Méthodes pour l'optimisation combinatoire

Dans le précédent chapitre nous avons présenté un problème inhérent à la conception /reconfiguration des lignes d'usinage sans évoquer les moyens offerts par la recherche opérationnelle pour le résoudre. Dans le présent chapitre, nous présentons d'abord une introduction à la théorie de la complexité pour sensibiliser le lecteur à la difficulté du problème posé. Ensuite, nous rapportons les méthodes les plus utilisées pour ce type de problèmes. Nous développons particulièrement les schémas basés sur les procédures de type séparation et évaluation car ce sont celles-ci que nous employons dans nos travaux.

2.1 Théorie de la complexité

La théorie de la complexité s'intéresse de façon générale aux problèmes de l'optimisation combinatoire et de façon plus particulière aux problèmes de décision qui leur correspondent. Les problèmes d'optimisation se caractérisent par des contraintes à satisfaire et un objectif à minimiser ou à maximiser. Pour les résoudre à l'optimum, il est toujours possible de parcourir tout l'espace de recherche en analysant toutes les solutions réalisables[1] pour ne garder que la meilleure du point de vue de l'objectif défini.

Un problème de décision est un problème pour lequel il n'y a que deux réponses possibles : oui ou non. Le problème de décision compare, en général, une valeur fixe à la valeur que peut prendre la fonction objectif. La procédure la plus basique pour y répondre est de parcourir tout l'espace des solutions réalisables (ce qui correspond au pire des cas). Ce processus permet, entre autres, de trouver une (voire plusieurs) solution(s) satisfaisant à la condition, dans ce cas, le processus de recherche est arrêté et la réponse donnée est positive. Toutefois si aucune solution parcourue n'est faisable, après que la totalité de l'espace de recherche ait été parcouru, alors une telle valeur n'existe pas et la réponse à donner est négative.

[1]Pour les problèmes difficiles sur lesquels nous reviendrons par la suite, cet espace est large et le nombre de solutions réalisables est très grand, c'est pourquoi il faut trouver le moyen d'éviter de parcourir la totalité des solutions.

Au sens de l'effort de calcul nécessaire, les problèmes d'optimisation sont au moins aussi difficile que les problèmes de décision qui leur correspondent. En effet, la résolution à l'optimum des problèmes d'optimisation nécessitent une énumération complète de l'espace de recherche (à moins d'avoir une preuve de la sous-optimalité de certains espaces qui pourraient, de ce fait, être ignorés). Par contre, pour des problèmes de décision, celle-ci dépend essentiellement de la valeur de l'objectif recherchée, ainsi l'exploration s'arrête dès qu'une telle solution est trouvée. Par conséquent, au mieux le problème de décision est plus facile que le problème d'optimisation correspondant. Au pire des cas, il lui est équivalent. Ainsi, même si la théorie de la complexité a été conçue pour les problèmes de décision, ses résultats sont aussi valables pour les problèmes d'optimisation correspondants.

2.1.1 Complexité d'un algorithme

La complexité d'un algorithme est définie par le temps de calcul nécessaire à son exécution pour la résolution d'un problème d'une taille donnée. Ce temps est, en réalité, exprimé en fonction du nombre d'instructions élémentaires nécessaires à l'exécution de l'algorithme par le processeur. Le choix de cette mesure est justifié par le fait que le nombre d'instructions est indépendant de la puissance des machines qui détermine le temps de calcul effectif. La théorie de la complexité s'intéresse, entre autres, à l'ordre de grandeur de la complexité dans le pire des cas qui est dite en grand $\mathcal{O}$. Pour estimer la complexité d'un algorithme, en grand $\mathcal{O}$, il suffit de borner le nombre d'instructions élémentaires qu'il engendre. Cette borne est exprimée en fonction de la taille des données n, et lorsqu'elle s'écrit en $p(n)$ tel que $p(n)$ est une fonction polynomiale, l'algorithme est dit polynomial et sa complexité est donnée par $\mathcal{O}(p(n))$. Notons que lorsque celle-ci est linéaire la complexité est également dite linéaire. Il existe d'autres types de complexité, par exemple : l'exponentielle soit en $\mathcal{O}(a^n)$. Les algorithmes de complexité en $\mathcal{O}(n!)$ ou en $\mathcal{O}(n^{logn})$ sont également considérés comme des algorithmes de complexité exponentielle.

Jusqu'à présent, nous n'avons abordé que la complexité des algorithmes de résolution, cependant, il est important de la distinguer de la complexité des problèmes eux-mêmes, et ce même si la complexité d'un problème est directement liée à celle de ses algorithmes. De manière informelle, la complexité d'un problème est déterminée par celle du meilleur algorithme connu qui le résout [Pas05]. Dans la suite, nous expliquons plus en détail ce lien en distinguant les deux classes de problèmes qui en découlent, la classe P qui contient les problèmes faciles et la classe NP qui comportent ceux qui sont dits difficiles.

2.1.2 La classe P

Cette classe comporte les problèmes pour lesquels il existe un algorithme de résolution de complexité polynomial, *i.e.* en $\mathcal{O}(p(n))$ tel que $p(n)$ est une fonction polynomiale. Les problèmes de cette classe sont dits faciles et leurs algorithmes de résolution sont souvent efficaces. Toutefois, en pratique, si la fonction polynomiale fait intervenir de haut coefficients avec un degré fort alors un algorithme exponentiel peut être plus efficace. Par exemple, un

algorithme en $\mathcal{O}(2^n)$ peut être plus efficace qu'un autre en $\mathcal{O}(n^{100})$. Si on prend l'exemple de $n = 10$ alors le premier nécessite 1024 instructions tandis que l'algorithme polynomial en engendrera 10^{100}.

2.1.3 La classe NP

Contrairement à ce qu'on peut imaginer, NP ne signifie pas Non Polynomial mais Polynomial Non déterministe. Les problèmes appartenant à cette classe sont définis comme des problèmes qui peuvent être résolus avec des algorithmes non déterministes.

Dans [GJ79], les auteurs proposent une définition pour les problèmes de décision appartenant à cette classe. Ils supposent connaître à l'avance une solution du problème, sans vraiment se soucier de la façon dont elle a été fournie. Dans ce cas, ils suggèrent de vérifier la faisabilité de la solution providentielle et si cette vérification se fait en un temps polynomial alors le problème correspondant appartient à la classe NP.

Les résultats de la théorie de la complétude sont basés sur la conjecture : P$\neq$NP et s'intéresse aux problèmes de décision appartenant à NP-P dits NP-complet. Toutefois, et d'après les définitions des deux classes, nous avons a priori : P$\subseteq$ NP, car toute instance qui peut être résolue avec un algorithme déterministe peut l'être avec un algorithme non déterministe.

Les problèmes d'optimisation qui nous intéressent sont des problèmes dit NP-difficiles, c'est-à-dire ceux pour lesquels les problèmes de décision correspondant sont dit NP-complets. Pour montrer qu'un problème p_1 est NP-complet, il suffit de trouver un autre problème NP-complet p_2 tel que le problème p_2 se réduit polynomialement à p_1. La réduction polynomiale nécessite de pouvoir transformer les données d'une instance de p_2 en des données d'une instance de p_1 avec un nombre polynomial d'étapes. De plus, il faut assurer qu'une solution de p_1 peut être également obtenu, après une transformation polynomiale d'une solution de p_2 (nous référons à l'ouvrage de [GJ79] pour plus de détails sur ce sujet).

2.1.4 Retombées pratiques

Lors de la résolution des problèmes NP-difficiles dont la complexité est exponentielle[2] nous sommes assez vite confrontés à une explosion combinatoire lorsque la taille des instances croît. Afin de sensibiliser le lecteur à la notion de l'explosion combinatoire, nous avons choisi d'apporter un exemple qui nous paraît des plus percutants et qui illustre bien ce phénomène. Soit le nombre qui est égal à : $2^{2^{2^{2^2}}}$, la valeur de ce nombre est supérieure à l'estimation du nombre d'atomes dans l'univers[3] [Col02]. Ainsi, il suffit de combiner cinq fois des deux en exposant pour atteindre des nombres astronomiques au sens propre du terme.

[2]Rappelons que cette classe comportent des algorithmes ayant une complexité en factoriel ou encore en $\mathcal{O}(n^{logn})$.

[3]Pour calculer ce nombre, il faut procéder de haut en bas, *i.e.* décomposer le nombre à partir de 2^2 du sommet ce qui fait 4, puis effectuer 2^4 ce qui fait 16, puis $2^{16} = 65536$, et enfin 2^{65536}.

Toutefois, la complexité d'un algorithme est mesurée dans le pire des cas, il est donc possible de résoudre certaines instances en un temps polynomial. Il y a des algorithmes qui sont efficaces dans la pratique, c'est le cas de l'algorithme du *simplex* qui a une complexité exponentielle [KM72] mais qui permet en pratique de résoudre en un temps polynomial même des instances difficiles.

Le choix d'une méthode pour la résolution d'un problème difficile doit se faire en fonction des objectifs à atteindre et doit prendre en compte la structure du problème. En effet, certaines approches se prêtent mieux à la résolution d'un type de problèmes par rapport à d'autres parce qu'elles conviennent mieux à leur structure. Les approches pour la résolution des problèmes de la classe NP peuvent être classées en deux catégories : les méthodes exactes et les méthodes approchées. Une méthode qui apporte une ou plusieurs solutions optimales en fournissant la preuve de l'optimalité est une méthode exacte. Lorsque l'enjeu de trouver la meilleure solution est important, que la taille des instances n'est pas très grande et que le temps alloué au calcul n'est pas restrictif, l'emploi d'une méthode exacte est envisageable.

Dans le cadre de cette thèse, nous nous sommes focalisés sur les méthodes exactes, c'est pourquoi nous les décrivons plus en détails dans la suite. Dans le paragraphe suivant, nous abordons une méthode de résolution lorsque les variables sont de type réel. Nous reviendrons par la suite sur les méthodes de résolution employant des variables entières et/ou binaires.

2.2 Programmation linéaire

Un programme linéaire permet de définir l'objectif à minimiser ou à maximiser[4] comme une expression linéaire des variables de décision (voir 2.1). Quant aux contraintes elles sont exprimées sous forme d'inégalités (voir 2.2) qui sont des expressions linéaires bornées[5]. Un programme linéaire (PL) peut être exprimé comme suit :

$$min \quad cx \tag{2.1}$$

$$s.c. \quad Ax \geq b \tag{2.2}$$

$$x \in \mathbb{R}^n_+ \tag{2.3}$$

avec $c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{mn}$

2.2.1 Résolution des programmes linéaires

L'algèbre linéaire offre une possibilité de résolution des programmes linéaires en se basant sur leur interprétation géométrique. En fait, l'espace des solutions réalisables S du programme linéaire correspond à un polyèdre de l'espace euclidien $\mathbb{R}$ lorsque celui-ci est non vide et non borné (si l'espace est non vide et borné c'est d'un polytope dont il s'agit, voir figure 2.1) [NW98]. L'espace S, appelé plus communément enveloppe convexe, est défini en

[4]max cx est équivalent à min (-cx)

[5]Notons que lorsque les deux bornes sont égales, les inégalités se réduisent à des égalités (équations).

fonction de ces facettes qui correspondent aux contraintes du PL et de ses points extrêmes qui sont définies par l'intersection des contraintes, les contraintes délimitant un point sont dites saturées à ce point, c'est le cas de la facette1 aux points p_1 et p_5 de la figure 2.1.

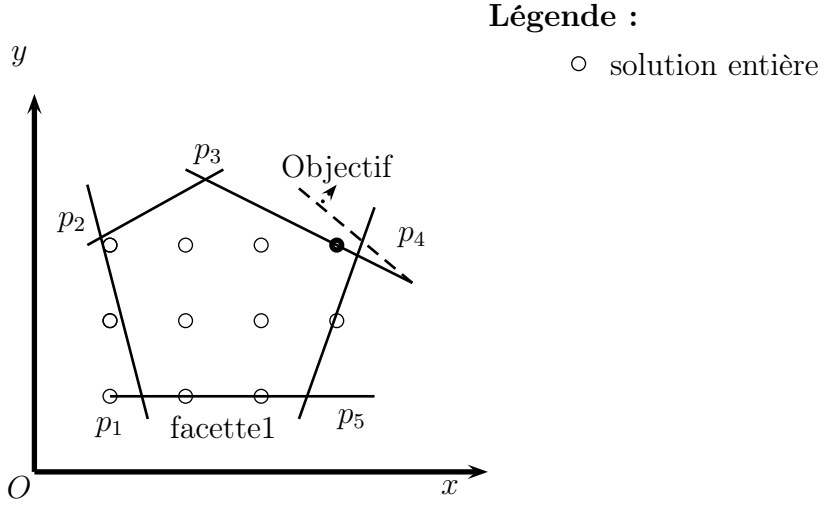

FIG. 2.1 – Enveloppe convexe

Lorsque S est non vide, le programme linéaire admet une ou plusieurs solutions réalisables, la meilleure solution coincide forcément avec un des points extrêmes (sommets) de l'enveloppe convexe de S, car quelle que soit la direction induite par l'objectif la meilleure valeur sera toujours sur la frontière de l'enveloppe convexe. Il arrive toutefois qu'il y ait plusieurs solutions optimales, dans ce cas, ces solutions correspondent à tous les points d'une facette.

Une façon de retrouver la meilleure solution est de parcourir les points extrêmes pour n'en garder que le meilleur du point de vue de la valeur de cx. C'est le raisonnement qui est proposé par l'algorithme du simplex qui a été introduit par [Dan51] dans sa forme primale. L'algorithme du simplex permet d'enclencher un processus itératif pour passer d'un point extrême à un autre en améliorant la qualité des solutions au fur et à mesure de l'avancement. La complexité du simplex n'est pas polynomial, néanmoins il est très efficace en pratique.

Pour certaines structures de problèmes, il est plus intéressant de faire coopérer les deux approches car le dual peut s'avérer plus facile à résoudre.

2.2.2 Programmes linéaires en nombres entiers

Lorsque les variables sont toutes entières le programme linéaire est dit en nombres entiers, il est noté PLNE. Un tel programme peut être exprimé par (2.4)-(2.5).

$$min \quad cx \tag{2.4}$$
$$s.c. \quad Ax \geq b \tag{2.5}$$
$$x \in \mathbb{N}_+^n \tag{2.6}$$

Pour ne considérer que les rationnels, on suppose : $c \in \mathbb{Z}^n, b \in \mathbb{Z}^m, A \in \mathbb{Z}^{mn}$

De façon plus générale, et selon la nature du problème, les variables de décision peuvent être de différents types, on dira que le programme est en variables mixtes s'il y a, à la fois, des variables réelles et entières. On note PLVM pour *Programme linéaire en variables mixtes* [NW98].

$$min \quad cx + hy \tag{2.7}$$
$$s.c. \quad Ax + Gy \geq b \tag{2.8}$$
$$x \in \mathbb{Z}_+^n, y \in \mathbb{R}_+^p \tag{2.9}$$

tel que : $c \in \mathbb{Z}^n, h \in \mathbb{Z}^p, b \in \mathbb{Z}^m, A \in \mathbb{Z}^{mn}, G \in \mathbb{R}^{mp}$

À l'inverse du PL, la solution optimale du PLNE n'est pas forcément contenue sur la frontière de son enveloppe convexe. En raison de l'intégralité des variables, les solutions réalisables peuvent ne pas se trouver sur la frontière définie par S. Ainsi, l'espace de solutions du PLNE est contenu dans S (l'enveloppe convexe du PL correspondant, voir la figure 2.1). Lorsque les contraintes d'intégralité sont relâchées le problème en nombres entiers se ramène au problème (PL) correspondant, il s'agit alors de relaxation du PLNE en PL et la valeur optimale du PL constitue une borne inférieure pour le PLNE correspondant.

Il existe plusieurs procédés pour trouver une solution optimale à un PLNE, parmi ceux-ci, nous citons les procédures arborescentes qui permettent le parcours total de l'espace de recherche par énumération. Ce schéma global gagne en intérêt lorsqu'il est amélioré en employant des techniques de détection de régions sub-optimales afin de les écarter de la recherche.

La procédure par séparation et d'évaluation (PSE), connue sous l'appellation anglo-saxonne de *Branch and Bound*, en est un exemple. Notre intérêt se porte particulièrement sur cette méthode car son caractère générique permet de l'employer dans plusieurs cadres sur lesquels nous reviendrons plus en détails dans les sections qui suivent (elle est notamment à la base des solvers des outils d'ILOG que nous utilisons dans nos travaux). Nous décrivons le schéma global d'une telle approche en mettant l'accent sur les techniques les plus utilisées pour améliorer les performances de cette approche en réduisant l'espace de recherche.

2.3 Procédure par séparation et évaluation

C'est une méthode générique d'énumération implicite qui exploite la structure spécifique des problèmes à résoudre pour réduire l'effort de recherche. L'énumération se fait en dé-

composant le problème original en sous-problèmes chacun séparant l'espace de recherche en plusieurs sous-espaces. Plus exactement, c'est lors d'un branchement qu'une décision est prise déterminant une décomposition en plusieurs sous-problèmes. Les branchements se font au fur et à mesure de la construction de l'arbre en commençant de la racine où aucune décision n'a été prise jusqu'aux feuilles potentielles. Par feuilles nous désignons les nœuds qui n'ont pas de fils possibles (lorsque toutes les décisions ont été prises) et par les feuilles potentielles, nous entendons celles qui peuvent aboutir à des solutions réalisables car toutes les décisions qui sont prises ne mènent pas forcément à des solutions réalisables. Ainsi, des contradictions avec certaines contraintes peuvent être générer, dans ce cas, le processus de recherche est abandonné en coupant la branche correspondante et en reprenant le développement des autres branches. Pour plus de détails sur les PSE nous référons aux ouvrages suivants : [Min83], [Sak84], [DMM97] et [NW98].

Les performances d'un arbre de séparation et d'évaluation sont directement liées à sa taille ; en général, plus sa taille est réduite plus l'approche est efficace. Il faut toutefois noter que l'approche peut être efficace pour des arbres relativement grands lorsque les traitements effectués aux nœuds ne nécessite pas de longs temps de calcul. Cependant, pour réduire la taille de l'arbre, des techniques d'évaluation de solutions permettent de déduire que certaines branches sont non prometteuses. Un mécanisme d'élagage peut ensuite être appliqué afin de «couper» les branches infructueuses qui ont été détectées. Pour ce faire, il suffit de détenir une solution réalisable et une borne de la meilleure valeur que pourrait prendre la fonction objectif au vu des décisions déjà prises, et ce quelles que soient les décisions à venir. La borne utilisée est une borne inférieure dans le cas d'un objectif à minimiser et une borne supérieure dans le cas contraire. Dans la suite, ne nous considérerons que le cas où l'objectif est à minimiser.

Le calcul de la borne inférieure est effectué à chaque nœud de l'arbre, il s'agit alors de la valeur du meilleur coût, que pourrait avoir toute solution développée à partir de ce nœud. Lorsque la valeur de la borne inférieure atteint ou dépasse la valeur de la meilleure solution réalisable rencontrée auparavant, alors aucune solution obtenue à partir de ce nœud ne peut correspondre à la solution optimale. Le mécanisme de l'élagage peut alors opérer en coupant cette branche infructueuse permettant ainsi de réduire l'effort de recherche.

Plus les techniques de calcul de la borne exploitent la structure spécifique du problème traité, meilleure est la qualité de la borne inférieure. Néanmoins, l'obtention de la borne ne doit pas susciter un temps de calcul trop élevé car le cumul de ces temps sur l'ensemble de l'arbre peut ralentir sa construction, ce qui aurait pour conséquences de réduire de façon significative les performances globales du schéma de résolution. Il est donc important de trouver la juste mesure entre l'effort à fournir dans le calcul de la borne et sa qualité.

2.4 Programmation dynamique

L' approche de programmation dynamique est basée sur les principes d'optimalité introduits dans [Bel57, BD62]. En pratique, il faut trouver une expression récursive décomposant le problème en plusieurs sous-problèmes [DM67]. Le principe est de résoudre le problème

initial à partir des solutions des sous-problèmes, pour plus de détails se référer à [NW98]. Ce processus commence d'un état initial x_0 qui permet de passer à un état suivant x_1 en prenant une décision u_1, ce processus est réitéré jusqu'à l'obtention d'un état terminal x_n après avoir pris la décision u_n. Ainsi, l'ensemble de décisions est noté $(u_1, \ldots, u_n)$, celui-ci engendre un coût $f(u_1, \ldots, u_n)$, tel que :

$$f(u_1, \ldots, u_n) = \sum_{i=1}^{n} f_i(x_{i-1}, u_i) \tag{2.10}$$

L'équation (2.10) montre que la prise d'une décision u_i correspondant à un état x_{i-1} engendre un coût $f_i(x_{i-1}, u_i)$ qui ne dépend ni des décisions antérieures ni des décisions ultérieures excepté le fait d'amener à l'état x_i.

2.5 Programmation par contraintes

2.5.1 Définition

Pour expliquer le principe de la programmation par contraintes, nous utilisons le *Problème de Satisfaction de Contraintes* (CSP). Une instance de ce problème est définie par un ensemble de variables $X = \{x_1, \ldots, x_n\}$. À chaque variable x_i correspond un domaine d_i contenant toutes les valeurs que peut prendre cette variable, ainsi l'ensemble des domaines est représenté par $D = \{d_1, \ldots, d_n\}$. Il existe plusieurs contraintes qui lient les variables entre elles $C = \{c_1, \ldots, c_m\}$ telles qu'une contrainte c_j combine plusieurs variables, on écrit : $c_j = \{x_{j_1}, \ldots, x_{j_n}\}$ et que la combinaison des valeurs de ses variables définit le domaine de la contrainte soit $dc_j = d_{j_1} \times d_{j_2} \times \ldots \times d_{j_n}$. Une variable x_i est dite *instanciée* lorsqu'une valeur appartenant à son domaine lui est assignée. L'instanciation est dite *complète* si toutes les variables sont instanciées, elle est *partielle* autrement. De plus, l'instanciation complète est *une solution réalisable* si les valeurs d'instanciation font que toutes les contraintes soient respectées. De même, l'instanciation partielle est une solution partielle si toutefois toutes les contraintes faisant intervenir des variables instanciées sont respectées.

2.5.2 Propagation des contraintes

À la différence de la plupart des approches pour la résolution des problème d'optimisation, la programmation par contraintes ne limite pas l'utilisation des contraintes à la validation des solutions, celle-ci les exploitent et les fait participer activement dans le processus de résolution [BLPN03], cette technique est appelée *propagation des contraintes*. Elle consiste en un mécanisme de déductions qui tend à inférer des contraintes afin de réduire les domaines des variables, on parlera alors de *filtrage des domaines*.

Prenons un exemple d'un CSP binaire qui fait intervenir deux variables x_1 et x_2 telles que leur domaine est égal respectivement à $d_1 = \{0, 1, 2\}$ et $d_2 = \{0, 1, 2\}$:

Si on considère la contrainte suivante : $x_1 < x_2$

La propagation de cette contrainte est obtenue en déduisant les valeurs de chacune des variables qui ne peuvent participer à une solution quelque soit la valeur de l'autre variable. En l'occurrence toute valeur de d_1 qui est supérieure ou égale au maximum de x_2 violera la contrainte, elle est donc inconsistante et doit être supprimée de d_1. Le même raisonnement au regard de la seconde variable x_2 permet de déduire que la valeur 0 ne peut en aucun cas satisfaire à la contrainte et est donc à soustraire de d_2. Intuitivement nous avons appliqué pour le filtrage effectué les règles suivantes :

$$d_1 \longleftarrow d_1 - \{\forall x_i \in d_1 | x_i \geq max(d_2)\} \Longleftrightarrow d_1 \longleftarrow d_1 - \{2\} = \{0, 1\}$$
$$d_2 \longleftarrow d_2 - \{\forall x_i \in d_2 | x_i \leq min(d_1)\} \Longleftrightarrow d_2 \longleftarrow d_2 - \{0\} = \{1, 2\}$$

Ces règles permettent d'assurer la consistance des domaines d_1 et d_2.

2.5.3 Démarche globale

La propagation des contraintes pour un problème NP-difficile est, en pratique, une méthode de résolution incomplète, dans le sens où il subsiste souvent plusieurs possibilitées d'instanciation pour les variables, ce qui fait que la propagation à elle seule ne suffit pas pour détecter toutes les inconsistances de l'espace de recherche. Pour compléter ce schéma, il est donc nécessaire de faire participer une technique de recherche pour instancier les variables aux valeurs résiduelles. Pour ce faire, la méthode usuelle est un algorithme de recherche arborescente pour lequel il faut :

1. définir la politique de branchement, en l'occurrence les contraintes à rajouter en termes d'instanciation de variables ou encore de domaines à subdiviser ;
2. déterminer une politique de retour en arrière *backtracking*, c'est-à-dire expliciter les décisions à prendre lorsqu'une incohérence est détectée. Plus exactement, quelle variable et quelle alternative d'instanciation doit être considérée ?

Les travaux de [SS77] ont introduit la séparation entre les méthodes de déduction (dans le cas de la programmation par contraintes la méthode de déduction est la propagation de contraintes) et les algorithmes d'exploration de l'espace de recherche. La représentation des contraintes, en accord avec la règle de programmation logique, introduite par Kowalski [Kow79], stipulant : algorithme = logique + contrôle.

L'ensemble est repris dans le schéma global, proposé dans [BLPN03], d'une approche basée sur la programmation par contraintes séparant la procédure de recherche et la propagation de contraintes que nous rapportons dans la figure 2.2. Ce schéma montre la séparation entre la définition du problème qui fournit les contraintes initiales et le processus de résolution qui fait coopérer la propagation de contraintes et l'algorithme de résolution représenté par une heuristique de recherche (backtracking). Le schéma met en avant l'ajout de contraintes additionnelles suite à des prises de décision dans l'algorithme de recherche d'une part et à

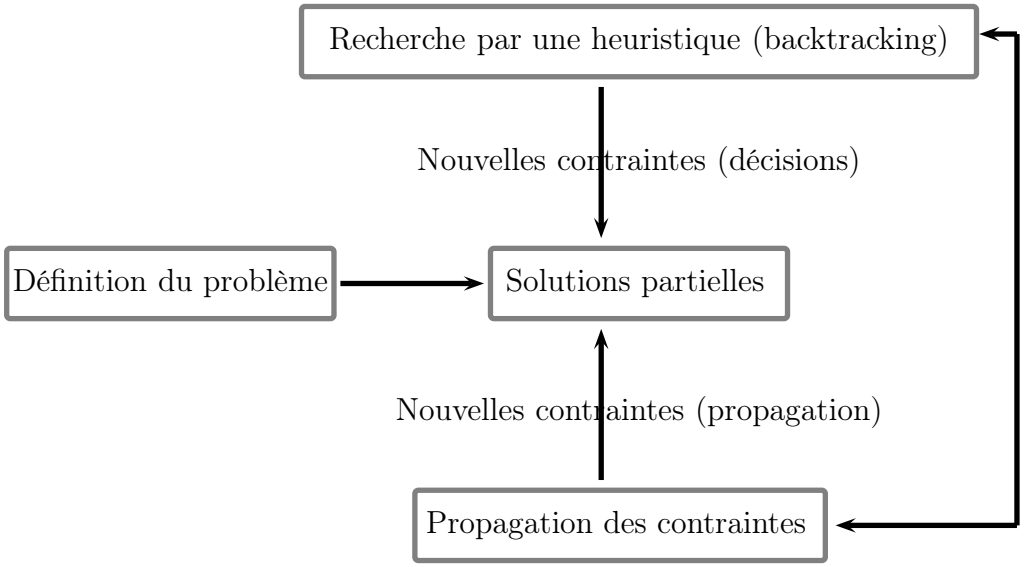

FIG. 2.2 – Interaction entre les différents intervenants d'une résolution par PPC

la propagation de contraintes inférant ces nouvelles décisions d'autre part (c'est l'arc qui lie la recherche par une heuristique et la propagation des contraintes).

Dans la suite, nous décrivons brièvement les méthodes approchées les plus importantes pour souligner notamment les différences avec les méthodes exactes vues jusque-là.

2.6 Méthodes approchées

L'objectif des méthodes approchées est de fournir des solutions de bonne qualité, voire optimales sans se soucier d'apporter la preuve de l'optimalité des solutions. Les méthodes approchées mettent en œuvre des règles heuristiques qui sont spécifiques aux problèmes traités.

Les règles heuristiques peuvent être intégrées dans des approches plus globales, dites méta-heuristiques. Contrairement aux règles heuristiques, les méta-heuristiques ont un caractère générique indépendant des spécificités des problèmes traités. [OL96] les définit comme étant un processus itératif de génération dirigeant une heuristique subordonnée et combinant plusieurs techniques afin d'exploiter et explorer l'espace de recherche en mettant en œuvre des stratégies pour structurer l'information dans le but d'atteindre des solutions proches de l'optimum.

Les méta-heuristiques peuvent être classées en méthodes amélioratives ou constructives. Les algorithmes génétiques sont un exemple du premier type ; cette méthode procède à partir d'une population initiale qu'elle améliore du point de vue de la valeur de la fonction objectif. Les mécanismes pour explorer et exploiter l'espace de recherche sont mis en œuvre grâce aux opérateurs de croisement et de mutation. Nous référons aux ouvrages suivants pour plus de

détails à ce sujet [Gol89, Dav91, Fal96, Koz92, Mul93].

La recherche tabou est également considérée comme une des méta-heuristiques les plus performantes. Cette méthode est un processus itératif explorant l'espace de recherche en passant d'une solution à une autre. La méthode tabou emploi un mécanisme nommé liste tabou afin d'éviter de rester bloquer dans des optimums locaux. L'idée est de garder en mémoire un certain nombre de solutions parmi les dernières parcourues dans le but de ne pas y revenir [DP06]. Il est toutefois permis de garder une solution de moindre qualité car celle-ci peut mener, moyennant une exploration de voisinage, à une bonne solution [GL97].

Par ailleurs, les colonies de fourmis, qui constituent une méthode constructive, permettent de trouver des solutions en s'inspirant du principe de base des fourmis. Le mécanisme met en place un système de collaboration de plusieurs agents (fourmis) pour permettre aux meilleurs de contribuer aux solutions futures (qui seront construites lors des itérations qui suivent) via une mémoire commune : la phéromone, grâce à laquelle une exploitation de l'espace de recherche est accomplie [Bru93, Tai95, DG97]. L'exploration est ajustée au moyen de paramètres qui règlent la proportion du choix aléatoire par rapport aux acquis par expérience. Il est à signaler que cette méthode est récente, il est donc difficile de juger de son efficacité contrairement aux deux méthodes précédentes.

Il est toutefois intéressant de noter qu'une méthode exacte tronquée est une méthode approchée. Ainsi, lorsqu'un algorithme exact est interrompu avant la fin de son déroulement, c'est-à-dire qu'il n'a pas apporté la preuve de l'optimalité de la solution trouvée celle-ci devient de ce fait une solution approchée.

2.7 Conclusion

Nous avons présenté dans ce chapitre un panorama des méthodes exactes et approchées qui sont utilisées dans l'optimisation combinatoire.

Chaque type de méthodes présente des inconvénients et des avantages qui lui sont propres selon les objectifs visés il peut être plus judicieux d'employer l'une ou l'autre. Concernant les méthodes approchées leur principal point faible réside dans le manque de stabilité des algorithmes au vu de la qualité des solutions apportées. Le plus souvent, ces méthodes ne garantissent aucun écart maximum de l'optimum risquant même de ne jamais converger. Même lorsqu'une méthode approchée est efficace pour un type de problème, il reste difficile de garantir que la qualité des résultats obtenus soit la même pour des données différentes. Dans cette thèse, nous avons fait le choix d'une résolution exacte car quand elle devient possible, son emploi est préférable aux méthodes approchées. De plus, pour les lignes dédiées, nous pouvons accorder du temps à la résolution du problème traité car c'est une décision qui n'intervient qu'une fois durant la vie du système lors de la conception préliminaire et une fois tous les 7 ans environ lors d'une reconfiguration.

À présent que nous avons introduit les méthodes de résolution de façon générale, nous consacrons le chapitre suivant à la classe de problèmes d'équilibrage de lignes d'assemblage qui représentent les problèmes les plus proches de ceux qui nous intéressent.