```

```
int n = 4 ;
range Indices = n+1..2*n+1 ;
```

Intervalles de flottants

```
range float Intervalle = 1.2..3.4 ;
```

Tableaux

```
int A[1..4] = [2, 7, 3, 9] ;
```

```
float A[1..4] = [2.2, 7.35, -3.4, 9.0] ;
```

```
string A[1..3] = [``Lundi'', ``Mercredi'', ``Jeudi''] ;
```

```
range Indices = 1..5 ;
int A[Indices] =  [2, 4, 6, 8, 10] ;
```

```
range Indices = 1..5 ;
int A[i in Indices] =  2 * i ;
```

```
int A[1..2][1..4] = [[3, 4, 5, 6], [5, 6, 7, 8]] ;
```

```
int A[i in 1..2][j in 1..4] = 2 * i + j ;
```

Ensembles

```
{int} A = {4, 7, 11} ;
```

```
{string} A = {``gaz'', ``chlorure''} ;
```

Tuples

```
tuple point {int x, int y} ;
```

```
tuple point {int x, int y} ;
point P = <2, 3> ;
```

```
tuple point {int x, int y} ;
point Ps[1..3] = [<1, 2>, <2, 3>, <3, 4>] ;
```

```
tuple point {int x, int y} ;
point Ps[i in 1..3] = <i, i+1> ;
```

```
tuple point {int x, int y} ;
point P = <2, 3> ;
int X = P.x ;
```

```
tuple point {int x, int y} ;
tuple rectangle {point ll, point ur} ;
point P1 = <1, 2> ;
point P2 = <3, 5> ;
rectangle R = <P1, P2> ;
int X = R.ll.x ;
```

10.3.8 Les variables en OPL

```
dvar int a ;

dvar float+ a ;

dvar string a ;

dvar int a in 0..10 ;

range Domaine = 0..10 ;
dvar int a in Domaine ;

{string} Jours = {‘‘Lundi’’, ‘‘Mercredi’’, ‘‘Jeudi’’} ;
dvar string a in Jours ;
```

10.3.9 Les tableaux de variables en OPL

```
dvar int+ a[1..4] ;

dvar float a[1..4] ;

dvar string a[1..3] ;

dvar int a[1..4] in 1..10 ;

range Domaine = 1..10 ;
dvar int a[1..4] in Domaine ;

range Indices = 1..4 ;
range Domaine = 1..10 ;
dvar int a[Indices] in Domaine ;
```

10.3.10 Les expressions et tableaux d’expressions en OPL

Fonctions de variables OPL ou d’autres expressions OPL.

```
dvar int a ;
dvar int b ;
dexpr int c = a + b ;
dexpr int d = a * c ;

dvar int a[1..4] ;
dvar int b ;
dexpr int c = sum(i in 1..4) a[i] ;
dexpr int d[i in 1..4] = a[i] + b ;
```

10.3.11 Survol des expressions mathématiques en OPL

```
forall (i in Indices) ... ;

sum (i in Indices) ... ;

prod (i in Indices) ... ;

max (i in Indices) ... ;

forall (i in [(j+1)..(2*j+k)]) ... ;

forall (i,j in Indices) ... ;

forall (ordered i,j in Indices) ... ;

forall (i,j in Indices : j = 2*i+1) ... ;

forall (i in Indices1, j in Indices2 : i <= 2*j) ... ;
```

10.3.12 Survol des opérations et des relations en OPL

Addition : +
Soustraction : `-`
Multiplication : `*`
Division : `/`
Division entière : `div`
Modulo : `mod`
Valeur absolue : `abs`
Exposant : `^`
Égal : `==`
Différent : `!=`
Supérieur : `>`
Inférieur : `<`
Supérieur ou égal : `>=`
Inférieur ou égal : `<=`
Et logique : `&&`
Ou logique : `||`
Négation logique : `!`
Implication logique : `=>`
Equivalence logique : `<=>`
Appartenance : `in`

Non appartenance : `not in`
Union ensembliste : `union`
Intersection ensembliste : `inter`
Différence ensembliste : `diff`

Chapitre 11

Références

11.1 Livres

Sur l'*optimisation* en général : [NKT89].

Sur la *théorie des graphes* : [GM90, LPS94].

Sur la *programmation linéaire* et la *programmation linéaire en nombres entiers* : [NW88, GPS00].

Sur les *problèmes de satisfaction de contraintes* et la *programmation par contraintes* : [Dec03, RBW06].

Sur la *théorie de la complexité* : [GJ79, Pap94].

Sur les problèmes spécifiques d'*ordonnancement* de tâches : [LR01, GOT04, BPN01].

Sur les méthodes de *recherche locale* : [AL97, HS05].

Sur de nombreux *algorithmes d'optimisation combinatoire* : [Ski98].

11.2 Sites web

.

Tout sur les problèmes NP-complets :
http ://www.nada.kth.se/~viggo/problemlist/compendium.html

Constraint Programming Online :
http ://slash.math.unipd.it/cp/

Association Française de Programmation par Contraintes (AFPC) :
http ://www.afpc-asso.org/

Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF) : http ://www.roadef.org/

Bibliographie

[AL97] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[BPN01] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based Scheduling : Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.

[Bri] P. Brisset. FaCile. http ://www.recherche.enac.fr/opti/facile/.

[Cav06a] J.B. Cavaillé. Programmation linéaire et non linéaire. Document de cours, Supaéro, 2006.

[Cav06b] J.B. Cavaillé. Éléments de théorie des graphes. Document de cours, Supaéro, 2006.

[Cos] Cosytec. CHIP. http ://www.cosytec.com/.

[Dec03] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Dia] D. Diaz. GNU Prolog. http ://gnu-prolog.inria.fr/.

[Ecl] EclipseTeam. ECLiPSe. http ://www.eclipse-clp.org/.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, 1979.

[GM90] M. Gondran and M. Minoux. *Graphes et Algorithmes*. Eyrolles, 1990.

[GOT04] Groupe GOTHA, editor. *Modèles et algorithmes en ordonnancement*. Eyrolles, 2004.

[GPS00] C. Guéret, C. Prins, and M. Sevaux. *Programmation Linéaire*. Eyrolles, 2000.

[Hen99] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[HM] P. Van Hentenryck and L. Michel. COMET. http ://www.comet-online.org/.

[HM05] P. Van Hentenryck and L. Michel. *Constraint-based Local Search*. MIT Press, 2005.

[HS05] H. Hoos and T. Stützle. *Stochastic Local Search : Foundations and Applications*. Morgan Kaufmann, 2005.

[Iloa] Ilog. CP Optimizer. http ://www.ilog.com/products/cpoptimizer/.

[Ilob] Ilog. CPLEX. http ://www.ilog.com/products/cplex/.

[Iloc] Ilog. OPL Development Studio. http ://www.ilog.com/products/oplstudio/.

[Lab] F. Laburthe. CHOCO. http ://choco-solver.net/.

[LPS94] P. Lacomme, C. Prins, and M. Sevaux. *Algorithmes de graphes*. Eyrolles, 1994.

[LR01] P. Lopez and F. Roubellat, editors. *Ordonnancement de la production*. Hermes Science, 2001.

[NKT89] G. Nemhauser, A. Rinnooy Kan, and M. Todd. *Handbooks in Operations Research and Management Science : Optimization*. North-Holland, 1989.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[Opt] Dash Optimization. Xpress-MP. http ://www.dashoptimization.com/.

[Pap94] C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[RBW06] R. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[SIC] SICStus. SICStus Prolog. http ://www.sics.se/isl/sicstuswww/site/.

[Ski98] S. Skiena. *The Algorithm Design Manual*. Telos/Springer-Verlag, 1998.

[SS] C. Schulte and P. Stuckey. GECODE. http ://www.gecode.org/.