UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences

Département de mathématiques

MEMOIRE DE MAGISTER

Spécialité : Recherche opérationnelle

RESOLUTION DU PROBLEME D'ORDONNANCEMENT $P|prec|C_{max}$ PAR UN ALGORITHME DE COLONIE DE FOURMIS

Par

MESSAOUDI OUCHENE Mohamed

Devant le jury composé de :

M. Blidia	Professeur, USDBlida,	Président du Jury
M. Boudh	Professeur, USTHB Alger,	Examinateur
R. Boudjemaa	Maître de Conférences-B, UMBBoumerdès,	Examinateur
A. Derbala	Maître de Conférences-A, USDBlida.	Promoteur

Blida, Janvier 2013

UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences

Département de mathématiques

MEMOIRE DE MAGISTER

Spécialité : Recherche opérationnelle

RESOLUTION DU PROBLEME D'ORDONNANCEMENT $P|prec|C_{max}$ PAR UN ALGORITHME DE COLONIE DE FOURMIS

Par

MESSAOUDI OUCHENE Mohamed

Devant le jury composé de :

M. Blidia	Professeur, USDBlida,	Président du Jury
M. Boudh	Professeur, USTHB Alger,	Examinateur
R. Boudjemaa	Maître de Conférences-B, UMBBoumerdès,	Examinateur
A. Derbala	Maître de Conférences-A, USDBlida.	Promoteur

Blida, Janvier 2013

REMERCIEMENTS

Je remercie tout d'abord, notre vénéré Allah,
Le tout puissant, à qui nous devons le tout.

Je tiens à exprimer ma gratitude à mon responsable de recherche Derbala Ali**,** maître de conférences à l'université Saad Dahlab de Blida, pour m'avoir permis de diriger mes travaux de cette recherche. Je le remercie également pour ces précieux conseils, et surtout sa disponibilité.

Je remercie Monsieur Blidia Mustafa, Professeur à l'université Saad Dahlab de Blida, pour avoir accepté d'examiner ce travail, et qui m'a fait l'honneur de présider ce jury. Qu'il trouve ici l'expression de ma profonde reconnaissance.

J'adresse mes sincères remerciements à Monsieur Boudhar Mourad, Professeur à l'université Houari Boumediene d'Alger, d'avoir accepté d'être membre de jury et de m'avoir fait l'honneur d'examiner mon travail.

Pour m'avoir fait l'honneur de participer au jury et d'examiner mon travail, je remercie Monsieur Boudjemaa Redouane, Maître de conférences à l'université M'hamed Bougara de Boumerdès

Je ne saurais oublier de remercier mes chers parents, ma grand-mère, ma femme, ma fille Asmaa, mon frère et mes chères sœurs qui étaient toujours à mes cotés et m'ont tant aidé et soutenu. Qu'ils trouvent ici l'expression de ma sincère gratitude et ma profonde reconnaissance.

Je remercie enfin mes chers amis, tous mes camarades pour leur éternelle bonne humeur et leur sympathie.

RÉSUMÉ

Le difficile problème étudié est celui de l'ordonnancement de tâches sur des machines parallèles identiques à contraintes de précédence afin de minimiser la longueur d'ordonnancement. Il est noté $P \mid prec \mid C_{max}$. Une recherche bibliographique a été entreprise. Une adaptation d'un algorithme de colonies de fourmis est réalisée pour le résoudre. Son application nous fournit une meilleure affectation de tâches aux machines. En face de chaque machine, les tâches sont exécutées selon trois règles de priorité PLC, VPLC et MAX. Avec ces trois règles et les deux informations heuristiques statiques et dynamiques appelées « visibilité », six versions de cet algorithme de colonies de fourmis ont été obtenues, étudiées et comparées. La justification et la finitude de l'algorithme sont exposées. Son implémentation est discutée. De nombreuses expérimentations ont été effectuées sur des données de problèmes générées aléatoirement. L'étude comparative de quatre métaheuristiques de type recuit simulé, recherche taboue, algorithme génétique et la version colonie de fourmis STA_MAX est réalisée. Les solutions obtenues par la version STA_MAX sont les meilleures. Un simulateur conçu par nos soins est présenté. Il permet de tester les différentes métaheuristiques.

Mots-Clés : Ordonnancement à machines parallèle, Liste de priorité, Optimisation par Colonies de fourmis, Métaheuristique.

ABSTRACT

The hard problem studied is the scheduling of tasks on identical parallel machines with constrained tasks to minimize schedule length or makespan. It is denoted P | prec |Cmax. A bibliographic research is carried out. A modified Ant Colony Algorithm is proposed. The application of this algorithm provides a better allocation of tasks to machines. In front of each machine, tasks are carried out according to three rules of priority PLC, VPLC and MAX. With these three rules and with both static and dynamic heuristics information called "visibility", six versions of the Ant Colony Algorithm are obtained, studied and evaluated. The justification and the finitness of this algorithm are presented and its implementation is discussed. Numerous experiments are conducted on data of problems generated randomly. A comparison among the four metaheuristics of simulated annealing, taboo search, genetic algorithm and the Ant colony version is done. The solutions obtained using the STA_MAX version are the best. An own designed simulator is presented and it permits to test various metaheuristics.

Keywords: Scheduling parallel, ant colony optimization, priority list, metaheuristics.

LISTE DES ALGORITHMES, ILLUSTRATIONS, GRAPHIQUES & TABLEAUX

TABLE DES MATIERES

INTRODUCTION

Le problème d'ordonnancement sur des machines parallèles identiques à contraintes de précédence noté $P|prec|C_{max}$ est classé NP difficile [Bru98]. En général de tel ordonnancement s'effectue en deux phases : l'affectation des tâches aux machines et l'exécution des tâches sur une machine. Notre objectif est de minimiser la longueur d'ordonnancement ou la date de fin d'exécution de la dernière tâche. Ce critère de « longueur d'ordonnancement » est important dans le cas $P||C_{max}$ car l'ordonnanceur assure l'équilibre de la charge entre les machines. Nous ne considérons que les ordonnancements non-préemptifs. Une fois une tâche débute son exécution, elle ne peut être interrompue. Le temps requis pour exécuter une tâche donnée est connu d'avance et ne dépend pas de la machine utilisée. Des relations de précédence peuvent exister entre les tâches et sont données par un graphe.

Notre motivation à faire cette étude et que des problèmes d'ordonnancement à machines parallèles restent *ouverts* et sont cités dans [Sch99]. En ordonnancement parallèle, les algorithmes les plus utilisés sont des heuristiques de types listes. Ils déterminent pour un ordre de tâches données par une liste, l'ordonnancement correspondant. Ils considèrent les tâches une à une et prennent la décision d'ordonnancer sur la base d'un ordonnancement partiel de tâches ordonnancées auparavant.

Dans le cas des problèmes NP difficiles et si le nombre de tâches est grand, la détermination d'un ordonnancement optimal est presque impossible du point de vue temps d'exécution. Dans ce cas, si on se contente d'une solution approchée, une heuristique ou une métaheuristique est à prescrire. La résolution approchée est pratique pour des applications d'ingénieries, qui ont un besoin urgent en estimation de ressources de calculs, tels le calcul parallèle, la synthèse d'un système digital et la compilation de haute performance ([Ayt03], [Dem02]).

Pour beaucoup de problèmes NP difficiles et dans une grande variété de domaines, les métaheuristiques ont reçu un intérêt considérable et se sont avérées efficaces dans des domaines variés tels industriel, économique, logistique, ingénierie, commerce, domaines scientifiques, etc. Phelipeau-Gelineau [Gel96] a développé un algorithme de recherche taboue pour résoudre le problème $P|prec|C_{max}$. Aytug et al. [Ayt03] fournissent un état de l'art sur l'utilisation des algorithmes génétiques dans la résolution des problèmes d'ordonnancement. Boumédiène et Derbala [Bou06] ont étudié et comparé six algorithmes de type liste avec un algorithme génétique.

Des exemples de nouvelles métaheuristiques sont les algorithmes évolutionnistes de type *colonies de fourmis*. Ces algorithmes s'inspirent du comportement des fourmis réelles qui communiquent entre elles à l'aide d'une substance chimique appelée la phéromone [Dor99]. Dans un algorithme d'Optimisation de Colonie de Fourmis (OCF), deux informations heuristiques statiques et dynamiques sont indispensables. Une règle d'affectation est à entreprendre pour le séquencement des tâches.

Dans le chapitre 1, les problèmes d'ordonnancement déterministes à machines parallèles sont présentés. Les concepts de base tels les notions de tâches, de ressources, les objectifs ou les critères d'évaluation sont définis. Les caractéristiques des tâches, des machines en parallèles et les propriétés des problèmes d'ordonnancement sont exposées. La classification des problèmes d'ordonnancement à machines parallèles est indiquée selon la notation de Blazewicz et al.[Bla94]. Les méthodes générales de résolution des problèmes d'ordonnancement sont exposées. Elles sont de types exactes telles les procédures par séparation et évaluation ou de la programmation dynamique. Elles sont approchées et de type recuit simulé, recherche taboue, algorithmes génétiques, colonies de fourmis, réseaux de neurones, de la logique floue et autres. La représentation des solutions, la complexité des problèmes et des algorithmes sont fournis. Un ensemble de résultats concernant les problèmes à machines parallèles identiques est présenté. Une recherche bibliographique sur le problème $P|prec|C_{max}$ est entreprise. Les problèmes d'ordonnancement sans contraintes de précédence et avec contraintes de précédence sont détaillés. Dans le chapitre 2 on introduit la métaheurisique colonies de fourmis. Ses principes généraux de fonctionnement sont présentés. En étudiant une version préliminaire, l'Ant System, l'algorithme de l'OCF produit est détaillé à partir des modifications d'algorithmes originaux. Des applications à des problèmes d'optimisation sont aussi fournies. Une analyse de la bibliographie concernant l'OCF a été faite.

Dans le chapitre 3, une adaptation d'un algorithme de fourmis pour la résolution du problème difficile d'ordonnancement $P|prec|C_{MAX}$ est proposée [Mes08]. Initialement, on applique l'algorithme d'OCF qui nous fournit une meilleure affectation de tâches aux machines. La seconde étape consiste, en face de chaque machine, à exécuter les taches, selon trois règles de priorité, le plus long chemin (PLC), une variante de PLC (VPLC) et le maximum entre PLC et la durée totale des tâches dans la descendance (MAX). Avec ces trois règles et les deux informations heuristiques statiques et dynamiques, six versions de cette méthode de colonies de fourmis ont été étudiées et comparées. Sa justification et sa finitude sont exposées. Son implémentation est discutée. De nombreuses expérimentations ont été effectuées sur des

données de problèmes générées aléatoirement. La méthode de colonie de fourmis statique munie de la règle VPLC a fourni de bonnes longueurs d'ordonnancement.

Dans le chapitre 4, l'étude comparative des quatre métaheuristiques de types recuit simulé, recherche taboue, un algorithme génétique et une version de colonie de fourmis dénotée STA_VPLC est réalisée. Nous proposons des variantes de ces métaheuristiques fournissant une meilleure affectation de tâches aux machines. Les quatre méthodes utilisées permettent d'obtenir des solutions réalisables en des temps raisonnables. Les solutions obtenues à l'aide de la version STA_VPLC sont, cette fois, meilleure que celle à laquelle aboutissent les algorithmes génétiques et la recherche taboue. Les temps d'exécution obtenus par application du recuit simulé sont démesurés. Les résultats dépendent fortement du type et de la taille du problème abordé. De même, il n'existe pas de paramètres adaptés à l'ensemble des problèmes. La topologie de l'espace des solutions du problème influe fortement sur l'efficacité de certaines stratégies et valeurs de paramètres par rapport à d'autres. Il est à souligner que la nature aléatoire de ces algorithmes, notamment pour les solutions initiales, rend le problème ardu. Pour le même benchmark, des simulations différentes avec exactement les mêmes paramètres peuvent produire des résultats différents.

Dans le chapitre 5, une présentation du simulateur conçu par nos soins est faite. Ce simulateur permet de tester rapidement les différentes métaheuristiques et d'étudier les effets de chaque paramètre tel la taille de la colonie, l'exposant de trace de phéromone, de la visibilité ou autre ou la stratégie adoptée pour résoudre le problème $P|prec|C_{max}$.

CHAPITRE 1 : ORDONNANCEMENT A MACHINES PARALLELES

1. Introduction

Un problème d'ordonnancement consiste à organiser les temps l'exécution d'un ensemble de tâches, sous des contraintes temporelles et des contraintes de disponibilité des ressources requises par les tâches [Lop01].

Un ordonnancement constitue une solution au problème de l'ordonnancement. Il décrit l'exécution des tâches et l'allocation des ressources au cours du temps, tout en satisfaisant un ou plusieurs objectifs [Gra66].

Nous présentons les problèmes d'ordonnancement déterministes à machines parallèles. Les concepts de base et les principes de méthodes générales de résolution des problèmes d'ordonnancement sont définis. Les problèmes d'atelier, où on doit exploiter au mieux pour réaliser un ensemble très varié de produits, sont notre intérêt. La complexité réside non pas dans les processus de fabrication, qui sont prédéterminés sous forme de gammes linéaires, mais plutôt dans la combinatoire qui naît de la prise en compte de la limitation des ressources existantes. Un ensemble de résultats sont présentés concernant les problèmes à machines parallèles identiques qui modélisent plusieurs problèmes théoriques et pratiques tels que la synthèse d'un système digital et la compilation de haute performance ([Ayt03], [Dem02]).

2. Concepts de base en ordonnancement

2.1 Les tâches

Une tâche est une entité élémentaire de travail localisée dans le temps par une date de début d'exécution de_i ou de fin d'exécution C_i, dont la réalisation est caractérisée par une durée p_i tel que $C_i = de_i + p_i$.

Nous ne considérons que les ordonnancements non-préemptifs. Une fois une tâche débute son exécution, elle ne peut être interrompue. Le temps requis pour exécuter une tâche donnée est connu d'avance et ne dépend pas de la machine utilisée.

Des relations de précédence peuvent exister entre les tâches et sont données par un graphe $G\ (V, E)$ où V est l'ensemble des tâches et un arc (i, j) est dans E si, et seulement si,

"*i précède j*". La tâche *j* ne peut commencer son exécution que si la tâche *i* ait terminée son exécution.

Dans ce cas les tâches sont dites *liées* ou *dépendantes*, sinon elles sont appelées *indépendantes*. En général et en ordonnancement d'atelier, une tâche n'est formée que d'une seule opération. Dans ce cas, tâche est opération sont synonymes.

2.2 Les ressources

Une ressource est un moyen technique ou humain requis pour la réalisation d'une tâche et disponible en quantité limitée ou illimitée. Plusieurs types de ressources sont à distinguer.

Une ressource est *renouvelable* si après avoir été utilisée par une ou plusieurs tâches, elle est à nouveau disponible en même quantité. Les hommes, les machines, etc. en constituent un exemple; la quantité de ressource utilisable à chaque instant est limitée. Dans le cas contraire, elle est *consommable* du type matières premières, budget, etc.

On distingue par ailleurs les ressources *disjonctives* ou non-partageables qui ne peuvent être disponible que pour l'exécution d'une tâche à la fois, du cas de machine-outil, robot manipulateur, etc. Les ressources *cumulatives* ou partageables peuvent être utilisées par plusieurs tâches simultanément, telles qu'un poste de travail, équipe d'ouvriers, etc.

Une machine peut constituer une ressource.

2.3 Objectifs et critères d'évaluation

Dans la résolution d'un problème d'ordonnancement, on peut choisir entre deux types de stratégies, visant respectivement à l'optimalité des solutions par rapport à un ou plusieurs critères, ou à leur admissibilité vis-à-vis des contraintes. L'approche par optimisation suppose que les solutions candidates à un problème puissent être ordonnancées de manière rationnelle selon un ou plusieurs critères d'évaluation numérique permettant d'apprécier la qualité des solutions. On cherche à minimiser ou maximiser de tels critères.

On note par exemple ceux qui sont :

- Liés aux temps : le temps total d'exécution ou le temps moyen d'achèvement d'un ensemble de tâches ; les différents types de retards par rapport aux dates fixées.
- Liés aux ressources : la quantité – maximale, moyenne ou pondérée – de ressources nécessaires pour réaliser un ensemble de tâches ; la charge de chaque ressource.

- Liés aux coûts de lancement, de production, de transport, de stockage,… etc.

2.4 Propriétés des problèmes d'ordonnancement

Un problème est *déterministe* ou *stochastique* si, respectivement, ses paramètres sont connus et donnés ou sont inconnus mais suivent une distribution de probabilité.

Dans un problème d'ordonnancement classique, une machine ne peut exécuter qu'une tâche à la fois et une tâche ne peut être exécutée que sur une machine à la fois.

Une affectation des tâches qui respecte les contraintes liées à l'environnement des machines et aux caractéristiques des tâches est dite *ordonnancement réalisable* ou *ordonnancement*. Si l'ordonnancement optimise le critère d'optimalité, il est dit *optimal*.

Un ordonnancement est *semi-actif* si aucun travail ne peut avancer sur la ressource où il se trouve, compte tenu des contraintes de gamme et de précédence.

Il est *actif* si aucune opération d'un travail *i* ne peut débuter son exécution plus tôt sans déplacer au minimum une autre opération.

Il est *sans délai* lorsque aucune machine n'est laissée inoccupée, ceci alors qu'une file d'attente contient au moins un travail susceptible de débuter son exécution sur cette machine. Lorsqu'une méthode parcourt l'ensemble des ordonnancements sans délai, elle n'est pas capable de trouver la solution optimale. On ne laisse pas une machine inoccupée alors qu'une file contient des tâches.

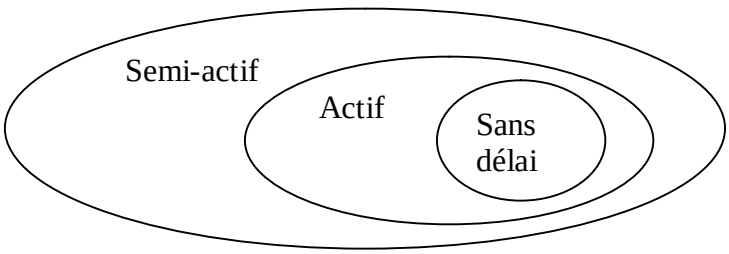


Figure 1.1 Les catégories d'ordonnancement

Un problème d'ordonnancement est défini par les machines, les tâches et par un critère d'optimalité.

2.4.1 Caractéristiques des machines en parallèles

Quand les machines sont *en parallèles,* elles font la même fonction.

On distingue trois types de machines parallèles dépendant de leurs vitesses d'exécution des tâches.

- Si elles ont la même vitesse d'exécution des tâches, elles sont dites *identiques*.
- Si elles diffèrent par leurs vitesses d'exécution et la vitesse d'une machine est constante et ne dépend pas des tâches, elles sont dites *uniformes*.
- Si les vitesses d'exécution des machines dépendent des tâches et sont différentes alors elles sont dites *quelconques*.

Dans le cas de machines disposées *en séries*, il y a trois modèles ou types d'exécution de tâches: le *flow shop*, l'*open shop* et le *job shop*.

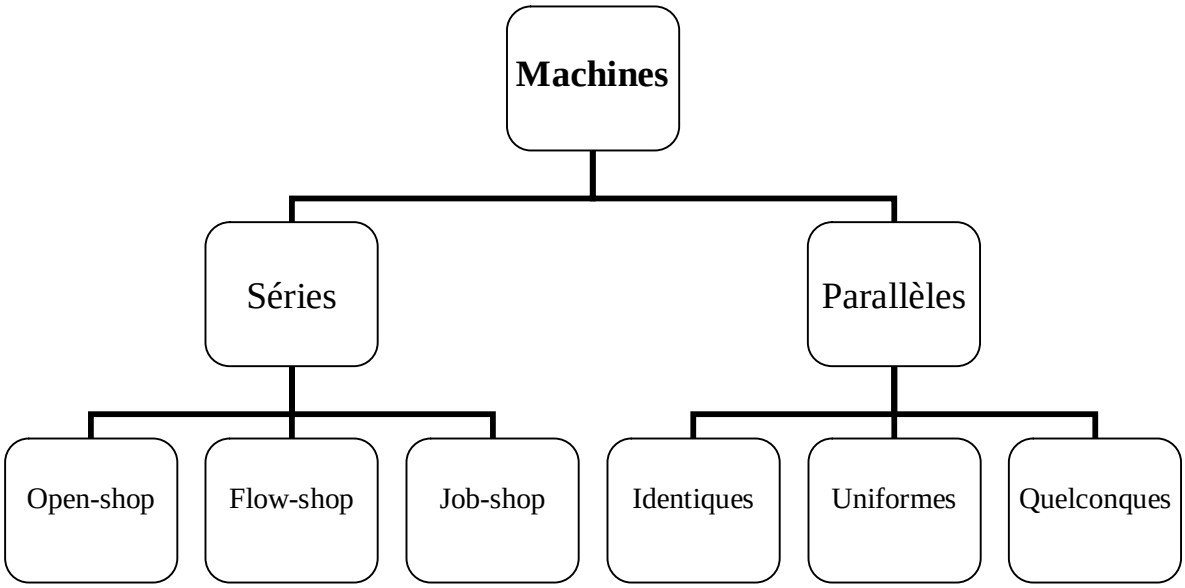


Figure 1.2 Types de machines

2.4.2 Caractéristiques des tâches [Bla94]

Soient $T = \{T1, T_2, \ldots, T_n\}$ un ensemble de « n » tâches et $M = \{M1, M_2, \ldots, M_m\}$ un ensemble de « m » machines. A chaque tâche $T_j \in T$, on associe les paramètres suivants :

1. La *durée d'exécution* p_{ij} de la tâche T_j sur la machine M_i. Dans le cas de machines identiques, le temps d'exécution d'une tâche ne dépend pas de la machine, d'où $p_{ij} = p_i, \quad i = 1..m$. Si les machines sont uniformes alors $p_{ij} = \frac{p_i}{b_i}$, $i = 1..m$ où p_i est une durée d'exécution de référence, mesurée usuellement sur la machine la moins rapide et b_i est le facteur vitesse d'exécution de la machine M_i.

2. La *date d'arrivée* ou *date de disponibilité* (*release date*) r_j. La date où la tâche T_j est prête pour l'exécution. Si les dates d'arrivée sont les mêmes pour toutes les tâches, on peut poser $r_j = 0, j = 1..n$.

3. La *date de fin d'exécution au plus tard* (*due date*) d_j, appelée aussi *date de fin échue* ou *date de fin souhaitée*. Si la tâche termine son exécution après cette date, elle encourt une pénalité.

4. La *date de fin impérative* $\tilde{d}_j$ (*deadline*). Si la tâche termine son exécution après cette date, elle ne risque pas seulement une pénalité mais des problèmes surgiront, soit l'atelier est bloqué ou la machine tombe en panne.

5. Le *poids* ou la *priorité* w_j (*weight*). Il exprime une urgence dans l'exécution de la tâche T_j.

6. La *date de fin d'exécution* C_j. En général, c'est une variable à déterminer.

7. Le *décalage* $L_j = C_j - d_j$ (*lateness*). Le temps total durant lequel, la tâche est autorisée à rester dans l'atelier.

8. Le *retard* $D_j = max\ \{C_j - d_j, 0\}$ (*tardiness*).

9. L'*avance* $E_i = max\ \{0, -L_i\}$ (*earliness*).

10. L'*indicateur de retard* $U_i = 0$ si $c_i \leq d_i$ et $U_i = 1$ sinon

2.4.3 Les fonctions objectives

Les critères d'optimalité les plus utilisés font intervenir la durée totale de l'ordonnancement, le délai d'exécution, les retards de l'ordonnancement, etc. La durée totale d'ordonnancement, notée C_{max}, est égale à la date d'achèvement de la tâche la plus tardive: $C_{max} = \max_j \{C_j\}$. C'est la *longueur d'ordonnancement* appelée aussi *makespan* (*Schedule length*).

Le *flow time pondéré* $\sum_{j=1}^{n} w_j C_j$ permet d'estimer le coût des stocks d'encours. En effet, une tâche T_j est présente dans l'atelier entre les instants r_j et C_j. Les stocks dont elle a besoin doivent être disponibles entre ces deux dates; d'où le coût $\sum_{j=1}^{n} w_j (C_j - r_j)$ égale, à un constant pré, à $\sum_{j=1}^{n} w_j C_j$.

Dans beaucoup de problèmes, il faut respecter les délais et les dates au plus tard d_j. Minimiser le *plus grand retard* $D_{max} = \max_j D_j$, la *somme des retards* $\sum_{j=1}^{n} D_j$ ou la *somme pondérée des tâches en retard* $\sum_{j=1}^{n} w_j D_j$ peuvent être d'un intérêt.

D'autres critères peuvent être définis et utilisés. Le *décalage maximum* $L_{max} = \max_j L_j$. Le *nombre de tâches en retard* $\sum_{j=1}^{n} U_j$. Le *nombre de tâches en retard pondéré* $\sum_{j=1}^{n} w_j U_j$.

2.5 Classification des problèmes d'ordonnancement à machines parallèles

Les problèmes d'atelier sont représentés selon une classification à trois champs $\alpha/\beta/\gamma$ qui a été introduite en *1979* par Graham et al. [Gra79], et Blazewicz et al. [Bla78].

Le champ $\alpha = \alpha_1 \ \alpha_2$ décrit l'environnement des machines.

Le paramètre $\alpha_1 \in \{ P, Q, R\}$ caractérise le type de machines parallèles utilisées :

$\alpha_1 = P$: machines identiques parallèles.

$\alpha_1 = Q$: machines parallèles uniformes.

$\alpha_1 = R$: machines parallèles quelconques.

$\alpha_2 \in \{\varnothing, k\}$ est un entier qui représente le nombre de machines dans le problème.

$\alpha_2 = \varnothing$: le nombre de machines est supposé être variable.

$\alpha_2 = k$: le nombre de machines est égal à k (k entier positif).

Le second champ $\beta = \beta_1 \beta_2 \beta_3 \beta_4 \beta_5 \beta_6 \beta_7 \beta_8$ décrit les tâches et les caractéristiques des ressources.

Le paramètre $\beta_1 \in \{\varnothing, pmtn\}$ indique la possibilité de la préemption de la tâche. La préemption n'est pas autorisée ou autorisée.

$\beta_2 \in \{\varnothing, res\}$ caractérise les ressources additionnelles. Aucune ressource additionnelle n'existe ou des contraintes de ressources spécifiées.

$\beta_3 \in \{\varnothing, prec, tree, chains\}$ indique les types de contraintes de précédence. Les tâches sont indépendantes, de contraintes de précédence quelconques, formant un arbre ou formant une union de chaînes.

$\beta_4 \in \{\varnothing, r_j\}$ toutes les instants au plutôt sont nuls ou sont différents.

$\beta_5 \in \{\varnothing, p_j = p, \underline{p} \leq p_i \leq \bar{p}\}$ les durées d'exécution sont arbitraires, égales ou dans l'intervalle définie.

$\beta_6 \in \{\varnothing, \tilde{d}\}$ décrit les deadlines. Aucune date de fin impérative ou des dates sont imposées.

$\beta_7 \in \{\varnothing, n_j \leq k\}$ décrit le nombre maximum d'opérations constituant une tâche. Aucune contrainte ou le nombre est inférieur ou égal à *k*.

$\beta_8 \in \{\varnothing, no\text{-}wait\}$ décrit, pour un problème à machines spécialisées, une propriété d'attente. La zone de stockage est de capacité illimitée ou sans attente, les capacités d'attente sont nulles. Une tâche qui termine son exécution sur une machine passe instantanément sur une autre machine.

Le troisième champ γ désigne le critère à utiliser.

$\gamma \in \{C_{max}\ ; \sum_{j=1}^{n} C_j\ ;\ L_{max}\ ;\ \sum_{j=1}^{n} D_j\ ; \sum_{j=1}^{n} w_j D_j\ /\ \sum_{j=1}^{n} w\ ; \sum_{j=1}^{n} U_j\ ;\ \sum_{j=1}^{n} w_j U_j\ ,\ etc.\ \}$.

Exemple :

- P_3 / *pmtn, prec* / L_{max} désigne le problème de minimiser le retard maximum de tâches liées par des contraintes de précédence arbitraires, à trois machines parallèles identiques. La préemption est autorisée.

2.6 Méthodes de résolution d'un problème d'ordonnancement

On décrit ici brièvement quelques méthodes générales d'optimisation que l'on utilisés souvent dans la résolution exacte ou approchée des problèmes d'ordonnancement.

2.6.1 Méthodes exactes

Elles recherchent un ordonnancement optimal minimisant l'un des critères présentés. Les techniques utilisées sont les méthodes par séparation et évaluation [Bak74], la programmation dynamique [Bel74] ou autres.

2.6.1.1 Les procédures par séparation et évaluation (PSE)

Les méthodes arborescentes Branch and Bound sont des méthodes exactes d'optimisation qui pratiquent une énumération intelligente de l'espace des solutions. Il s'agit

d’énumérations complètes améliorées. Elles partagent l'espace des solutions en sous ensembles de plus en plus petits, la plupart étant éliminés par des calculs de bornes avant d'être construits explicitement.

On peut inventer plusieurs méthodes arborescentes. Cependant elles auront trois composantes communes :

- Une règle de séparation
- Une fonction d'évaluation
- Une stratégie d'exploration

La séparation consiste à décomposer l’ensemble des solutions en plusieurs sous-ensembles qui sont décomposés à leur tour selon une démarche itérative.

Ce processus peut se mettre sous la forme d’un arbre d’énumération ; les nœuds de l’arbre correspondent aux sous-ensembles des solutions et les feuilles correspondent à des solutions réalisables. Pour accélérer la recherche de la solution optimale en évitant l’exploration inutile de certains nœuds, on utilise une procédure d’évaluation qui calcule, à chaque niveau de l’arbre, la valeur de chaque nœud et permet ainsi de décider du nœud à développer. Dans le cas d’un problème de minimisation, cela revient à calculer la borne inférieure pour tous les nœuds fils du niveau considéré et la descente dans l’arbre est poursuivie avec le nœud donnant la borne inférieure. Par ailleurs une borne supérieure de la solution optimale est calculée et est utilisée pour éviter l’exploration de nœuds dont la valeur de la borne inférieure est supérieure à la valeur de la borne supérieure. Cette borne supérieure est réactualisée lorsqu’une solution réalisable de valeur inférieure est atteinte.

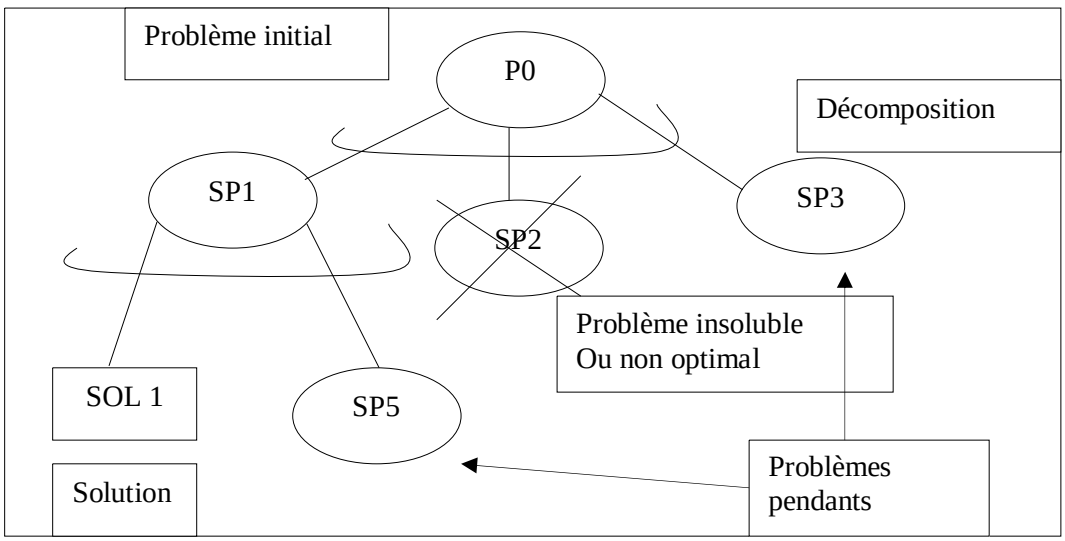


Figure 1.3 Espace de recherche associé à une PSE

Des remontées et descentes par d'autres nœuds de l'arbre sont effectuées tant que la borne calculée est inférieure ou égale à la valeur actualisée de la borne supérieure. L'exploration de certaines branches de l'arbre est ainsi évitée, ce qui permet de ne pas énumérer toutes les solutions réalisables. Il faut donc souligner que l'efficacité de la méthode en termes du nombre de nœuds explorés est déterminée par la qualité de la borne initiale et par la procédure d'évaluation considérée.

2.6.1.2 La programmation dynamique

La programmation dynamique est une technique mathématique qui a pour objet d'aider à prendre des décisions séquentielles indépendantes les unes des autres. Il n'y a pas de formalisme mathématique standard. C'est une approche de résolution où les équations dites de Bellmann doivent être spécifiées selon le problème à résoudre. Tout problème de programmation dynamique peut être défini par un processus de décision séquentielle. Un processus de décision séquentielle est formé :

- D'un système dynamique à temps discret évoluant pendant un nombre fini de périodes.
- D'une fonction coût additive au fil des périodes.

Dans cette famille, un problème de dimension *n* est décomposé en *n* problèmes de dimension *1* chacun. Le système est constitué de *n* étapes que l'on résout séquentiellement, le passage d'une étape à l'autre se fait à partir des lois d'évolution du système et d'une décision.
Cette méthode a été introduite par Bellmann [Bel74]. Son principe d'optimalité est basé sur l'existence d'une équation récursive permettant de décrire la valeur optimale du critère à une étape donnée en fonction de sa valeur à l'étape précédente. Pour appliquer la programmation dynamique à un problème combinatoire, le calcul du critère pour un sous-ensemble de taille *k* nécessite la connaissance de ce critère pour chaque sous-ensemble de taille *k-1*, ce qui porte le nombre de sous-ensembles considérés à *2n* où *n* est le nombre d'éléments considérés dans le problème, d'où sa complexité est exponentielle.

2.6.2 Méthodes approchées

Une méthode approchée ou heuristique est un algorithme qui a pour but de trouver une solution réalisable d'un problème sans garantie d'optimalité.

2.6.2.1 Le recuit simulé [Kir83]

Cette technique provient de l'observation de la formation d'une structure cristalline quand un métal refroidit. Le recuit simulé est une technique permettant de trouver, au bout d'un temps raisonnable, une solution pour des problèmes de complexité élevée. Elle est utilisée pour la détermination de la répartition statique des tâches. Elle est considérée comme une technique stochastique et ne garantie pas de solution optimale. Elle sera développée ultérieurement dans le chapitre 4.

2.6.2.2 Les algorithmes génétiques [Hol75]

Ils sont fondés sur les principes de l'évolution biologique des espèces. Ils sont basés sur une notion étendue de voisinage. Plusieurs solutions sont maintenues simultanément et évoluent par des techniques de recombinaison et de mutation. Une des difficultés rencontrées est le codage des solutions.

2.6.2.3 Les réseaux de neurones

Ils ont été appliqués à la résolution approchée des problèmes d'optimisation en général et à d'ordonnancement en particulier. Leur avantage est la possibilité d'utiliser des circuits analogiques dont l'évolution converge vers des états stables correspondant à une solution approchée du problème d'ordonnancement. Grâce à leur vitesse élevée de convergence, leur utilisation en-ligne peut être envisageable. Leur inconvénient est le blocage à la rencontre d'un minimum local.

2.6.2.4 La logique floue

Elle apporte des solutions pour les problèmes qu'on ne sait pas spécifier complètement et quantitativement. On associe un degré de satisfaction à la fin d'exécution de chaque tâche. Si elle se termine avant sa date critique, la satisfaction est stable. Le degré d'insatisfaction augmente au fur et à mesure que l'on s'éloigne de cette date critique.

Pour l'ordonnancement stochastique, quand les temps d'exécution des tâches sont aléatoires, deux approches sont souvent utilisées : la théorie des files d'attentes et les processus Bandits.

2.7 Représentation des solutions

On représente souvent un ordonnancement par un diagramme de GANTT, un repère orthogonal dans un plan. L'axe des abscisses représente le temps et l'axe des ordonnées représente l'ensemble des machines. Pour chaque machine, on représente la séquence de taches effectuées dans le temps.

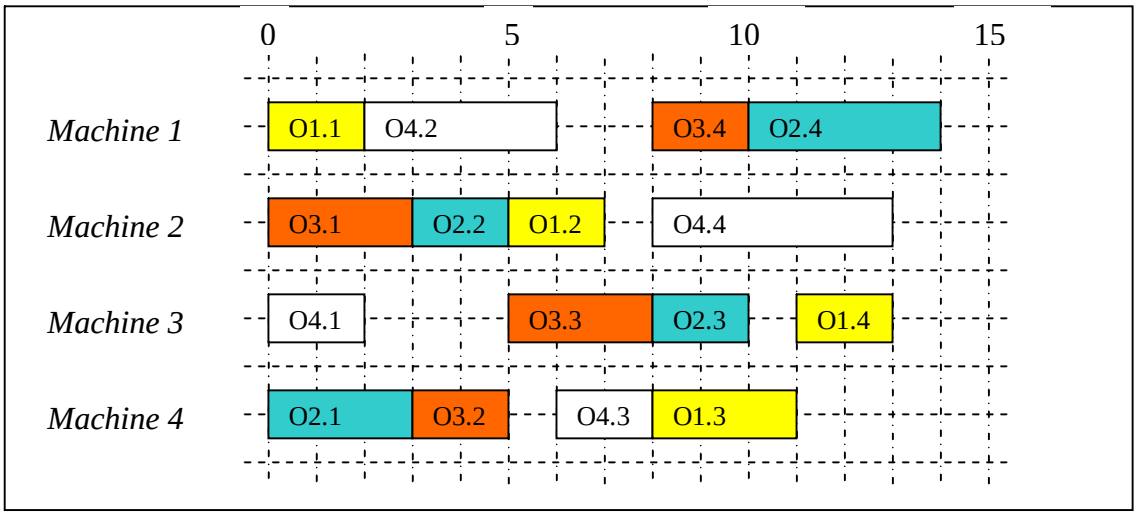


Figure 1.4 Exemple de diagramme de Gantt « ressource »

3. Les problèmes d'ordonnancement à machines parallèles

Le problème à machines parallèles se caractérise par le fait que plusieurs machines sont disponibles pour l'exécution d'un travail qui n'en nécessite qu'une. Ce problème est une généralisation du problème à une machine et un cas particulier de problème d'atelier multimachines. Il est fréquemment rencontré dans des applications réelles, en particulier dans les traitements informatiques [Bla96a].

C'est aussi un problème d'affectation. Déterminer une solution c'est :

- Décider sur quelle machine effectuer chaque opération ;
- Déterminer la séquence d'opérations sur chaque machine.

On appelle *charge* d'une machine la somme des durées d'exécution des tâches qui lui sont affectées. Nous ne considérons que les problèmes d'ordonnancement statiques et déterministes, à machines parallèles identiques, non préemptifs et à contraintes de précédence quelconques. Le critère d'optimalité est la minimisation de la longueur d'ordonnancement de

problème noté $P / prec / C_{max}$. Il s'interprète comme l'équilibre de la charge du travail entre les machines.

4. Complexité des problèmes et des algorithmes

Par abus de langage, une méthode de résolution *polynomiale* ou en *temps polynomial*, désigne une méthode où le temps d'exécution est limité supérieurement par une fonction polynomiale de la taille *N* du problème. On qualifie de *facile* les problèmes pour lesquels de telles méthodes existent. Par opposition, pour les problèmes combinatoires *difficiles*, il faut se résigner à des méthodes dont le temps de résolution n'est limité supérieurement que par une fonction *exponentielle* de *N*. Un exemple de complexité exponentielle est en O (2^N).

- Un problème est dit de la classe P s'il peut être résolu par un algorithme polynomial. Il est dit facile. Sinon les problèmes sont dits difficiles.
- Un algorithme non déterministe est muni d'une instruction qui permette, chaque fois qu'elle est appliquée, de faire le bon choix.
- Un problème appartient à la classe NP s'il peut être résolu par un algorithme polynomial non déterministe. NP signifie non déterministe polynomial.
- On conjecture que P ≠ NP. La classe NP contient des problèmes qui sont plus difficiles que ceux de la classe P si toutefois P ≠ NP [Coo70].
- Soient P1 et P2, deux problèmes de reconnaissance, on dit que P1 se réduit en temps polynomial à P2, s'il existe un algorithme pour P1 qui fait appel comme un sous programme, à un algorithme de résolution de P2 et si cet algorithme de P1 est polynomial lorsque la résolution de P2 est comptabilisée comme une opération élémentaire.
- Un problème de reconnaissance (PE) est dit " NP-complet ", si tout problème de la classe NP peut se réduire polynomialement à lui.
- Si un problème P se réduit en temps polynomial à P' et si P' peut être résolu par un algorithme polynomial, il en est de même de P.
- La propriété de « NP-complet » signifie que si on avait un algorithme polynomial pour un problème NP-complet alors on aurait un algorithme polynomial pour tous les problèmes de la classe et dans ce cas P = NP.
- Cook [Coo70], a jeté les fondements de la complexité des algorithmes.

- Si un problème P est « NP-complet » et si on peut mettre en évidence une réduction polynomiale de P à P', alors P' est « NP-complet ».

5. Recherche bibliographique sur le problème P / Prec / Cmax

Contrairement au problème *1 // Cmax* qui est trivial, il devient intéressant de rechercher à minimiser la durée totale lorsqu'on considère plusieurs machines en parallèle (*Pm // Cmax*). En pratique, cette minimisation peut conduire dans certains cas à mieux répartir la charge, en vue d'une utilisation rentable du parc de machines (afin d'éviter les sous/sur-utilisations).

5.1 Problème d'ordonnancement sans contraintes de précédence

En l'absence de contraintes de précédence, le problème *Pm // Cmax* est NP-difficile [Gar78], même à partir de deux machines [Sch00] ; les méthodes exactes sont essentiellement des PSE.

Dans le cas d'opérations interruptibles (P_m / *pmtn* / C_{max}), l'algorithme de Mc Naughton fournit un ordonnancement de durée minimale pour une complexité en *O (n)* [Mcn59].

- La première phase consiste à calculer la grandeur : $C^*_{\max} = \max\left(\frac{1}{m}\sum_{i=1}^{n} P_i, \max_i\{P_i\}\right)$

qui montre que soit les machines sont totalement utilisées dans un ordonnancement optimal, soit la durée totale est donnée par l'opération la plus longue.

- On procède ensuite aux phases suivantes :
  1. Sélectionner une opération et l'ordonnancer sur la première machine à $t = 0$.
  2. Ordonnancer au plus tôt sur la même machine une opération non encore sélectionnée. Recommencer tant que la durée totale sur la machine ne dépasse pas C_{max*}.
  3. Transférer la partie du travail ordonnée après C_{max*} sur la machine suivante. Revenir en *2*.

Si la préemption est interdite, *Pm // Cmax,* on peut utiliser une heuristique qui consiste à ranger les opérations suivant un ordre *LPT* (*Longest Processing Time first*) et à les affecter dans cet ordre, successivement, à la machine la moins chargée. Les opérations les plus courtes sont donc conservées à la fin de l'ordonnancement afin d'équilibrer au mieux la charge. Cette heuristique permet d'obtenir une performance relative illustrée par le ratio entre la durée obtenue par l'heuristique et la durée optimale [Sev01]:

$$\frac{C_{\max}(LPT)}{C^*_{\max}} \leq \frac{4}{3} - \frac{1}{3m}$$

Cette inégalité signifie que la longueur d'un ordonnancement *LPT* peut être au plus près de *33%* supérieure à celle optimale.

5.2 Problèmes d'ordonnancement avec contraintes de précédence

- En revanche, le P_m / *prec* / C_{max} avec $2 \leq m < n$ est NP-difficile au *sens fort* [Bac74].
- Pour le P_m / p_i=*1, tree* / C_{max}, l'algorithme de Hu [Hu61] fournit une solution optimale. Il réalise d'abord un étiquetage des nœuds, en partant du nœud initial. Un ordonnancement est ensuite déterminé en affectant à la machine qui se libère le plus tôt, les opérations de plus petite étiquette.
- En présence de contraintes de précédence, même avec des temps d'exécution unitaires, *Pm/ prec,* p_j = *1/* C_{max} est NP-complet au *sens fort* [Ull76].
- Dans le cas de deux machines, P_2 / *prec,* p_j = *1/*C_{max} est *polynomial* [Cof72]. Une extension heuristique est proposée sur la base de cet algorithme pour le cas où les durées ne sont plus égales et les structures de précédence plus générales (problème NP-difficile).
- Les problèmes Q_2 / *chains,* p_j = *p/* C_{max} et *P/ intree,* p_j = *p/* L_{max} [Bru99] [Bru77] sont *polynomiaux*.
- Le problème P_2 */prec,* p_j = *p/* L_{max} [Gar76] est *polynomial*.

- Les problèmes *P/ chains,* p_j = *1/* C_{max} et *P/ tree,* p_j = *1/* C_{max} [Hu61] sont *polynomiaux.*
- Un algorithme a été développé par Muntz et Coffman [Cof84] lorsque deux machines identiques sont utilisées (P_2 / *prec, pmtn* / C_{max}).
- Des même auteurs, un autre algorithme permet de résoudre optimalement le P_m / *tree* / C_{max} en utilisant le concept de « *partie de machine utilisée* ». Cela revient à faire l'hypothèse que la durée de l'opération *i* sur la machine *k* est inversement proportionnelle à la vitesse v_k de la machine $\frac{p_i}{v_k}$ (problème Q_m / *tree* / C_{max}). Les deux algorithmes de Muntz et Coffman sont basés sur la procédure d'étiquetage de Hu.

- Phelipeau-Gelineau [Phe96] a développé un algorithme de recherche Taboue pour résoudre le problème P / prec/ C_{max}.
- Aytug et al. [Ayt03] fournissent un état de l'art sur l'utilisation des algorithmes génétiques dans la résolution des problèmes d'ordonnancement. Les articles sont référencés par type de problème qu'ils résolvent.
- Pour le problème P // C_{max}, Chiu et al. [Chi99] ont été incapables de comparer leur algorithme génétique avec d'autres heuristiques.
- Boumédiène et Derbala [Bou06] ont étudié et comparé six méthodes de listes définies dans la littérature [Phe96]. La performance des méthodes est reproduite graphiquement où ils mesurent la qualité et l'efficacité de chaque liste. D'autre part, un algorithme génétique, basé sur deux types de croisement, est implémenté et comparé avec les listes.

CHAPITRE 2 : OPTIMISATION PAR COLONIE DE FOURMIS (OCF)

1. Introduction

Pour appréhender le travail présenté dans ce mémoire, il est important d'analyser les travaux réalisés sur le thème des colonies de fourmis. L'algorithme des colonies de fourmis est une métaheuristique dont le comportement est inspiré de celui des fourmilières réelles. Son principe général de fonctionnement est présenté. Une version préliminaire, l'Ant System, est étudiée. L'algorithme de l'OCF produit est détaillé à partir des modifications d'algorithmes originaux. Des applications à des problèmes d'optimisation sont aussi fournies.

Pour une bonne introduction à la notion des métaheuristiques, il est conseillé de consulter les livres de Alba [Alb05], Dréo et al. [Dre03], Goldberg [Gol94], Holland [Hol75], Michalewicz [Mic91] et Talbi [Tal09].

L'optimisation par colonie de fourmis constituera l'outil de résolution du problème d'ordonnancement dans ce présent travail.

2. L'Optimisation par Colonie de Fourmis (OCF)

2.1 Principe général

L'algorithme s'inspire d'une expérience menée en 1989 par Goss *et al.* [Gos89]. Des fourmis avaient été mises en contact avec une source de nourriture reliée à la fourmilière par un pont ayant deux branches de longueurs différentes, tel qu'illustré à la Figure 2.1. Les fourmis parvenaient après quelque temps à emprunter toujours le chemin le plus court entre la fourmilière et la nourriture. Il a été observé que la probabilité que toutes les fourmis empruntent le chemin le plus court augmentait avec la différence de longueur entre les deux branches du pont.

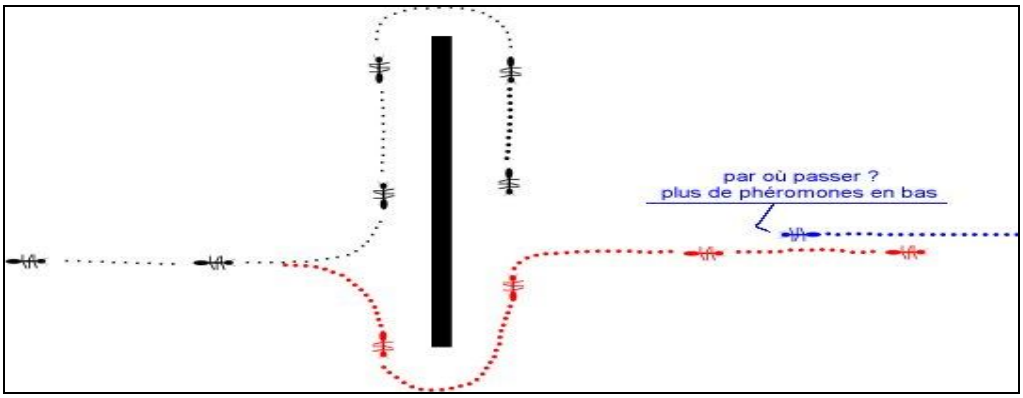


Figure 2.1 Disposition de l'expérience de Goss *et al.* [Dor99a]

Goss *et al.* [Gos89] ont établi que les fourmis partageaient des informations sur leurs expériences d'exploration à l'aide d'une substance chimique appelée *phéromone*. Elles déposent par terre une certaine quantité de cette substance lorsqu'elles se déplacent et en même temps, scrutent le sol pour « lire » les informations laissées par leurs prédécesseurs. Lorsqu'elles ont à faire un choix quant à la direction à prendre, les fourmis sélectionnent un chemin de façon probabiliste en fonction de la quantité de phéromone présente sur les différentes voies possibles. Le comportement observé par Goss s'explique par le fait que les fourmis qui empruntent le chemin le plus court parviennent à la source de nourriture et reviennent à la fourmilière plus vite que celles qui ont pris le plus long chemin. En laissant toujours une trace de phéromone, elles rendent le chemin emprunté plus attirant pour les fourmis suivantes. Comme la phéromone s'évapore avec le temps, le chemin le plus long est de moins en moins emprunté et sa trace disparaît presque complètement. Le temps nécessaire à la stabilisation du système est imputable au fait qu'au départ aucune phéromone n'est présente sur aucune route. Les fourmis ont alors autant de chances de choisir le plus court que le plus long chemin.

L'OCF est un algorithme à l'intérieur duquel des agents indépendants, les fourmis artificielles, construisent des solutions de manière probabiliste due à une mémoire collective des expériences antérieures, la trace de phéromone. Le premier problème résolu à l'aide d'un algorithme de fourmi est celui du voyageur de commerce (*Traveling Salesman Problem - TSP*) [Rob49]. La résolution de ce problème consiste à trouver dans un graphe pondéré un cycle hamiltonien de longueur minimale, où un cycle hamiltonien est un chemin élémentaire fermé.

2.2 Version préliminaire de l'OCF : L'Ant System

L'Ant System (AS) a été introduit en 1991 par Dorigo et al. [Dor91b]. Il est le premier algorithme de fourmis fourni et le prototype de plusieurs autres méthodes [Man99b].

Le fonctionnement général de l'AS consiste à construire des chemins pour un ensemble de fourmis, de manière probabiliste en se basant sur la mémoire collective, la trace de phéromone, et sur une vision locale du problème, et ce, pour un nombre de cycles donné. Lorsque toutes les fourmis ont terminé la construction d'une solution, un cycle est fermé.

Pour communiquer entre elles, les fourmis utilisent une matrice de phéromone qui garde en mémoire les informations associée aux solutions. Les valeurs de cette représentation s'accroît avec le temps puisque la matrice est mise à jour par les fourmis au cours de l'algorithme.

Les fourmis sont des agents simples et possèdent les caractéristiques suivantes :

(1) Elles choisissent la prochaine ville à visiter avec une probabilité qui est fonction de la distance entre les villes et de la quantité de phéromone présente sur l'arête les reliant.

(2) Pour les amener à construire des chemins hamiltoniens, la transition vers les villes déjà empruntées est interdite jusqu'à ce qu'un chemin soit totalement emprunté.

(3) Lorsqu'elles ont emprunté totalement un chemin, les fourmis déposent une certaine quantité de phéromone sur chaque arête *(r, s)* empruntée.

Le Tableau 2.1 résume les termes de la notation utilisée dans l'algorithme de l'AS. Il contient des variables utilisées et des paramètres dont les valeurs sont à fixer. Ces termes seront définis et interprétés plus en détails dans les paragraphes qui suivent.

Variable	**Description**
τ_{rs}	Quantité de phéromone présente entre *r* et *s*
$\Delta\tau_{rs}$	Quantité de phéromone à ajouter à *Trs*
η_{rs}	Visibilité locale entre r et s
P_{rs}^{k}	Probabilité pour la fourmi *k* de choisir l'élément *s*, connaissant *r*
L_k	Évaluation de la solution de la fourmi *k*
$tabou_k$	Les éléments déjà placés pour la fourmi *k*
y, *t*	Index de la liste tabou, Temps actuel (compteur de cycles)
Paramètre	**Description**
m, NC_{MAX}	Nombre de fourmis, Nombre maximal de cycles
τ_0 , ρ	Quantité de phéromone initiale, Persistance de la trace
α, β	Exposant de la trace de phéromone, Exposant de la visibilité locale

Tableau 2.1 Notation de l'AS

Un pseudo-code de l'Ant System est fourni ci dessous.

Dans l'étape 1, on initialise le compteur de cycles et la trace de phéromone.

Un cycle commence à l'étape 2 où l'indice *y* des listes tabou est remis à 1 et chaque fourmi place sa ville de départ dans sa liste taboue.

À l'étape 3, les fourmis construisent des chemins en parallèle qui sont évalués à l'étape 4, où la meilleure solution trouvée jusqu'à maintenant est mise à jour.

À l'étape 5, il y a mise à jour globale de la trace.

À l'étape 6, le compteur de cycles est incrémenté et on décide de poursuivre ou non l'exécution, selon que le nombre de cycles NC_{MAX} est atteint ou qu'une autre condition d'arrêt préalablement définie est atteinte.

Algorithme 2.1 Pseudo-code de l'Ant System [Dor96b]

1. $t = 0$
   Initialiser chaque arête (r, s) à la valeur de τ_0
2. $y = 1$
   POUR chaque fourmi $k = 1$ à m *{Une fourmi sur chaque ville}*
   Placer la fourmi k sur la ville k et affecter $tabou_k(y) = k$
3. **POUR** $y = 2$ à h
   POUR chaque fourmi $k = 1$ à m
   Choisir une ville s selon P_{rs}^k, calculé selon l'Équation 2.4
   $tabou_k(y) = s$
4. **POUR** chaque fourmi $k = 1$ à m
   Calculer la longueur L_k du chemin décrit par k
   Mettre à jour la meilleure solution globale
5. **POUR** chaque arête (r, s)
   Mettre à jour τ_{rs} selon l'Équation 2.1
6. $t = t + 1$
   SI $(t < NC_{MAX})$ **ALORS**
   Vider toutes les listes tabou
   Aller à l'étape 2
   SINON
   ARRÊTER

La trace de phéromone est indiquée par une matrice τ de nombres réels ayant la même structure que la matrice d'adjacence du graphe de problème. La valeur de $\tau_{rs}(t)$ indique la quantité de phéromone présente sur l'arc (r, s) à l' instant *t.*

Au départ, tous les éléments de la matrice de trace sont initialisés à une valeur constante positive τ_o proche de zéro. L'accumulation de la trace encourage les fourmis à retourner sur les arêtes empruntées antérieurement. Elle agit dans ce cas comme élément

d'intensification de la recherche. La mise à jour de la trace se fait de manière globale, à la fin de chaque cycle et selon l'Équation 2.1, où ρ est un paramètre fixé.

$$\tau_{rs}(t+1) = \rho.\tau_{rs}(t) + \Delta\tau_{rs} \qquad (2.1)$$

$\Delta\tau_{rs}$ est calculé pour chaque arête *(r, s),* selon l'Équation 2.2, où $\Delta\tau_{rs}^{k}$ est la quantité de phéromone laissée par une fourmi *k* pour l'arc *(r, s).* Cette quantité se calcule selon l'Équation 2.3, où Λ est une constante et L_k est la longueur du chemin de la fourmi *k.* La quantité de phéromone laissée par une fourmi sur une arête est fonction de la qualité de la solution finale qu'elle a produite. Toutes les fourmis participent à la mise à jour de la trace et de ce fait elles n'ajoutent aucune phéromone sur les arêtes qu'elles n'ont pas empruntées pour construire leur solution.

$$\Delta\tau_{rs} = \sum_{k=1}^{m} \Delta\tau_{rs}^{k} \qquad (2.2)$$

$$\Delta\tau_{rs}^{k} = \begin{cases} \dfrac{\Lambda}{L_k} & \text{si la fourmi k emprunte l'arc (r, s} \\ 0 & \text{sinon} \end{cases} \qquad (2.3)$$

Le coefficient ρ est appelé *persistance de la trace* et sa valeur est comprise entre 0 et 1. À chaque cycle, une partie de la phéromone sur toutes les arêtes s'évapore et seules celles empruntées en reçoivent en fonction de la qualité des solutions dont elles font partie. Le mécanisme d'évaporation a pour but d'éviter les accumulations de trace à l'infini et reproduit le phénomène d'évaporation de la phéromone naturelle. Ainsi, les arêtes dont la trace n'est pas renouvelée deviennent moins attirantes pour les fourmis.

Une liste taboue $tabou_k$ est associée à chaque fourmi *k* afin d'assurer les chemins hamiltoniens. Lorsqu'un chemin est construit, $tabou_k$ contient, dans l'ordre, les villes parcourues par la fourmi *k* et $tabou_k(y)$ est la $y^{ième}$ ville visitée par cette fourmi. Les listes taboues sont vidées après chaque cycle et une fois la mise à jour de la trace effectuée.

Lorsqu'une fourmi *k* doit faire le choix de la prochaine ville à visiter, elle calcule la probabilité P_{rs}^{k} de transition vers chaque ville potentielle $s \notin tabou_k$ où *r* est la dernière ville visitée par la fourmi *k.* Cette probabilité est fonction de τ_{rs} et d'une visibilité locale, η_{rs}, qui doit refléter l'attrait d'une ville de manière locale. Pour la fourmi adaptée au TSP, la valeur de η_{rs} est donnée par $1/D_{rs}$. Cette valeur donne à la ville *s* un attrait inversement proportionnel à la distance entre *r* et *s.* On calcule P_{rs}^{k} selon l'Équation 2.4, appelée *règle de transition.*

$$P_{rs}^{k} = \begin{cases} \dfrac{[\tau_{rs}]^{\alpha} \cdot [\eta_{rs}]^{\beta}}{\sum_{w \notin \text{tabou}_k} [\tau_{rw}]^{\alpha} \cdot [\eta_{rw}]^{\beta}} & \text{si} \quad s \notin \text{tabou}_k \\ 0 & \text{sinon} \end{cases} \qquad (2.4)$$

Cette règle de transition est aussi appelée *règle aléatoire-proportionnelle (random proportional rule).* Notons que $\sum P_{rs} = 1$ et que les exposants α et β servent à balancer l'importance relative de la trace de phéromone en fonction de la visibilité locale. Si $\alpha = 0$ ou $\beta = 0$, on obtient respectivement une heuristique de construction gourmande.

Les valeurs que prennent habituellement les paramètres de l'AS sont indiquées par Dorigo *et al.* ([Dor91b], [Dor96b]). Le nombre de fourmis *m* doit être en *O(h),* où *h* est la taille du problème. La persistance de la trace ρ est de l'ordre de 0,5.

Les valeurs données à α et β ont un impact important sur la qualité des solutions obtenues. Pour de grandes valeurs de α combinées à des valeurs de β qui ne sont pas suffisamment grandes, l'algorithme entre rapidement dans un état de *chemin-unique,* où les fourmis empruntent toujours les mêmes arêtes sans trouver de bonne solution. Si on ne donne pas une valeur suffisamment grande à α ou trop grande à β, alors l'algorithme ne peut produire que des solutions de qualité moyenne. Dorigo *et al.* [Dor96b] concluent que α doit être autour de 1, et β entre 2 et 5 pour le problème du TSP testé.

La complexité temporelle de l'AS est en $O(NC_{MAX} \cdot h^2 \cdot m)$. Une relation linéaire entre le nombre de villes *h* et le meilleur nombre de fourmis *m* est observé. Cette complexité peut se réduire *en* $O(NC_{MAX} \cdot h^3)$ [Dor96a].

2.3 L'algorithme d'Optimisation par Colonie de Fourmis

L'OCF proprement dite a été introduite par Dorigo et Gambardella [Dor97b], toujours pour une application au TSP. Quatre éléments majeurs sont modifiés ou ajoutés à l'AS pour créer l'OCF.

La règle de transition présentée à l'Équation 2.4 est modifiée pour offrir un meilleur balancement entre l'exploration de nouvelles zones de l'espace de solutions et l'exploitation des connaissances accumulées sur le problème. Il en résulte l'Équation 2.5, avec *s* étant la ville choisie par la fourmi *k* et *r* étant la dernière ville visitée, où Q_o est un paramètre ($0 < Q_o < 1$), *Q* est un nombre généré aléatoirement, distribué uniformément entre [0,1] et ψ est une ville choisie aléatoirement en fonction des probabilités calculées selon l'Équation 2.4. On procède ainsi à une intensification de la recherche avec une probabilité de Q_o, puisque l'on

choisit alors l'argument maximal des $\{[\tau_{rw}]^{\alpha}.[\eta_{rw}]^{\beta}\}\ \forall\ w \notin tabou_k$, qui correspondent à l'attrait des différentes villes encore à visiter pour *k.* Avec une probabilité de (1 - Q_o), on optera pour la diversification en choisissant *s* selon la règle de transition de l'AS, ce qui ajoute à la recherche un caractère probabiliste. Cette règle est appelée *règle proportionnelle pseudo aléatoire (pseudo-random-proportionnal rule).*

$$s = \begin{cases} \arg\max_{w \notin tabou_k} \{[\tau_{rw}]^{\alpha}.[\eta_{rw}]^{\beta}\} & \text{si } Q \le Q_0 \\ \psi & \text{sinon} \end{cases} \qquad (2.5)$$

Afin de profiter d'une information locale supplémentaire, Dorigo *et al.* [Dor99b] ont introduit une liste de candidats. L'ensemble des choix possibles à partir d'une ville est restreint aux *lc* villes les plus proches non encore sélectionnées. Cette limitation des candidats a également l'avantage de réduire le temps nécessaire au calcul des probabilités.

La mise à jour de la trace de phéromone est modifiée afin de mieux orienter la recherche. Contrairement à l'AS, où toutes les fourmis participent à la mise à jour, une seule fourmi met à jour la trace de façon globale à la fin de chaque cycle et en fonction de la qualité de sa solution. Dorigo *et al.* [Dor99b] ont expérimenté une mise à jour par la meilleure fourmi globale *gb (global-best),* celle ayant produit la meilleure solution trouvée depuis le début de l'algorithme, et par la meilleure fourmi du cycle *(iteration-best).* Les deux stratégies donnent des résultats presque identiques avec une légère préférence pour la meilleure globale. La mise à jour *globale* est effectuée à la fin de chaque cycle, selon l'Équation 2.6, où ρ_{global} est la persistance globale de la trace et L_{gb} est la longueur du chemin de la meilleure fourmi globale avec $0 < \rho_{global} < 1$.

$$\tau_{rs}(t+1) = \rho_{global}.\tau_{rs}(t) + (1-\rho_{global}).\Delta\tau_{rs}$$

$$où \quad \Delta\tau_{rs} = \begin{cases} \frac{1}{L_{gb}} & \text{si gb emprunte (r, s)} \\ 0 & \text{sinon} \end{cases} \qquad (2.6)$$

Les auteurs ont introduit une seconde mise à jour de la trace, la mise à jour *locale,* pour contribuer à la diversification de la recherche. Chaque fourmi, lorsqu'elle ajoute une ville à sa solution, diminue la quantité de phéromone sur l'arête *(r, s)* qu'elle vient d'emprunter, selon l'Équation 2.7, où ρ_{local} est la persistance locale de la trace avec $0 < \rho_{local} < 1$. Trois valeurs possibles ont été considérées pour $\Delta\tau$ [Dor97b]. L'emploi de $\Delta\tau = 0$ produit les pires résultats. Les auteurs ont également essayé $\Delta\tau = 0$ et $\Delta\tau = \gamma.\max_{w \notin tabou_k} \tau_{rw}$, où γ est un paramètre dans [0,1[.

Les OCF avec $\Delta\tau = \tau_0$ et avec $\Delta\tau = \gamma.\max_{w \notin \text{tabou}_k} \tau_{rw}$ offrent des performances semblables. Comme le calcul de $\Delta\tau = 0$ requiert moins de temps, cette valeur a été retenue.

$$\tau_{rs}(t+1) = \rho_{local}\tau_{rs}(t) + (1-\rho_{local})\Delta\tau \qquad (2.7)$$

Pour résumer les modifications apportées à l'AS, on établit l'algorithme détaillé de l'OCF. On y observe quatre nouveaux éléments.

En (a), le choix de la prochaine ville est fait selon la règle proportionnelle-pseudo-aléatoire.

Dans la même étape, en (b), on observe l'utilisation d'une liste de candidats.

La mise à jour locale se trouve en (c) et en (d) la mise à jour globale est effectuée par la meilleure solution globale.

Le lecteur peut se référer au Tableau 2.1 pour la notation de l'OCF.

La seule différence est la duplication de la persistance de la trace ρ en ρ_{local} et ρ_{global} pour les mises à jour locale et globale de la trace.

Les valeurs habituellement données aux paramètres de l'OCF ne sont pas les mêmes que pour l'AS. Dorigo et Gambardella [Dor97b] estiment qu'à l'exception de τ_o, les valeurs données aux paramètres sont largement indépendantes de la taille du problème. Le nombre de fourmis *m* doit désormais être fixé expérimentalement [Dor99a]. Ils ont proposé une formule pour calculer le nombre idéal de fourmis. Les auteurs concluent que *m* devrait être près de 10.

Alors que ρ devait être près de 0,5 pour l'AS, il en va autrement pour l'OCF. Plus les valeurs de ρ_{local} et ρ_{global} sont près de 1 sans l'égaler, meilleurs sont les résultats, et souvent $\rho_{local} = \rho_{global}$. Les valeurs de α *et* β déterminées pour l'AS sont conservées pour l'OCF. La valeur de Q_o a été fixée à 0,9, ce qui favorise l'exploitation de l'espace des solutions. Après expérimentations, la quantité de la trace initiale τ_o est affectée à une valeur égale à $(h.L_{oe})^{-1}$, où *h* est la taille du problème et L_{oe} est la longueur estimée du chemin optimal. Cette estimation est donnée par l'heuristique du plus proche voisin *{nearest neighbor heuristic).*

Algorithme 2.2 Pseudo-code de l'OCF [Dor97b]

1. $t = 0$
   Initialiser chaque arête (r, s) à la valeur de τ_0

2. $y = 1$
   POUR chaque fourmi $k = 1$ à m *{Une fourmi sur chaque ville}*
   Placer la fourmi k sur la ville k et affecter $tabou_k(y) = k$

3. **POUR** $y = 2$ à h
   POUR $k = 1$ à m
   Choisir une ville s parmi les lc villes candidates (a)(b)
   selon l'Équation 2.5
   Mettre à jour τ_{rs} selon l'Équation 2.7 (c)
   $tabou_k(y) = s$

4. **POUR** chaque fourmi $k = 1$ à m
   Calculer la longueur L_k du chemin décrit par k
   Mettre à jour la meilleure solution globale

5. **POUR** chaque arête $(r, s) \in tabou_{gb}$ *{Mise à jour globale}*
   Mettre à jour τ_{rs} selon l'Équation 2.6 (d)

6. $t = t + 1$
   SI $(t < NC_{MAX})$ **ALORS**
   Vider toutes les listes tabou
   Aller à l'étape 2
   SINON
   ARRÊTER

Finalement, l'ajout d'une méthode de recherche locale est abordé. Les auteurs concluent que leur méthode est comparable aux autres métaheuristiques, et veulent par cet ajout la rendre compétitive par rapport aux méthodes élaborées spécifiquement pour le TSP.

3. Recherche bibliographique sur l'application d'algorithmes de fourmis

Depuis leur introduction en 1991, les algorithmes de fourmis ont subi de nombreuses modifications dans le but d'en améliorer la performance ou de les adapter à des problèmes particuliers. Cette section aborde les principales modifications apportées au cours des années. L'hybridation des fourmis avec la recherche locale se montre souvent efficace. Plusieurs auteurs ont greffé leur colonie d'une procédure d'amélioration locale exécutée avant la mise à

jour de la trace. Mais l'hybridation la plus complète est probablement celle de Gambardella, Taillard et Dorigo, le HAS-QAP, pour « *Hybrid Ant System for the Quadratic Assignment Problem* » [Gam99]. L'idée consiste à donner une solution de départ à chaque fourmi et laisser celle-ci se déplacer dans l'espace de solutions à l'aide d'une structure de voisinage basée sur la trace de phéromone accumulée. La mise à jour est uniquement globale et un mécanisme effaçant périodiquement toutes les traces de phéromone est utilisé pour éviter une convergence trop rapide.

Maniezzo [Man99a] propose une modification de la règle de transition visant à réduire le temps de calcul et à éliminer un paramètre. La règle donnée à l'Équation 2.4 est modifiée pour obtenir l'Équation 2.8. Cette règle de transition est équivalente à la précédente : elle permet au décideur de donner une importance relative à la trace par rapport à la visibilité locale en plus d'éliminer un paramètre et d'utiliser des opérateurs moins exigeants en termes de temps.

$$P_{rs}^{k} = \begin{cases} \dfrac{\alpha.\tau_{rs} + (1-\alpha).\eta_{rs}}{\sum_{w \notin \text{tabou}_k} \alpha.\tau_{rw} + (1-\alpha).\eta_{rw}} & \text{si} \quad s \notin \text{tabou}_k \\ 0 & \text{sinon} \end{cases} \qquad (2.8)$$

Gagné, Gravel et Price [Gag01] ont adapté l'OCF à un problème d'ordonnancement d'une machine unique *(Single Machine Scheduling)* à objectifs multiples. Dans un premier temps, les auteurs modifient la règle de transition en utilisant une matrice de distances pour chaque objectif et donc une visibilité locale pour chacun. De plus, l'algorithme est bonifié d'une fonction d'anticipation *(lookahead function).* Cette fonction reflète la qualité potentielle de la solution en cours de construction et est insérée à la suite des visibilités locales dans la règle de transition. Son calcul prend en considération la partie construite, l'élément à ajouter et une borne inférieure de l'évaluation de la partie non complétée.

Une procédure d'amélioration locale est ajoutée et appliquée aux meilleures solutions trouvées. Après avoir conclu en évaluant le potentiel de ces améliorations [Gag02a], les auteurs utilisent l'algorithme pour la résolution de problèmes de nature industrielle sur lesquels ils avaient travaillé par le passé [Gra02]. Plusieurs auteurs ont travaillé avec succès sur la parallélisation d'algorithmes de fourmis. Notons entre autres Bullhheimer *et al.* [Bul98] et Talbi *et al.* [Tal01].

Nombre d'auteurs proposent des modifications relatives à la gestion de la trace de phéromone.

Stùtzie et Hoos [Stu97a] introduisent le MAX-MIN Ant System dans lequel la force de la trace de phéromone est restreinte à un intervalle donné. Des bornes inférieure et supérieure pour la trace sont établies en fonction de la taille de l'instance à résoudre. Le but de cette limitation est d'éviter la convergence et la stagnation trop rapide observée lorsque la mise à jour est faite uniquement par la meilleure fourmi du cycle.

Taillard et Gambardella [Tai97b] proposent le *Fast Ant System* (FANT). Il s'agit d'un OCF avec recherche locale, possédant trois particularités: (1) Une seule fourmi est utilisée, ce qui peut être considéré simplement comme une valeur spécifique de paramètre; (2) La trace de phéromone est traitée uniquement avec des nombres entiers et modifiée en fonction des améliorations de la meilleure solution globale ; et (3) II n'y a pas d'évaporation de la phéromone à chaque cycle, mais plutôt une réinitialisation occasionnelle pour éviter la stagnation.

Merkle et Middendorf [Mer00] présentent une nouvelle manière d'utiliser la trace de phéromone pour l'adapter au problème de retard total pondéré sur machine unique *(single machine total weighted tardiness problem).* Le problème consiste à ordonner des commandes en minimisant le retard total généré, où le retard total est une somme pondérée des unités de temps de retard de chaque commande par rapport à sa date de livraison prévue. Les auteurs utilisent une trace qui associe une position de la séquence à une commande et les mises à jour sont faites de manière standard.

Beaucoup d'auteurs ont présenté des algorithmes dérivés de l'OCF original. La majeure partie des modifications apportées concerne évidemment la trace de phéromone, surtout la façon de la mettre à jour et les valeurs que peuvent prendre ses éléments. Peu ou pas de chercheurs se sont arrêtés à modifier la structure même de la trace de phéromone, tous utilisent une matrice à deux dimensions et en adaptent le contenu au problème traité.

On aborde quelques problèmes d'optimisation fréquemment traités à l'aide des algorithmes de fourmis.

- Le *problème d'affectation quadratique (Quadratic Assignment Problem - QAP)* est également souvent résolu à l'aide d'algorithmes de fourmis. Le but de la résolution est de produire une permutation qui minimise le coût total d'affectation, qui est calculé selon les distances entre les endroits et le coût de transport entre les activités.

Maniezzo et Colorni [Man99c] présentent un algorithme d'OCF performant dans lequel une méthode d'amélioration locale est exécutée sur chaque solution, la trace est accumulée pour chaque affectation activité-endroit et la visibilité locale associant une activité à un endroit est représentée par une borne inférieure du coût d'une solution contenant cette affectation. Taillard et Gambardella [Tai97a] ont appliqué l'OCF au QAP d'une manière différente, (à l'aide de l'HAS-QAP). La trace de phéromone est la même et une procédure d'amélioration locale est également ajoutée.

- Le routage des messages dans les réseaux de télécommunication représente également un domaine d'application important de l'OCF. Il s'agit d'un problème multi-objectifs stochastique et distribué qui consiste à établir des tables de routage qui minimisent le temps nécessaire à l'envoi de messages entre des paires de nœuds et balancent le trafic sur le réseau *(Load Balancing)*. Pour la résolution par OCF, des fourmis sont lancées sur le réseau à intervalles plus ou moins constants, à partir d'un point de départ et avec une destination aléatoire. Il existe une matrice de phéromone pour chaque nœud qui est en fait une matrice de probabilités de taille $((\textit{nombre}\ \text{de nœuds} - 1) \times (\textit{nombre}\ \text{de nœuds} - 1))$, où pour chaque nœud de destination, il existe une probabilité de router le message vers chaque nœud connecté. Lorsqu'une case de la matrice est modifiée, au moins une case correspondant à un autre nœud connecté doit l'être également pour conserver des probabilités valables. La matrice d'un nœud est mise à jour par les fourmis qui y passent. Schoonderwoerd *et al.* ([Sch97a], [Sch97b]) ont utilisé ce principe de dépôt de phéromone dans un algorithme nommé *l'Ant-Based Control* (ABC).

- En terminant, des algorithmes de fourmis ont été appliqués à beaucoup d'autres problèmes pour lesquels de bons résultats ont été obtenus. Parmi ces problèmes se trouvent le problème d'ordonnancement séquentiel *(sequential ordering problem)* [Gam97], le problème de tournée de véhicules *(véhicule routing problem)* ([Bul99b], [Bul99c]), le problème d'ordonnancement sur une machine unique *(single machine scheduling)* (uni et multi-objectifs) [Gag02b] [Gag02a] [Gra02] et le problème d'optimisation de l'aménagement du clavier *(Optimization of keyboard arrangement problem)* [Egg03], pour n'en nommer que quelques-uns.

4. Conclusion

Une analyse de la bibliographie concernant l'OCF a été faite. Les concepts clés, la formalisation et les principaux efforts de la communauté scientifique y ont été abordés. Les

métaheuristiques représentent un domaine en pleine expansion et un intérêt en raison de leur facilité relative d'adaptation et de leur excellent compromis qualité-temps. Elles fournissent des solutions de très bonne qualité, à un coût moindre que celui des méthodes exactes.

Dans le chapitre suivant, nous proposons une adaptation d'un algorithme de fourmis pour la résolution du problème difficile d'ordonnancement $P \mid prec \mid C_{max.}$

CHAPITRE 3 :

ADAPTATION D'UN ALGORITHME DE FOURMIS POUR LA RESOLUTION DU PROBLEME D'ORDONNANCEMENT DIFFICILE $P|prec|C_{max}$

1. Introduction

Le problème d'ordonnancement sur des machines parallèles identiques à contraintes de précédence noté $P|prec|C_{max}$ est classé NP difficile [Bru98]. En général de tel ordonnancement s'effectue en deux phases : l'affectation des tâches aux machines et l'exécution des tâches sur une machine. Une résolution exacte n'est en général efficace que si le nombre de tâches est réduit.

Notre objectif est de minimiser la longueur d'ordonnancement ou la date de fin d'exécution de la dernière tâche. Ce critère est important car l'ordonnanceur assure l'équilibre de la charge entre les machines. Nous ne considérons que les ordonnancements non-préemptifs. Une fois une tâche débute son exécution, elle ne peut être interrompue. Le temps requis pour exécuter une tâche donnée est connu d'avance et ne dépend pas de la machine utilisée. Des relations de précédence peuvent exister entre les tâches.

En ordonnancement parallèle, les algorithmes les plus utilisés sont de types listes. Ils déterminent pour un ordre de tâches données par une liste, l'ordonnancement correspondant. Ils considèrent les tâches une à une et prennent la décision d'ordonnancer sur la base d'un ordonnancement partiel de tâches ordonnancées auparavant.

Pour beaucoup de problèmes difficiles et dans une grande variété de domaines, les métaheuristiques ont reçu un intérêt considérable et se sont avérées efficaces dans des domaines variés tels industriel, économique, logistique, ingénierie, commerce, domaines scientifiques, etc.

- Phelipeau-Gelineau [Gel96] a développé un algorithme de recherche Taboue pour résoudre le problème $P|prec|C_{max}$.
- Aytug et al. [Ayt03] fournissent un état de l'art sur l'utilisation des algorithmes génétiques dans la résolution des problèmes d'ordonnancement.
- Boumédiène et Derbala [Bou06] ont étudié et comparé six algorithmes de type liste avec un algorithme génétique.

Des exemples de nouvelles métaheuristiques sont les algorithmes évolutionnistes de type *colonies de fourmis*. Ces algorithmes s'inspirent du comportement des fourmis réelles qui communiquent entre elles à l'aide d'une substance chimique appelée la phéromone [Dor99]. Dans un algorithme d'Optimisation de Colonie de Fourmis (OCF), deux informations heuristiques statiques et dynamiques sont indispensables. Une règle d'affectation est à entreprendre pour le séquencement des tâches.

Dans le paragraphe 2, une adaptation d'un algorithme de colonies de fourmis est exposée. Initialement, on applique l'algorithme d'OCF qui nous fournit une meilleure affectation de tâches aux machines. La seconde étape consiste à exécuter les tâches en face de chaque machine selon trois règles de priorité, le plus long chemin (PLC), une variante de PLC (VPLC) et le maximum entre PLC et la durée totale des tâches dans la descendance (MAX).

Avec ces trois règles et les deux informations heuristiques statiques et dynamiques, six versions de cette méthode de colonies de fourmis ont été étudiées et comparées. Sa justification et sa finitude sont exposées dans le paragraphe 3. Dans le paragraphe 4, son implémentation est discutée. Dans le paragraphe final, de nombreuses expérimentations ont été effectuées sur des données de problèmes générées aléatoirement. La méthode de colonie de fourmis statique munie de la règle VPLC a fourni de bonnes longueurs d'ordonnancement. Son temps de calcul est légèrement plus important que celui des autres versions.

2. L'algorithme d'OCF adapté

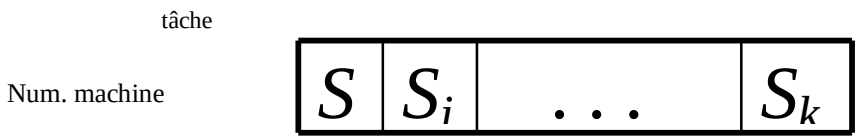
« N » tâches ($j = 1…n$) sont à exécuter sur « m » machines parallèles. Supposons que chaque tâche est constituée d'une seule opération qui peut être exécutée sur n'importe quelle machine. Les dates de début d'exécution au plus tôt r_j sont nulles. Le temps d'exécution d'une tâche est p_j unités de temps.

Le Tableau 3.1 résume les termes de la notation utilisée pour l'algorithme d'OCF adapté. Les variables utilisées et les paramètres dont les valeurs à fixer, sont décrits.

Variable	Description
$\tau_{ij}(t)$	la valeur de phéromone représentant le composant de la variable *i (tâche)* de valeur *j (machine)* à l'itération ***t.***
t	Compteur d'itérations.
q	une variable aléatoire distribuée uniformément sur [*0,1*].
X_{ij}	la variable *i* a flippée (tâche) de valeur *j* (machine).
$P_{X_{ij}}^{K}(t)$	Probabilité pour la fourmi *k* de choisir la variable X_{ij} à l'itération t.
$\eta_{ij}(t)$	la valeur de l'information heuristique à l'itération *t*.
J^k	l'ensemble de tous les couples (tâche, machine) non tabous (qui ne sont pas encore affectés).
Paramètre	**Description**
Col	Nombre de fourmis.
Max_Iter	Nombre maximal d'itérations.
q_0	paramètre de l'algorithme pris entre *0* et *1*.
τ_0	une quantité de phéromone initiale.
α	Exposant de la trace de phéromone
β	Exposant de l'information heuristique.
ρ	persistance de la trace de phéromone comprise entre 0 et 1.

Tableau 3.1 Notation de l'algorithme d'OCF adapté.

Une solution ou ordonnancement est une affectation des tâches aux machines. Chaque solution *S* est un vecteur d'entiers à « *n* » composantes. Chaque composante s_i est associée à une tâche et indiquera le numéro de la machine sur laquelle elle s'exécutera.



$i, j, k \in [0, m-1]$

L'obtention d'une nouvelle solution S_2 à partir d'une solution S_1 se fait par un *flip* d'une seule composante de S_1. Le *flip* d'une composante fixée d'un vecteur représente le changement du numéro de la machine dans cette composante.

Une stratégie de l'OCF pour résoudre le problème $P\ |prec|C_{max,}$ consiste à faire démarrer chaque fourmi d'une solution initiale complète générée aléatoirement, et de lui appliquer des flips répétitifs afin de construire une solution finale (stratégie d'amélioration).

Algorithme 3.1 OCF à stratégie d'amélioration.

(1) *t = 1*.
Initialiser la trace phéromone.
(2) *Pour* chaque fourmi *k =1 à col*
(2.a) Générer une solution S_0.
(2.b) Améliorer la solution S_0, pour construire S_k.
(2.c) Evaluer la solution S_k
(2.d) Appliquer la mise à jour « locale » de phéromone.
(3) Sélectionner la meilleure solution dans cette itération
(4) Appliquer la mise à jour « globale » de phéromone.
(5) *t := t+1*.
Si ($t \leq$ Max_*Iter*) *alors* Aller à l'étape **(2)**
Sinon Sélectionner la meilleure solution globale.
Arrêter
(6) Ordonnancer les tâches selon une règle de priorité.

La trace de phéromone est implémentée en utilisant une *table-phéromone* où le nombre de lignes est égal au nombre de tâches et le nombre de colonnes est égal au nombre de machines. Toutes les entrées de cette table sont initialisées à une valeur réduite $\tau_0 > 0$.

Nous rappelons que le temps d'exécution sur une machine j est donné par :

$$temps_{M_j} = \sum_{j=1}^{NM_j} P_j$$

NMj est le nombre de tâches qui s'exécutent sur la machine *Mj*.

Le temps d'exécution parallèle est donné par le maximum des temps d'exécution sur les *m* machines :

$$Temps_{par} = \max_{\{j,\, j\in\{1,\ldots,m\}\}} \left(Temps_{Mj}\right)$$

La fonction d'évaluation utilisée pour une solution donnée, notée *F*, est une combinaison de deux fonctions de performance, le temps d'exécution parallèle des tâches $Temps_{par}$ et de la précédence entre ces tâches. L'information heuristique η, appelée aussi *visibilité*, peut être dans deux états. Le premier où elle est à priori disponible pour exécution. Cette situation est la plus fréquente dans les problèmes statiques où sa valeur est fixée. Dans le second état, elle est rendue disponible durant l'exécution. C'est la situation typique dans des problèmes dynamiques où elle évolue selon la fonction d'évaluation *F* ci-dessus définie.

Une version d'un algorithme d'OCF est un algorithme statique ou dynamique munie d'une règle de priorité.

Les méthodes de liste sont des méthodes itératives où à chaque étape on complète une solution partielle en cherchant à faire le choix le plus avantageux. Le choix effectué à une itération est définitif. On s'interdit de le remettre en cause au cours des étapes ultérieures.

Pour déterminer la séquence des tâches sur chaque machine, nous avons utilisé trois meilleures listes de priorité [Bou06].

- *En fonction des plus longs chemins* [Jac55]. Pour chaque tâche *i*, on calcule la longueur du plus long chemin *PLC (i)* de cette tâche jusqu'à une tâche sans successeur. *PLC (i)* se calcule de la manière suivante :

$$PLC(i) = \begin{cases} P_i & \text{i : sans successeurs} \\ \max\limits_{j \text{ successeur de i}} \{PLC(j) + p_i\} \end{cases}$$

- En fonction d'une variante du plus long chemin VPLC définie par l'expression suivante :

$$VPLC(i) = \begin{cases} P_i & \text{i : sans successeurs} \\ \sum\limits_{j \text{ successeur de i}} \{VPLC(j) + p_i\} \end{cases}$$

- En fonction de la valeur de l'expression suivante : $\max\limits_i \left\{ PLC(i), \dfrac{descendance(i)}{m} + p(i) \right\}$.

  Les descendants d'une tâche sont les tâches qui sont successeurs, ou successeurs du successeur, etc.

  Cette règle permet de faire intervenir la valeur du plus long chemin et la durée totale des tâches dans la descendance.

3. Justification et finitude de l'algorithme 3.1

- L'étape **(1)** initialise le compteur d'itérations et la trace de phéromone sur la table-phéromone ; elle représente l'utilisation précédente de cet élément.
- Une itération commence à l'étape **(2)**. Dans cette étape et pour chaque fourmi k, on affecte les tâches aléatoirement ou selon une stratégie bien définie aux machines (l'étape **(2.a)**).
- A l'étape **(2.b)**, chaque fourmi essaye d'améliorer son affectation en choisissant à chaque étape le composant à flipper dans l'affectation actuelle, selon la règle de transition d'état suivante :

$$\text{si } q < q_0 \qquad X_{ij} = \arg\max_{(i,j)\in J^k} \left[(\tau_{ij}(t))^{\alpha} \times (\eta_{ij}(t))^{\beta} \right]$$

$$\text{si } q > q_0 \qquad P_{X_{ij}}^{K}(t) = \frac{\tau_{ij}(t)^{\alpha} \times [\eta_{ij}(t)]^{\beta}}{\sum_{(l,h)\in J^k} \tau_{lh}(t)^{\alpha} \times [\eta_{lh}(t)]^{\beta}}$$

- Naturellement, la trace de phéromone d'une fourmi s'évapore lentement. Pour accélérer cette évaporation, une persistance de la trace de phéromone est fixée. Une évaporation de phéromone est effectuée, dans la *table-phéromone*, sur le composant choisi par la règle suivante : $\tau_{ij}' = \rho \times \tau_{ij}$.
- Chaque affectation construite est évaluée à l'étape **(2.c)**.
- A l'étape **(2.d)**, on applique la mise à jour locale aux composants de la solution finale, selon la règle : $\tau_{ij}' = \tau_{ij} + (1-\rho) \times \tau_0$.
- A l'étape **(3)**, une fois que toutes les fourmis ont achevé la construction de leurs solutions, on choisit la meilleure solution trouvée.
- A l'étape **(4)**, il y a une mise à jour globale de la trace, qui suit la règle : $\tau_{ij}' = \tau_{ij} + (1-\rho) \times \frac{1}{F^*}$ avec *F**, l'évaluation de la meilleure solution.
- A l'étape **(5)**, le compteur d'itérations est incrémenté et on décide de poursuivre ou non l'exécution. On sélectionne la meilleure affectation globale.
- Pour déterminer la séquence des tâches sur chaque machine, on utilise une liste de priorité à l'étape **(6)**.
- On arrête l'algorithme quand le nombre total d'itérations atteint une valeur fixée.

4. Implémentation

Evaluons l'algorithme de résolution du problème $P|prec|C_{max}$. Pour la méthode colonies de fourmis, les deux informations heuristiques (statique & dynamique) ont été définies.

Pour chacune de ces deux informations, les trois règles d'affectation adoptées pour construire l'ordonnancement ont été utilisées (PLC, VPLC et Max). Nous présentons donc 6 versions de cette méthode : Sta_PLC, Dyn_PLC, Sta_VPLC, Dyn_VPLC, Sta_MAX et Dyn_MAX.

Sta_MAX signifie que l'information heuristique est statique et que la règle de priorité utilisée est *MAX*. Ces versions sont testées sur des bancs d'essais (benchmarks), générées

aléatoirement par un outil de simulation. Ce générateur permet d'avoir des instances de problème $P|prec|C_{max}$, dont les temps d'exécution sont uniformément distribués dans l'intervalle [a, b] avec 0<a<b.

Le générateur fournit un graphe de précédence aléatoire, sans circuit, de densité arbitraire dans] 0 ,1[.

4.1 Critères de performance

Pour apprécier la qualité des méthodes, il est important de comparer les résultats obtenus avec la durée d'un ordonnancement optimal, ce qui est en général impossible à déterminer. Pour comparer les différentes versions de la méthode colonies de fourmis, utilisons la notion de performance relative [Bou06]. Pour une instance donnée, elle est définie comme le rapport entre la durée de l'ordonnancement calculé par cette méthode et la quantité calculée par la relation max (LB_1, LB_2), avec :

$$LB_1 = \max_{\text{i sans prédécesseur}} PLC(i) \qquad LB_2 = \frac{\sum_i p_i}{m}$$

La performance relative moyenne d'une méthode est la moyenne de ses performances relatives sur un ensemble d'instances donné.

4.2 Choix des paramètres

Dans la littérature, les auteurs conseillent d'utiliser $\tau_0 = (n\,L)^{-1}$ où « *n* » est le nombre de sommets du graphe associé au problème et « *L* » la longueur d'une tournée trouvée par la méthode du plus porche voisin dans ce graphe. Dans notre cas, τ_0 est de 0,1.

Le nombre de fourmis est un paramètre important. Il doit désormais être fixé expérimentalement [Dor99].

En ce qui concerne les paramètres α et β, qui accordent une importance aux différents éléments dans la construction des solutions, ils sont fixés respectivement à 1et 2.

La valeur de q_0 a été fixée à 0,8, alors que ρ devait être près de 0,5.

5. Tests et résultats généraux

Une instance d'un problème est définie quand toutes les données lui sont associées.

Les temps d'exécution de tâches et le graphe de précédence entre les tâches constituent un exemple de données d'un problème $P|prec|C_{max}$. Un échantillon est constitué d'un nombre

d'instances. Nous rappelons que le critère de performance pris en considération est le rapport relatif moyen défini ci-dessus. Le paramètre influençant ce rapport relatif moyen est la densité du graphe de précédence des tâches. La densité d'un graphe $\Gamma = (X, U)$ a été défini comme étant le rapport entre le nombre d'arcs générés par l'outil de simulation et le nombre d'arcs d'un graphe complet obtenu à partir du même ensemble de sommets X. Elle est souvent exprimée en pourcentage. Plus ce pourcentage est grand, plus le graphe est qualifié de *dense*.

5.1 Influence de la densité du graphe de précédence

De nos expérimentations et pour chaque type de densité, faible, moyenne et élevé, des résultats sont obtenus et représentés graphiquement.

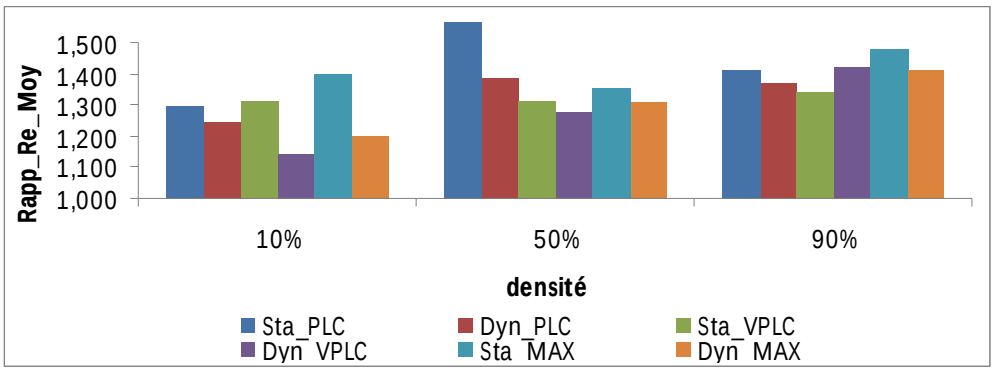


Figure 3.1 Influence de la densité du graphe de précédence pour les versions d'OCF.

Nous remarquons que pour les graphes de faible densité et de densité moyenne la version Dyn_VPLC est la meilleure. Pour les graphes de densité élevée, la version Sta_VPLC est meilleure.

5.2 Influence du nombre de tâches et nombre de machines

Il apparaît à l'issue des expérimentations que le nombre de tâches est un autre paramètre influençant les résultats de la méthode colonies de fourmis. Nous avons remarqué que la version statique basée sur la règle VPLC est la mieux adaptée pour un nombre de tâches < 100 (figure 3.2). Lorsque le nombre de taches est grand, on obtient des résultats très satisfaisants avec les versions Stat_MAX et Dyn_MAX.

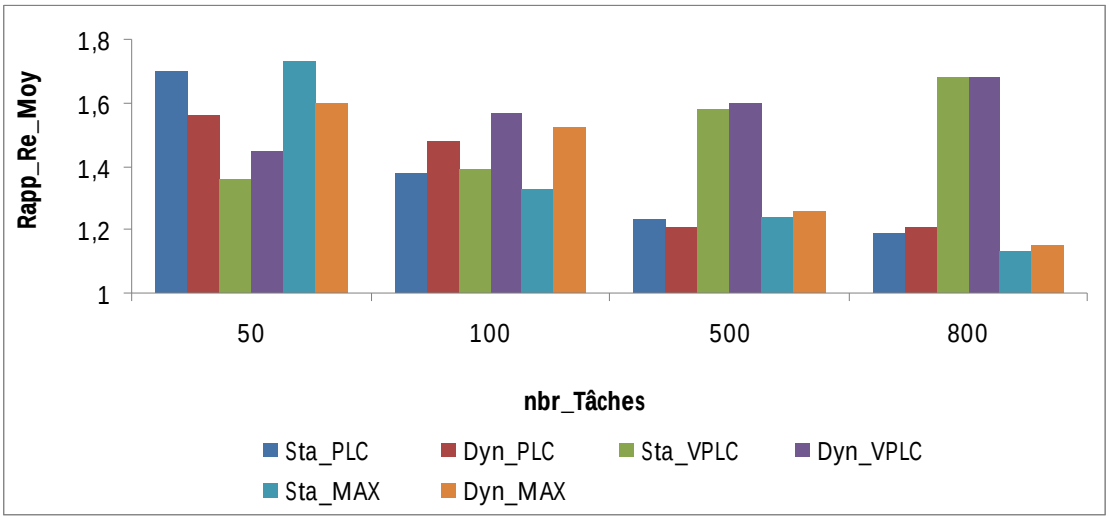


Figure 3.2 Influence du nombre de tâches pour les versions d'OCF.

Concernant l'influence de la variation du nombre des machines, si elle est grande, les performances des versions restent presque constantes. Lorsque cette variation est petite, on obtient des bons résultats (figure 3.3).

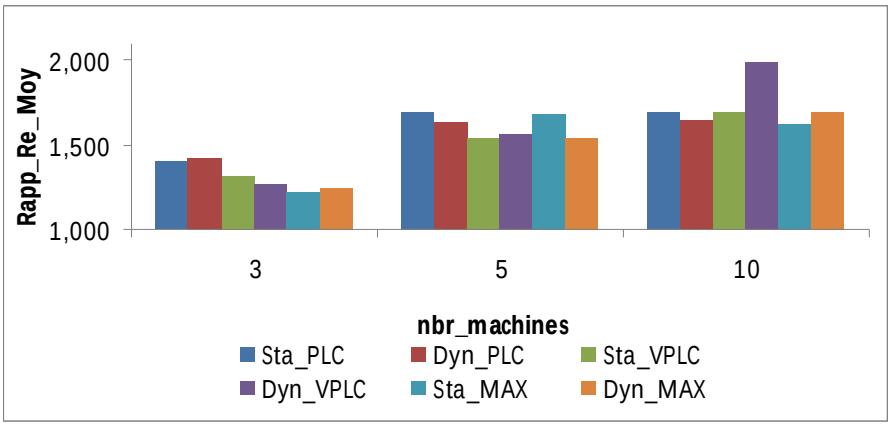


Figure 3.3 Influence du nombre de machines pour les versions d'OCF.

Les expérimentations les plus significatives, à savoir les instances les plus performantes, sont réalisées avec des instances comportant 100 tâches, 2 machines et dont une densité faible du graphe du précédence.

5.3 Cas particulier: Le problème P2 |prec, pj =1 |Cmax

Nous avons aussi considéré les instances qui appartiennent à la classe où le nombre de machine est limité à deux et les temps d'exécution des tâches sont unitaires. L'algorithme de

Coffman & Graham [Cof72] nécessite au plus un temps proportionnel à n^2 pour trouver un ordonnancement de longueur minimale à tel problème. La borne de cet algorithme est $R = 2 - \frac{2}{m}, m \geq 2$. Nous avons étudié les performances de l'algorithme d'OCF proposé dans ce cas particulier.

Ci-joint un tableau de tests réalisés pour le problème $P2 \mid prec, p_j = 1 \mid C_{max}$.

Makespan (C_{max})							
Instance (n_m_densité)	*Sta_PLC*	*Dyn_PLC*	*Sta_VPLC*	*Dyn_VPLC*	*Sta_Max*	*Dyn_Max*	*Coff_Gra*
5_2_10%	3,12	3,14	3,09	3,1	3,33	3,33	3,01
15_2_10%	8,81	8,9	8,9	8,92	8,84	8,77	8,02
25_2_10%	15,1	14,88	15,04	15,02	14,95	14,62	13,05
5_2_50%	3,56	3,73	3,40	3,56	3,71	3,58	3,06
15_2_50%	10,81	10,94	10,92	10,88	10,97	10,79	8,47
25_2_50%	17,62	17,9	17,36	17,51	17,6	17,64	14,09
5_2_75%	4,19	4,11	3,92	4,00	4,04	4,03	3,18
15_2_75%	11,61	11,57	11,39	11,57	11,43	11,33	8,88
25_2_75%	18,46	18,21	18,36	18,27	18,56	18,3	14,67
Temps _CPU (seconds)							
Instance (n_m_densité)	*Sta_PLC*	*Dyn_PLC*	*Sta_VPLC*	*Dyn_VPLC*	*Sta_Max*	*Dyn_Max*	*Coff_Gra*
5_2_10%	0,025	0,030	0,030	0,036	0,039	0,047	<1 ms
15_2_10%	0,097	0,108	0,101	0,116	0,110	0,131	<1 ms
25_2_10%	0,243	0,258	0,245	0,266	0,254	0,276	<1 ms
5_2_50%	0,025	0,029	0,031	0,036	0,039	0,048	<1 ms
15_2_50%	0,096	0,109	0,102	0,117	0,112	0,128	<1 ms
25_2_50%	0,242	0,258	0,253	0,294	0,257	0,293	<1 ms
5_2_75%	0,025	0,029	0,030	0,036	0,040	0,048	<1 ms
15_2_75%	0,096	0,108	0,102	0,116	0,112	0,127	<1 ms
25_2_75%	0,243	0,258	0,244	0,266	0,256	0,277	<1 ms

Tableau 3.2 Résultats expérimentaux du problème $P2 \mid prec, p_j = 1 \mid C_{max}$.

Le tableau3.2 montre l'évolution des longueurs d'ordonnancement (makespan) et les temps de calculs obtenus par l'application des six versions d'OCF et l'algorithme de Coffman & Graham [Cof72], par rapport à la variation des paramètres : nombre de tâches « *n* » et la densité du graphe de précédence pour *m*=2.

Nous remarquons que pour des instances de petites tailles les 6 versions sont très proches en termes de performance avec l'algorithme de Coffman & Graham. Toutefois, les deux versions Sta_VPLC et Dyn_VPLC deviennent meilleures pour des instances de taille moyenne et de densité faible. Par ailleurs, les 6 versions seront moins performantes pour des instances de grandes tailles.

5.4 Synthèse des résultats et conclusion

Un nombre important d'expérimentations, de l'ordre de 1000, a été réalisé. Pour un échantillon de 100 jeux d'essai, les expérimentations sont réalisées avec un nombre ne dépassant pas 800 tâches et 10 machines. Le tableau 3.3 résume les différents résultats expérimentaux obtenus pour les 6 versions d'OCF implémentées.

Temps_CPU (seconds)						
Instance (n_m_densité)	*Sta_PLC*	*Dyn_PLC*	*Sta_VPLC*	*Dyn_VPLC*	*Sta_Max*	*Dyn_Max*
10_3_10%	1,34	1,25	1,34	1,08	1,45	1,08
10_3_50%	1,86	1,34	1,48	1,03	1,48	1,08
10_3_90%	1,31	1,24	1,19	1,21	1,43	1,12
50_3_25%	1,67	1,61	1,33	1,43	1,27	1,31
50_5_25%	1,63	1,87	1,57	1,75	1,77	1,53
100_3_10%	1,27	1,25	1,39	1,14	1,39	1,2
100_3_50%	1,39	1,34	1,19	1,45	1,25	1,5
100_3_90%	1,54	1,24	1,31	1,56	1,63	1,5
50_10_25%	1,41	1,51	1,33	1,72	1,45	1,3
100_3_10%	1,38	1,48	1,39	1,57	1,33	1,52
500_3_75%	1,23	1,21	1,58	1,6	1,24	1,26
800_3_75%	1,19	1,21	1,68	1,68	1,13	1,15
Rapport relatif moyen						
Instance (n_m_densité)	*Sta_PLC*	*Dyn_PLC*	*Sta_VPLC*	*Dyn_VPLC*	*Sta_Max*	*Dyn_Max*
25_3_10%	0,067	0,066	0,07	0,061	0,079	0,058
25_3_50%	0,08	0,063	0,068	0,063	0,073	0,067
25_3_90%	0,72	0,062	0,072	0,06	0,069	0,06
25_5_50%	0,715	0,739	0,703	0,791	0,742	0,779
25_8_50%	1,62	1,68	1,74	1,77	1,72	1,69
100_3_10%	0,76	0,726	0,753	0,753	0,692	0,753
100_3_50%	0,728	0,747	0,729	0,806	0,723	0,784
100_3_90%	0,695	0,755	0,751	0,725	0,77	0,791
50_10_25%	4,34	4,44	4,65	4,73	4,64	4,94
100_3_10%	2,610	2,690	2,540	2,880	2,470	2,950
500_3_75%	62,4	64,2	64,2	66,6	61,2	69,6
800_3_75%	165	181,2	168,6	171	165,6	177,6

Tableau 3.3 .Résultats expérimentaux du problème $P|prec|C_{max}$.

Les résultats sont regroupés dans un graphique ci-dessous et pour chacune des versions. Il est à 3-dimensions. Deux axes d'ordonnées parallèles sont associés aux temps CPU moyen et au rapport relatif moyen. Sur l'axe des abscisses, les 6 versions d'OCF sont représentées.

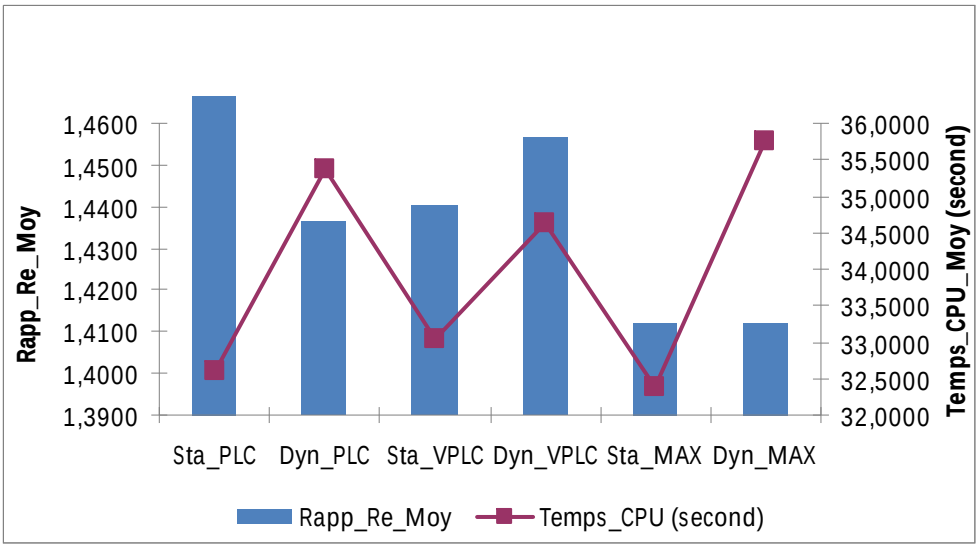


Figure 3.4 Résultats globaux pour les versions d'OCF.

De la lecture de ce graphe, les deux meilleures versions minimales sont reconnues comme étant Sta_MAX et Dyn_MAX. La version de colonies de fourmis statique munie de la règle MAX nécessite moins de temps de calcul. Elle est un bon compromis entre la qualité de la solution obtenue et le temps de calcul.

CHAPITRE 4 :

ETUDE COMPARATIVE DE QUATRE METAHEURISTIQUES POUR LA RESOLUTION DU PROBLEME $P|prec|C_{\max}$

Le problème d'ordonnancement étudié dans ce mémoire $P|prec|C_{\max}$ est prouvé NP-difficile [Bru98]. Une étude comparative de quatre métaheuristiques de types recuit simulé, recherche taboue, un algorithme génétique et une version de colonie de fourmis dénotée STA_VPLC est réalisée [Mes08].

On propose des variantes de ces métaheuristiques fournissant une meilleure affectation de tâches aux machines. L'exécution des tâches en face de chaque machine se fera selon la meilleure liste parmi les trois étudiées au chapitre précédent.

Nous présentons donc trois méthodes qui ont été étudiées et comparées avec la meilleure version de colonie de fourmis.

1. Les méthodes de Liste

Une méthode de liste se caractérise par la règle de priorité adoptée pour construire la liste et la règle d'affectation. Le but de nos expérimentations a été de tester, d'évaluer et de comparer trois règles de priorités PLC, VPLC et Max {,} qu'on rappelle leurs définitions ci-dessous.

- *PLC* [Jac55]. Pour chaque tâche *i,* on calcule la longueur du plus long chemin *PLC (i)* de cette tâche jusqu'à une tâche sans successeur. *PLC (i)* se calcule de la manière suivante :

$$PLC(i) = \begin{cases} P_i & \text{i : sans successeurs} \\ \max\limits_{j \text{ successeur de i}} \{PLC(j) + p_i\} \end{cases}$$

- VPLC définie par l'expression suivante :

$$VPLC(i) = \begin{cases} P_i & \text{i : sans successeurs} \\ \sum\limits_{j \text{ successeur de i}} \{VPLC(j) + p_i\} \end{cases}$$

- MAX définie par l'expression suivante : $\max\limits_i \left\{ PLC(i), \dfrac{descendance(i)}{m} + p(i) \right\}$.

Les descendants d'une tâche sont les tâches qui sont successeurs, ou successeurs du successeur, etc.

La règle d'affectation utilisée est celle sans délai. Elle ne laisse pas les machines oisives si des tâches sont prêtes à s'exécuter.

Pour chaque une des trois listes, le but serait de pouvoir comparer les résultats donnés par ces listes avec la durée d'un ordonnancement optimal. Dans ce cas l'écart entre les solutions de type Liste et l'optimum ou la distance à l'optimum en pourcentage sera déterminée. Sachant qu'en général la valeur de l'optimum C_{max} est inconnue, une borne inférieure *LB* de cette valeur optimale sera utilisée.

LB est définie par la relation max (LB_1, LB_2), avec :

$$LB_1 = \max_{\text{i sans prédécesseur}} PLC(i) \qquad LB_2 = \frac{\sum_i p_i}{m}$$

Le rapport d'approximation expérimental est donnée par la formule : $\rho(I) = \frac{C_{\max}^{L(I)}}{LB(I)}$ avec:

- I : une instance,
- $C_{\max}^{L(I)}$: Solution obtenue par une Liste L,
- *LB (I)* : La borne inférieure.

Une analyse expérimentale sur un échantillon de 100 instances a été menée. Pour chaque une d'elle, on calcule le rapport $\rho(I)$. Les résultats obtenus sont représentés par l'allure graphique dans un repère orthogonal.

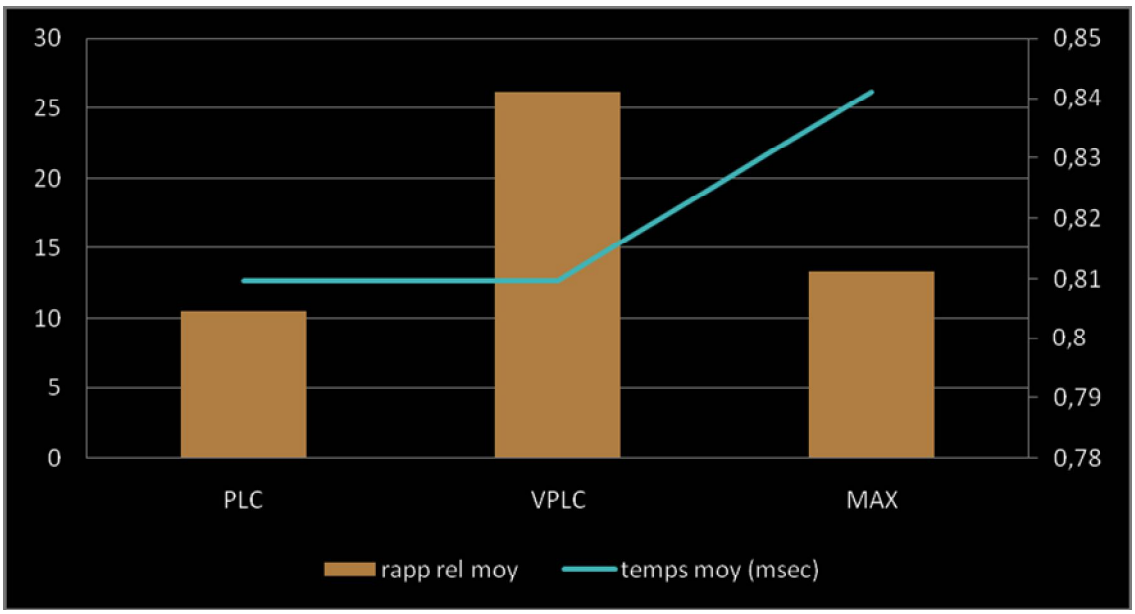


Figure 4.1 Performance des listes associées à une règle d'affectation sans délai.

De ce comportement graphique, les deux meilleures listes sont reconnues comme étant

PLC et MAX. La liste PLC nécessite moins de temps de calcul. Sur chaque instance de l'échantillon, on calcule le nombre de fois au la meilleure longueur est obtenue pour chaque liste. Les résultats sont donnés dans le diagramme de performance.

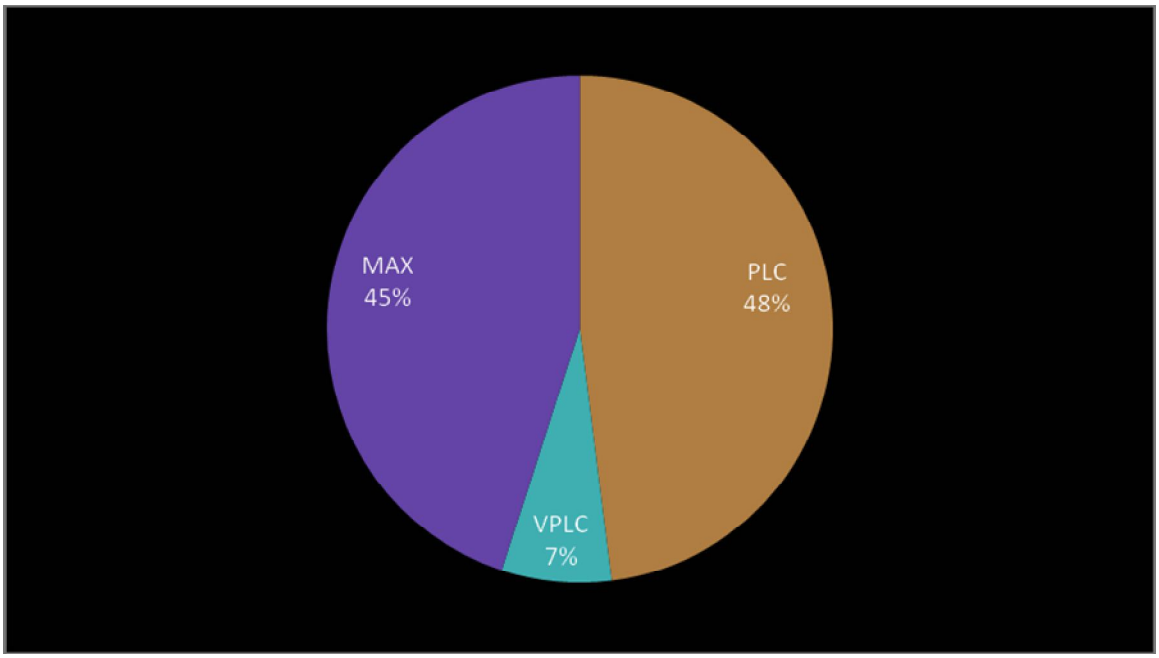


Figure 4.2 Pourcentage de performance des listes.

La règle PLC réalise 48 % de meilleurs cas. La règle VPLC est moins performante pour le problème $P|prec|C_{\max}$.

Le pourcentage où la règle MAX est efficace est de 45 %.

Pour la résolution de notre problème $P|prec|C_{\max}$, nous confirmons graphiquement que la règle PLC est meilleure parmi les trois listes.

2. Description et Implémentation d'un recuit simulé (RS)

Le recuit simulé a été inventé par Kirkpatrick, Gelatt et Vecchi en *1983*. Ils ont pu résoudre quasi-optimalement des problèmes de voyageur de commerce à *5000* sommets. Ils s'inspirent de la méthode de simulation de Métropolis (*1950*) en mécanique statistique.

L'analogie s'inspire du recuit des métaux en métallurgie. Un métal refroidi trop vite présente de nombreux défauts microscopiques, c'est l'équivalent d'un minimum local pour

un *POC*. Si on le refroidit lentement, les atomes se réarrangent, les défauts disparaissent et le métal a une structure ordonnée, équivalent du minimum global pour un *POC*.

L'énergie d'un système est représentée par un réel *T*, la température. Une méthode de recuit simule s'obtient :

- On tire au sort une transformation, une solution *s'* de *V(s)* au lieu de chercher la meilleure ou la première solution voisine améliorante.
- On construit la solution résultante *s'* et sa variation de cout $\Delta f = f(s') - f(s)$.
- Si $\Delta f \leq 0$, le cout diminue et on affecte la transformation améliorante comme dans une recherche locale ($s = s'$).
- Si $\Delta f > 0$, le cout remonte, c'est un rebond, qu'on va pénaliser d'autant plus que la température est basse et que Δf est grand. Une fonction exponentielle a les propriétés désirées. On calcule une probabilité d'acceptation $a = exp(-\Delta f / T)$, puis on tire au sort une valeur *p* de *[0,1]*. Si $p \leq a$, la transformation est acceptée bien qu'elle dégrade le cout, et on fait $s = s'$. Sinon, la transformation est rejetée : on conserve *s* pour l'itération suivante.

Pour assurer la convergence, *T* est diminuée lentement à chaque itération, par exemple $T = KT$, $K<1$ mais proche de un. On peut aussi décroitre *T* par paliers.

On s'arrête quand *T* atteint un seuil fixe ε, proche de zéro. Le réglage des paramètres est assez délicat. Il est prudent de prévoir deux tests d'arrêt supplémentaires : un limitant le nombre d'itérations à une valeur *MaxIter*, et un limitant le nombre d'itérations sans changement de cout à une valeur *MaxGel* : *MaxGel* doit être assez grand.

Contrairement à une recherche locale, un recuit simulé donne une solution finale qui n'est pas la meilleure trouvée.

Il vaut mieux stocker en cours de route toute solution améliorante. Les valeurs typiques sont : *MaxIter* = 10000, ε = 0,01, *MaxGel* = 200, *K* = 0,9995

Algorithme 4.1 Algorithme Général du Recuit Simulé

- Construire une solution initiale quelconque *s*

  z= f(s)* {meilleur cout obtenu}

  s=s* {meilleur solution connue}

- Initialiser *T*, ε, *MaxIter* et *MaxGel*

  NIter := 0 {nombre d'itérations}

  NGel := 0 {nombre d'itération sans améliorations}

- ***Répéter***

  NIter :=NIter + 1

  Tirer au sort une solution *s'* dans *V(s)*

  Calculer la variation de cout Δf

  Si $\Delta f \leq 0$ ***Alors*** *Accept :=vrai*

  Sinon Tirer au sort *p* dans *[0,1]*

  Accept :=vrai si $p \leq exp(-\Delta f / T)$

  FS

  Si *Accept* ***alors*** Si $\Delta f = 0$ alors *NGel :=NGel+1*

  Sinon *Ngel :=0*

  Si *f(s') < z** ***alors*** *z* :=f(s'), s* :=s'*

  FS

  s :=s'

  FS

  T:=KT

  Jusqu'à (*T*≤ ε) ***ou*** (*NbIter =MaxIter*) ***ou*** (*NGel=MaxGel*)

3. Description et Implémentation d'une Recherche Taboue (RT)

3.1 Principe

Les recherches taboues ont été inventées par Glover en 1985. Elles sont de conception plus récente que le recuit simulé, n'ont aucun caractère stochastique et paraissent meilleures à temps d'exécution égal. Elles sont caractérisées par trois points :

- A chaque itération, on examine complètement le voisinage *V(s)* de la solution actuelle *s*, et on va sur la meilleure solution *s'*, même si le cout remonte.

- On s'interdit de revenir sur une solution visitée dans un passé proche grâce à une liste taboue *T* stockant de manière compacte la trajectoire parcourue. On cherche donc *s'* dans *V(s)-T*.

- On conserve la meilleure solution trouvée, contrairement au recuit, c'est rarement la dernière.

On stoppe après un nombre maximal d'itérations sans améliorer la meilleure solution ou quand *V(s)-T= {}*. Il ne se produit que sur de très petits problèmes pour lequel le voisinage tout entier peut se retrouver enferme dans *T*.

Une méthode taboue échappe aux minima locaux, même si *s* est un minimum local, l'heuristique va s'échapper de la région *V(s)* en empruntant un « *col* ».

En début de calcul, la méthode trouve une suite de solutions améliorées, comme une recherche locale. On voit ensuite le cout osciller puis redescendre vers un meilleur minimum local. Les améliorations deviennent de plus en plus rares au cours des itérations.

3.2 La liste taboue

Le point délicat est la capacité *NT* de la liste taboue *T*. Glover montre que *NT* de 7 à *2*0 suffit pour empêcher les cyclages, quelle que soit la taille du problème. *T* fonctionne comme une « mémoire à court terme ».

A chaque itération, la *NT* ième transformation de *T*, la plus ancienne, est écrasée par la dernière effectuée. En pratique, *T* se gère simplement avec une structure de données de type « file ». En théorie, il faudrait stocker les *NT* dernières solutions sur lesquelles l'algorithme est passé. En ordonnancement, il faudrait conserver les *NT* dernières permutations des *n* taches. Si la solution actuelle est *s*, la prochaine solution sur laquelle on va se déplacer doit être dans *V(s)-T*. Il faut vérifier que la permutation générée n'est pas déjà dans *T*.

Les voisinages considérés étant souvent grands, il est évident que le test répétitif de présence dans *T* est très couteux, sans parler de la mémoire nécessaire pour coder le détail de *NT* solutions. Le stockage explicite des solutions n'est donc jamais pratique.

On recommande les critères moins couteux pour éviter le cyclage. Ces critères sont aussi plus restrictifs : ils peuvent interdire certains mouvements qui ne conduiraient pas à un cyclage. Les plus utilisés de ces critères consiste à stocker dans *T* les transformations ayant permis de changer de solution, et d'interdire de faire les transformations inverses pendant NT itérations, on stockerait les deux taches permutées à chaque itération, et on s'interdirait de les

remettre dans une permutation pendant NT itérations. Un critère plus simple est d'interdire d'utiliser les NT dernières valeurs de la fonction objective. Il suffit de stocker un nombre par itération.

Algorithme 4.2 Algorithme Général de la Recherche Taboue

- Construire une solution initiale quelconque *s*

  z = f(s)=z* {meilleur cout obtenu}

  s =s* {meilleur solution connue}
- Initialiser *MaxIter* {nombre maximum d'itérations}

  T = {} {la liste taboue est vide}
- *NIter :=0*
- ***Répéter***

  NIter =NIter + 1

  z'' = + ∞

  Pour toute solution *s'* de *V(s)*

  Si *s'* n'est pas dans *T* ***alors***

  Si *f(s') < z''* ***alors*** { mise a jour du voisin « le moins pire »}

  S''=s' , z''=f(s')

  Stocker la transformation dans *u*

  FS

  FS

  FP

  Si z''<+ ∞ ***alors*** {si un voisin non tabou a été trouve}

  s=s'' , z=z''

  Enlever la transformation en tête de *T* {la plus ancienne}

  Ajouter *u* en fin de *T*

  Si *z <z''* ***alors*** {mise à jour de la meilleure solution}

  s=s , z*=z*

  FS

  FS

  Jusqu'à (*NbIter =MaxIter*) ***ou*** (*z''=+ ∞*)

4. Description et Implémentation d'un Algorithme Génétique (AG)

Cette classe de méthodes a été inventée par Holland dans les années soixante, pour imiter les phénomènes d'adaptation des êtres vivants. L'application aux problèmes d'optimisation a été développée ensuite par Goldberg. Par analogie avec la reproduction des êtres vivants, on part d'une population initiale de N ordonnancements réalisables. Il faut coder chaque solution, un ordonnancement réalisable, comme une chaîne de caractères par analogie à un chromosome d'une cellule vivante. Le chromosome est formé de sous chaînes appelées " gènes ", chacun codant une caractéristique de la solution.

Une itération est appelée " génération ". A chaque génération, on choisit au hasard NC paires de chromosomes à reproduire, NC < N, avec une probabilité croissante de leur adaptation. Chaque paire d'ordonnancements (x, y) choisie subit une opération de "croisement" (crossover). Un gène est choisi aléatoirement dans x, puis permuté avec le gène de même position dans y. On pratique également un faible " taux de mutation " : NM chromosomes sont choisis et subissent une modification aléatoire d'un gène. La nouvelle population est formée de la population précédente et des chromosomes nouveaux générés par mutation ou croisement.

On définit pour chaque solution une " mesure d'adaptation au milieu " appelée " fitness ". Pour l'ordonnancement, cette fonction est sa longueur. On élimine alors les solutions les moins adaptées pour maintenir un effectif constant de N solutions. On recommence le même processus à l'itération suivante. L'intérêt est que les bonnes solutions sont encouragées à échanger par croisement leurs caractéristiques et à engendrer des solutions encore meilleures.

De plus, pour un POC, problème d'optimisation combinatoire, les solutions finales vont être concentrées autour des minima locaux, et la méthode fournira un choix de plusieurs solutions possibles au décideur.

Notons G la fonction d'évaluation choisie. Pour engendre la génération $X^{(n+1)}$ de solutions à partir de la population courante $X^{(n)}$, on applique le schéma décrit ci-dessous :

- Sélectionner dans $X^{(n)}$ un ensemble de paires de solutions de hautes qualités.
- Appliquer à chacune des paires de solutions sélectionnées un opérateur de croisement qui produit une ou plusieurs solutions " enfants ".
- Remplacer une partie de $X^{(n)}$ formée de solutions de basse qualité par des solutions " enfants " de bonne qualité;

- La population $X^{(n+1)}$ est obtenue après avoir appliqué un opérateur de mutation aux solutions ainsi obtenues.

Algorithme 4.3 Algorithme Génétique en général

Initialisation : soit $X^{(0)} \subseteq X$, une population initiale;

Etape n : soit $X^{(n)} \subseteq X$, la population courante;

- sélectionner dans $X^{(n)}$ un ensemble de paires de solutions da haute qualité;
- appliquer à chacune des paires de solutions sélectionnées un opérateur de croisement qui produit une ou plusieurs solutions " enfants ".
- remplacer une partie de $X^{(n)}$ formée de solutions de basse qualité par des solutions enfants de haute qualité;
- appliquer un opérateur de mutation aux solutions ainsi obtenues; les solutions éventuellement mutées constituent la population $X^{(n+1)}$;

Si la règle d'arrêt est satisfaite, stop;

Sinon, passer à l'étape n + 1.

4.1 Codage

Un ordonnancement est fournit par la liste des tâches qui le compose. Ce codage de l'ordonnancement est indirect. Les numéros associés aux tâches ont été utilisés. Nous confondons ordonnancement et liste selon l'ordonnancement. La permutation de tâches représentant la liste est le chromosome, il représente la solution. Une population sera donc un ensemble de permutations de « n » tâches.

Pour construire la population initiale, on utilise les trois règles PLC, VPLC et Max {,}.

4.2 La sélection

La sélection aussi bien celle des individus de " haute qualité " que celle des individus de " basse qualité ", comporte généralement un aspect aléatoire. Chaque individu x_i de la population parmi laquelle se fait la sélection se voit attribuer une probabilité p_i d'être choisi d'autant plus grande que son évaluation est haute (basse dans le cas d'une sélection de " mauvais individus "). On tire un nombre " r " au hasard (uniformément sur [0, 1]). L'individu " k " est choisi tel que : $\sum_{i=1}^{k-1} p_i < r_i \leq \sum_{i=1}^{k} p_i$.

La probabilité que x_k soit choisi est aussi bien égale à p_k. Cette procédure est appelée " la roulette russe ". Elle est itérée jusqu'à ce que l'obtention d'une population de taille désirée.

4.3 Le croisement

Soit deux solutions x et y sélectionnées parmi les solutions de haute qualité. Un opérateur de croisement (crossover) fabrique une ou deux nouvelles solutions x' et y' en combinant x et y.

Si x et y sont deux vecteurs de 0 et 1, un opérateur de croisement classique (two-point crossover) consiste à sélectionner aléatoirement deux positions dans les vecteurs et à permuter les séquences de 0 et 1 figurant entre ces deux positions dans les deux vecteurs.

Pour des vecteurs x = 0 1 1 0 1 1 0 0 et y = 1 1 0 0 1 0 1 0, si les positions " après 2 " et " après 5 " sont choisies, on obtient après croisement :

x' = 0 1 | 001 | 100 et y' = 1 1 | 1 0 1 | 0 1 0.

De nombreuses variantes d'un tel opérateur peuvent être imaginées. Ces variantes doivent être adaptées au codage des solutions et favoriser la transmission des " bonnes sous-structures " des solutions parents aux enfants. Pour le problème d'ordonnancement avec le codage liste des tâches, le two-point crossover ne convient pas.

Si x = A B C D E F G H et y = B E F H A D G C, en croisant entre les positions " après 2 " et " après 5 ", on obtiendrait : x' = AB | FHA | FGH et y' = BE | CDE | DGC qui ne sont pas des permutations.

Nous avons utilisé un opérateur de croisement spécialement conçu pour les listes données l'OX-crossover. Les deux solutions parentes sont " préparées " avant l'échange des séquences situées entre les deux positions choisies au hasard.

Dans cet exemple de tâches, la zone d'échange de x est préparée à accueillir la séquence des tâches F, H, A de y. Pour ce faire, on remplace chacune des tâches F, H et A dans le vecteur x par une place vide symbolisée par une *, soit *B | CDE |*G* et en commençant à la droite de la zone d'échange, on tasse les tâches restantes de la permutation dans l'ordre de la permutation x en oubliant les *, ce qui donne : DE | ***| GBC. Les * se retrouvent dès lors dans la zone d'échange, alors que l'ordre de parcours des autres tâches n'a pas été changé. On procède de même pour y : B* | FHA |*G* devient HA | ***| GBF. On procède alors à l'échange des séquences, ce qui donne deux permutations enfants x' et y' : x' = DE | FHA | GBC et y' = HA | CDE |GBF.

Un certain nombre de paires d'enfants sont ainsi générés et remplacent une partie des parents choisis parmi les moins performants.

4.4 La Mutation

Une mutation est une perturbation introduite pour modifier une solution individuelle, par exemple la transformation d'un 0 en un 1 ou inversement dans un vecteur binaire.

Dans l'ordonnancement, une mutation peut correspondre à une *transposition* de deux tâches. En général, on décide de muter une solution avec une probabilité assez faible de 0,1. Le but de la mutation est d'introduire un élément de diversification et d'innovation.

4.5 Critère d'arrêt

Comme toute procédure itérative, un AG s'arrête si une certaine condition d'arrêt est vérifiée. La plus utilisée est lorsqu'un impératif de temps de calcul est imposé ou lorsque la performance de l'algorithme est à tester. Un nombre de générations est produit signifie qu'un pourcentage de l'espace des solutions est exploré.

5. Analyse expérimentale de RS, RT & AG

Le recuit simulé, la recherche taboue et l'algorithme génétique sont appliqués aux benchmark de 100 instances générés aléatoirement avec différents jeux de paramètres. Les temps d'exécution sont distribués uniformément sur un intervalle [a, b] et avec un nombre ne dépassant pas 100 tâches et 10 machines. Les résultats obtenus lors de ces simulations sont recensés dans les tableaux ci dessous.

Soulignons que les temps d'exécution concernent uniquement les traitements effectifs des différents algorithmes, et excluent les opérations d'affichage, et d'entrées/sorties.

5.1 Le recuit simulé

Le tableau 4.1 présente les résultats obtenus par l'application du recuit simulé aux différentes instances avec différents paramètres. En dépit de la variation opérée sur les paramètres dans les différentes exécutions, toutes les simulations effectuées avec l'algorithme du recuit simulé produisent des résultats durant un temps d'exécution mesurable.

Instance	Nbr d'itérations	Température Initiale	Température Min (ε).	K	Temps d'exéc. Global (Sec.)	Err Rel Moy
25_3_10%	100	*10*	0,01	0,9995	17,643	1,592
25_3_50%	100	*10*	0,01	0,9995	17,054	1,903
25_3_90%	100	*10*	0,01	0,9995	17,538	1,543
25_5_50%	100	*10*	0,01	0,9995	17,891	1,941
25_8_50%	100	*10*	0,01	0,9995	18, 380	1,476
25_10_50%	100	*10*	0,01	0,9995	18,646	1,397
50_3_10%	70	*10*	0,1	0,9995	31,939	1,569
50_3_50%	70	*10*	0,1	0,9995	32,295	1,293
50_3_90%	70	*10*	0,1	0,9995	33,492	1,176
100_3_50%	70	*10*	0,1	0,9995	111,023	1,117
100_3_90%	70	*10*	0,1	0,9995	111,001	1,082

Tableau 4.1 Résultats expérimentaux du RS.

5.2 La recherche taboue

Les résultats obtenus par l'application de la recherche taboue avec différents jeux de paramètres et différentes stratégies sont résumés au tableau 4.2. La stratégie de génération du voisinage est identique pour toutes les simulations : c'est le changement d'une position binaire par solution (simple truth value change) [Ben02].

Instance	Nbr d'itérations	Taille Max Liste Tabou	Stratégie génération	Stratégie sélection	Temps d'exéc. Global (Sec.)	Err Rel Moy
25_3_10%	50	*5*	STVC	RANDOM	2,502	1,926
25_3_50%	50	*12*	STVC	RANDOM	4,497	1,303
25_3_90%	50	*7*	STVC	RANDOM	3,482	1,081
25_5_50%	50	*12*	STVC	RANDOM	7,176	1,563
25_10_50%	50	*5*	STVC	RANDOM	10,28	1,791
50_3_50%	50	*7*	STVC	RANDOM	13,836	1,29
50_3_90%	50	*7*	STVC	RANDOM	13,836	1,284
100_3_10%	50	*7*	STVC	RANDOM	26,866	1,200
100_3_50%	50	*5*	STVC	RANDOM	24,839	1,457
100_3_90%	50	*12*	STVC	RANDOM	29,784	1,199

Tableau 4.2 Résultats expérimentaux de RT.

La taille la plus intéressante pour la liste taboue se situe à 7. Une plus grande taille n'apporte rien de plus, par contre, une augmentation du temps de calcul. En revanche, une liste plus petite entraîne l'apparition de cycles, empêchant l'algorithme de converger vers une meilleure solution comme l'illustre la figure 4.3.

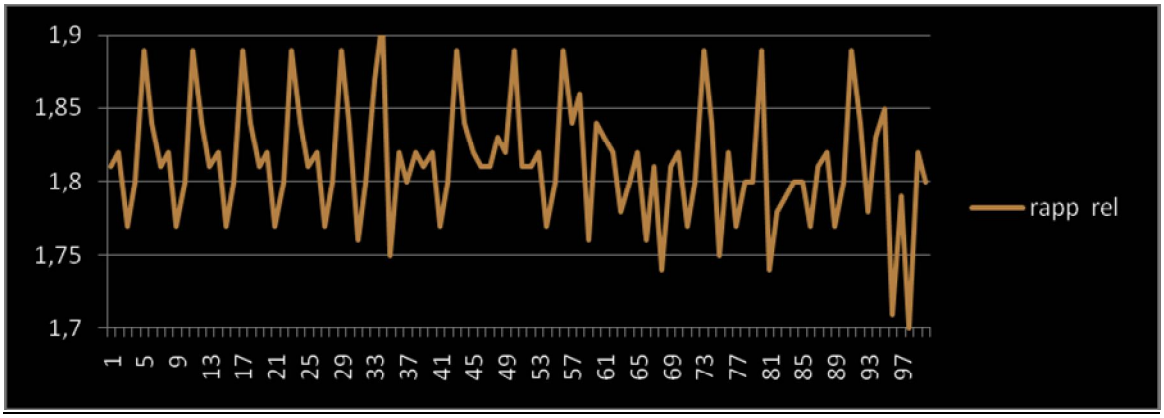


Figure 4.3 Simulation avec une taille de liste taboue réduite.

5.3 L'algorithme génétique

Le tableau 4.3 résume des résultats obtenus en utilisant des algorithmes génétiques avec différentes stratégies et différents jeux de paramètres. La méthode de sélection utilisée pour tous les tests est "la roue de la fortune" [Bou06]. Il est à noter que, dans tous les cas, la population initiale est générée aléatoirement.

Instance	Nbr Générations	Taille population	Proba. Cross-Over	Proba. Mutation	Temps d'exéc. Global (Sec.)	Err Rel Moy
25_3_10%	100	*100*	1 - CRP	1 - CRP	10,502	1,104
25_3_50%	60	*100*	0,9 - CRP	0,1 - CRP	10,497	1,221
25_3_90%	100	*100*	0,9 - CRP	0,7 - CRP	13,482	1,156
25_5_50%	100	*100*	0,5 - CRP	0,5 - CRP	14,176	1,186
25_10_50%	60	*100*	0,9 - CRP	0,1 - CRP	12,28	1,212
50_3_50%	25	*100*	0,9 - CRP	0,1 - CRP	14,836	1,365
50_3_90%	100	*100*	0,7 - CRP	0,7 - CRP	22,836	1,127
100_3_10%	60	*100*	0,9 - CRP	0,1 - CRP	31,866	1,269
100_3_50%	100	*100*	0,7 - CRP	0,7 - CRP	44,839	1,178
100_3_90%	25	*100*	0,9 - CRP	0,9 - CRP	29,784	1,321

Tableau 4.3 Résultats expérimentaux de l'AG.

La stratégie Children Replace Parents (les enfants remplacent les parents) [Ben02] consiste à remplacer tous les anciens individus par ceux obtenus à la nouvelle génération, évitant ainsi que certaines solutions ne soient reproduites trop fréquemment, conduisant la recherche vers un optimum local.

L'utilisation de cette stratégie permet de sortir des simulations effectuées avec ces paramètres, de l'optimum local et de trouver une solution réalisable.

L'influence de l'agrandissement de la population sur le temps d'exécution est évidente et facile à comprendre, les opérations sur chaque génération étant répétées un plus grand nombre de fois. La figure *4.4* illustre un exemple d'exécution utilisant les algorithmes génétiques.

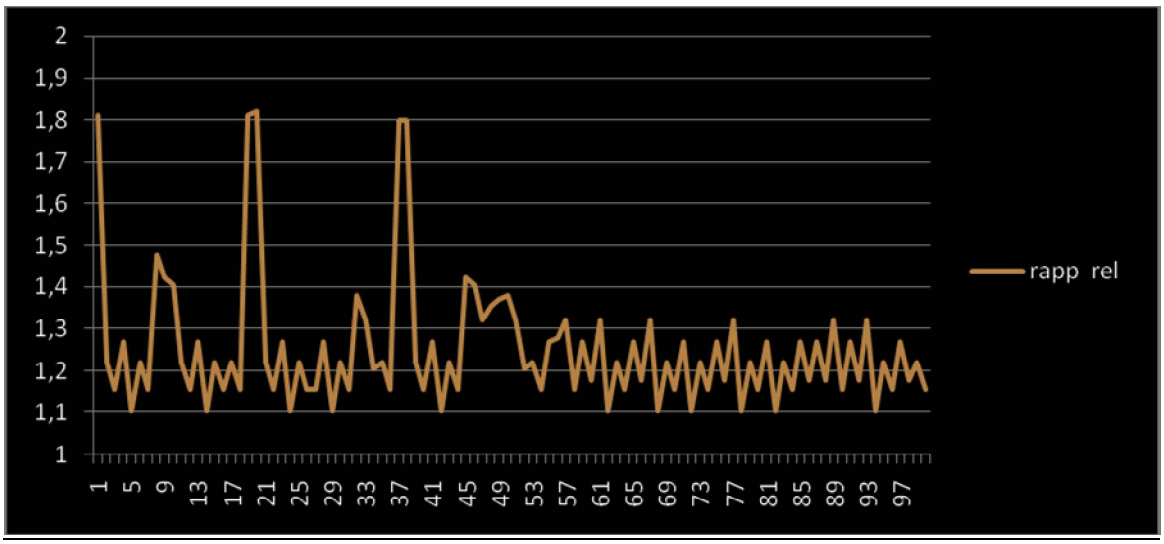


Figure 4.4 Exemple d'exécution des algorithmes génétiques.

6. Comparaison de la version STA VPLC & les trois variantes

Afin d'obtenir une comparaison globale des quatre métaheuristiques *OCF, RS, RT et AG*, nous déroulons ces méthodes pour *100* instances à *3* machines et *90%* de densité. Le nombre de tâches prend les valeurs discrètes suivantes : *5, 10, 25, 50, 75 et 100.*

Le premier critère de comparaison est le temps d'exécution donné par chacune des méthodes citées. Ces temps de réponse sont récapitulés dans le tableau ci-dessous.

Méthodes / Nbr Tâches	*OCF*	*RS*	*RT*	*AG*
5	*0,054*	*4,31*	*0,846*	*3,867*
10	*0,102*	*9,835*	*1,739*	*8,125*
25	*0,630*	*17,538*	*7,176*	*14,176*
50	*1,843*	*33,492*	*13,836*	*22,836*
75	*4,067*	*65,125*	*20,701*	*31,129*
100	*7,572*	*111,001*	*29,784*	*44,839*

Tableau 4.4 Temps d'exécution en fonction du nombre de tâches.

Les résultats montrent que la version *STA_VPLC* est meilleures que les trois autres méthodes, puisqu'elle donne des solutions satisfaisantes dans un temps raisonnable.

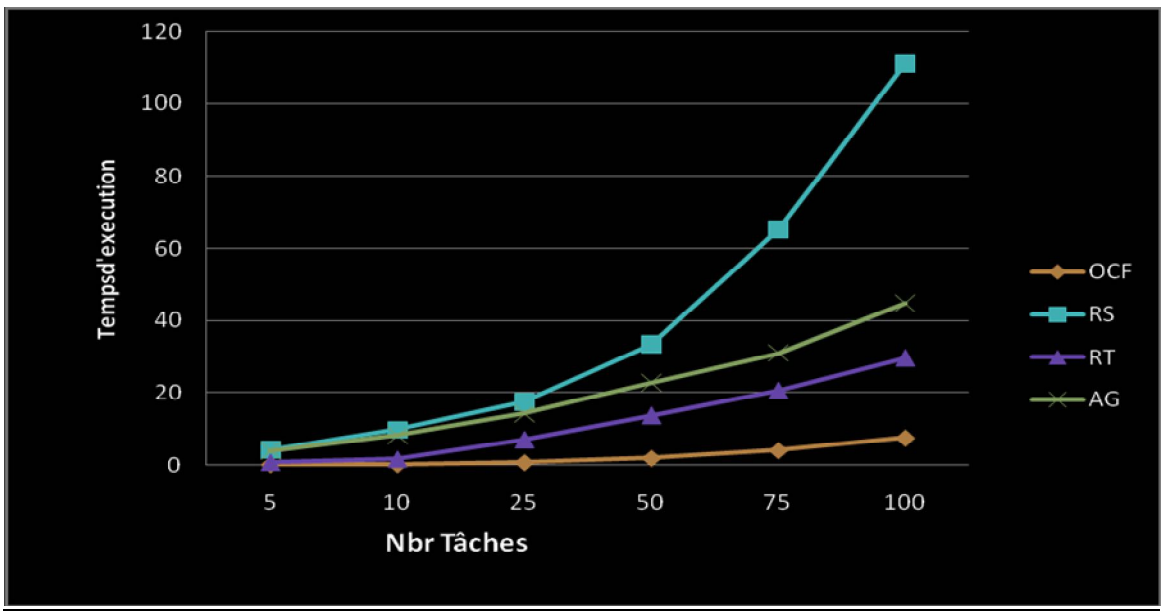


Figure 4.5 Graphe comparatif de méthodes : temps d'exécution

Graphiquement, si le nombre de tâches est inférieur à 10, toutes les méthodes citées ont des temps d'exécution presque identiques. Dans ce cas aucune de méthodes n'est à préférer. Si le nombre de tâches dépasse 25, *STA_VPLC* et *RT* prennent des temps raisonnable. *RS* fournisse des temps de réponse démesurés.

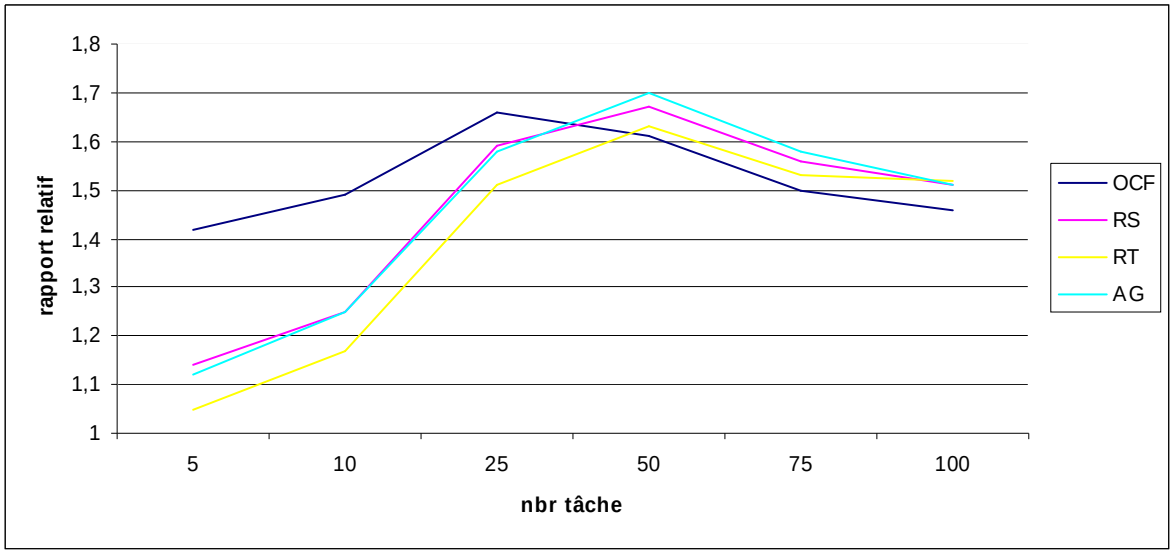


Figure 4.6 Graphe comparatif de méthodes : rapport relatif moyen

De point de vue rapport relatif moyen, si le nombre de tâches est inférieur à 50, la méthode recherche taboue est la meilleure. Si le nombre de tâches dépasse 50, *STA_MAX* fournisse des résultats très satisfaisants.

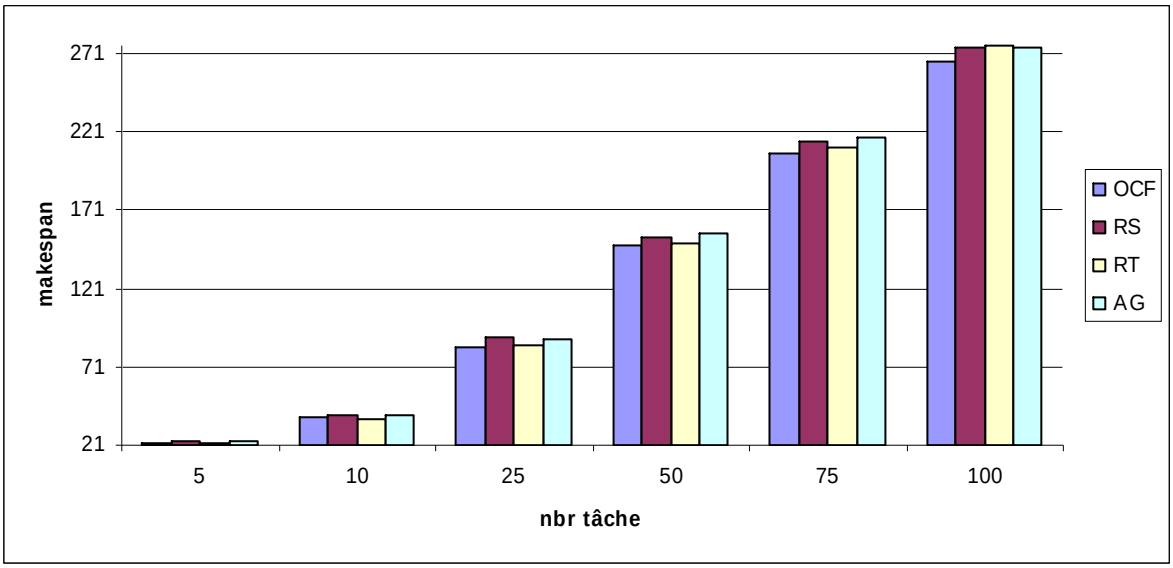


Figure 4.7 Graphe comparatif de méthodes : Makespan

Concernant le dernier critère qui est la longueur d'ordonnancement, et si le nombre de tâches augmente, toutes les méthodes citées ont des Makespan presque identiques.

Les quatre méthodes utilisées dans ce chapitre, permettent d'obtenir des solutions réalisables en des temps acceptables compte tenu du nombre de combinaisons possibles.

Les solutions obtenues à l'aide de la version *STA_MAX* sont, cette fois, meilleure que celle à laquelle aboutissent les algorithmes génétiques et la recherche taboue. Ceci confirme le fait qu'il est impossible de définitivement opter pour une métaheuristique particulière. Les résultats dépendent fortement du type et de la taille du problème abordé. De même, il n'existe pas de paramètres adaptés à l'ensemble des problèmes. La topologie de l'espace des solutions du problème influe fortement sur l'efficacité de certaines stratégies et valeurs de paramètres par rapport à d'autres. Il est à souligner que la nature aléatoire de ces algorithmes, notamment pour les solutions initiales, implique une difficulté supplémentaire. En effet, des simulations différentes avec exactement les mêmes paramètres peuvent produire des résultats différents, pour le même benchmark.

Toutes les considérations présentées à travers les différents tests dans ce chapitre, mettant en relief l'intérêt de disposer d'un simulateur tel qu'il est présenté. Ce simulateur permet de tester rapidement différentes métaheuristiques et d'étudier les effets de chaque paramètre ou stratégie pour résoudre le problème $P|prec|C_{\max}$.

CHAPITRE 5 : PRESENTATION SUCCINCTE DU SIMULATEUR CONFECTIONNE

Pour l'implémentation des algorithmes proposés dans ce mémoire, le simulateur est conçu en utilisant le langage de programmation évolué *Delphi*, version gratuite n°*7*. Il est un environnement de programmation permettant de développer des applications Windows. Il incarne la suite logique de la famille *Turbo Pascal*, avec ses nombreuses versions. Il est un outil moderne, qui fait appel à une conception visuelle des applications.

Le simulateur réalisé a été déroulé sur un processeur Intel Celeron M cadencé à *1.3* GHz et de *760* Mo de RAM. Il est formé principalement de *15* formes que nous présentant brièvement dans la suite.

L'utilisation du simulateur

Après exécution du programme, la fenêtre présentée dans la figure *5.1* apparaîtra. Elle contient trois types de problèmes d'ordonnancement.

Figure 5.1 La forme initiale du Simulateur.

Si on choisit le problème *P/ prec/* C_{max}, une deuxième forme apparaîtrait. Cette forme, présentée dans la figure *5.2*, contient quatre menus principaux.

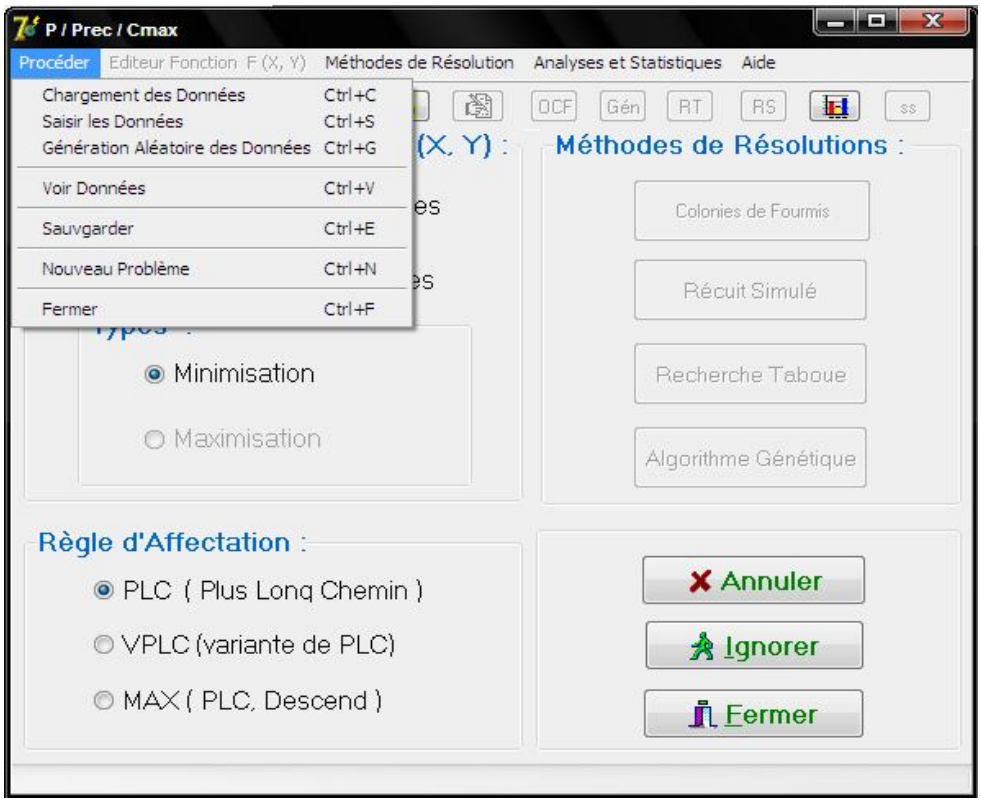
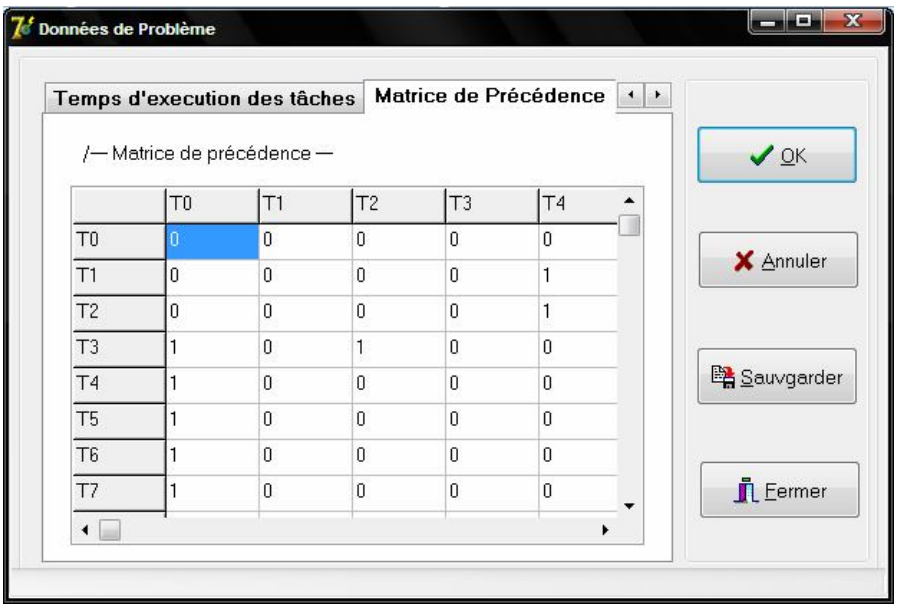


Figure 5.2 Fenêtre principale.

Le menu « Procéder » sélectionné dans l'image, contient : le chargement ou l'importation des données à partir d'un fichier_ORD, la saisie manuelle et la génération aléatoire ou automatique des données du problème. A l'aide d'un éditeur illustré ci dessous, nous pouvons consulter ces données. L'éditeur permet également de sauvegarder les données dans un fichier à extension ORD.



	T0	T1	T2	T3	T4
T0	0	0	0	0	0
T1	0	0	0	0	1
T2	0	0	0	0	1
T3	1	0	1	0	0
T4	1	0	0	0	0
T5	1	0	0	0	0
T6	1	0	0	0	0
T7	1	0	0	0	0

Figure 5.3 Editeur des données.

Il est ensuite nécessaire de sélectionner et paramétrer les caractéristiques utilisées par la fonction « coût », fonction dont l'objectif des métaheuristiques et de la minimiser.

L'utilisateur sélectionne les caractéristiques qui doivent figurer dans la fonction, puis une fonction arithmétique est définie. Elle relie ces caractéristiques à l'aide de l'éditeur illustré à la figure *5.4*. Ce dernier permet aussi de sauvegarder ou de lire une fonction à partir d'un fichier.

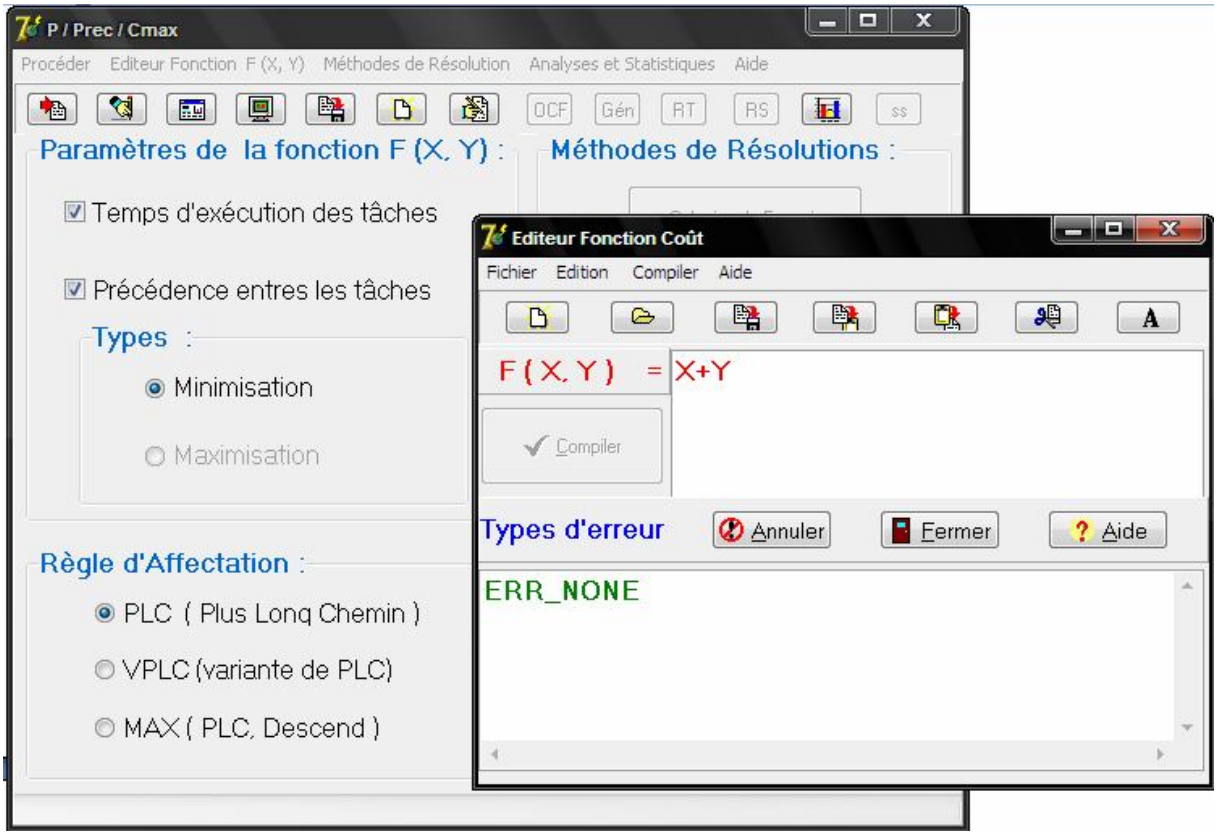


Figure 5.4 Editeur de la fonction objective.

Des opérations mathématiques telles que le logarithme népérien, l'exponentielle, la racine carrée ou les opérateurs arithmétiques simples peuvent être utilisées. La fonction est validée par une étape de compilation qui effectue une analyse syntaxique. Cette dernière figure illustre l'introduction d'une fonction de cout comportant deux paramètres *X* et *Y* qui représentent respectivement le temps d'exécution des taches et la précédence entre elles.

Il est alors nécessaire de choisir la méthode de résolution. Pour cela, quatre listes ainsi quatre métaheuristiques ont été implémentées dans notre simulateur. L'interface de sélections des méthodes de résolution s'affiche ci dessous.

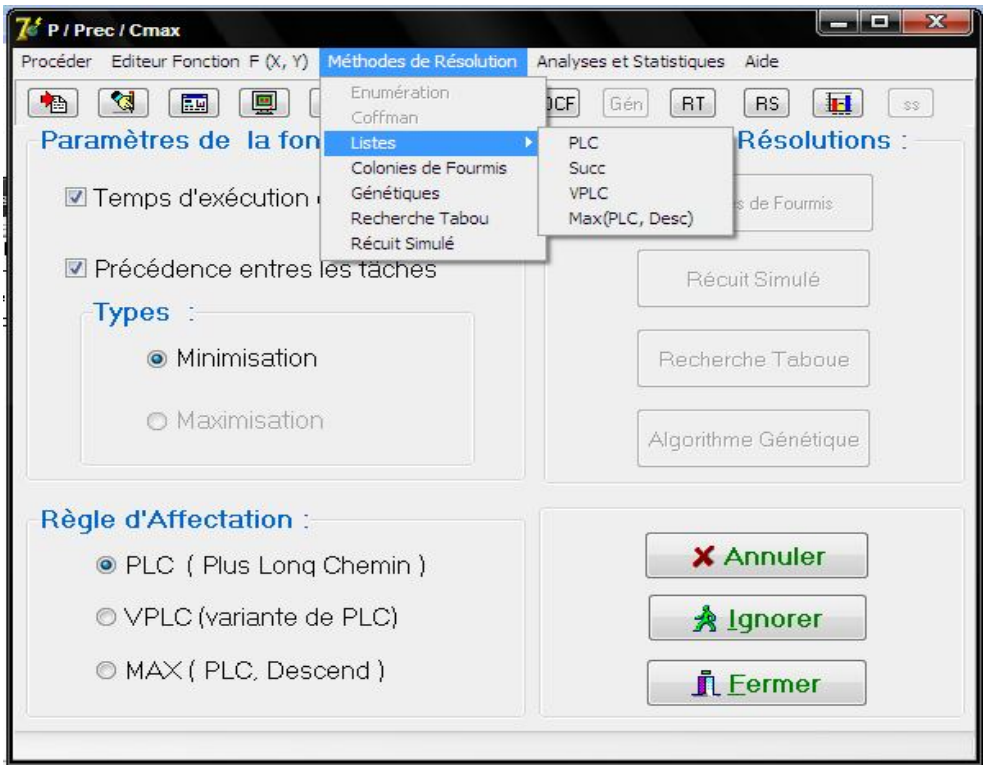


Figure 5.5 Choix d'une méthode de résolution.

La sélection d'une règle d'affectation parmi les trois règles disponibles s'impose.

Nous fixons, pour chaque métaheuristique, les paramètres nécessaires à son exécution. Un exemple d'introduction de paramètres pour la méthode de colonie de fourmis est ainsi fourni.

Figure 5.6 Introduction des paramètres pour la colonie de fourmis.

L'exécution peut alors commencer, et l'utilisateur dispose à la fin de chaque méthode de deux outils, les résultats obtenus et leurs évolutions durant l'exécution, lui permettant de faire

une synthèse. L'ordonnancement final ainsi déterminé et sa longueur sont représentés dans ce diagramme de Gantt. Son temps d'exécution apparaît dans la même fenêtre de représentation du dit diagramme.

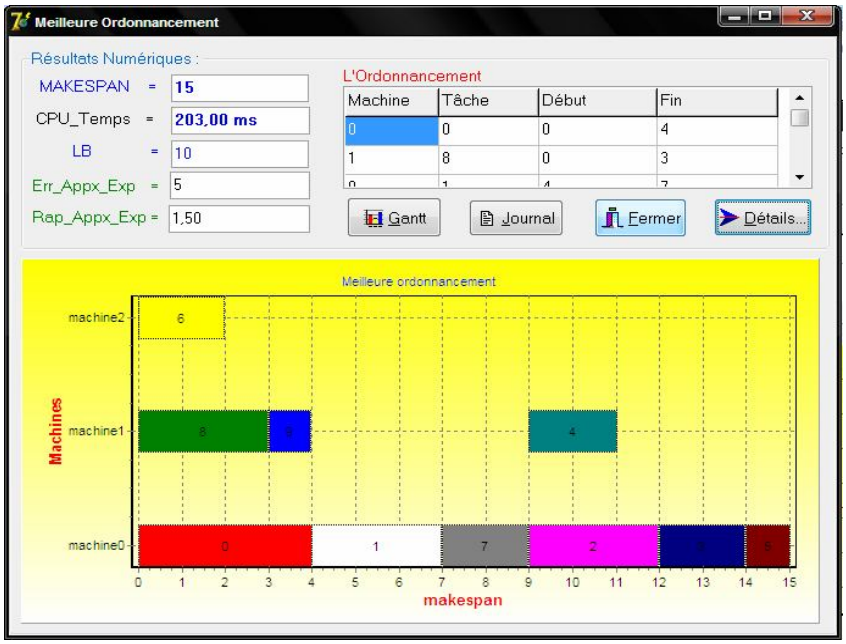


Figure 5.7 L'ordonnancement trouvé par la colonie de fourmis.

L'évolution de cette exécution peut être représentée graphiquement.

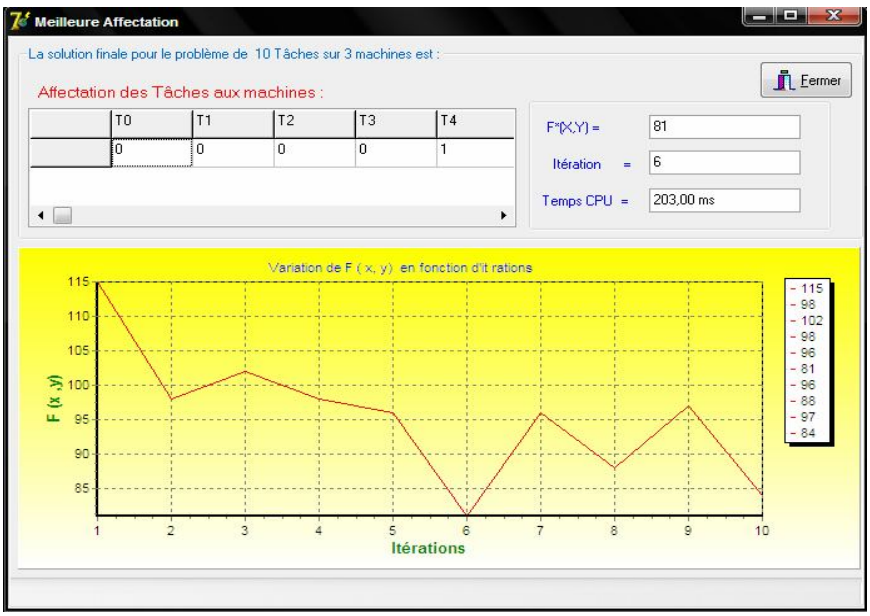


Figure 5.8 Evolution graphique de l'exécution de l'algorithme OCF.

Le dernier menu est celui de l'Analyse et des Statistiques des méthodes de résolution du problème indiqué. Nous pouvons générer aléatoirement des échantillons de ce problème, les faire exécuter par les quatre métaheuristiques implémenter et les comparer.

Une fenêtre illustre un exemple d'exécution de l'algorithme fourmis appliqué à 100 problèmes d'ordonnancement à 10 tâches, 03 machines en parallèle et un graphe de précédence entre les tâches de densité 50 %. La génération des échantillons se fait aléatoirement.

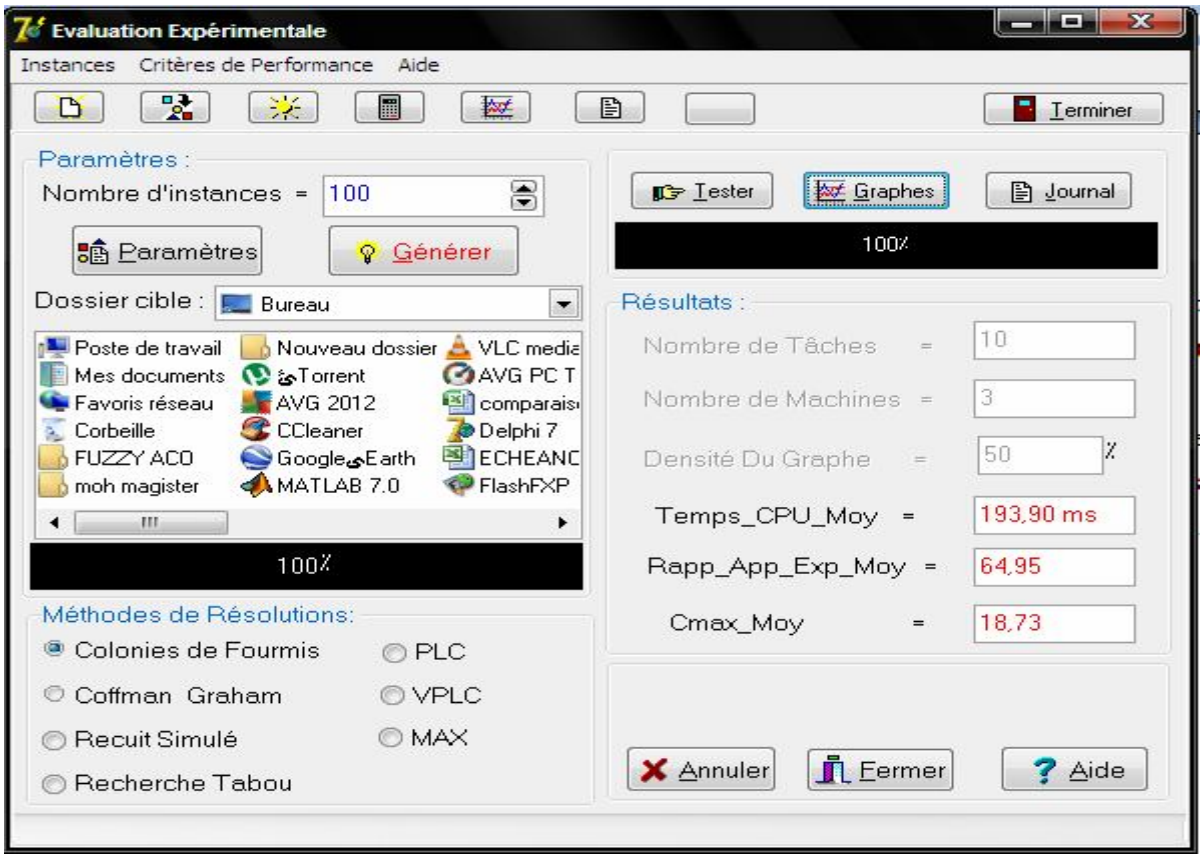


Figure 5.9 Exemple d'une évaluation expérimentale.

A cette étape de développement en programmation, ce simulateur ne peut pas faire l'objet d'adaptation à d'autres classes de problèmes d'ordonnancement. Des améliorations dans l'implémentation, la présentation des problèmes d'ordonnancement à machines en série, en parallèles, dans l'obtention d'une autre interface beaucoup plus conviviale est à entreprendre.

CONCLUSION ET PERSPECTIVES DE RECHERCHE

Après l'étude comparative des quatre métaheuristiques, des problèmes d'élaboration et de confection d'algorithmes d'approximation, du choix des paramètres des métaheuristiques, par exemple, résident encore. S'il faut une étude ultérieure, il faut celle de la domination parmi les métaheuristiques disponibles dans la bibliographie, une étude à projeter dans la confection d'algorithmes où les résultats ne dépendent pas fortement du type et de la taille du problème abordé. Des questions peuvent se poser et sont, par exemple, celle de l'existence des paramètres adaptés à l'ensemble des problèmes, de la topologie de l'espace des solutions du problème qui influe fortement sur l'efficacité de certaines stratégies. Des problèmes restent ouverts et sont cités dans [Sch99]. Nous les reportons ci-dessous.

Problème ouvert 1.

Elaborer un algorithme d'approximation polynomial pour $P|prec|C_{max}$ ou pour $P|prec, p_j=1|C_{max}$ avec un facteur de garantie du plus mauvais cas de $2 - \delta$, $\delta > 0$.

Un algorithme de complexité exponentielle en m est aussi d'intérêt.

Elaborer un algorithme d'approximation polynomial pour le problème $P2|prec, p_j=1|C_{max}$.

Elaborer un algorithme d'approximation polynomial pour le problème $P3|prec, p_j=1|C_{max}$.

Problème ouvert 2.

Pour $P|prec, p_j=1, c=1|C_{max}$ améliorer le facteur de garantie à $7/3 - \delta$ ou améliorer la borne d'inapproximation à $5/4 + \delta$.

Pour $P\infty|prec, p_j=1, c=1|C_{max}$ améliorer le facteur de garantie à $4/3 - \delta$ ou améliorer la borne d'inapproximation à $7/6 + \delta$.

Dire à quelle condition il existe un algorithme d'approximation polynomial avec un facteur de garantie constant pour $P\infty|prec, p_j=1, c|C_{max}$.

Dire à quelle condition il existe un algorithme d'approximation polynomial avec un facteur de garantie constant pour $P\infty|prec, p_j=1, c_{jk}|C_{max}$.

RÉFÉRENCES

[Alb05] E. Alba. Parallel Metaheuristics. A New Class of Algorithms. Wiley-Interscience, Inc. Publication, Hoboken, New Jersay, USA, 2005.

[Ayt03] Aytug, H., M. Khouja, and F.E. Vergara. Use of genetic algorithms to solve production and operations management problems: a review. Int. J. Prod. Res, 41, N°17 (2003) pp.3955-4002.

[Bac74] K.R. Backer et Su Z.-S., «Sequencing with due-dates and early start times to minimize maximum tardiness", Naval Research Logistics Quarterly, vol. 21, no. 1, (Mar. 1974), pp.171-176.

[Ben02] Benatchba K., Koudil M., Drias H., Oumsalem H. et Chaouche K., "PARME, un environnement pour la résolution du problème Max-Sat", à paraître dans CARI'02, 6ème Colloque Africain sur la Recherche en Informatique, Octobre 2002.

[Bla94] J. Blazewicz, K.H.Ecker, G.Schmidt and J.Weglarz. Scheduling in Computer and Manufacturing Systems. Second, Revised Edition, Springer-Verlag, 1994.

[Bla78] Blazewicz, J. Simple algorithms for multiprocessor scheduling to meet deadline, Infor. Process. Lett.6, pp. 162-64, 1978.

[Bou06] Boumédiène-Merouane Hocine et Derbala Ali. Les Problèmes d'Ordonnancement à Machines Parallèles de Tâches Dépendantes : une Evaluation de Six Listes et d'un Algorithme Génétique. Actes du colloque international sur l'optimisation et les systèmes d'information, COSI06, Alger, Algérie, 11-13 Juin 2006, pp. 279-289.

[Bru98] P. Brucker and S. Knust 1998. Complexity results of scheduling problems, page web: http://www.mathematik.uni-osnabrueck.de/research/OR/class/.

[Bul98] B. Bullnheimer, G. Kotsis et C. Strauss (1998). Parallelization strategies for the Ant System. Dans High Performance Algorithms and Software in Nonlinear Optimization, R. De Leone, A. Murli, P. Pardalos et G. Toraldo, Editors, Kluwer Academic Publishers: Dordrecht, NL, p. 87-100.

[Bul99] B. Bullnheimer, R. F. Hartl et C. Strauss (1999). Applying the Ant System to the Vehicle Routing Problem. Dans Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, Voss S., Martello S., Osman I.H. et Roucairol C, Editors: Kluwer, Boston.

[Chi99]: Chiu N., Fang S. et Lee Y., "Sequencing parallel machining operations by genetic algorithms", Comuter and Industrial Engineering 36, (1999), 259-280

[Cof72] E.G. Coffman, and R.L. Graham. Optimal scheduling for two-processor systems, Acta Informatica, NO1., pp. 200-213, 1972.

[Cof84]: Coffman E.G., Frederickson Jr. G.N. et Luecker G.S., "A note on axpected makespans for largest-first sequences of independent task on two processors", Math. Oper. Res. 9, (1984), 260-266

[Coo70]: Cook S.A., “The complexity of theorem-proving procedures”, Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC), (1970), 151-158

[Dem02] M. Demange and V. Paschos. Autour de nouvelles notions pour l’analyse des algorithmes d’approximation :formalisme unifié et classes d’approximation. RAIRO Operations Research (2002) 237-277.

[Dor91b] M. Dorigo, V. Maniezzo et A. Colorni (1991 b). Positive feedback as a search trategy, Rapport technique no. 91-016. Dip. Elettronica, Politecnico di Milano, Italy.

[Dor96a] M. Dorigo, A. Colorni et V. Maniezzo (1996a). Ant System: Optimization by a Colony of Cooperating Agents. IEEE transactions on Systems, Man, and Cybernetics, Part-B, Vol. 26 (1), p. 1-13.

[Dor96b] M. Dorigo, V. Maniezzo et A. Colorni (1996b). The Ant System : Optimization by a colony of cooperating agents. Cybernetics-Part B, Vol. 26(1), p. 1-13.

[Dor97b] M. Dorigo et L. M. Gambardella (1997b). Ant Colony system: A Cooperative learning approach to the Traveling Salesman Problem. IEEE Transactions on Evolutionary Computation, Vol. 1 (1), p. 53-66.

[Dor99a] M. Dorigo et G. D. Caro (1999a). Chapter 2 : The Ant Colony Optimization Meta-Heuristic. Dans New Ideas in Optimization, D. Corne, M. Dorigo et F. Glover, Editors: London, p. 11-32.

[Dor99b] M. Dorigo, G. D. Caro et L. M. Gambardella (1999b). Ant Algorithms for Discrete Optimization. Artificial Life, Vol. 5 (2), p. 137-172.

[Dre03] J. Dréo, A, Petrowski, P. Siarry et E. Taillard. Métaheuristiques pour l’optimisation difficile. Eyrolles, 2003.

[EGG'03] J. Eggers, D. Feillet, S. Kehl, M. Oliver Wagner et B. Yannou (2003). Optimization of the keyboard arrangement problem using an Ant Colony algorithm. European Journal of Operational Research, Vol. 148 (3), p. 672-686.

[Gag01] C. Gagné, M. Gravel et W. Price (2001). A look-ahead addition to the ant colony optimization metaheuristic and its application to an industrial scheduling problem. Dans le proceeding de The 4th Metaheuristics International Conference, Porto, Portugual,

[Gag02a] C. Gagné, M. Gravel et W. L. Price (2002a). Algorithme d'optimisation par colonie de fourmis avec matrices de visibilité multiples pour la résolution d'un problème d'ordonnancement industriel. INFORMS Journal on Computing, Vol. 40 (2), p. 259-276.

[Gag02b] C. Gagné, W. Price et M. Gravel (2002b). Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence dependent setup times. Journal of the Operational Research Society, Vol. 53 (8), p. 895-906.

[Gam97] L. M. Gambardella et M. Dorigo (1997). HAS-SOP: An Hybrid Ant System for the Sequential Ordering Problem, Rapport technique no. IDSIA 97-11. IDSIA: Lugano, Switzerland.

[Gam99] L. M. Gambardella, É. Taillard et M. Dorigo (1999). Ant Colonies for the Quadratic Assignment Problem. Journal of the Operational Research Society, Vol. 50, p. 167 176.

[Gel96] L., Gelineau, , Problème d'ordonnancement multiprocesseur avec communication par diffusion. CNET ,Technopole anticipa2, Pierre Marzin (sep1996).

[Gos89] S. Goss, S. Aron, J. L. Deneubourg et J. M. Pasteels (1989). Selforganized shortcuts in the Argentine ant. Naturwissenschaften, Vol. 76, pp. 579-581.

[Gra66]: Graham R.L., "Bounds of certain multiprocessing anomalies", Bell System Tech. J. 45, (1966), 1563-1581.

[Gra78]: Garey M.R. et Johnson D.S., "Complexity results for multiprocessor scheduling under resource constraints», SIAM Journal on Computing 4, (1978), 397-411.

[Gra79]: Graham R.L, Lawler E.L. et Lenstra J.K. et Rinnoy Kan A.HG., "Optimisation and approximation in deterministic sequencing and scheduling: asurvey", Discrete Mathematics 5, (1979), 287-326.

[Gra02] M. Gravel, W. L. Price et C. Gagné (2002). Scheduling continuous casting of aluminium using a multiple-objective ant colony optimization metaheuristic. European Journal of Operational Research, Vol. 143 (1), pp. 218-229.

[Hol75] J. H. Holland. Adaptation in Natural and Artificial Systems. Ann Arbor: The University of Michigan Press, 1975.

[Hu61]: Hu T.C., «Parallel sequencing and assembly line problems», Oper. Res. 9, (1961), 841-848.

[Jac55] J.R. Jackson. Scheduling production line to minimize maximum tardiness. Technical Report 43, Management science research project, University of California, Los Angeles, 1955.

[Lop01]: Lopez P. et Roubellat F., "Ordonnancement de la production", Hermes Science Publications, Paris 2001.

[Man99a] V. Maniezzo (1999a). Exact and Approximate Nondeterministic Tree- Search Procedures for the Quadratic Assignment Problem. INFORMS Journal on Computing, Vol. 11 (4), p. 358-369.

[Man99b] V. Maniezzo et A. Carbonaro (1999b). Ant Colony Optimization : An overview. Dans le proceeding de MIC'99 III Metaheuristics International Conference, Brazil,

[Man99c] V. Maniezzo et A. Colorni (1999c). The Ant System Applied to the Quadratic Assignment Problem. IEEE Transactions on Knowledge and Data Engineering, Vol. 11 (5), p. 769-778.

[Mcn59]: McNaughton R., «scheduling with deadlines and loss functions», Management Sci. 6, (1959), 1-12.

[Mer00] D. Merkle et M. Middendorf (2000). An ant algorithm with a new pheromone evaluation rule for total tardiness problems. Dans le proceeding de Real-World Applications of Evolutionary Computing, Springer, Berlin, p. 287-296.

[Mes08] M. Messaoudi Ouchene et A. Derbala. Adaptation d'un Algorithme de Fourmis pour la résolution du problème difficile d'ordonnancement P | prec |Cmax. Proceedings de l'International Symposium on Operational Research, ISOR 2008, Algiers, Algeria: November 2-6, 2008, pp 667-677.

[Mic91] Michalewicz Z., «Genetic Algorithms + Data Structures = Evolution programs», Springer Verlag, New-York, 1991.

[Rob49] J. B. Robinson (1949). On the Hamiltonian game (a travelingsalesman problem). RAND Research Memorandum RM-303.

[Sch97a] R. Schoonderwoerd, O. Holland et J. Bruten (1997a). Ant-like Agents for Load Balancing in Telecommunications Networks. Dans le proceeding de Agents'97, Marina del Rey, CA, ACM Press, p. 209-216.

[Sch97b] R. Schoonderwoerd, O. Holland, J. Bruten et L. Rothkrantz (1997b). Ant-based Load Balancing in Telecommunications Networks. Adaptive Behavior, Vol. 5 (2), p. 169-207.

[Sch99] P.,Schuurman,, and G.J. Woeginger, 1999. Polynomial time approximation algorithms for machine scheduling: ten open problems. J. sched. 2 (1999), pp.203-213.

[Sch00]: Schuurman P., "Approximating schedules", PROEFSCHRIFT, Eindhoven, The Netherlands 2000.

[Stu97a] T. Stùtzle et H. Hoos (1997a). Improvements on the Ant System: Introducing the MAX-MIN Ant System. Dans le proceeding de ICANNGA97 - Third International Conference on Artificial Neural Networks and Genetic Algorithms, University of East Anglia, Norwich, UK, Wien: Springer Verlag,

[Tai97a] É. Taillard et L. M. Gambardella (1997a). An Ant Approach for Structured Quadratic Assignment Problems. Dans le proceeding de 2nd Metaheuristics International Conference (MIC-97), Sophia- Antipolis, France,

[Tai97b] É. Taillard et L. M. Gambardella (1997b). Adaptive memories for the quadratic assignment problem, Rapport technique no. IDSIA-87-97. IDSIA: Lugano, Suisse.

[Tal01] E.-G. Talbi, O. Roux, C. Fonlupt et D. Robillard (2001). Parallel ant colonies for the quadratic assignment problem. Future Generation Computer Systems, Vol. 17 (4).

[Tal09] E.-G. Talbi. Metaheuristics. From Design to implementation, Wiley-Interscience, Inc. Publication, Hoboken, New Jersay, USA, 2009.