Conception d'une metaheuristique et application au problème d'ordonnancement de projet à moyens limités

Mireille Palpant
Sous la direction de Christian Artigues et Philippe Michelon
Laboratoire d'Informatique d'Avignon

15 juin 2001

Abstract

Nous présentons une métaheuristique pour la résolution des problèmes d'optimisation combinatoire NP-difficiles que nous appliquons, dans un but d'illustration, au problème d'ordonnancement de projet à moyens limités. Ce problème consiste à planifier dans le temps la réalisation d'un ensemble de tâches régies par des contraintes temporelles (contraintes de précédence entre tâches, tâches non préemptives) et des contraintes de ressources (durant toute son exécution, une tâche requiert une quantité constante d'unités de ressources, chaque ressource étant limitée par une capacité fixe au cours du temps). L'objectif est alors de minimiser la durée totale d'exécution de l'ensemble de ces tâches.

La méthode proposée ici fait cohabiter résolution exacte et recherche locale tout en intégrant des procédés de diversification et d'intensification dans la recherche des solutions. Nous obtenons des résultats très encourageants sur un ensemble de problèmes issus de la littérature.

1 Introduction

Résoudre exactement des problèmes d'optimisation combinatoire NP-difficiles est très ardu. En effet, pour de tels problèmes, les méthodes exactes requièrent un effort calculatoire qui croît exponentiellement avec la taille des instances du problème (explosion combinatoire) et, rapidement, les heuristiques ou metaheuristiques deviennent l'unique moyen d'obtenir une bonne solution en un temps raisonnable. Ces heuristiques sont principalement basées sur des méthodes de recherche locale (méthodes de descente, tabou [6], recuit simulé [10]) ou des méthodes évolutives (algorithmes génétiques [8], colonies de fourmis [4], Scatter Search [5]).

Cependant, même pour les problèmes d'optimisation combinatoire les plus difficiles, les méthodes exactes demeurent très efficaces pour de petites tailles d'instances, la solution optimale étant trouvée et prouvée en un temps très court. Cette simple constatation constitue le point de départ d'une approche hybride, assez originale dans le contexte des problèmes d'optimisation discrète, visant à combiner résolution exacte et heuristique au sein une même méthode. Nous pouvons par exemple citer l'algorithme Forget And Extend [3] pour les problèmes à ordonnancement de type "job-shop" ou encore la méthode Mimausa [16] appliquée au problème quadratique en variables binaires.

L'objet de ce mémoire est précisément de présenter une metaheuristique basée sur cette approche, résolvant optimalement au cours d'itérations successives un certain nombre de sous-problèmes du problème initial.

Nous commencerons dans la section suivante par présenter de façon générale la méthode,

puis nous nous attarderons sur le point le plus critique qui est : "comment sélectionner les sous-problèmes successifs à optimiser ?". Différentes stratégies seront présentées dans cette optique. Dans le but d'illustrer la méthode et de présenter des résultats expérimentaux, nous présenterons dans la section 3 une application au problème d'ordonnancement de projet à moyens limités — ou RCPSP pour Resource Constrained Project Scheduling Problem —. Après une brève description du RCPSP, nous poursuivrons avec des caractéristiques avancées de la méthode, liées à ce problème particulier. Enfin, nous conclurons par des résultats expérimentaux et quelques remarques.

2 Une description générale de la méthode

2.1 Principe

Notre metaheuristique se propose de combiner résolution exacte et recherche locale afin de résoudre un problème d'optimisation combinatoire P, tel que :

P : $\min_{X \in \mathcal{X}} f(X)$, avec $X = (x_1, \ldots, x_n)$, n étant la taille du problème.

Elle se propose pour cela de combiner des procédures d'*intensification* et de *diversification* [6] dans la recherche des solutions. L'intensification consiste à "creuser" le sous-espace des solutions admissibles à partir d'une solution réalisable. La diversification quant à elle réoriente la recherche vers un nouveau sous-espace, soit en perturbant la solution courante, soit en déterminant un nouveau point de départ.

La méthode développée est composée de deux phases :

- une phase d'initialisation, au cours de laquelle nous recherchons une solution initiale réalisable X_0 à l'aide d'une heuristique.
- une phase itérative, au cours de laquelle des sous-problèmes successifs du problème initial sont résolus exactement. Cette phase s'achève lorsqu'un nombre maximal d'itérations MAX est atteint, lorsqu'un sous-problème de taille n a été résolu exactement (en effet, dans ce cas le problème initial est résolu exactement) ou lorsque la valeur de la meilleure solution connue $f(X^*)$ atteint une borne inférieure BI (car P est un problème de minimisation).

Au cours d'une itération i, nous cherchons à résoudre optimalement un sous-problème $\tilde{P}_i$ du problème initial P. Plus précisément, l'ensemble des variables est partitionné en deux sous-ensembles de p et $n-p$ éléments respectivement. Le premier sous-ensemble contient les variables qui sont sélectionnées pour appartenir au sous-problème : ce sont les variables dites "libres". Les autres variables ("fixes") étant fixées à leur valeur courante, le sous-problème induit par les variables "libres" est alors créé puis optimisé en un temps maximum t_{opt} par une méthode exacte. La solution courante X_i est ensuite reconstruite à partir de la solution précédente X_{i-1} et du résultat de cette optimisation $\tilde{X}_i$. Cette opération peut consister en la simple mise à jour des valeurs des variables qui ont été modifiées par la méthode exacte ou bien demander un traitement plus complexe. Une fois cette opération réalisée, on met éventuellement à jour la valeur de la meilleure solution connue X^*. Si l'on parvient à MAX_ITER_SANS_AMEL itérations sans amélioration de la solution courante, celle-ci est alors perturbée afin de diversifier la recherche. Finalement, la valeur de p est ajustée en fonction du temps dispensé par la méthode exacte pour la résolution du sous-problème. On passe ensuite à l'itération suivante si aucun critère d'arrêt n'a été satisfait.

Nous pouvons remarquer que contrairement aux méthodes de recherche locale classiques, la solution ne change pas systématiquement à chaque itération. En effet, si les valeurs courantes

des variables libérées correspondent déjà à la solution optimale du sous-problème, la solution courante demeure totalement inchangée.

Nous pouvons raisonnablement espérer que l'optimisation successive de différents sous-problèmes du problème initial, améliore progressivement la qualité de la solution globale et que la solution finale soit de très bonne qualité. De plus, la méthode perturbant ponctuellement la solution courante, nous espérons visiter plusieurs minima locaux dont l'un soit l'optimum.

L'algorithme général de la méthode peut être présenté comme suit :

```
générer une solution initiale réalisable X_0 à l'aide d'une heuristique ;
X* : = X_0 ;
nb_iter_sans_amel : = 0 ;
initialiser t_opt ;
i : = 1 ;
REPETER
    sélectionner p variables ;
    construire le problème réduit P̃_i et calculer sa solution optimale X̃_i ;
    SI X̃_i obtenue avant t_opt ALORS
        SI p = n ALORS
            retourner X̃_i ;
        FIN SI
        construire une nouvelle solution S_i à partir de X̃_i et X_{i-1} ;
        SI f(X_i) < f(X*) ALORS
            X* : = X_i ;
            nb_iter_sans_amel : = 0 ;
        SINON
            nb_iter_sans_amel : = nb_iter_sans_amel + 1 ;
        FIN SI
    SINON
        nb_iter_sans_amel : = nb_iter_sans_amel + 1 ;
    FIN SI
    SI nb_iter_sans_amel = MAX_ITER_SANS_AMEL ALORS
        perturber X_i ;
        nb_iter_sans_amel : = 0 ;
    FIN SI
    ajuster la valeur de p en fonction du temps pris par la méthode exacte ;
    i : = i + 1 ;
JUSQU'A i = MAX OU f(X*) = BI ;
retourner X* ;
```

Plusieurs paramètres entrent en jeu dans cette méthode :

- la taille p du sous-problème : elle ne doit pas être trop élevée afin que le problème réduit soit résolu dans un temps relativement court, ni trop petite afin qu'une sous-partie substantielle du problème initial soit tout de même résolue. Cependant, la détermination de cette valeur n'est pas chose aisée car le temps requis par la méthode exacte dépend non seulement de la taille du sous-problème mais également des instances et de la valeur de la borne supérieure. Nous nous proposons d'auto-ajuster la valeur de cette variable en fonction des temps de calcul dispensés par la méthode exacte pour la résolution des sous-problèmes.

- le paramètre MAX_ITER_SANS_AMEL : il détermine au bout de combien d'ité- rations sans amélioration de la solution courante la solution doit être perturbée. Sa valeur doit être justement dosée afin de permettre la combinaison harmonieuse des procédures d'intensification et de diversification de l'heuristique.
- le paramètre t_{opt} : son bon réglage est essentiel à l'obtention d'une bonne solution finale en même temps qu'un temps d'exécution acceptable. Ce paramètre est étroitement lié à la taille du sous-problème.
- le nombre maximum d'itérations MAX.
- la sélection des variables libérées : nous donnons plusieurs stratégies dans le paragraphe suivant.

2.2 Choix du sous-problème

Le point crucial de la méthode réside dans le choix des variables à passer au sous-problème. Nous proposons ici les différentes façons que nous avons envisagées pour les choisir :

- "aléatoire" : toutes les variables sont choisies aléatoirement.
- "contraint" : on choisit une variable de départ et on constitue ensuite un "bloc" de variables de telle sorte que chaque variable soit liée à une autre par une contrainte.
- "mixte" : certaines variables sont choisies aléatoirement, d'autres suivant diverses contraintes qui les lient aux autres variables précédemment choisies. Le critère de contrainte varie bien sûr en fonction des applications.

En ce qui concerne les méthodes de tirage aléatoire et mixte, nous affectons à chaque variable une valeur de seuil, comprise entre 0 et 1, qui est fonction des caractéristiques de la variable. Cette valeur permettra de retenir ou non la variable lors de la génération du sous-problème. Ainsi, lorsqu'une variable est considérée pour appartenir au sous-problème, elle est effectivement retenue si un nombre tiré de façon aléatoire (compris entre 0 et 1) est inférieur ou égal à son seuil. Plusieurs possibilités de seuil sont envisageables :

- Seuil 1 : $s(x) = 1$ si la variable x présente une certaine caractéristique
  C sinon, $0 \leq C \leq 1$ étant à déterminer

  Cette premier seuil permet de privilégier une classe de variables présentant une caractéristique donnée.
  Exemple : $s(x) = 1$ si la variable x est présente dans toutes les contraintes actives, 0 sinon.

- Seuil 2 : $s(x) = 1 - C * ratio(x)$, $0 \leq C \leq 1$ étant à déterminer.

  Ce deuxième seuil permet d'affecter à chaque variable une valeur qui sera fonction d'une caractéristique pondérée donnée.
  Exemple : $ratio(x)$ est le nombre de contraintes actives impliquant la variable x sur le nombre total de contraintes.

3 Application au RCPSP

Nous allons dans un premier temps présenter brièvement le problème du RCPSP [14] ainsi que diverses heuristiques de la littérature pour sa résolution. Nous nous intéresserons ensuite à la façon avec laquelle nous avons appliqué notre metaheuristique à ce problème particulier. Enfin, nous présentons nos résultats sur des problèmes tirés de la littérature.

3.1 Présentation du problème

Le RCPSP est un problème d'optimisation NP-difficile en raison de sa nature fortement combinatoire. Le cas étudié ici est celui de la minimisation de la durée totale du projet.

Le RCPSP consiste en l'ordonnancement d'un ensemble donné de tâches sur une ou plusieurs ressources dont la capacité est fixée et connue. Les tâches sont liées entre elles par des contraintes de précédence, et chaque tâche nécessite une quantité donnée de chacune des ressources tout au long de son exécution. Une instance de RCPSP peut se formuler comme suit :

Un projet est la donnée d'un ensemble $A = \{1, \ldots, n\}$ de tâches et d'un ensemble $\mathcal{R} = \{1, \ldots, m\}$ de ressources. Une quantité constante R_k d'unités de ressource $k \in \mathcal{R}$ est disponible à chaque instant. Toute tâche $i \in A$ doit être exécutée, une et une seule fois et sans interruption, en p_i unités de temps. Durant ce temps, i monopolise alors une quantité positive ou nulle r_{ik} d'unités de chaque ressource k. Enfin, une relation de précédence entre les tâches est modelisée par un graphe orienté acyclique (A, E).

Cette instance peut être représentée par un graphe valué G, le graphe *potentiel-tâches*, dont l'ensemble V des sommets est l'ensemble A augmenté de deux tâches fictives 0 et $n+1$ de durées et de consommations nulles. L'ensemble des arcs du graphe est $E \cup \{(0, i), (i, n+1) \mid i \in A\}$ et tout arc (i, j) est valué par p_i.

Le problème d'ordonnancement consiste alors à caracteriser le n-uplet des dates de début d'exécution des tâches $\mathcal{S} = (S_1, S_2, \ldots, S_n)$ de sorte que :

- à tout moment et pour toute ressource k, la quantité d'unités de ressource k utilisée par toute les tâches en cours d'exécution est inférieure ou égale à la capacité de k ;
- pour tout arc $(i, j) \in E$, l'exécution de la tâche j ne peut commencer qu'après la complétion de i, à savoir $S_j \geq S_i + p_i$;
- la date d'achèvement $C_{\max} = \max\{S_i + p_i \mid i \in A\}$ du projet est minimale.

Le problème du RCPSP dans le cadre de la minimisation de la durée totale du projet peut être modélisé de la façon suivante :

$$\min C_{max} \qquad (1)$$

$$\text{s-à :} \quad C_{max} \geq S_i + p_i \quad \forall i \in A \qquad (2)$$

$$S_j \geq S_i + p_i \quad \forall (i,j) \in E \qquad (3)$$

$$\sum_{i \in \mathcal{A}(t)} r_{ik} \leq R_k \quad \forall k \in \mathcal{R}, \forall t \qquad (4)$$

où $\mathcal{A}(t)$ est l'ensemble des activités en cours à l'instant t : $S_i \leq t < S_i + p_i$

Nous présentons dans la figure 1 un exemple de RCPSP à 7 tâches et 2 ressources (de capacités respectives 3 et 1) ainsi qu'un ordonnancement réalisable de celui-ci. Nous avons représenté le graphe G valué par les durées des tâches. Les consommations de ressources par les tâches sont indiquées au-dessus ou en-dessous des tâches de la facon suivante : $\{r_{i1}, r_{i2}\}$, r_{ij} déterminant la quantité de ressource j consommée par la tâche i.

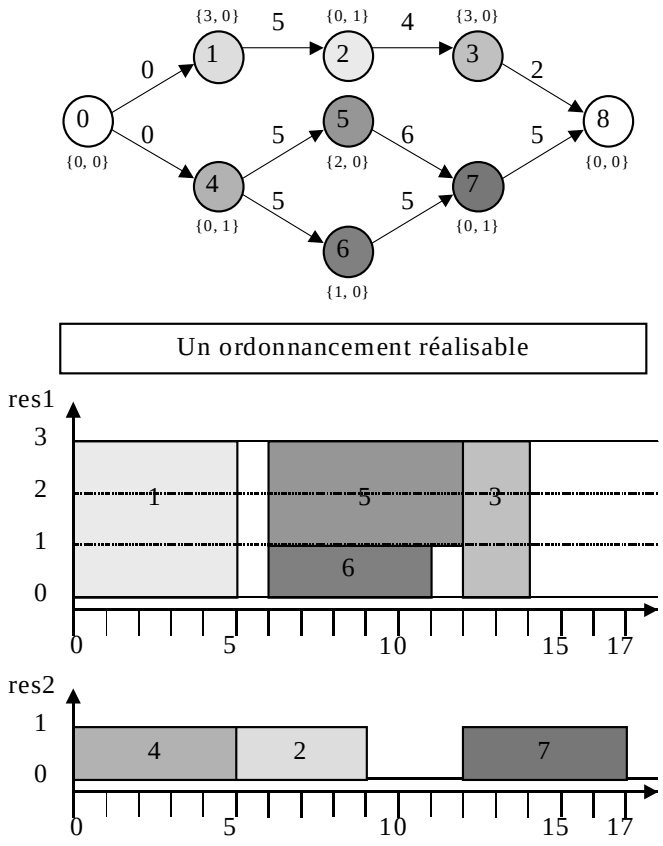

Figure 1: exemple de RCPSP

3.2 Les heuristiques pour le RCPSP dans la littérature

Diverses heuristiques ont été développées pour le problème du RPCSP : méthodes contructives, méthodes de recherche locale ou encore algorithmes génétiques. Un état de l'art complet peut être trouvé dans [13] et [14].

- Les méthodes constructives (à simple passe) basées sur des règles de priorité produisent des solutions admissibles de la forme $\mathcal{S} = (S_1, S_2, \ldots, S_n)$ en partant d'une solution initiale vide et en fixant à chaque étape $i = 1, \ldots, n$ une date de début définissant ainsi une solution partielle courante. Un indice de priorité est affecté à chaque tâche et, à chaque étape, la tâche de plus petit indice est sélectionnée. La décision qui est prise à une étape donnée n'est jamais remise en question. C'est pourquoi les résultats donnés par de telles méthodes se révèlent généralement de mauvaise qualité. Mais celles-ci se distinguent néammoins par leur rapidité et leur grande simplicité. Elles font appel à des modes de construction d'ordonnancement — ou SGS pour Schedule Generation Schemes —. Il existe deux SGS différents qui divergent au niveau de l'incrémentation : sur les tâches pour le SGS sériel [9] (on ordonnance une tâche à une date donnée puis on passe à la tâche suivante), sur le temps pour le SGS parallèle [9] (on ordonnance le maximum de tâches possible à une date donnée puis on passe à la date suivante).

  Nous décrivons une peu plus en détail la méthode sérielle que nous avons utilisé. Elle choisit à chaque étape une tâche ordonnançable (dont tous les prédécesseurs sont déjà

ordonnancés). Si cette tâche ne peut pas être ordonnancée à la plus petite date permise par ses prédécesseurs du fait d'un manque de ressource, alors elle sera ordonnancée plus tard, à la date la plus petite telle que la capacité de ressource disponible soit suffisante.

- Les méthode à passes multiples emploient différentes stratégies. L'une d'elles consiste à exécuter plusieurs SGS en utilisant une règle de priorité différente à chaque fois, tandis qu'une autre combine un SGS et une règle de priorité uniques en biaisant cette dernière grâce à une distribution probabiliste afin d'obtenir des ordonnancements différents à chaque itération [11]. Une dernière méthode consiste à exécuter une procédure de "forward-backward" [15] qui alterne passes avant et passes arrière. La passe avant applique la procédure sérielle classique décrite ci-dessus conjointement à l'utilisation d'une règle de priorité pour construire un ordonnancement au plus tôt. La passe arrière quant à elle génère un ordonnancement au plus tard en ordonnançant les tâches prioritaires successives à la plus grande date permise par leurs successeurs et les disponibilités de ressource. L'ordonnancement débute alors à la date finale fournie par la passe avant précédente et ordonnance les tâches antérieurement à cette date.

- Les méthodes de recherche locale se basent dans une large proportion sur une représentation en liste d'activités et sur le SGS sériel. Nous pouvons ainsi citer la procédure de Recuit Simulé de Bouleimen et Lecocq [2] ou encore un algorithme basé sur la recherche Tabou de Valls et al. [20].

- Les algorithmes génétiques. Nous pouvons donner comme exemple les algorithmes développés par Hartmann [7] qui emploient tous la méthode sérielle mais des représentations différentes (liste d'activités, clé aléatoire, règle de priorité).

- D'autres méthodes ont également été proposées :

  - Branch and Bound tronqué : exécution d'un SGS avec branchement occasionnel [17], ou bien Branch and Bound dont le temps d'exécution est limité [19] (dans ce cas, on branche d'abord sur les activités les plus prometteuses).
  - méthodes basées sur les arcs disjonctifs [1] et [18] : l'idée est d'étendre les relations de précédence en ajoutant des arcs indiquant les incompatibilités de ressources entre les activités.

3.3 Caractéristiques avancées de la méthode pour le RCPSP

Nous allons voir comment nous avons traité les différents aspects de la metaheuristique dans le cadre du problème du RCPSP. Ainsi, nous présenterons dans un premier temps l'heuristique que nous avons retenu pour la construction de la solution initiale, puis nous verrons de quelle façon nous avons généré les sous-problèmes successifs à optimiser. Nous nous attacherons ensuite à exposer la méthode de génération d'une nouvelle solution à partir du résultat de l'optimisation et de la solution courante. Nous verrons également de quelle manière nous avons apporté de la diversification en perturbant ponctuellement la solution courante et nous terminerons avec la gestion des divers paramètres.

3.3.1 Choix de l'heuristique de construction de la solution initiale

L'heuristique que nous avons choisie est basée sur le schéma de construction en série et intègre également une procédure de "forward-backward", qui va alterner passes avant et passes arrière jusqu'à ce que les résultats obtenus au cours de deux passes successives convergent vers une valeur commune.

Une telle heuristique a déjà été utilisée par Li et Willis [15] sous une forme légèrement différente. En effet, la procédure de Li et Willis donne parfois au cours de la passe arrière un ordonnancement non réalisable comportant des tâches ordonnancées dans le passé. Pour éviter un tel désagrément, et afin de ne jamais dégrader la solution courante (ce qui n'est pas le but principal de la méthode, l'heuristique utilisée devant être rapide et la solution fournie non necéssairement de très bonne qualité), nous avons choisi les règles de priorité suivantes :

- pour la passe avant, la plus petite des dates de début données par la passe arrière hormis lors de la première passe où la règle utilisée est MinLST[1]
- pour la passe arrière, la plus grande des dates de fin calculées à partir des dates de début données par la passe avant

Cette heuristique sert à générer une solution initiale réalisable mais permet également de déterminer les tâches critiques, c'est-à-dire les tâches dont les dates de début au plus tôt — ou EST pour Earliest Starting Time — (calculées par la passe avant) et au plus tard — ou LST pour Latest Starting Time — (calculées par la passe arriere) coïncident.

La convergence de notre heuristique peut être prouvée de la façon suivante :

Proposition 1.1 :
Soit $S=(s_1,\ldots,s_n)$ un ordonnancement réalisable et $S'=(s'_1,\ldots,s'_n)$ obtenu à partir de S de la façon suivante :
$\forall i \in A, s'_i \leq s_i$ (1).
Il est trivial d'admettre que (1) est une condition suffisante pour dire que S' ne dégrade pas S (car $s'_n \leq s_n$).

Proposition 1.2 :
Soit $Sf=(sf_1,\ldots,sf_n)$, un ordonnancement réalisable obtenu après une passe avant à partir de l'ordonnancement $Sb=(sb_1,\ldots,sb_n)$ donné par la passe arrière précédente. Nous allons démontrer que (1) est vraie pour Sf.

Démonstration :
Supposons (1) non vraie pour Sf. Alors :
$\exists i \in A$ tel que $sf_i > sb_i$
$\exists j \in A$ tel que $sb_j > sb_i$ et $sf_j < sf_i$
En effet, pour que l'exécution de la tâche i soit reculée, il est nécessaire qu'une tâche j débutant après i dans Sb soit ordonnancée avant i.

Nous pouvons déduire qu'au moment où la tâche j a été ordonnancée, la tâche i ne devait pas être ordonnançable car sinon elle aurait été choisie du fait de la règle de priorité utilisée (car $sb_j > sb_i$).
La figure 2 représente le résultat d'une passe arrière au cours de laquelle la tâche i a été ordonnancée à une date plus petite que la tâche j. Nous avons appelé pred(i) et pred(j) l'ensemble de toutes les tâches appartenant à un chemin entre la tâche 0 et respectivement i et j.

[1] plus petite date de début au plus tard (calculée à partir du plus long chemin entre une tâche et la tâche fictive de fin dans le graphe de précédence)

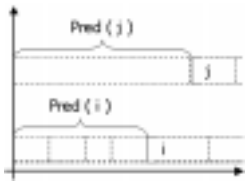

Figure 2: exemple de passe arrière avec $sb_i < sb_j$

Supposons maintenant que lors de la passe avant suivante, j est ordonnançable avant i. Pour cela, nous pouvons considérer le cas extrême où il n'existe qu'un chemin entre la tâche 0 et j. Ce chemin contient alors uniquement le prédécesseur de j dans le projet, lequel est ordonnancé en premier. Ainsi, j se retrouve ordonnançable avant i. Or, nous pouvons clairement constater qu'au moment où la tâche j est ordonnançable, elle est toujours dominée, et ceci de façon récursive, soit par une tâche appartenant à pred(i), soit par la tâche i elle-même, ce qui infirme la validité de notre raisonnement.

D'où (1) est vraie pour Sf et Sf ne dégrade donc pas Sb.

Le raisonnement permettant de prouver qu'une passe arrière ne dégrade pas le résultat donné par la passe avant précédente est de même nature.

Théorème :
Notre heuristique de "forward-backward" converge en un nombre fini de pas.

Démonstration :
Les résultats obtenus au cours des passes successives de l'heuristique valent au maximum $\sum_{i \in A} p_i$ (borne supérieure UB) et au minimum la valeur du chemin critique dans le graphe de précédence des tâches (borne inférieure LB).

Nous avons démontré que la valeur de la durée totale du projet ne peut pas, d'une passe à l'autre, être dégradée. De même, celle-ci ne peut rester constante plus de deux passes successives, auquel cas le test d'arrêt de l'heuristique est satisfait. Ainsi, nous pouvons conclure que notre heuristique converge vers une valeur comprise entre UB et LB en un nombre fini de pas valant au maximum UB-LB+1 (dans le cas où la première passe donne un makespan égal à UB et où le résultat n'est amélioré que de 1 à chaque itération).

3.3.2 Choix du sous-problème

Les variables fixées et libérées sont les dates de début des tâches dans la solution courante. Le sous-problème est déterminé en figeant une partie des tâches et en libérant les autres. On obtient ainsi un sous-problème plus général que le RCPSP lui-même car les tâches fixées peuvent être considérées comme des périodes d'indisponibilité partielle de chacune des ressources qu'elles occupent. Ainsi, nous pouvons définir un profil de disponibilité des ressources qui sera fonction du temps : une quantité variable $R_k(t)$ d'unités de ressource $k \in \mathcal{R}$ est disponible à chaque instant. De plus, chaque tâche libérée peut avoir à s'exécuter à l'intérieur d'une fenêtre de temps $[\underline{S}_i, \overline{S}_i]$donnée par ses prédécesseurs et ses successeurs fixés dans le projet.

Nous avons répresenté en figure 3 un exemple de sous-problème obtenu à partir de l'exemple précédent (cf. figure 1) en libérant les tâches 3, 4 et 5 et en figeant les autres à leur valeur courante. Nous obtenons ainsi les ensembles $\tilde{A} = \{3,4,5\}$ et $\tilde{E} = \{(i,j) \mid i,j \in \tilde{A}, (i,j) \in E\}$ à partir desquels nous construisons le graphe réduit $\tilde{G} = (\tilde{V}, \tilde{E})$, avec $\tilde{V} = \tilde{A} \cup \{0, n+1\}$. Nous définissons également les fenêtres d'exécution pour chacune des tâches. Finalement, nous avons reporté un schéma indiquant la variation de capacité des ressources en fonction du temps.

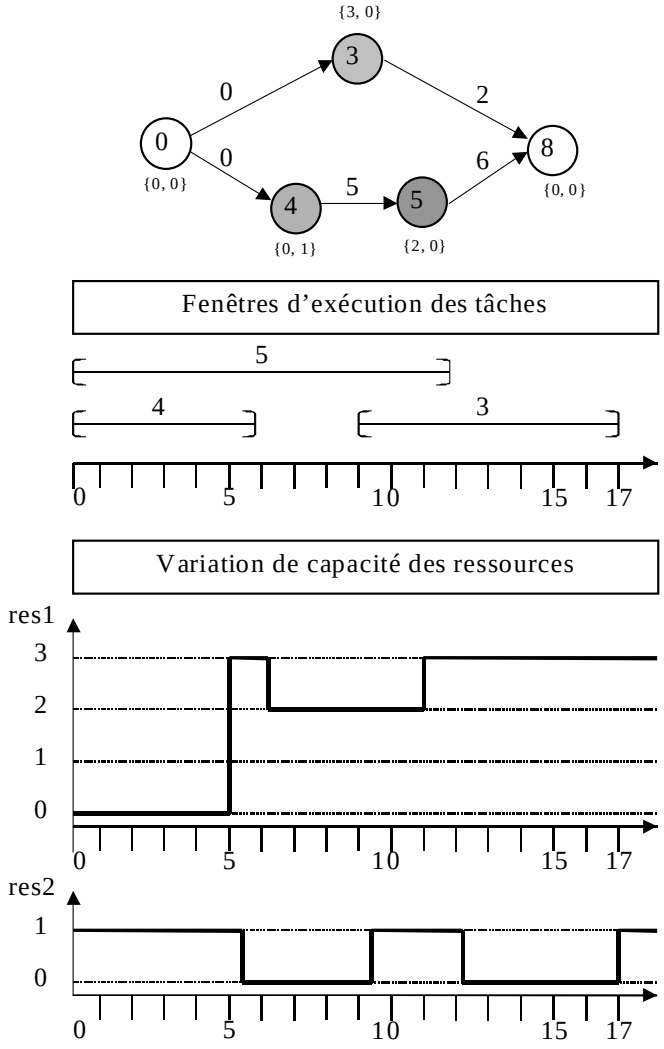

Figure 3: exemple de sous-problème

Le sous-problème, dont l'objectif est le même que le problème initial, consiste alors à caractériser le p-uplet des dates de début d'exécution des tâches libres présentes dans l'ensemble $\tilde{A}$ de sorte que :

$$\tilde{P} \ : \ \min \tilde{C}_{max} \qquad (1)$$

$$\text{s-à :} \quad \tilde{C}_{max} \geq \tilde{S}_i + p_i \quad \forall i \in \tilde{A} \qquad (2)$$

$$\tilde{S}_i \leq \tilde{S}_i + p_i \quad \forall (i,j) \in \tilde{E} \qquad (3)$$

$$\tilde{S}_i \geq \underline{S}_i \quad \forall i \in \tilde{A} \qquad (4)$$

$$\tilde{S}_i + p_i \leq \overline{S}_i \quad \forall i \in \tilde{A} \qquad (5)$$

$$\sum_{i \in \tilde{A}(t)} r_{ik} \leq R_k(t) \quad \forall k \in \mathcal{R}, \forall t \qquad (6)$$

où $\tilde{A}(t)$ est l'ensemble des activités libres en cours à l'instant t : $\tilde{S}_i \leq t < \tilde{S}_i + p_i$

Par ailleurs, nous avons envisagé d'utiliser les méthodes suivantes pour la détermination des tâches à libérer :

- 1 : "aléatoire". Les p tâches sont choisies aléatoirement.
- 2 : "aléatoire-prédécesseurs1". Les tâches sont choisies aléatoirement, mais lorsqu'une tâche est choisie, ses prédécesseurs dans le projet sont également choisis, dans la limite des p places disponibles (possibilité de faire de même avec les successeurs).
- 3 : "aléatoire-prédécesseurs2". Les tâches sont choisies aléatoirement, mais lorsqu'une tâche est choisie, les tâches la précédant immédiatement dans l'ordonnancement sont également choisies, dans la limite des p places disponibles (possibilité de faire de même avec les tâches qui débutent au moment où la tâche tirée s'achève).
- 4 : "aléatoire-tous prédécesseurs". Les tâches sont choisies aléatoirement, mais lorsqu'une tâche est choisie, ses prédécesseurs dans le projet ou les tâches qui la précèdent immédiatement dans l'ordonnancement sont également choisies, dans la limite des p places disponibles (possibilité de faire de même avec les successeurs et les tâches qui suivent immédiatement dans l'ordonnancement).
- 5 : "bloc". On choisit aléatoirement une tâche de départ et on constitue ensuite un "bloc" de tâches en considérant des tâches successives et en prenant les tâches qui s'exécutent même partiellement en même temps que la tâche courante ou qui la précèdent ou la suivent immédiatement dans l'ordonnancement, dans la limite des p places disponibles.

Nous pouvons faire quelques remarques sur les méthodes de génération de sous-problèmes. Pour les méthodes 2, 3 et 4, il est possible de combiner les deux aspects (prédécesseurs et successeurs dans le projet, tâches qui précèdent et suivent la tâche tirée dans l'ordonnancement). La méthode 5 semble la plus prometteuse dans l'optique de résoudre des problèmes de grande taille car toutes les autres risquent de fournir un sous-problème trop fortement contraint et donc peu aisément modifiable par la méthode exacte lorsque la taille du sous-problème est trop faible. Pour ces 4 premières méthodes de tirage, il parait donc nécessaire d'augmenter de façon significative la taille du sous-problème afin d'obtenir de bons résultats, mais il faut dans cette optique gérer au mieux le temps alloué pour la résolution des sous-problèmes. En effet, dans certains cas défavorables, les tâches pourraient être tirées d'une telle façon qu'il soit impossible de résoudre le sous-problème exactement dans des temps raisonnables. C'est pourquoi je préconise l'utilisation de la méthode 5 car celle-ci ne fait que très peu intervenir l'aléatoire.

3.3.3 Caractéristiques concernant les seuils

Pour les méthodes de sélection aléatoire ou partiellement aléatoires, nous définissons les seuils (cf. 2.2) comme suit :

- Seuil1 : $s(i) = \begin{array}{l} 1 \text{ si } i \text{ est une tâche critique} \\ C \text{ sinon, } 0 \leq C \leq 1 \text{ étant à déterminer} \end{array}$

  Ce premier seuil privilégie les tâches critiques.
- Seuil2 : $s(i) = 1 - C * \frac{LST_i - EST_i}{marge\_max}$, avec $marge\_max = \max_{i \in A} LST_i - EST_i$. $0 \leq C \leq 1$ est à déterminer.

  Ce deuxième seuil affecte à chaque tâche une probabilité qui sera proportionnelle à sa marge pondérée. Elle privilégie également les tâches critiques, celles-ci possédant une marge nulle.

3.3.4 Reconstruction de la solution courante

Pour reconstruire la solution courante, nous allons utiliser la même heuristique que pour la construction de la solution initiale, c'est à dire l'application successive de passes avant et passes arrière de la méthode sérielle. Cette méthode ne diffère d'avec la précédente qu'au niveau de la règle de priorité utilisée lors de la première passe. Celle-ci affecte à chaque tâche un indice de priorité correspondant à la date de début dans la solution courante pour les tâches fixées et à la date de début déterminée par la méthode exacte pour les tâches libérées.

La raison de l'utilisation de cette heuristique est qu'elle assure la génération d'un ordonnancement réalisable en même temps qu'une non dégradation de la solution courante du fait des règles de priorité utilisées. Ainsi, elle présente le double avantage de recaler la solution fournie par la méthode exacte dans la solution courante tout en l'améliorant éventuellement au cours de passes successives.

3.3.5 Perturbation de la solution courante

Pour diversifier notre recherche, nous avons choisi de reconstruire totalement la solution courante (principe des méthodes "multi-start").

Pour cela, nous appliquons une nouvelle fois notre heuristique de "forward-backward" en utilisant une règle de priorité aléatoire lors de la première passe avant. Ainsi, lorsque plusieurs tâches sont candidates, celle qui sera effectivement considérée pour l'ordonnancement est tirée au hasard. Cet aléa nous permet d'obtenir un ordonnancement réalisable que nous espérons différent à chaque fois.

3.3.6 Gestion des paramètres

- la taille p du sous-probleme : nous avons choisi de commencer avec une valeur relativement faible qui s'auto-ajuste toutes les cinq itérations en fonction du temps moyen mis par la méthode exacte pour résoudre les cinq sous-problèmes successifs. Cet auto-ajustement est fonction de plusieurs seuils qui déterminent si et dans quelles proportions nous pouvons incrémenter ou décrémenter la valeur de p.

- le paramètre MAX_ITER_SANS_AMEL définissant le nombre d'itérations sans amélioration au bout duquel la solution courante est dégradée : sa valeur est équivalente à la taille du problème à résoudre. Par exemple, pour un problème à 30 tâches, elle sera égale à 30.

- le paramètre t_{opt} donnant le temps maximum alloué à la méthode exacte pour résoudre un sous-problème : nous avons choisi de le fixer à 0.5 seconde afin d'obtenir un temps d'exécution global qui ne soit pas trop élevé sans toutefois laisser un temps trop bref à la méthode exacte pour la résolution des sous-problèmes.

- le nombre maximum d'itérations MAX : sa valeur est égale à dix foix la valeur du paramètre MAX_ITER_SANS_AMEL afin de permettre en moyenne au cours de l'exécution 10 départs de recherche.

- méthode de sélection des variables du sous-problème : nous avons choisi d'utiliser la méthode 5 par "bloc" car elle est celle qui donne les meilleurs résultats (cf. étude comparative en 3.4.2).

3.4 Résultats expérimentaux

3.4.1 Présentation des problèmes

Les performances de notre algorithme ont été éprouvées sur des instances à 30, 60, 90 et 120 tâches, générées par le générateur automatique PROGEN et publiées dans Kolisch et al. [12]. Ces problèmes comportent quatre ressources chacun et sont générés en faisant varier trois paramètres :

- la complexité du reseau NC (Network Complexity), c'est à dire la densité des contraintes de précédence entre tâches
- le facteur de ressource RF (Resource Factor) qui détermine le nombre moyen de ressources occupées par les tâches durant leur exécution
- la force de ressource RS (Resource Strength) dont la valeur est proportionnelle à la capacité des différentes ressources

3.4.2 Résultats

Notre algorithme a été codé en C++ et nous avons utilisé la bibliohèque ILOG Scheduler (qui résout des problèmes d'ordonnancement en utilisant la propagation par contraintes) pour la résolution exacte des sous-problèmes.

Nous avons tout d'abord testé les performances des différentes méthodes de sélection du sous-problème sur des instances à 30 tâches et les résultats sont reportés dans le tableau ci-dessous. En première colonne sont indiquées les différentes méthodes. Les pourcentages moyens de déviation à l'optimum qui leur sont associés sont reportés dans la colonne de droite.

méthode	*dev_opt*
aléatoire	1.5338
aléatoire-prédécesseurs1	1.2370
aléatoire-prédécesseurs2	1.0885
aléatoire-tous prédécesseurs	0.9524
bloc	0.0187

Tableau 1 : pourcentages moyens de déviation à l'optimum donnés par les différentes méthodes sur des problèmes à 30 tâches

Les performances de notre méthode ont été comparées à divers algorithmes de la littérature : l'algorithme génétique de Hartmann [7] et une méthode basée sur le recuit simulé de Bouleimen et Lecocq [2] utilisant tous deux la représentation en liste d'activités et le SGS sériel, la méthode parallèle à passe multiple de Kolisch [11] utilisant le sampling et l'algorithme CARA de Valls et al. [20] basé sur la recherche Tabou. Les résultats sont présentés dans les tableaux ci-dessous.

Les heuristiques présentées en colonne 1 sont triées selon des performances décroissantes. La deuxième colonne indique le pourcentage moyen de déviation à l'optimum pour les instances à 30 tâches ou le pourcentage moyen de déviation à la valeur du chemin critique pour les autres instances. La troisième colonne présente le pourcentage maximal de déviation par rapport à la meilleure solution actuelle (toutes méthodes confondues) pour la méthode CARA et notre heuristique. Le nombre d'instances pour lesquelles ces deux algorithmes trouvent la meilleure solution est reporté en colonne 4 et leurs temps moyens d'exécution sont indiqués en colonne 5. En ce qui concerne les autres algorithmes, nous n'avons pas pu reporter ces diverses

informations. Cependant, nous pouvons noter que les résultats qui sont ici communiqués ont été obtenus après 5000 itérations.

algorithme	*dev_opt*	*dev_max*	*nb_meil_sol*	*temps_moyen*
Notre méthode	0.0187	2.0618	474	22.23
Valls et al. [20] (2000)	0.0562	3.4483	463	1.61
Bouleimen, Lecocq [2] (1998)	0.23	-	-	-
Hartmann [7] (1997)	0.25	-	-	-
Kolish [11] (1996)	1.29	-	-	-

Tableau 2 : resultats sur les J30

algorithme	*dev_LB*	*dev_max*	*nb_meil_sol*	*temps_moyen*
Notre méthode	10.9325	3.2520	426	58.03
Valls et al. [20] (2000)	11.4546	6.3158	375	2.76
Hartmann [7] (1997)	11.89	-	-	-
Bouleimen, Lecocq [2] (1998)	11.90	-	-	-
Kolish [11] (1996)	13.23	-	-	-

Tableau 3 : resultats sur les J60

algorithme	*dev_LB*	*dev_max*	*nb_meil_sol*	*temps_moyen*
Notre méthode	10.5376	4.0322	407	93.91
Valls et al. [20] (2000)	11.1234	5.4054	368	4.63

Tableau 4 : resultats sur les J90

algorithme	*dev_LB*	*dev_max*	*nb_meil_sol*	*temps_moyen*
Notre méthode	33.1640	8.3333	244	318.32
Valls et al. [20] (2000)	34.5330	8.1081	199	43.94
Hartmann [7] (1997)	36.74	-	-	-
Bouleimen, Lecocq [2] (1998)	37.68	-	-	-
Kolish [11] (1996)	38.75	-	-	-

Tableau 5 : resultats sur les J120

Les résultats obtenus par notre méthode sont meilleurs pour tous les types d'instances que les résultats obtenus par les heuristiques de la littérature, tant au niveau du pourcentage moyen de déviation à l'optimum ou à la borne inférieure qu'au niveau du nombre de meilleures solutions trouvées. Il est à noter que ces dernières comportent également les solutions que nous avons amélioré (6 bornes supérieures pour les instances à 60 tâches, 9 pour celles à 90 tâches et 1 pour celles à 120 tâches). Le seul point d'ombre demeure la relative lenteur de notre méthode par rapport à celle de Valls et al.

4 Conclusion et perspectives

Nous avons développé une méthode relativement originale pour la résolution des problèmes d'optimisation combinatoire NP-difficiles et l'avons testé avec succès sur certaines instances du RCPSP. Les résultats obtenus sont particulièrement encourageants malgré un temps d'exécution moyen.

C'est pourquoi notre premier effort devra se porter sur ce point que nous espérons améliorer non seulement par une meilleure programmation, mais également par la recherche de nouvelles méthodes de génération de sous-problèmes plus performantes, par une meilleure gestion des temps de calcul dispensés par la méthode exacte pour la résolution des sous-problèmes ou par un meilleur réglage des divers paramètres de la méthode.

Par la suite, nous pourrons nous intéresser à la robustesse de notre heuristique en testant ses performances avec d'autres outils de résolution exacte, comme la programmation linéaire par exemple.

Nous essayerons également de raffiner les mécanismes propres aux heuristiques, notamment les phases de diversification et d'intensification. Afin d'apporter de la diversification, nous avons envisagé l'optique de perturber la solution courante au lieu de partir d'un nouveau point de départ. En ce qui concerne l'imbrication des processus, nous pensons développer une phase "normale" qui ne consisterait ni en diversification, ni en intensification, et qui permettrait de gérer au mieux les deux autres phases.

References

[1] C. E. Bell and J. Han. A new heuristic solution method in resource-constrained project scheduling. *Naval Research Logistics*, 38:315–331, 1991.

[2] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem. Technical report, Université de Liéges, 1998.

[3] Y. Caseau and F. Laburthe. Effective forget-and-extend heuristics for scheduling problems. In *CP-AI-OR'99, Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, Ferrara, Italy, 1999.

[4] A. Colorni, M. Dorigo, V. Manezzio, and M. Trubian. Ant system for job-shop scheduling. *Belgian Journal of Operations Research, Statistics and Computer Science*, 34, 1994.

[5] F. Glover. Genetic algorithms and scatter search: unsuspected potentials. *Statistics and Computing*, 4, 1994.

[6] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[7] S. Hartmann. A competitive genetic algorithm for the resource-constrained project scheduling. Technical report, Université de Kiel, 1997.

[8] J. H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[9] J. E. Kelley. The critical-path method: Resources planning and scheduling. *Industrial scheduling*, pages 347–365, 1963.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[11] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90:320–333, 1996.

[12] R. Kolisch, A.Sprecher, and A. Drexl. Benchmark instances for project scheduling problems. *Handbook on recent Advances in Project Scheduling*, pages 197–212, 1998.

[13] R. Kolisch and S. Hartmann. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 127(2):394–407, 2000.

[14] R. Kolisch and R. Padman. An integrated survey of deterministic project scheduling. *Omega*, 29(3):249–272, 2001.

[15] K. Y. Li and R. J. Willis. An iterative scheduling technique for resource-constrained project scheduling. *European Journal of Operational Research*, 56:370–379, 1992.

[16] T. Mautor and P. Michelon. Mimausa : A new hybrid method combining exact solution and local search. In *MIC'97, 2nd Metaheuristics International Conference*, Sophia Antipolis, France, 1997.

[17] B. Pollack-Johnson. Hybrid structures and improving forecasting and scheduling in project management. *Journal of Operations Management*, 127(2):394–407, 2000.

[18] L. R. Shaffer, J. B. Ritter, and W. L. Meyer. *The critical-path method*. McGraw Hill, 1965.

[19] A. Sprecher. Solving the rcpsp efficiently at modest memory requirements. Technical report, Université de Kiel, 1996.

[20] V. Valls, F. Ballestin, and M. S. Quintanilla. Resource-constraint project scheduling: A critical activity reordering heuristic. In *PMS'2000, 7th International Workshop on Project Management and Scheduling*, pages 282–283, Osnabruck, Germany, 2000.