LES MÉTA-HEURISTIQUES : quelques conseils pour en faire bon usage

Alain HERTZ
Ecole Polytechnique - GERAD
Département de mathématiques et de génie industriel
CP 6079, succ. Centre-ville,
Montréal (QC) H3C 3A7, Canada
E-mail address: Alain.Hertz@gerad.ca

1. Introduction

La solution optimale à un problème d'optimisation ne peut que très rarement être déterminée en un temps polynomial. Il est donc souvent nécessaire de trouver des modes de résolution qui fournissent une solution de bonne qualité dans un laps de temps raisonnable : c'est ce que font les heuristiques. Depuis une vingtaine d'années, les heuristiques les plus populaires, et également les plus efficaces, sont des techniques générales, appelées méta-heuristiques, qu'il s'agit d'adapter à chaque problème particulier. Parmi ces techniques générales, nous nous pencherons dans ce chapitre sur deux approches ayant clairement démontré leur utilité dans de nombreux domaines, y compris la gestion des systèmes de production. La première de ces approches, appelée *Recherche Locale*, est couramment utilisée depuis plus de 20 ans et comprend, entre autres, la technique du Recuit Simulé [Kirkpatrick et al., 1983] et la Recherche Taboue [Glover, 1986]. La deuxième approche, appelée *Méthode Évolutive*, est plus récente puisqu'elle date du début des années 90. Elle s'est particulièrement faite connaître par l'intermédiaire des algorithmes génétiques dont l'origine remonte aux travaux de Holland [Holland, 1975].

Le but de ce chapitre n'est pas de donner une description précise de l'ensemble des méta-heuristiques couramment utilisées en optimisation. Nous visons plutôt à guider toute personne désirant adapter une méta-heuristique à un problème particulier d'optimisation. Pour ce faire, nous décrivons tout d'abord brièvement, dans la prochaine section, les 2 approches susmentionnées. Dans la Section 3, nous commençons par indiquer les concepts les plus importants qu'il s'agit d'adapter adéquatement pour pouvoir résoudre un problème d'optimisation particulier à l'aide d'une méta-heuristique. Ce travail d'adaptation d'une approche générale à un problème particulier peut être réalisé de diverses manières. C'est pourquoi, dans la suite de la Section 3, nous présentons diverses stratégies d'adaptation, en mettant en relief les avantages et les inconvénients de chacune d'elles. La Section 4 contient un résumé des conseils délivrés dans ce chapitre.

2. Description générale des principales méta-heursitiques

Soit S un ensemble de solutions à un problème d'optimisation, et soit f une fonction qui associe une valeur f(s) à chaque solution $s \in S$. L'ensemble S est défini de telle sorte qu'une solution n'en fait partie que si elle respecte l'ensemble des contraintes du problème considéré. L'objectif d'un algorithme d'optimisation est de déterminer une solution dans S qui minimise la fonction f. En d'autres termes, il s'agit de déterminer une solution $s^* \in S$ telle que

$$f(s^*) = \min_{s \in S} f(s)$$

Le but de cette section n'est pas de donner une description précise et détaillée des principales techniques de Recherche Locale et des Méthodes Évolutives. Nous nous en tiendrons à l'essentiel, et nous mettrons en relief quelques difficultés que l'on peut rencontrer lors de leur utilisation. Le lecteur désirant obtenir plus de détails sur ces techniques peut consulter le célèbre ouvrage de Reeves [Reeves, 1995] dans lequel les principales méta-heuristiques sont présentées de manière très didactique, avec des exemples d'applications tel que l'ordonnancement de la production. Comme le domaine des méta-heuristiques évolue très vite, le lecteur préférera peut-être consulter des ouvrages plus récents, tel que celui de Dréo et al. qui est écrit en français [Dréo, 2003].

2.1 La Recherche Locale

Les techniques de Recherche Locale associent un *voisinage* N(s) à chaque solution $s \in S$, et construisent une séquence s_0, s_1, … de solutions où s_0 est une solution initiale généralement produite par un algorithme constructif simple, et où chaque s_i fait partie du voisinage $N(s_{i-1})$ de s_{i-1}. Toute solution dans le voisinage N(s) de s est dite *voisine* à s. Divers critères d'arrêt peuvent être considérés, tel qu'un temps limite imposé ou un écart par rapport à une borne inférieure jugé suffisant. Une description générale des techniques de Recherche Locale est donnée dans la figure 1.

1) Générer une solution initiale s et poser $s^*=s$
2) Tant qu'aucun critère d'arrêt n'est satisfait faire
    a. Générer une solution s' dans le voisinage N(s) de s.
    b. Poser s=s' et mettre à jour s^* si $f(s)<f(s^*)$.

Figure 1 : description générale d'un algorithme de Recherche Locale

La méthode de Recherche Locale la plus simple est probablement la méthode de descente qui génère, à chaque itération, la solution voisine s' qui minimise f dans N(s). Ainsi donc, la solution s' générée à l'étape 2.a est telle que

$$f(s') = \min_{s'' \in N(s)} f(s'')$$

L'algorithme s'arrête dès qu'il n'y a pas d'amélioration de la fonction f entre deux itération successives (c'est-à-dire dès que la solution voisine générée en 2.a n'est pas meilleure que la solution courante s). Le principal défaut de la méthode de descente est son arrêt au premier minimum local rencontré. Divers algorithmes ont été proposés, principalement dès le milieu des années 80, pour contourner cet obstacle que représentent les optima locaux. La technique du *Recuit Simulé* [Kirkpatrick et al., 1983], par exemple, choisit s' au hasard dans N(s). Cette solution voisine à s est acceptée comme nouvelle solution courante si elle est meilleure que s. Dans le cas contraire, s' n'est acceptée comme nouvelle solution courante qu'avec une certaine probabilité. Si s' est refusée, une nouvelle solution est pigée au hasard dans N(s) et ainsi de suite. Le processus s'arrête lorsque la solution s n'a plus été modifiée depuis un certain temps. La probabilité d'accepter une solution voisine s' avec $f(s')>f(s)$ comme nouvelle solution courante diminue avec le temps.

La Recherche Taboue [Glover, 1986] est similaire à la méthode de descente, en ce sens que la solution s' choisie dans N(s) est la meilleure possible. Cependant, l'algorithme ne s'arrête pas si $f(s')\geq f(s)$, le principe sous-jacent étant qu'il se peut que l'on doive détériorer la solution courante pour pouvoir atteindre un optimum global. Ce fait est illustré dans la Figure 2.

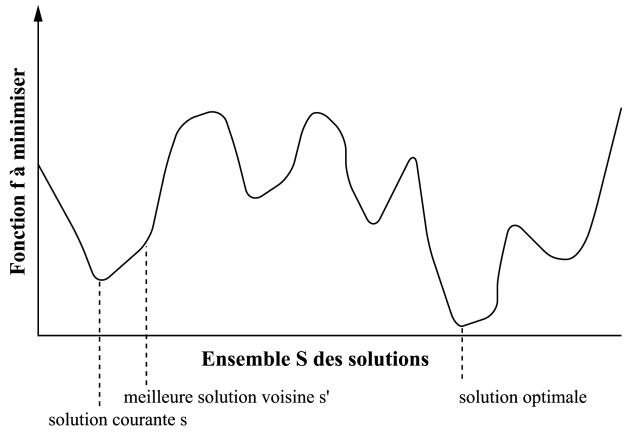

Figure 2 : La recherche de la solution optimale peut nécessiter une détérioration de la solution courante

Lorsque la meilleure solution $s'\in N(s)$ s est telle que $f(s')>f(s)$, il est possible que s soit la meilleure solution dans N(s'), et l'algorithme risque alors de cycler autour des solutions s et s'. Pour contourner cette difficulté, la Recherche Taboue considère une liste de solutions dites 'taboues', en ce sens qu'il est interdit de se rendre vers une solution faisant partie de cette liste pendant un certain nombre d'itérations. Ainsi, lorsque la Recherche Taboue se déplace d'une solution s vers une solution voisine s', la solution s est introduite dans la liste taboue, empêchant le retour immédiat vers s.

2.2 Les Méthodes Évolutives

Depuis le début des années 90, une autre famille d'heuristiques est devenue très populaire : les Méthodes Évolutives [Hertz et Kobler, 2000]. Contrairement à la Recherche Locale qui tente d'améliorer itérativement une solution courante, les Méthodes Évolutives travaillent sur une *population* de solutions en appliquant un processus cyclique composé d'une phase de *coopération* et d'une phase *d'adaptation individuelle* qui se succèdent à tour de rôle. Dans la phase de coopération, les solutions de la population courante sont comparées entre elles, puis combinées, dans le but de produire de nouvelles solutions qui héritent des bons aspects de chaque membre de la population. Dans la phase d'adaptation individuelle, chaque solution dans la population peut évoluer de manière indépendante. On peut utiliser le même type de critère d'arrêt que dans la Recherche Locale, ou alors on peut décider de stopper une Méthode Évolutive dès que les solutions dans la population sont jugées trop similaires. Une description générale des Méthodes Évolutives est donnée dans la figure 3.

1) générer un ensemble P de solutions dans S
2) Tant qu'aucun critère d'arrêt n'est satisfait faire
    a. Appliquer un opérateur d'adaptation individuelle à chaque solution dans P
    b. Appliquer un opérateur de coopération afin d'effectuer des échanges d'information parmi les solutions de P et ainsi mettre à jour P

Figure 3 : description générale d'une Méthode Évolutive

L'algorithme génétique [Goldberg, 1989] est un exemple particulier de Méthode Évolutive pour lequel l'adaptation individuelle du point 2.a est réalisée à l'aide de l'opérateur dit de *mutation*, alors que l'échange d'information du point 2.b est gouverné par un opérateur de *reproduction* et un opérateur de *combinaison* (ou *crossover*). Il est de plus en plus courant de nos jours d'appliquer brièvement une Recherche Locale à chaque membre de la population à titre d'adaptation individuelle, et on parle alors généralement d'*algorithme hybride*.

3. Principaux ingrédients des méta-heuristiques

Pour adapter une méta-heuristique à un problème d'optimisation particulier, il est nécessaire de modéliser adéquatement un certain nombre d'ingrédients. Ces principaux ingrédients sont :

- L'ensemble S des solutions; cet ensemble définit le domaine dans lequel la recherche d'un optimum va s'effectuer.

- La fonction objectif f à minimiser. Cette fonction induit une topologie sur l'espace de recherche, avec des montagnes (mauvaises solutions), des vallées (régions autour d'un minimum local), etc.
- Le voisinage d'une solution (dans le cas d'une Recherche Locale) ou l'opérateur de combinaison (dans le cas des Méthodes Évolutives). Ce dernier concept définit les déplacements possibles dans l'espace de recherche. Dans le cas d'une Recherche Locale, le domaine S est exploré en parcourant un chemin qui se dirige à chaque pas d'une solution vers une solution voisine. Dans le cas des Méthodes Évolutives, l'opérateur de combinaison permet de créer de nouvelles solutions dans S à partir des solutions courantes présentes dans la population.

Nous indiquons ci-dessous quelques points importants à prendre en compte lors de l'adaptation d'un de ces concepts à un problème particulier. Pour illustrer nos propos, nous traiterons principalement les problèmes d'ordonnancement d'atelier de type *flow shop* et *job shop* avec pour objectif la minimisation du *makespan*, c'est-à-dire la durée totale d'exécution des tâches.

3.1 L'espace des solutions

Tel qu'indiqué ci-dessus, l'espace S des solutions représente le domaine dans lequel la recherche d'un optimum va s'effectuer. Le premier réflexe de toute personne désireuse d'adapter une méta-heuristique à un problème d'optimisation particulier, est de définir S comme l'ensemble des solutions *réalisables*, c'est-à-dire celles qui satisfont toutes les contraintes du problème considéré. Cependant, un tel choix n'est pas forcément le plus judicieux. En effet, il existe de nombreux problèmes pour lesquels la détermination d'une solution réalisable est un problème difficile (les mathématiciens utilisent le terme NP-dur), le meilleur exemple étant la confection d'horaires de personnel. De plus, même dans le cas où il est facile de déterminer des solutions réalisables, il peut être difficile de définir un voisinage qui ne modifie que légèrement une solution courante tout en conservant la réalisabilité des solutions. Ainsi, il peut parfois être plus adéquat de définir S comme un ensemble de solutions ne satisfaisant pas forcément toutes les contraintes du problème considéré. La fonction a optimiser devra alors bien sûr tenir compte de la violation éventuelle de certaines contraintes.

Pour illustrer ces propos, considérons le problème du job shop. Il est possible de coder une solution à l'aide de m permutations, où m correspond au nombre de machines, et la $i^{ème}$ permutation donne l'ordre des tâches devant être exécutées sur la $i^{ème}$ machine. Par exemple, supposons que 3 pièces P_1, P_2, P_3 doivent être usinées sur les machines M_1, M_2 et M_3. Supposons que la gamme opératoire de P_1 est (M_1,M_2,M_3), que celle de P_2 est (M_1,M_3,M_2), et que celle de P_3 est (M_2,M_1,M_3). Supposons finalement que l'usinage de chaque pièce sur chaque machine dure une unité de temps. La solution représentée ci-dessous dans la Figure 4 peut être codée grâce au triplet (3,1,2), (3,1,2), (3,2,1) de

permutations. Nous remarquons cependant qu'un triplet de permutation ne définit pas nécessairement une solution réalisable. Le triplet (3,1,2), (1,3,2), (3,2,1) n'est par exemple pas réalisable. En effet, celle solution indique que P_3 doit précéder P_1 sur M_1, alors que P_1 doit précéder P_3 sur M_2 . Comme P_3 passe sur M_2 avant de passer sur M_1, on en déduit que P_1 doit passer sur M_2 avant de passer sur M_1, ce qui contredit sa gamme opératoire. En définissant l'espace S de recherche, il faut donc décider si celui-ci contient toutes les permutations possibles des tâches sur les machines, ou seulement celles qui induisent une solution réalisable.

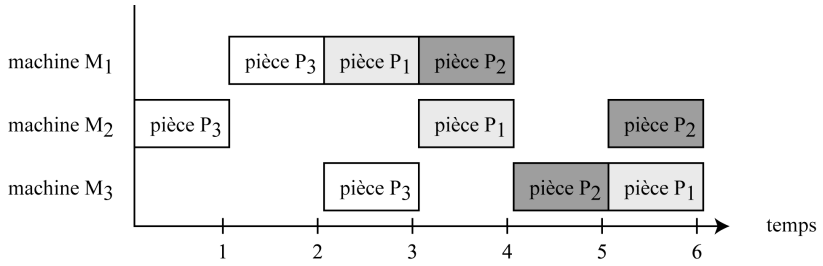

Figure 4 : Une solution à un problème de jobshop

3.2 La fonction à optimiser

La fonction f à optimiser induit une topologie sur l'espace de recherche. La recherche d'un optimum correspond donc à la recherche de la vallée la plus profonde dans S. Même dans le cas où S ne contient que des solutions réalisables, le meilleur choix pour f n'est pas nécessairement la fonction objectif du problème considéré. En effet, si la topologie manque de relief, il peut être difficile de guider la recherche vers un optimum global. À titre d'exemple, pour un problème de job shop dont l'objectif est de minimiser le makespan, il se peut que plusieurs solutions aient le même makespan alors qu'une de ces solutions est peut-être préférable aux autres car des machines terminent peut-être leur usinage plus tôt. Considérons par exemple le problème de job shop décrit dans la section précédente. La solution de la figure 5 a le même makespan que celle de la figure 4.

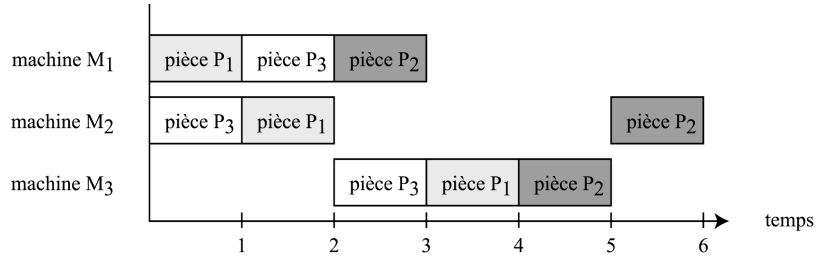

Figure 5 : Une solution à un problème de jobshop

Cependant, on pourrait estimer que celle de la figure 5 est meilleure car seule une machine termine au temps 6 alors que dans la solution de la figure 4, deux machines

atteignent le makespan. Une fonction qui donnerait plus de relief à la topologie du problème de job shop serait par exemple la somme des carrés des temps de complétion de chaque machine. La solution de la figure 4 aurait ainsi une valeur de $4^2+6^2+6^2=88$ alors que la solution de la figure 5 aurait une valeur de $3^2+5^2+6^2=70$ et serait donc jugée meilleure que celle de la figure 4.

Nous avons indiqué dans la section 3.1 qu'il peut être judicieux d'inclure des solutions non réalisables dans l'espace de recherche S. Dans ce cas, la fonction f à optimiser doit non seulement mesurer la valeur des solutions admissibles, mais également pénaliser les solutions non réalisables. L'ordre de grandeur de la pénalité à accorder à une violation de contrainte n'est pas simple à déterminer. Une trop petite pénalité ne guide pas suffisamment la recherche vers des solutions réalisables, alors que de trop grandes pénalités induisent des montagnes infranchissables dans l'espace de recherche, ce qui correspond à interdire les solutions non réalisables. La technique la plus courante consiste à choisir une pénalité initiale quelconque, à augmenter celle-ci lorsque l'algorithme n'explore pas assez de solutions réalisables, et à la diminuer lorsque la recherche reste confinée dans une région ne contenant que des solutions réalisables [Gendreau et al., 1994].

3.3 Stratégies d'exploration

La Recherche Locale explore l'espace des solutions en générant à chaque étape une solution s' voisine de la solution courante s, alors qu'une Méthode Évolutive explore cet espace en générant de nouvelles solutions sur la base de celles de la population courante. L'exploration de l'espace des solutions peut être réalisée selon les quatre stratégies suivantes.

Stratégie 1 : Génération de solutions réalisables.
Stratégie 2 : Génération répétée de solutions jusqu'à obtention d'une solution réalisable.
Stratégie 3 : Génération de solutions (non nécessairement réalisables), et pénalisation des violations de contraintes dans le cas d'une solution non réalisable.
Stratégie 4 : Génération de solutions (non nécessairement réalisables), et réparation de ces solutions en les rendant réalisables (si elles ne le sont pas déjà).

Chaque stratégie a ses avantages et ses inconvéniens. La première stratégie peut être complexe à mettre en place car il n'est pas toujours évident de définir des variations sur une ou des solutions qui garantissent de fournir des solutions réalisables. L'avantage cependant de cette première stratégie est qu'elle permet de se concentrer sur les solutions réalisables, sans avoir à se soucier de pénalités à accorder à des solutions non réalisables. La deuxième stratégie permet également de confiner la recherche dans l'espace des solutions réalisables. De plus, aucun opérateur spécialisé n'est nécessaire pour garantir la réalisabilité puisque les solutions non réalisables générées sont rejetées. Le défaut de cette deuxième stratégie est qu'il peut être nécessaire de générer de nombreuses solutions

avant d'avoir la chance de tomber sur une solution réalisable. La troisième stratégie est probablement la plus simple car elle permet de définir à peu près n'importe quel générateur de solutions puisque les violations de contraintes sont acceptées. Cependant, elle requiert l'ajout d'une composante dans la fonction objectif qui pénalise les violations de contrainte. La quatrième stratégie donne également beaucoup de liberté lors du choix d'un générateur de solutions, et la non réalisabilité d'une solution est cette fois-ci réparée à l'aide d'une procédure adéquate. La difficulté dans cette quatrième stratégie réside dans la phase de réparation. En effet, tout en réparant la solution générée, il faut s'assurer qu'elle ne s'écarte pas trop de la solution courante s (dans le cas d'une Recherche Locale) ou de la population courante (dans le cas d'une Méthode Évolutive), car sinon il devient difficile, voire impossible, de guider la recherche d'un optimum.

Pour illustrer ces différentes stratégies, nous allons considérer une Recherche Locale pour le problème de minimisation du makespan dans un problème de job shop. Un voisinage pour la première stratégie peut être défini comme suit. Soit P un chemin critique pour la solution courante s. Pour l'exemple de la figure 4, il existe 2 tels chemins :

- (3,2)→(3,1)→(1,1)→(2,1)→(2,3)→(1,3) et
- (3,2)→(3,1)→(1,1)→(2,1)→(2,3)→(2,2)

(où une paire (x,y) signifie que P_x est usinée sur M_y). Il a été démontré dans [Van Laarhoven et al., 1992] qu'en permutant deux pièces consécutives sur une même machine sur un chemin critique, on obtient toujours une solution réalisable. Dans notre exemple, on pourrait donc inverser P_3 et P_1 sur M_1, P_1 et P_2 sur M_1 ou P_2 et P_1 sur M_3. Les trois solutions voisines qu'on peut ainsi obtenir sont représentées dans la figure 6.

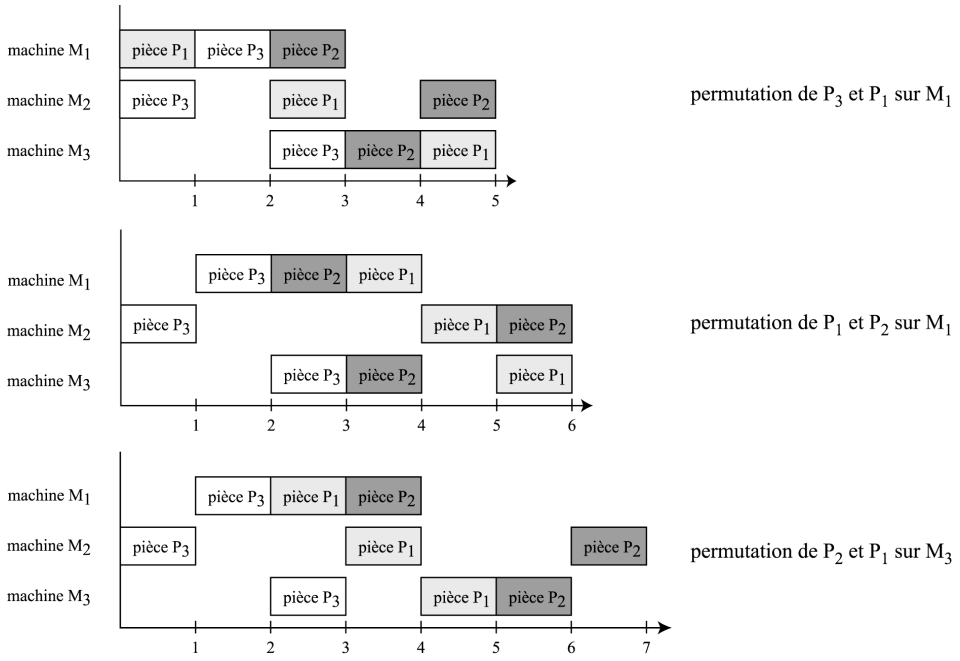

Figure 6 : 3 solutions voisines à la solution de la Figure 4

La deuxième stratégie pourrait consister à permuter deux pièces consécutives sur une machine, sans tenir compte des chemins critiques. Ainsi, par exemple, on pourrait modifier la solution de la Figure 4 en permutant les pièces P_1 et P_3 sur la machine M_2, et on obtiendrait une solution non réalisable puisqu'elle correspondrait au triplet (3,1,2), (1,3,2), (3,2,1) de permutations, et nous avons vu dans la Section 3.1 que celui-ci n'est pas réalisable. La deuxième stratégie pourrait donc consister à générer de tels déplacements jusqu'à l'obtention d'une solution réalisable.

La troisième stratégie accepte les violations de contraintes tout en les pénalisant. La solution (3,1,2), (1,3,2), (3,2,1) pourrait donc être acceptée comme voisine de la solution de la figure 4. Pour évaluer cette solution non réalisable, on pourrait par exemple itérativement supprimer une machine de la gamme opératoire d'une pièce, jusqu'à obtention d'une solution réalisable. La fonction objectif mesurerait alors le makespan du sous-problème ainsi obtenu et y ajouterait une pénalité proportionnelle aux nombres de tâches supprimées. Pour la solution non admissible (3,1,2), (1,3,2), (3,2,1) , on pourrait par exemple supprimer la machine M_1 de la gamme opératoire de P_3 (c'est-à-dire qu'on supprime l'usinage de la pièce P_3 sur M_1). On obtiendrait alors la solution admissible représentée dans la figure 7.

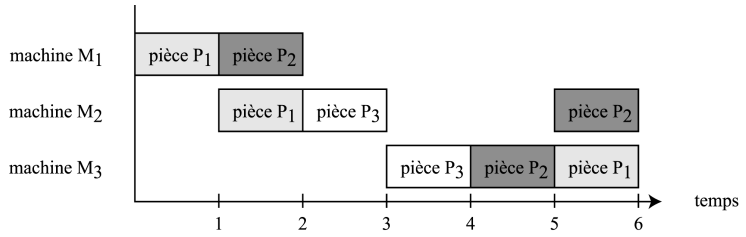

Figure 7 : Solution obtenue à partir de la figue4 en permutant P_1 et P_3 sur M_2 et en suppimant P_3 sur M_1.

La quatrième stratégie préférera réparer la solution (3,1,2), (1,3,2), (3,2,1) plutôt que la pénaliser. On peut réparer une solution non réalisable en considérant chaque permutation sur chaque machine comme des ordres de priorité, des souhaits à respecter si possible. On peut donc construire une solution du jobshop en faisant avancer le temps unité par unité, et à chaque fois qu'une machine est libre, on lui donne la tâche disponible la plus prioritaire à effectuer. (Une tâche est « disponible » lorsque celle qui la précédait dans la gamme opératoire de la pièce considérée est terminée). La solution réalisable qu'on obtiendrait donc avec les ordres (3,1,2), (1,3,2) et (3,2,1) est illustrée dans la Figure 7.

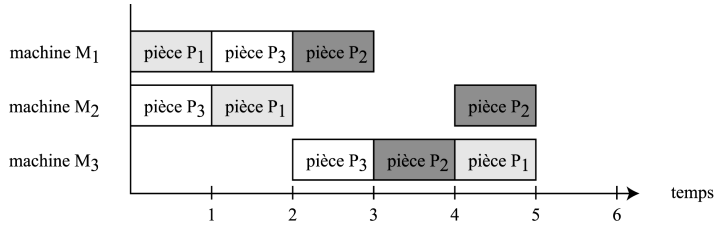

Figure 7 : Solution obtenue à partir de la figue 4 en permutant les pièces P_1 et P_3 sur M_2 et en la réparant

3.4 Voisinage d'une solution

Définissons le graphe orienté G dont les sommets sont les solutions dans S, et tel qu'il existe un arc d'une solution s vers une solution s' si et seulement si $s' \in N(s)$. La Recherche Locale peut être vue comme un cheminement dans G le long des arcs du graphe. Il est important de définir un voisinage N qui vérifie la propriété suivante :

$\forall s \in S$, il existe un chemin dans G reliant s à une solution optimale. (Propriété P)

En effet, si tel n'est pas le cas, il se peut qu'il soit totalement impossible pour la Recherche Locale d'atteindre une solution optimale à partir d'une solution initiale, en cheminant dans le graphe G. À titre d'exemple, supposons qu'on doive planifier n tâches $T_1,\ldots,T_n$ en aussi peu de temps que possible. Certaines paires de tâches ne peuvent pas être réalisées simultanément (par exemple parce qu'elles requièrent le même superviseur), et chaque tâche à une fenêtre de temps à l'intérieur de laquelle elle doit impérativement être réalisée. Pour résoudre ce problème, on pourrait imaginer une Recherche Locale dans laquelle S serait l'ensemble des planifications réalisables et où le voisinage d'une solution consisterait à modifier l'horaire d'exactement une tâche. Un tel choix ne serait pas judicieux car, comme le montre l'exemple suivant, le voisinage choisi ne définit pas forcément un graphe ayant la propriété P susmentionnée.

L'exemple comprend six tâches $T_1,\ldots,T_6$ ayant chacune une durée d'une heure. On suppose que la tâche T_i ne peut pas être réalisée en même temps que la tâche T_{i+1} ($i=1,\ldots,5$) et que la tâche T_1 ne peut pas être réalisée en même temps que T_6. De plus aucune tâche ne peut démarrer plus tôt que 8h00 et chaque tâche doit impérativement être terminée à 11h00. Un horaire possible consiste à planifier les tâches T_1 et T_4 à 8h00, les tâches T_2 et T_5 à 9h00, et les tâches T_3 et T_6 à 10h00. Cette planification, que nous noterons H respecte toutes les contraintes et correspond donc à une solution réalisable. Cependant, cette solution n'a pas de solution voisine puisque si on change l'horaire d'une tâche tout en respectant les fenêtres de temps, on viole nécessairement une contrainte de non simultanéité. Par exemple, la tâche T_1 ne peut pas débuter à 9h00 car elle serait réalisée en même temps que T_2, et elle ne peut pas débuter à 10h00 à cause de la tâche T_6. L'horaire H n'est cependant pas optimal car on aurait pu planifier les tâches T_1, T_3 et T_5 à 8h00 et les tâches T_2, T_4 et T_6 à 9h00 pour finir le tout une heure plus tôt. Le graphe orienté G induit par l'espace S des solutions et par le voisinage N ne contient donc aucun chemin de H vers une solution optimale puisque H n'a pas de successeur. Le graphe G ne satisfait donc pas la propriété P.

Le voisinage le plus simple qu'on puisse imaginer et qui induit nécessairement un graphe G vérifiant la propriété P, consiste à définir $N(s)=S$ pour toute solution $s \in S$. Une telle modélisation signifie que la Recherche Locale aura la liberté de construire une séquence de solutions dans S, sans que ces solutions soient nécessairement corrélées entre elles. Une telle stratégie n'est pas recommandée car il est alors difficile de guider la recherche

d'un optimum. De plus, la détermination de la meilleure solution voisine $s' \in N(s)$ devient alors un problème aussi complexe que le problème d'optimisation original. Idéalement, le voisinage N(s) d'une solution s ne devrait contenir que des solutions similaires à s, en ce sens que leur 'aspect' et leur 'valeur' mesurée par la fonction f ne diffèrent pas beaucoup de ceux de s.

Illustrons ces propos en considérant un problème de flow shop de permutation avec temps de réglage (supposés symétriques). On peut définir l'espace S comme l'ensemble des permutations des pièces $P_1,\ldots P_n$ à usiner, et le voisinage N(s) d'une solution s comme l'ensemble des solutions obtenues en permutant 2 pièces dans s. On peut cependant préférer restreindre N(s) à l'ensemble des solutions obtenues en permutant deux pièces consécutives dans s. Cette deuxième possibilité est peut-être plus appropriée pour le problème considéré car elle induit une plus petite variation dans la fonction objectif. En effet, notons $\pi(i)$ l'indice de la pièce en $i^{ème}$ position et soit $s=(P_{\pi(1)}, \ldots,P_{\pi(n)})$ une solution au problème. Si on permute $P_{\pi(i)}$ avec $P_{\pi(i+1)}$, on supprime les temps de réglage entre $P_{\pi(i-1)}$ et $P_{\pi(i)}$ et entre $P_{\pi(i+1)}$ et $P_{\pi(i+2)}$ alors qu'on en rajoute un entre $P_{\pi(i-1)}$ et $P_{\pi(i+1)}$ et entre $P_{\pi(i)}$ et $P_{\pi(i+2)}$, ce qui donne un total de 4 variations. Par contre, si on permute $P_{\pi(i)}$ avec $P_{\pi(j)}$, $j>i+1$, alors on supprime les temps de réglage entre $P_{\pi(i-1)}$ et $P_{\pi(i)}$, entre $P_{\pi(i)}$ et $P_{\pi(i+1)}$, entre $P_{\pi(j-1)}$ et $P_{\pi(j)}$, et entre $P_{\pi(j)}$ et $P_{\pi(j+1)}$, alors qu'on en rajoute entre $P_{\pi(i-1)}$ et $P_{\pi(j)}$, entre $P_{\pi(j)}$ et $P_{\pi(i+1)}$, entre $P_{\pi(j-1)}$ et $P_{\pi(i)}$, et entre $P_{\pi(i)}$ et $P_{\pi(j+1)}$, pour un total de 8 variations. Si l'on veut se mouvoir dans l'espace de recherche en évitant les grandes variations de la fonction objectif, la permutation de pièces consécutives semble donc plus adéquate.

3.5. L'opérateur de combinaison

Dans le début des années 90, il était très populaire d'utiliser l'opérateur de combinaison de type *uniforme* dont le principe de base est de créer une solution dite *enfant* en copiant de manière équilibrée dans deux solutions appelées *parents*. Plus précisément, supposons que chaque solution soit codée à l'aide d'un vecteur à n composantes, et soient $(x_1,\ldots,x_n)$ et $(y_1,\ldots,y_n)$ deux solutions parents : une solution enfant $(z_1,\ldots,z_n)$ est créée en fixant $z_i=x_i$ ou y_i, chaque choix étant réalisé avec une probabilité égale à ½ .

L'opérateur de croisement de type uniforme ne peut pas être utilisé dans le cas où une solution est codée à l'aide d'une permutation puisque la solution enfant ne correspondrait plus alors nécessairement à une permutation. À titre d'illustrations, supposons que l'on doive permuter trois objets a, b et c. Une combinaison uniforme des solutions (a,b,c) et (b,c,a) pourrait induire la solution enfant (a,b,a) si les copies se font dans le premier parent pour les deux premières positions et dans le deuxième parent pour la troisième position. La solution (a,b,a) n'est pas réalisable puisque a apparaît deux fois alors que c est absent. Plusieurs opérateurs de combinaison ont été proposés dans le cas d'un codage à l'aide de permutations [Portmann et Vignier, 2001]. On peut par exemple copier un parent pour la moitié de ses positions, et remplir les positions manquantes avec les éléments manquants, mais en respectant l'ordre de ces éléments dans le deuxième parent.

Ainsi, par exemple, si on combine les parents (a,b,c,d) et (c,d,a,b), et si on copie le premier parent aux positions 1 et 4, on obtient d'abord la solution partielle (a,-,-,d). Comme c précède b dans le deuxième parent, on rajoute c avant b pour obtenir la solution enfant (a,c,b,d).

Il est de plus en plus reconnu que pour qu'une Méthode Évolutive soit efficace, l'opérateur de combinaison doit tenir compte de la structure particulière du problème traité. En d'autres termes, des opérateurs de combinaison spécialisés s'avèrent généralement plus performants que des opérateurs généraux du type 'uniform crossover'. L'idée maîtresse qui doit guider le choix d'un opérateur de combinaison est l'échange d'*information pertinente*. Si une solution s est jugée meilleure qu'une solution s', il faut tâcher de comprendre ce qui rend s meilleur que s', et c'est ce type d'information qui doit être transmis lors de la création d'une solution enfant.

Pour illustrer ces propos, considérons un problème de flow shop de permutation sur une machine, avec temps de réglage. Une solution s est meilleure qu'une solutions s' si les tâches de s sont positionnées de telle sorte qu'elles induisent un temps total de réglage moins important que dans s'. Lors de l'échange d'information, il faut donc essayer de conserver ces temps de réglage. Prenons par exemple les solutions (a,b,c,d,e) et (e,d,b,a,c), si on copie le premier parent dans la moitié de ses positions, par exemple aux positions 1, 3 et 4, on obtient la solution partielle (a,-,c,-,e) qui est complétée en (a,d,c,b,e). Cette solution enfant utilise des temps de réglage très différents de ceux des deux solutions parents et peut donc être de valeur très mauvaise. Une solution enfant telle que (b,a,c,d,e) combine les deux parents de manière plus adéquate puisque tous les temps de réglage de la solution enfant apparaissent dans au moins un des deux parents.

Si l'information pertinente à échanger semble être l'ordre relatif des éléments de la permutation, l'opérateur susmentionné n'est pas forcément le plus adéquat. En effet, si on croise les solutions (a,b,c,d,e) et (e,d,b,a,c), nous avons vu qu'on peut obtenir la solution enfant (a,d,c,b,e). Cette solution enfant a placé c avant b alors que les deux solutions parents ont préféré positionner b avant c. Pour éviter ce genre de situation, on peut préférer, par exemple, l'opérateur suivant qui se base sur une représentation matricielle des permutations. Plus précisément, à chaque solution s on associe une matrice $M=(m_{ij})$ telle que $m_{ij}=1$ si i précède j dans s, $m_{ij}=-1$ si j précède i dans s, et $m_{ij}=0$ si i=j. Notons M_1 et M_2 les matrices associées à deux solutions s_1 et s_2. En sommant M_1 avec M_2, et en divisant la matrice résultante par 2, on obtient une matrice M_3 qui contient des éléments non nuls à chaque endroit où s_1 et s_2 s'accordent sur l'ordre relatif des tâches. Pour que M_3 corresponde à une permutation, il faut encore remplir les cases hors diagonale qui contiennent la valeur 0. Pour ce faire, on peut choisir de compléter l'information manquante en la copiant dans M_1 ou dans M_2 de manière aléatoire. Par exemple, si $s_1=(6,1,3,4,2,5)$ et $s_2=(1,3,6,4,5,2)$, les matrices M_1, M_2 et M_3 sont représentées ci-dessous.

$$M_1 = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & -1 \\ -1 & 0 & -1 & -1 & 1 & -1 \\ -1 & 1 & 0 & 1 & 1 & -1 \\ -1 & 1 & -1 & 0 & 1 & -1 \\ -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad M_2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ -1 & 0 & -1 & -1 & -1 & -1 \\ -1 & 1 & 0 & 1 & 1 & 1 \\ -1 & 1 & -1 & 0 & 1 & -1 \\ -1 & 1 & -1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 1 & 1 & 0 \end{pmatrix} \quad M_3 = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ -1 & 0 & -1 & -1 & 0 & -1 \\ -1 & 1 & 0 & 1 & 1 & 0 \\ -1 & 1 & -1 & 0 & 1 & -1 \\ -1 & 0 & -1 & -1 & 0 & -1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

À ce stade, on a l'ordre partiel représenté dans la figure 7. Il faut encore définir les ordres relatifs entre 1 et 6, entre 3 et 6, et entre 2 et 5. Si on choisit de copier l'ordre entre 1 et 6 dans le premier parent, on positionnera donc 6 avant 1, et comme 1 précède 3, nous savons désormais que 6 doit précéder 3. En copiant l'ordre relatif entre 2 et 5 dans le deuxième parent, on obtient la solution enfant (6,1,3,4,5,2).

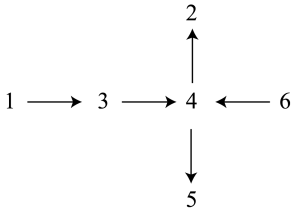

Figure 8 : Ordre partiel entre 6 tâches

4. Remarques finales

En résumé, pour adapter une méta-heuristique à un problème d'optimisation particulier, que ce soit dans le domaine de la gestion de la production, des ressources humaines, ou dans un tout autre domaine, il est important de bien adapter les principaux ingrédients de ces techniques qui sont l'espace de solution S, la fonction objectif f, et le voisinage N dans le cas d'une Recherche Locale ou l'opérateur de combinaison dans les Méthodes Évolutives. La fonction objectif donne du relief à l'espace dans lequel la recherche d'un optimum est effectuée alors que le voisinage ou l'opérateur de combinaison permettent de générer de nouvelles solutions à partir de la solution ou de la population de solutions courante. En définissant le voisinage N ou l'opérateur de combinaison C, if faut tâcher d'assurer une sorte de continuité dans la fonction objectif, en ce sens que les solutions générées par N ou C ne doivent pas trop différer de la solution courante ou des solutions parents quant à la valeur mesurée par f. Si tel n'est pas le cas, il devient très difficile de guider adéquatement la recherche d'un optimum. Dans le cas d'une Recherche Locale, il est également important de s'assurer qu'une solution optimale soit atteignable à partir de n'importe quelle solution initiale.

Nous avons également vu qu'il n'est pas nécessaire d'imposer la réalisabilité des solutions dans S. La fonction objectif doit alors pénaliser les solutions non réalisables. Il est cependant également possible d'imposer la réalisabilité des solutions dans S. Dans un tel cas, on a le choix entre diverses stratégies. On peut :

- définir un voisinage ou un opérateur de combinaison sophistiqué qui tienne compte des contraintes du problème considéré et assure donc la réalisabilité des solutions générées,
- utiliser un voisinage ou un opérateur de combinaison peu sophistiqué qui ne garantit pas la réalisabilité de la ou des solutions générées, mais qui peut produire une solution réalisable s'il est utilisé de manière répétée,
- utiliser un voisinage ou un opérateur de combinaison qui ne garantit pas la réalisabilité de la ou des solutions générées, et réparer les solutions générées dans le cas où elles ne sont pas réalisables.

Il existe donc de nombreuses manières d'adapter une méta-heuristique à un problème. Personne à ce jour n'a découvert une recette miracle. Des conseils sont prodigués dans [Zufferey, 2002; Hertz et Widmer, 2003]. Cependant, certains choix peuvent s'avérer plus judicieux pour certains problèmes que pour d'autres. Le but de ce chapitre était de mettre en relief les différents ingrédients auxquels un utilisateur d'une méta-heuristique doit prêter attention, en indiquant l'impact de chaque choix avec ses avantages et ses désavantages. Nous espérons que tous ces conseils s'avéreront utiles.

Références.

Dréo, J., Pétrowski, A., Siarry, P., et Taillard, É. *Métaheuristiques pour l'optimisation difficile*. Eyrolles, 2003.

Gendreau, M., Hertz, A., et Laporte, G. *A Tabu Search Heuristic for the Vehicle Routing Problem*. Management Science, tome 40, n° 10, pages 1276-1290, 1994.

Glover, F. *Future paths for integer programming and links to artificial intelligence*. Computers and Operations Research, tome 13, pages 533-549, 1986.

Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine learning*. Addison-Wesley, 1989.

Hertz, A., et Kobler, D. *A framework of the description of evolutionary algorithms*. European Journal of Operational Reseach, tome 126, n° 1, pages 1-12, 2000.

Hertz, A., et Widmer, M. *Guidelines for the use of meta-heuristics in combinatorial optimization*. European Journal of Operational Research, tome 151, pages 247-252, 2003.

Holland, J.H. *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, Mass., 1975.

Kirkpatrick, S., Gelatt, C., et Vecchi, M. *Optimisation by simulated annealing*. Science, tome 220, n° 4598, pages 671-680, 1983.

Portmann, M.C., et Vignier, A. *Algorithmes génétiques et ordonnancement*. Ordonnancement de la Production, édité par Lopez, P., et Roubellat, F., pages 95-130, Hermes Science, Paris 2001

Reeves, C. *Modern Heuristic Techniques for Combinatorial Problems*. Advances topics in computer science. Mc Graw0Hill, 1995.

Van Laarhoven, P.J.M., Aarts, E.H.L., et Lenstra, J.K. *Job shop scheduling by simulated annealing*. Operations Research, tome 40, pages 113-125, 1992.

Zufferey, N. *Heuristiques pour les problèmes de coloration des sommets d'un graphe et d'affectation de fréquences avec polarités*. Thése de doctorat n° 2668, École Polytechnique Fédérale de Lausanne, Suisse, pages141-146, 2002.