UCL Université catholique de Louvain

Louvain School of Management (LSM)

LOUVAIN School of Management

An iterated local search approach to the job scheduling and tool switching problem

Mémoire réalisé par
Joseph Seynave

Promoteur
Daniele Catanzaro

Lecteur
Martin Frohn

Année académique 2017-2018
Master en Ingénieur de gestion

RÉSUMÉ

Le problème d'ordonnancement de tâches, chacune nécessitant un ensemble spécifique d'outils, fait partie des enjeux courants des industries. Posséder un algorithme capable de déterminer l'ordonnancement optimal d'une série de tâches et l'ordre de chargement des outils sur une machine flexible peut entrainer des gains conséquents. Un tel ordonnancement optimal permettrait en effet d'avoir un nombre minimal de changements d'outils à effectuer entre les différentes tâches. Ces changements d'outils requièrent non seulement du temps mais ont également un certain coût.

Dans un premier temps, le *Job Sequencing and Tool Switching Problem* sera présenté. Ensuite, les méthodes de résolution exactes existantes dans la littérature seront exposées. Cette partie plus théorique se terminera par une présentation des principales métaheuristiques en mettant particulièrement en avant la *recherche locale itérative*.

Dans un second temps, un algorithme de *recherche locale itérative* sera proposé. Les tests réalisés sur un ensemble d'instances montrent que la métaheuristique implémentée permet d'obtenir des solutions de qualité satisfaisante.

Mots-clés : Job Sequencing and Tool Switching Problem ; ordonnancement ; recherche locale itérative ; recherche opérationnelle ; problème combinatoire.

ABSTRACT

Job scheduling problems, each requiring a specific set of tools, are common industry issues. Having an algorithm capable of determining the optimal scheduling of a series of tasks and the order in which tools are loaded onto a flexible machine can lead to significant gains. Such optimal scheduling would in fact allow a minimum number of tool changes to be made between the different tasks. Indeed, these tool changes not only take time but also have a cost.

First, the Job Sequencing and Tool Switching Problem will be presented. Then, the exact resolution methods existing in the literature will be exposed. This more theoretical part will end with a presentation of the main metaheuristics, with particular emphasis on iterative local search.

In a second step, an iterative local search algorithm will be proposed. The tests carried out on a set of instances show that the metaheuristic implemented makes it possible to obtain solutions of acceptable quality.

Keywords: Job Sequencing and Tool Switching Problem; scheduling; Iterated Local Search; operational research; combinatorial optimization.

TABLE DES MATIERES

LISTE DES TABLEAUX

LISTE DES FIGURES

INTRODUCTION

En recherche opérationnelle, le *Job Scheduling and Tool Switching Problem* (SSP) est un problème combinatoire NP-difficile. Ce problème est commun notamment dans le département supply chain des industries. Il permet de déterminer l'ordre optimal dans lequel une série de tâches doivent être exécutées. La réalisation d'une tâche requiert un ensemble d'outils mais pour chaque tâche, cet ensemble est spécifique. Pour exécuter une tâche ou fabriquer un produit, les outils seront chargés sur une machine flexible. Outre l'ordre dans lequel les produits seront fabriqués, il est aussi crucial de déterminer quels outils charger sur la machine pour la fabrication de chacun des produits (Tang & Denardo, 1988). En effet, sur la machine se trouveront les outils nécessaires pour la réalisation de la tâche courante mais aussi d'autres outils inutiles pour la tâche en question. Trouver quels outils placer sur la machine est essentiel car tous les outils ne peuvent pas être chargés en même temps sur la ligne de production. Entre la fabrication de deux produits, certains outils doivent être enlevés de la machine et d'autres doivent y être ajoutés. Le meilleur ordonnancement des tâches sera donc celui qui minimisera le nombre de changements d'outils entre ces tâches. Un bon ordonnancement des tâches et une bonne gestion des outils impactent les performances du système de fabrication (Gray, Seidmann, Stecke, 1993).

Différentes techniques sont possibles pour solutionner ce problème. Dans ce mémoire, les méthodes exactes et les métaheuristiques seront abordées. Cependant, une métaheuristique en particulier sera utilisée et appliquée : la recherche locale itérative. L'Iterated Local Search (ILS) a la faculté d'être très malléable et d'offrir beaucoup de choix quant à son implémentation (Johnson & McGeoch, 1997). Cette méthode de résolution sera décrite et l'algorithme implémenté ainsi que les résultats obtenus seront présentés.

Afin de ne pas surcharger ce mémoire d'annexes, certains fichiers relatifs à ce travail se trouvent sur un drive accessible via ce lien :
https://drive.google.com/open?id=1Yv1NcZVXvT1M8ZKElpuBsx59DXbd00cR
Ce drive contient les fichiers *Mosel* des principaux algorithmes implémentés ainsi que les fichiers *Excel* reprenant les résultats obtenus. Une brève description des fichiers présents sur le

drive se trouve en annexe 1. Cependant, consulter ces fichiers n'est pas indispensable à une bonne compréhension de ce mémoire.

CHAPITRE I

THE JOB SEQUENCING AND TOOL SWITCHING PROBLEM

Ce premier chapitre aura pour but de décrire le problème qui va être abordé tout au long de ce mémoire. Il s'agit d'un problème dont l'objectif est de trouver l'ordonnancement optimal de plusieurs tâches nécessitant différents outils afin de minimiser le nombre de changements d'outils entre ces tâches.

1.1 DESCRIPTION DU PROBLÈME

Le Job Sequencing and Tool Switching Problem (SSP) est un problème courant, principalement dans les industries métallurgiques et électroniques. Ces secteurs ont vu des systèmes de fabrication flexibles remplacer les systèmes rigides. Les machines flexibles font parties de ces nouveaux systèmes de fabrication. Elles offrent la possibilité de produire, sur une même machine, différents produits et de pouvoir changer l'ordre dans lequel ceux-ci seront fabriqués (Amaya, Cotta & Fernández-Leiva, 2012 ; Crama, 1997). Ces dernières ont aussi la faculté d'exécuter différentes opérations sans changements abruptes entre celles-ci (Paiva & Carvalho, 2017). Dans le cadre de ce mémoire, le problème se concentrera sur une seule machine flexible.

Sur cette machine, ou ligne de production unique, un ensemble de jobs doit être exécuté. Dans la suite de ce travail, le terme "job" renverra à la fabrication d'un produit spécifique (Paiva & Carvalho, 2017). Chacun des jobs nécessite un sous-ensemble d'outils particuliers parmi l'ensemble d'outils à disposition. Ces jobs seront donc exécutés l'un à la suite de l'autre sur la machine flexible. Les outils nécessaires pour un job seront chargés sur la machine juste avant le début de sa production. Cette machine flexible a un nombre de compartiments fixe pouvant chacun accueillir un et un seul outil. La capacité, ou le nombre de compartiments, est telle que pour chaque job pris un à un, tous les outils de celui-ci puissent être chargés sur la ligne de production. Cependant, l'ensemble des outils utilisés par tous les jobs ne savent pas être chargés dessus en même temps (Adjiashvili, Bosio & Zemmer, 2015 ; Bard, 1988 ; Catanzaro, Gouveia, & Labbé, 2015 ; Paiva & Carvalho, 2017 ; Salonen, Raduly-Baka, & Nevalainen, 2006 ; Tang & Denardo, 1988). Ces deux conditions se retrouvent ci-dessous sous forme mathématique.

$$\sum_{t=1}^{T} a_{tj} \leq Capacité \quad \forall\, j \in J$$

$$\sum_{t=1}^{T}\sum_{j=1}^{J} a_{tj} > Capacité$$

avec a_{tj} = 1 si le job j nécessite l'outil t.

Etant donné le fait que la machine exécutera un job en utilisant les outils placés dans ses différents compartiments, entre la fin de l'exécution d'un job et le début du suivant, il est parfois nécessaire de changer les outils chargés sur la machine. Pour faire cela, la machine doit être arrêtée afin qu'un appareil puisse permuter un/plusieurs outils de la machine avec un/plusieurs autre(s) de la zone de stockage d'outils. Le fait de retirer un outil et d'en ajouter un nouveau comptera dans la suite du travail pour un changement (Crama, 1997 ; Tang et Denardo, 1988). Etant donné le fait que ces changements consomment de l'argent et du temps (interruption de la production pour chacun d'eux), le but sera de minimiser ce nombre de changements en déterminant l'ordonnancement optimal. Nous posons l'hypothèse que les coûts de changements d'outils sont uniformes et indépendants des outils changés (Catanzaro et al., 2015 ; Crama, 1997 ; Tang & Denardo, 1988). De plus, l'ordre dans lequel les outils sont chargés sur la machine n'a pas d'importance.

1.2 COMPLEXITÉ DU PROBLÈME

Ce problème d'ordonnancement de jobs en minimisant le nombre de changements d'outils est un problème d'optimisation combinatoire NP complet (Catanzaro et al., 2015). La complexité computationnelle ou le temps mis par le meilleur algorithme connu pour résoudre ce problème est grand (Stützle, 1998b). En effet, le nombre de solutions réalisables (le nombre d'ordonnancements possibles) est tellement grand qu'on ne saurait pas toutes les énoncer. Cependant, chaque solution réalisable peut être vérifiée en un temps polynomial. Tang et Denardo (1988) ont fait le parallèle entre le problème du voyageur de commerce (Traveling Salesman Problem, TSP) et le SSP. Et étant donné que le TSP est un problème NP complet, le SSP l'est aussi. Ce problème peut être décomposé en deux sous-problèmes :

- Un problème d'ordonnancement qui consiste à trouver l'ordonnancement optimal (Catanzaro et al., 2015). Crama, Kolen, Oerlemans, et Spieksma (1994) ont montré que

ce problème d'ordonnancement était un problème NP difficile. En effet, pour un problème de N jobs, il existe N ! ordonnancements possibles.

- Un problème d'outils qui consiste à déterminer quels outils charger sur la machine en fonction de l'ordonnancement donné afin de minimiser le nombre de changements d'outils (Catanzaro et al., 2015). Tang et Denardo (1988) ont mis en évidence le fait que le problème d'outils pouvait être résolu en un temps polynomial grâce à la procédure KTNS qui sera décrite dans le point 5.5.2.

1.3 ILLUSTRATION DU PROBLÈME

Dans ce point, un exemple va être illustré. La figure 1 est caractérisé par une matrice binaire outils-jobs $T * J$ où :

a_{tj} = 1 si le job j nécessite l'outil t,

a_{tj} = 0 sinon,

pour t = 1,2,…..,T (l'ensemble des outils) et j = 1,2,….., J (l'ensemble des jobs) (Crama et al., 1994). Dans cet exemple-ci, T vaut 7, J est égal à 5 et la capacité de la machine équivaut à 4.

Jobs / Outils	1	2	3	4	5
1	1	0	0	1	1
2	0	1	0	0	1
3	1	1	0	0	0
4	0	0	1	1	0
5	0	0	1	1	0
6	0	0	0	0	1
7	1	0	1	0	0
	Capacité de la machine = 4				

Figure 1: Problème avec 5 jobs, 7 outils et une capacité de 4

Dans la figure 1, on voit bien que tous les jobs ne nécessitent pas toujours un nombre d'outils égal à la capacité de la machine. Cependant, aucun job n'a besoin de plus d'outils qu'il n'y a de compartiments sur la machine. Un ordonnancement s est une permutation de l'ensemble J ne contenant qu'une seule fois chaque job, chacun à une place de s (Crama et al., 1994). Un exemple d'ordonnancement, *4-2-1-5-3*, est illustré dans la figure 2. Cet ordonnancement est tout à fait aléatoire et n'est donc probablement pas un ordonnancement optimal. Comme pour la figure 1, la matrice outils-jobs permet de savoir quels outils sont présents sur la machine au début du lancement d'un job. Dans beaucoup de cas, le nombre d'outils nécessaires pour un job est inférieur à la capacité de la machine. Dès lors, les outils nécessaires pour un job sont chargés sur la machine en priorité mais, si c'est possible, il est aussi intéressant de garder certains outils

supplémentaires sur la machine, même s'ils ne seront pas utilisés pour l'exécution du job courant (Gray, Seidmann & Stecke, 1993). Il a été démontré que garder sur la ligne de production certains outils autres que ceux nécessaires pour la fabrication du produit en question permettait d'éviter de faire des changements inutiles (Catanzaro et al., 2015 ; Paiva & Carvalho, 2017). Dans la figure 2, les outils présents sur la machine et entourés sont les outils que l'on garde sur la machine afin de diminuer le nombre de changements mais qui ne sont pas utilisés pour le job. Le choix de garder certains outils plutôt que d'autres se base sur la procédure Keep the Tool Needed Soonest (KTNS) qui sera décrite dans le chapitre 5.

Outils \ Jobs	4	2	1	5	3
1	1	(1)	1	1	(1)
2	0	1	(1)	1	0
3	(1)	1	1	0	0
4	1	(1)	0	0	1
5	1	0	0	0	1
6	0	0	0	1	0
7	0	0	1	(1)	1
	Capacité de la machine = 4				

Figure 2 : Exemple d'ordonnancement et sa matrice Outils-Jobs associée

Dans la matrice, quand $a_{t,j-1}$ =0 et $a_{t,j}$ =1, cela équivaut à un changement. De plus, il faut également compter le nombre d'outils chargés sur la machine pour le lancement du premier job (Paiva & Carvalho, 2017). Dans l'exemple de la figure 2, le nombre d'outils chargés au départ pour le job 4 compte pour quatre changements. Pour le job 2, un changement est nécessaire : retirer l'outil 5 afin de pouvoir ajouter l'outil 2 sur la ligne de production. Le job 1 et le job 5 ont eux aussi besoin d'un changement, respectivement enlever l'outil 4 pour mettre l'outil 7 et retirer l'outil 3 pour l'outil 6. Enfin, le job 3 demande quant à lui deux changements : il faut enlever les outils 2 et 6 pour rajouter le 4 et le 5. Le nombre de changements total pour cet ordonnancement de jobs est donc égal à neuf.

Le but recherché dans ce travail sera donc de trouver l'ordonnancement optimal qui minimisera le nombre de changements d'outils car dans les systèmes de fabrication flexibles, avoir un bon planning et charger les bons outils sur la ligne de production détermine l'efficience du processus (Crama et al., 1994 ; Gray et al., 1993).

Ce premier chapitre a mis en évidence les caractéristiques du Job Sequencing and Tool Switching Problem (SSP) à savoir les jobs, les outils, la machine flexible, la capacité de cette

machine et le lien entre un ordonnancement et le nombre de changements d'outils associé. Nous allons maintenant nous attarder aux différentes manières existantes pour résoudre ce problème.

CHAPITRE II

RÉSOLUTION EXACTE DU SSP

Ce second chapitre va présenter l'une des manières existantes pour résoudre un problème. La technique utilisée dans ce chapitre est appelée « résolution exacte » car on va tenter de résoudre ce problème en trouvant l'ordonnancement optimal global minimisant le nombre de changements d'outils.

Les méthodes de résolution exacte permettent de résoudre un problème de manière exacte et donc de trouver une solution optimale globale. Pour parvenir à la solution optimale globale, certaines méthodes exactes vont tester toutes les solutions possibles et garder la meilleure (Phan, 2013). Afin d'augmenter l'efficacité des méthodes exactes, des techniques ont été mises en place afin de ne pas devoir visiter toutes les solutions réalisables possibles. L'exemple le plus connu est l'algorithme de Branch and Bound (Stützle, 1998b). Pour le SSP, Tang et Denardo (1988), ainsi que Bard (1988), ont été les premiers à établir une formulation de Programmation Linéaire en nombre entier.

2.1 FORMULATION DE TANG ET DENARDO (1988)

Avant d'introduire la formulation de Tang et Denardo (1988), il est nécessaire de comprendre les notations utilisées. On considère un ensemble $J = \{1,2, \dots n\}$ de jobs, un ensemble $K = \{1,2, \dots n\}$ de positions dans l'ordonnancement, un ensemble $T = \{1,2, \dots m\}$ d'outils et une capacité maximum C. On notera :

- T_j comme l'ensemble des outils nécessaires pour exécuter le job $j \in J$;
- J_t comme l'ensemble des jobs utilisant l'outil $t \in T$;
- et ${}^{c}J_t$ comme le complément de J_t.

Posons également trois variables de décisions :

- u_j^k= 1 si le job j est exécuté à la position k, 0 sinon ;
- v_t^k= 1 si l'outil t est présent sur la machine à la position k, 0 sinon ;
- w_t^k= 1 si l'outil t est présent sur la machine à la position k et absent à la position $k-1$, 0 sinon (Catanzaro et al., 2015).

$min \sum_{t\in T} v_t^1 + \sum_{t\in T} \sum_{k\in K\setminus\{1\}} w_t^k$ (1a)

s.t. $\sum_{j\in J} u_j^k = 1 \qquad \forall k \in K$ (1b)

$\sum_{k\in K} u_j^k = 1 \qquad \forall j \in J$ (1c)

$\sum_{j\in J_t} u_j^k \leq v_t^k \qquad \forall k \in K, t \in T$ (1d)

$\sum_{t\in T} v_t^k \leq C \qquad \forall k \in K$ (1e)

$v_t^k - v_t^{k-1} \leq w_t^k \qquad \forall k \in K\setminus\{1\}, t \in T$ (1f)

$u_j^k \in \{0,1\} \qquad \forall k \in K, j \in J$ (1g)

$v_t^k, w_t^k \in \{0,1\} \qquad \forall k \in K, t \in T$ (1h)

La fonction objective (1a) permet de minimiser le nombre de changements d'outils. La première partie de la fonction prend en compte le nombre d'outils chargés sur la machine au départ pour le premier job de l'ordonnancement. Dans la seconde partie, le nombre de changements d'outils pour les jobs de la position 2 à n est minimisé. Pour calculer cela, seuls les ajouts d'outils sur la machine sont pris en compte. En effet, cela ne sert à rien de compter les ajouts et les retraits étant donné qu'un changement équivaut à un ajout et un retrait. Les deux premières contraintes (1b et 1c) sont les contraintes d'assignements. Elles permettent de s'assurer qu'un seul job soit exécuté par la machine à chaque moment et qu'un job ne soit exécuté qu'une seule fois. La troisième contrainte (1d) impose le fait qu'un job ne puisse pas être exécuté à un moment donné si un des outils dont il a besoin n'est pas présent sur la machine. La contrainte (1e) s'assure que le nombre d'outils chargés sur la machine ne dépasse pas la capacité maximale de celle-ci. La contrainte (1f) définit la manière dont un changement d'outils est comptabilisé, à savoir quand un outil est présent en position k mais pas en position $k-1$ (Catanzaro et al., 2013 ; Tang & Denardo, 1988).

Ce premier programme de Programmation Linéaire en nombre entier permet d'avoir des solutions exactes mais a plusieurs inconvénients. Le premier concerne les temps de calculs. En effet, ceux-ci deviennent vite très grands. Ainsi, pour le problème d'ordonnancement seul, le temps de computation est déjà $O(n!)$. L'algorithme de Tang et Denardo (1988) a été implémenté afin de se rendre compte de l'impact que pouvait avoir l'augmentation du nombre de jobs sur le temps de calcul. Dans le tableau 1, l'ordonnancement optimal à déterminer était basé sur un certain nombre de jobs (5 ou 10 ou 15), sur 7 outils et une machine ayant une capacité égale à 5. Seul le nombre de jobs a donc varié dans cette expérimentation. Le fichier *Excel* reprenant les données du problème et les résultats se trouve sur le drive.

Nombre de jobs	Temps de calcul afin de trouver le meilleur ordonnancement (en secondes)
5	0.1
10	5.8
15	1897.7

Tableau 1 : Temps de calcul pour le SSP résolu avec l'algorithme de Tang et Denardo (1988)

L'implémentation, présente sur le drive, a été faite sur Fico Xpress Ive 8.5 (version 1.24.24) dont le langage de programmation associé est Mosel 4.8.3. Les expériences computationnelles ont été menées sur un ordinateur Intel Core i5-7500 CPU@ 3.4GHz avec une mémoire RAM de 8,00 Go avec comme système d'exploitation Windows 10 (version 16299.431).

Dans le tableau 1, on peut clairement voir que le temps de calcul n'augmente pas proportionnellement au nombre de jobs, cette augmentation est au contraire exponentielle. Ben Ismail (2012) a également évalué les temps de calcul nécessaires pour le problème du voyageur de commerce en fonction du nombre de villes à devoir visiter et est arrivé aussi à des conclusions montrant cet accroissement rapide des temps de calcul en fonction du nombre de villes.

Le second inconvénient est qu'une solution faisable et optimale peut être atteinte si :

- $u_j^k = 1/n \quad \forall\, j \in J, k \in K$;
- $v_t^k = |J_t|/n \quad \forall\, k \in K, t \in T$;
- et $w_t^k = 0 \quad \forall\, k \in K, t \in T$.

Dans ce cas, le nombre de changement sera égal à 0 (Catanzaro et al., 2015). Pour tenter de résoudre ce problème, Laporte, Salazar-González, et Semet, (2004) ont proposé une autre formulation de Programmation Linéaire en nombre entier.

2.2 FORMULATION DE LAPORTE ET AL. (2004)

Leur nouvelle formulation fait un parallèle entre le problème d'ordonnancement de jobs et le problème du voyageur de commerce (TSP). Pour faire cela, ils ont introduit un job fictif *0* qui représente le point de départ et le point d'arrivée ; l'ensemble d'outils de ce job fictif est $T_0 = \emptyset$. Ils ont modélisé le SSP comme une version particulière du TSP dans laquelle le circuit hamiltonien[1] doit commencer et finir au job 0, doit visiter tous les jobs restants une fois

[1] Circuit passant par tous les points d'un graphe une et une seule fois (Skiena, 1990).

exactement et dont le poids des arêtes équivaut au nombre de changements d'outils entre le job i et le job j. Avant de présenter leur formulation, passons à une présentation des différentes notations utilisées :

- $J_0 = J \cup \{0\}$;
- $J^i = J \setminus \{i\} \ \forall i \in J$;
- $J_0^i = J_0 \setminus \{i\} \ \forall i \in J$;
- et G qui correspond à un graphe complet contenant tous les jobs de J_0 (Catanzaro et al., 2015).

Posons aussi trois variables de décisions :

- $x_{ij} = 1$ si le job j suit directement le job i dans l'ordonnancement, 0 sinon (pour tous $i, j \in J_0, i \neq j$) ;
- $y_j^t = 1$ si l'outil $t \in T$ est sur la machine quand le job j est exécuté, 0 sinon (pour tous $j \in J_0, t \in T$) ;
- et $z_j^t = 1$ si l'outil $t \in T$ est inséré sur la machine pour l'exécution du job j, 0 sinon (pour tous $j \in J_0, t \in T$) (Catanzaro et al., 2015).

$$min \sum_{j \in J} \sum_{t \in T_j} z_j^t \quad (2a)$$

$$\sum_{j \in J_0^i} x_{ij} = 1 \qquad \forall i \in J_0 \quad (2b)$$

$$\sum_{i \in J_0^j} x_{ij} = 1 \qquad \forall j \in J_0 \quad (2c)$$

$$\sum_{i,j \in S : i \neq j} x_{ij} \leq |S| - 1 \qquad \forall S \subset J_0, 2 \leq |S| \leq n - 1 \quad (2d)$$

$$\sum_{t \in T} y_j^t \leq C \qquad \forall j \in J \quad (2e)$$

$$x_{ij} + y_j^t - y_i^t \leq z_j^t + 1 \qquad \forall i, j \in J, i \neq j, t \in T \quad (2f)$$

$$x_{0j} + y_j^t \leq z_j^t + 1 \qquad \forall j \in J, t \in T \quad (2g)$$

$$y_j^t = 1 \qquad \forall j \in J, t \in T_j \quad (2h)$$

$$z_j^t = 0 \qquad \forall j \in J, t \in T \setminus T_j \quad (2i)$$

$$y_j^t, z_j^t \in \{0,1\} \qquad \forall j \in J, \ t \in T \quad (2j)$$

$$x_{ij} \in \{0,1\} \qquad \forall i, j \in J_0 \quad (2k)$$

La fonction objective (2a) cherche à minimiser le nombre de changements d'outils. Les contraintes (2b) et (2c) sont les contraintes d'assignements qui s'assurent pour chaque job qu'il ne soit précédé que d'un job et qu'il précède lui-même un seul job. La contrainte (2d) permet d'éviter la formation de sous-tours. La contrainte (2e) vérifie que le nombre d'outils chargés sur la machine pour chaque job ne dépasse pas la capacité. La contrainte (2f) s'assure que si le job j suit le job i et que le job j nécessite l'outil t, ce dernier est soit déjà présent sur la machine

pour le job i, soit il est inséré sur la machine juste avant l'exécution du job j. La contrainte (2g) est un cas spécial de la contrainte (2f) dans lequel on prend en compte le chargement de la machine entre le job fictif *0* et le premier job de l'ordonnancement. La contrainte (2h) fait en sorte que les outils nécessaires pour le job j soient présents sur la machine. La contrainte (2i) assure le fait qu'un changement d'outil soit réalisé uniquement si l'outil est requis pour l'exécution du job (Catanzaro et al., 2015 ; Laporte et al., 2004).

Cette nouvelle formulation de Laporte et al., (2004) est plus sophistiquée que celle de Tang et Denardo (1988). Cependant, le temps de computation est très grand. Ainsi, une instance de plus de 10 jobs ne peut être résolue en une heure (Catanzaro et al., 2015).

2.3 ETAT DE L'ART ACTUEL

Durant les 20 dernières années, peu d'approches de solutions exactes ont été réalisées. Ghiani, Grieco et Guerriero (2010) se sont basés sur le problème du circuit hamiltonien de coût minimum pour proposer une formulation du SSP avec une fonction objective non linéaire. Ils ont développé un algorithme de KTNS greedy pour le problème de *tooling* et un algorithme de branch-and-cut pour la partie *sequencing* du problème. Leurs résultats ont été meilleurs que ceux de Laporte et al. (2004) car une instance de 45 jobs a pu être résolue en 1h. Catanzaro et al. (2015) se sont eux basés sur les modèles du point 2.1 et 2.2 pour établir trois nouvelles formulations de Programmation Linéaire en nombre entier. Leurs nouveaux modèles permettent d'atteindre une meilleure relaxation linéaire par rapport à ce qui se faisait jusque-là. Les temps de calcul de leurs nouveaux modèles sont aussi beaucoup plus faibles et des problèmes basés sur un plus grand nombre de jobs et d'outils peuvent désormais être résolus. Cependant, un gap important persiste dans leurs nouveaux modèles (Catanzaro et al., 2015).

Ce second chapitre s'attardait donc à une des manières possibles pour résoudre un tel problème à savoir grâce à une méthode de résolution exacte. Deux d'entre elles ont été développées car elles sont à la base de nombreux travaux. Ce chapitre montre néanmoins les limites de la résolution exacte pour des problèmes plus complexes avec des instances plus grandes. Le chapitre suivant va s'attarder à une autre manière de résolution.

CHAPITRE III

MÉTHODE DE RÉSOLUTION APPROCHÉE

Le précédent chapitre expliquait l'une des méthodes de résolution de problèmes : la résolution exacte. Il existe également des méthodes de résolution approchées. Les deux grandes méthodes de résolution approchées sont les heuristiques et les métaheuristiques. Il existe bien des heuristiques dont certaines configurées pour solutionner le SSP. Ce chapitre consacre plus de temps aux métaheuristiques auxquelles appartient la technique de résolution qui sera employée dans ce travail.

3.1 LES PROBLÈMES D'OPTIMISATION COMBINATOIRE

Les problèmes d'optimisation combinatoire, dont le SSP fait partie, ont pour but de trouver la meilleure solution possible dans un espace discret de solution réalisable (Marmion, 2011). Résoudre ces problèmes combinatoires via une méthode de résolution exacte s'avère bien souvent très compliqué dû au temps de computation que cela demanderait. En effet, le nombre de solutions à explorer dans ce type de problème est très grand et elles ne peuvent être toutes visitées en un temps de calcul acceptable (Stützle, 1998b). Néanmoins, il existe d'autres techniques afin de résoudre de tels problèmes. Pour éviter le problème de l'explosion combinatoire, on peut avoir recours aux métaheuristiques qui ne visiteront délibérément qu'une partie des solutions. La résolution de problèmes via une métaheuristique ne permet donc pas d'être certain d'avoir la solution optimale globale du problème étant donné que certaines solutions n'ont pas été considérées (Stützle, 1998b). Cependant, le temps de calcul requis par une métaheuristique est acceptable : d'un ordre de complexité polynomial (Hao & Solnon, 2014). Les mondes scientifiques et industriels font régulièrement face à des problèmes combinatoires. C'est notamment pour cette raison que les métaheuristiques se sont de plus en plus développées (Blum & Roli, 2003). Avant que ces dernières ne se développent, d'autres méthodes approximées ont été mises au point afin d'avoir une alternative aux méthodes exactes : les heuristiques.

3.2 LES HEURISTIQUES

Une heuristique est une méthode de résolution de problèmes qui va fournir une solution rapidement. Cependant, la solution fournie n'offre aucune garantie quant à sa qualité. Cette solution est souvent proche de la solution optimale mais rien ne dit qu'il n'existe pas une solution encore meilleure. Les heuristiques sont souvent utilisées pour les problèmes combinatoires (Michallet, 2013). La caractéristique d'une heuristique est qu'elle est faite pour un problème. Or, il existe des heuristiques avec des caractéristiques plus générales qui peuvent par la suite être configurées de manière plus précise pour des problèmes particuliers. Une heuristique peut donc être utilisée pour approcher la solution optimale d'un problème ou alors, elle peut aussi aider une méthode exacte à procéder plus rapidement (Stützle, 1998b).

Blum et Roli (2003) distingue deux catégories parmi les méthodes approximées : les méthodes constructives et les méthodes de recherche locale. La première part d'une solution partielle initiale et y ajoute des composants au fur et à mesure jusqu'à arriver à une solution complète. Cette technique va construire l'ordonnancement de jobs pour le SSP en ajoutant à chaque étape un job à l'ordonnancement partiel. La seconde méthode part d'une solution initiale (complète) et va itérativement essayer de la remplacer par une meilleure solution. Dans ce cas-ci, l'ordonnancement sera modifié à chaque itération afin de tenter de minimiser le nombre de changements d'outils. Les méthodes constructives ont l'avantage d'être plus rapides mais retournent bien souvent des solutions de qualité inférieure aux méthodes de recherches locales.

Plusieurs auteurs ont travaillé sur le SSP en ayant recours aux heuristiques : Djellab H., Djellab K., et Gourgand (2000) avec leur heuristique itérative de meilleure insertion, Fathi et Barnette (2002) avec une procédure de recherche locale associée à une méthode heuristique, Chaves, Senne, et Yanasse (2012) avec leur heuristique basée sur une phase de construction et une recherche locale itérative,…(Paiva & Carvalho, 2017). On peut donc voir que plusieurs auteurs ont déjà approché le SSP avec des heuristiques. Les heuristiques ne seront pas décrites davantage car elles ne concernent pas la technique employée dans ce travail pour la résolution du SSP. Cependant, étant donné que les métaheuristiques se basent sur des heuristiques, certaines d'entre-elles seront développées par la suite en tant que parties imbriquées dans une métaheuristique.

3.3 LES MÉTAHEURISTIQUES

3.3.1 Caractéristiques d'une métaheuristique

Les métaheuristiques se basent sur des heuristiques mais explorent un espace de recherche de manière plus efficiente. Les heuristiques permettent de résoudre un problème en exploitant les caractéristiques de celui-ci et peuvent éventuellement trouver la bonne solution. Les métaheuristiques peuvent être appliquées à des problèmes différents, elles sont plus génériques. Leur principale caractéristique est qu'elles ont la faculté de pouvoir s'échapper du piège tendu par les optima locaux (Marmion, 2011 ; Gendreau & Potvin, 2010).

Etant donné le fait qu'il n'y ait pas de définition précise d'une métaheuristique, Blum et Roli (2003) ont résumé différentes propriétés les caractérisant :

- Les métaheuristiques sont des stratégies qui guident la recherche locale.
- Le but est d'explorer efficacement l'espace de recherche pour trouver des solutions (presque) optimales.
- Les techniques vont des simples procédures de recherches locales aux processus d'apprentissages complexes.
- Les algorithmes métaheuristiques sont approximés et souvent non-déterministes.
- Ils peuvent être composés de mécanismes pour éviter de rester piégé dans une zone de l'espace de recherche.
- Le concept de base permet un niveau abstrait de description.
- Elles ne sont pas spécifiques à un problème.
- Les métaheuristiques peuvent utiliser des connaissances spécifiques à un domaine via une heuristique qui est contrôlée par un niveau supérieur.
- Les formes les plus avancées de métaheuristiques utilisent l'expérience de recherche (grâce à une mémoire p.ex.) pour guider la recherche.

3.3.2 La notion de voisinage

Dans la suite de ce mémoire, le concept de voisinage sera utilisé à plusieurs reprises. Pour les problèmes combinatoires, l'ensemble des solutions possibles peut très vite devenir très grand. Dans ce cas, afin de trouver la solution optimale du problème, il faudrait pouvoir explorer chacune des possibilités de l'ensemble de solutions. Or, explorer toutes les solutions prend un temps d'exécution proportionnel à la taille de cet ensemble de solutions (Marmion, 2011). Un voisinage ($N(s)$) est un ensemble de solutions réalisables dites proches d'une solution (s). Le critère de proximité correspond au fait qu'il est facilement possible de passer de la solution s à

une solution de son voisinage ou encore que *s* partage avec les solutions réalisables de son voisinage une structure similaire ou un certain nombre de composants similaires (Ambite & Knoblock, 2001 ; Marmion, 2011). Le nombre de solutions que peut générer un opérateur à partir d'une solution *s* définira donc la taille du voisinage. Dès lors, on ne va plus explorer l'ensemble des solutions du problème mais explorer des voisinages.

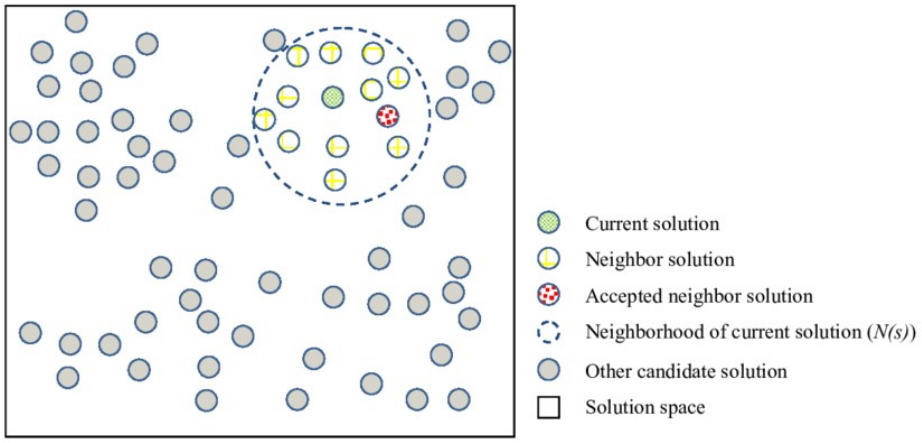

Figure 3 : Représentation d'un voisinage (Sylejmani, 2013, p. 20)

La figure 3 représente un espace de solutions possibles pour un problème. La solution courante (en vert) est la solution depuis laquelle démarre l'exploration. Le voisinage déterminé (cercle bleu) contient les solutions réalisables (en jaune et celle en rouge) proches de la solution courante. Après exploration du voisinage, la meilleure solution s'avère être celle en rouge.

3.3.3 Les notions d'intensification et de diversification

Deux notions sont très importantes quand le sujet des métaheuristiques est abordé : l'intensification et la diversification. La balance entre les deux est un facteur important influençant la performance d'une métaheuristique (Blum & Roli, 2003 ; Yang, Deb & Fong, 2014).

L'intensification est une stratégie qui consiste à intensifier la recherche autour des zones où l'on a déjà trouvé des bonnes solutions (Hao & Solnon, 2014). Cette stratégie cherche à améliorer la qualité de la solution ou les chances de trouver une meilleure solution dans un futur proche en suivant la direction donnée par la fonction objective (Hoos & Stützle, 2005). L'intensification se base en général sur l'expérience accumulée lors de la recherche pour encore améliorer la solution courante dans une région locale (Blum & Roli, 2003 ; Yang et al., 2014).

La diversification évite que l'on stagne durant la recherche dans une zone où l'optimum n'est que local mais qu'au contraire on explore d'autres espaces de recherche (Hoos & Stützle, 2005; Yang et al., 2014).

3.3.4 Différentes métaheuristiques

Les stratégies de recherches des métaheuristiques peuvent être dissociées en deux philosophies : l'une considérée comme une extension des algorithmes de recherche locale et l'autre utilisant une technique d'apprentissage afin de savoir quelles zones de recherche peuvent être intéressantes à explorer (Blum & Roli, 2003). La suite de ce chapitre sera dédiée à plusieurs métaheuristiques. Pour chacune d'entre-elles, une description de ses propriétés caractéristiques sera proposée et un pseudocode de son algorithme sera expliqué sur base du SSP.

3.3.4.1 Le recuit simulé ou Simulated Annealing (SA)

- ***Caractéristiques***

Le recuit simulé est une des plus vieilles métaheuristiques et une des premières à mettre en place une stratégie pour s'échapper d'un optimum local. Cet algorithme a été présenté pour la première fois par Kirkpatrick, Gelatt, et Vecchi (1983) ainsi que par Černý (1985) pour les problèmes d'optimisation combinatoire. Le recuit simulé vient de l'industrie métallurgique. La principale caractéristique est que certaines solutions de moins bonne qualité peuvent malgré tout être choisies comme solution courante (Hoos & Stützle, 2005).

- ***Algorithme (Blum & Roli, 2003)***

$s \leftarrow$ GenerateInitialSolution
$T \leftarrow T_0$
while (stop criterion not met) do
    $s' \leftarrow PickAtRandom(N(s))$
    $if\ \big(f(s') < f(s)\big)\ then)$
        $s \leftarrow s'$
    $else$
        $s \leftarrow s' \quad with\ probability\ e^{-\frac{f(s')-f(s)}{T}}$
    $end-if$
    $Update(T)$
$EndWhile$

- ***Fonctionnement***

Un ordonnancement de jobs est généré de manière soit aléatoire soit via une heuristique. Cet ordonnancement sera l'ordonnancement initial (s) du recuit simulé. Un critère, appelé température (T), est aussi initialisé. Nous reparlerons dans les lignes qui suivent de ce paramètre. Ensuite, on entre dans la boucle de cette métaheuristique et on en sortira une fois que le critère d'arrêt sera atteint. A chaque itération, un ordonnancement (s') appartenant au voisinage de s sera choisi de manière aléatoire. Pour évaluer l'ordonnancement s', trois éléments sont nécessaires : le nombre de changements d'outils de l'ordonnancement s' : $f(s')$, le nombre de changements d'outils de l'ordonnancement s : $f(s)$ et le paramètre T.

L'ordonnancement s' deviendra la nouvelle solution courante si $f(s') < f(s)$. Mais si $f(s') > f(s)$, l'ordonnancement s' remplacera quand même s avec une probabilité égale à $e^{-\frac{f(s')-f(s)}{T}}$ (Blum & Roli, 2003). De plus, à chaque itération, ce paramètre T va diminuer. Ainsi, au début, la probabilité d'accepter des ordonnancements avec un grand nombre de changements d'outils est élevée. Par contre, au plus T diminue, au moins la probabilité d'accepter un s' dont $f(s') > f(s)$ sera élevée (Blum & Roli, 2003). Le recuit simulé est donc défini par deux facteurs :

- A température fixe, au plus $f(s') - f(s)$ est grand, au plus la probabilité que s' remplace s est faible.
- Par contre, au plus T est élevé, au plus la probabilité que s' remplace s est élevée.

Enfin, il est nécessaire de déterminer la valeur initiale de T et la façon dont il va diminuer au fur et à mesure des itérations. La manière dont le paramètre T va diminuer peut varier au cours du problème et il est préférable de le paramétrer en fonction du problème car il est compliqué de tirer une manière de faire qui soit bonne pour tous les problèmes (Blum & Roli, 2003).

La méthode du recuit simulé est aujourd'hui plus souvent utilisée comme partie d'une métaheuristique plutôt qu'en tant que telle, elle sert alors souvent de critère d'acceptation d'une solution (Blum & Roli, 2003).

3.3.4.2 La recherche taboue ou Tabu Search (TS)

- ***Caractéristiques***

La recherche taboue fonctionne en cherchant dans le voisinage de la solution courante la meilleure solution. La principale caractéristique de la recherche taboue est qu'elle utilise la mémoire. Une liste taboue est mise en place et a pour but d'interdire de visiter les solutions qu'elle contient. La liste taboue permet non seulement de sortir d'un optimum local mais aussi d'éviter des cycles (Ghedira, 2007 ; Glover, 1986 ; Glover & Laguna, 1997).

- ***Algorithme (Blum & Roli, 2003)***

$s \leftarrow GenerateInitialSolution$
$TabuList \leftarrow \emptyset$
while $(stop\ criterion\ not\ met)$ *do*
    $s \leftarrow ChooseBestOf(N(s) \setminus TabuList)$
    $Update(TabuList)$
endwhile

- ***Fonctionnement***

La recherche taboue va commencer par générer un ordonnancement de départ s, celui-ci sera généré soit de manière aléatoire, soit via une heuristique. Il est cependant intéressant qu'il soit

généré via une heuristique car cela permet d'avoir un nombre d'itération moins élevé à réaliser (Jain, Rangaswamy, & Meeran, 2000). Au tout début, une liste taboue va également être initialisée. Par la suite, on entre dans la boucle de l'algorithme et on n'en sortira que lorsque le critère d'arrêt sera atteint. Au départ de l'ordonnancement s, une recherche locale va être effectuée. Cette recherche locale va explorer le voisinage de s de manière exhaustive afin de trouver le meilleur ordonnancement s' minimisant le nombre de changements d'outils ($f(s')$). Cependant, tout le voisinage ne sera pas exploré, seuls les ordonnancements admis le seront (Blum & Roli, 2003). Pour cet exemple, la liste taboue ne reprend pas des ordonnancements mais reprend des composants de solutions (Hertz & Widmer, 1996). Prenons que le fait de permuter deux jobs *i* et *j* détermine les ordonnancements appartenant au voisinage de s. Une fois tout le voisinage exploré, l'ordonnancement avec le nombre minimum de changements d'outils sera l'optimal local (s') ayant été obtenu grâce à la permutation du job *k* avec le job *l*. On mettra alors dans la liste taboue le job *l* afin d'interdire de permuter ce job *l* durant les *X* prochaines itérations (Hertz & Widmer, 1996). Donc, les ordonnancements admis correspondent aux ordonnancements du voisinage de s auxquels ont été retirés les ordonnancements pour lesquels un mouvement sur un composant appartenant à la liste taboue était requis (Hertz & Widmer, 1996). Lorsque l'on rentre dans la liste taboue un élément (ordonnancement, composant, mouvement,…), on retire l'élément le plus ancien de la liste selon la méthode FIFO (Blum & Roli, 2003 ; Nowicki & Smutnicki, 1996).

Dans l'exemple illustré ci-dessus, la liste taboue ne contient pas des ordonnancements mais des composants d'ordonnancement. Pour rendre la recherche taboue encore plus efficiente, on peut également créer autant de listes taboues que d'attributs que l'on souhaite interdire (Blum & Roli, 2003). Les attributs d'une solution courante sont par exemple : ses composants, le mouvement effectué pour y arriver, la différence entre elle et une autre solution. Une liste taboue sera créée par attribut ; si on définit trois attributs, il y aura trois listes taboues, chacune reprenant les attributs des X dernières solutions courantes. Ces listes taboues forment les conditions taboues (Blum & Roli, 2003). Le fait de mettre, dans une liste taboue, l'attribut d'une solution courante empêche de visiter les solutions partageant ce même attribut (Nowicki & Smutnicki, 1996). Or, ces solutions ont parfois un potentiel intéressant. C'est la raison pour laquelle un critère d'aspiration a aussi été mis en place (Blum & Roli, 2003). Il a pour but d'accepter une solution, malgré le fait qu'elle soit catégorisée dans la liste taboue, si elle remplit certaines conditions (p.ex. : si elle est meilleure que la solution courante avec laquelle elle partage l'attribut) (Nowicki & Smutnicki, 1996). La taille de la liste taboue est un facteur

important (Blum & Roli, 2003). Si la taille est petite, le risque de cycle est plus grand. Si la taille est grande, le critère d'aspiration doit être bien configuré car les solutions admises d'un voisinage seront moins nombreuses et le risque de rater un optimum sera plus grand. De plus, la taille de la liste peut être dynamique (Ghedira, 2007).

- Une taille minimale et une taille maximale peuvent être choisies et à chaque itération, la taille de la liste diminue ou augmente.
- Si $f(s)<f(s^*)$, la liste est vidée
- Si $f(s)<f(s^*)$, la taille de la liste est réduite de un.
- Si $f(s)>f(s^*)$, la taille de la liste est augmentée de un.

La mémoire d'une liste taboue peut aussi être caractérisée par une notion de court terme ou de long terme. Al-Fawzan et Al-Sultan (2003) ont proposé 5 versions de recherche taboue pour voir si une mémoire de long terme ou de court terme donnait de meilleurs résultats. Ils en ont déduit qu'une mémoire de long terme basée sur la fréquence et sur des données récentes offrait de meilleurs résultats par rapport à l'algorithme de base.

3.3.4.3 La procédure de recherche adaptative randomisée gourmande ou Greedy Randomized Adaptative Search Procedure (GRASP)

- ***Caractéristiques***

Cette métaheuristique est composée de deux éléments principaux : une heuristique constructive et une recherche locale. Le premier élément permet de construire une solution de départ alors que le second élément va chercher à améliorer cette solution de départ (Feo & Resende, 1995). Cette métaheuristique est souvent moins performante que les autres car elle fonctionne sans mémoire, mais est cependant rapide (Blum & Roli, 2003).

- ***Algorithme (Blum & Roli, 2003)***

$while(stop\ criterion\ not\ met)do$
    $s \leftarrow \emptyset$
    $\alpha \leftarrow DetermineCandidateListLenght$
    $while(solution\ not\ complete)do$
        $RCL_\alpha \leftarrow GenerateRestrictedCandidateList(s)$
        $x \leftarrow SelectElementAtRandom(RCL_\alpha)$
        $s \leftarrow s \cup \{x\}$
        $UpdateGreedyFunction(s)$
    $endwhile$
    $LocalSearch(s)$
    $MemorizeBestFoundSolution$
$endwhile$

- ***Fonctionnement***

Comme pour les autres métaheuristiques, l'algorithme sera exécuté jusqu'à ce que le critère d'arrêt soit rencontré. L'heuristique constructive va créer une solution de départ en ajoutant itérativement un job x à l'ordonnancement partiel s (Blum & Roli, 2003). Ce job est choisi aléatoirement depuis une liste RCL_α. Cette dernière est définie au préalable par une taille (α) et est composée des α meilleurs jobs. Un job est qualifié de « meilleur » en fonction du bénéfice qu'il apporterait à la solution partielle s'il y était inséré (Blum & Roli, 2003). Il s'agit, par exemple, des jobs qui entraineraient la plus faible augmentation du nombre de changements d'outils s'ils étaient insérés dans l'ordonnancement partiel. Quand un job est ajouté à l'ordonnancement partiel, la liste RCL_α est renouvelée avec les nouveaux α meilleurs jobs (Blum & Roli, 2003). Les itérations se succèdent ainsi jusqu'à avoir une solution de départ. La taille de la liste des meilleurs jobs est déterminée avant de rentrer dans la boucle. Cette taille peut soit être constante tout au long de la métaheuristique GRASP, soit être modifiée aléatoirement avant le début de chaque heuristique constructive (Blum & Roli, 2003).

Une fois qu'une solution de départ est déterminée, une recherche locale est appliquée sur celle-ci. Pour plus d'efficacité, la recherche locale doit être configurée de telle sorte qu'elle s'agence bien avec l'heuristique constructive (Blum & Roli, 2003).

Une fois la recherche locale finie, la solution trouvée s est gardée en mémoire si $f(s) < f(s*)$ où $s*$ est la meilleure solution trouvée jusque-là. Ensuite, une nouvelle solution de départ est générée par l'heuristique constructive et ainsi de suite (Blum & Roli, 2003).

3.3.4.4 La recherche dans des voisinages variables ou Variable Neighborhood Search (VNS)

- ***Caractéristiques***

Cette métaheuristique présentée par Hansen et Mladenović (1997) a comme particularité le fait de combiner une recherche locale avec une structure de voisinage qui change dynamiquement.

- ***Algorithme (Blum & Roli, 2003)***

$Select\ a\ set\ of\ neighborhood\ structures\ N_k, k = 1, \dots, k_{max}$
$s \leftarrow GenerateInitialSolution$
$while(stop\ criterion\ not\ met)do$
    $k \leftarrow 1$
    $while(k < k_{max})do$
        $s' \leftarrow PickAtRandom(N_k(s))$
        $s'' \leftarrow LocalSearch(s')$
        $if(f(s'') < f(s))then$
            $s \leftarrow s''$
            $k \leftarrow 1$
        $else$
            $k \leftarrow k + 1$
        $end - if$
    $endwhile$
$endwhile$

- ***Fonctionnement***

Tout d'abord, un ensemble $k = \{1,2, \dots k_{max}\}$ de structures de voisinages est défini. Souvent, les structures de voisinages sont définies de telle sorte que la cardinalité des voisinages augmente : $|N_1| < |N_2| < \dots < |N_{k_{max}}|$ (Blum & Roli, 2003). De plus, au début de la métaheuristique, avant de rentrer dans la boucle principale, un ordonnancement initial (s) est généré. On sortira de cette boucle uniquement lorsque le critère d'arrêt sera atteint.

Le corps de la recherche dans des voisinages variables démarre de l'ordonnancement courant s et lui applique une procédure *PickAtRandom*. Cette procédure va sélectionner aléatoirement un s' dans le $k^{ème}$ voisinage de s (Blum & Roli, 2003). Une recherche locale va être appliquée sur cet ordonnancement s'. La recherche locale ne se soucie pas du tout des structures de voisinages et retournera un ordonnancement s'' qui sera comparé avec s. Si $f(s'') < f(s)$, s'' remplacera s et l'index de voisinage k sera réinitialisé. Si ce n'est pas le cas, k sera incrémenté et on retournera à la procédure *PickAtRandom* (Blum & Roli, 2003).

Le fait de changer de structure de voisinage s'il n'y a pas d'amélioration de la solution après la recherche locale correspond au processus de diversification (Blum & Roli, 2003).

Une recherche locale, la descente dans des voisinages variables (VND), exploite les propriétés de la VNS. Dans la VND, l'ordonnancement s' sélectionné dans un voisinage k est la meilleure solution de ce voisinage (Blum & Roli, 2003). Cette solution sera comparée avec la solution courante s et la remplacera si elle est d'une meilleure qualité. Sinon, k sera incrémenté et on cherchera un s' dans le voisinage suivant (Blum & Roli, 2003).

3.3.4.5 La recherche locale guidée ou Guided Local Search (GLS)

- ***Caractéristiques***

Cette métaheuristique est caractérisée par le fait qu'elle change dynamiquement la fonction objective. Cette fonction objective sera modifiée de telle sorte que l'optimum local courant devienne moins désirable et que la recherche puisse s'échapper de cet optimum en changeant de paysage de recherche (Blum & Roli, 2003). En effet, des pénalités vont augmenter la fonction objective, la rendant moins attractive et cela à chaque fois que la recherche locale se trouve dans un optimum local (Voudouris & Tsang, 1995). Pour faire cela, la GLS se base sur les caractéristiques des solutions : les propriétés ou éléments qui les distinguent les unes des autres.

La fonction objective est modifiée de la sorte :

$$f'(s) = f(s) + \gamma \sum_{i=1}^{m} p_i * I_i(s)$$

Où - $I_i(s)$ = 1 si la caractéristique *i* est présente dans la solution s, 0 sinon ;

- γ est un paramètre de régulation qui établit la pertinence d'une caractéristique compte tenu de la fonction objective originale ;

- p_i est un paramètre de pénalité qui selon sa valeur accorde plus ou moins d'importance à une caractéristique (Blum & Roli, 2003).

- ***Algorithme (Blum & Roli, 2003)***

$s \leftarrow GenerateInitialSolution$
$while(stop\ criterion\ not\ met)do$
    $s \leftarrow LocalSearch(s, f')$
    $for\big(all\ feature\ i\ with\ maximum\ utility\ Util(s,i)\big)do$
        $p_i \leftarrow p_i + 1$
    $endfor$
    $Update(f', p)$
$endwhile$

- ***Fonctionnement***

L'algorithme va partir d'un ordonnancement initial (s) et appliquer une recherche locale, $LocalSearch(s, f')$, jusqu'à ce qu'un optimum local soit trouvé (Blum & Roli, 2003). Ensuite, les paramètres de pénalité ayant une utilité maximale seront incrémentés. L'utilité d'une caractéristique est définie de la sorte :

$$Util(s,i) = I_i(s) * \frac{c_i}{1 + p_i}$$

où c_i est le coût assigné à chaque caractéristique i par rapport à l'importance relative qu'elle a par rapport aux autres (Blum & Roli, 2003). En plus d'incrémenter les paramètres de pénalité, parfois une règle de multiplication, $Update(f', p)$, leur sera appliquée afin de les lisser :

$$p_i \leftarrow p_i * \alpha \quad \text{où } \alpha \in \{0,1\}$$

Une fois les pénalités mises à jour, on recommencera une $LocalSearch(s, f')$.

3.3.4.6 Algorithme évolutionniste ou Evolutionary Computation (EC)

- ***Caractéristiques***

L'evolutionary computation est basé sur la sélection naturelle de Charles Darwin où les individus les mieux adaptés (fitness) survivent plus longtemps (Ghedira, 2007). Cette métaheuristique, comme la suivante, fait partie des méthodes basées sur une population, c'est-à-dire qu'il n'y a plus une seule solution par itération mais un ensemble de solutions qui sont traitées (Hertz & Kobler, 2000). Le principe est le suivant : des opérations de recombinaison et de mutation sont appliquées sur une population courante d'individus, ce qui donnera une nouvelle population d'individus (Blum & Roli, 2003). Les individus sont choisis pour être dans la population en fonction de leur *fitness* (valeur de la fonction objective, mesure de qualité,…) ; si celui-ci est haut, la probabilité de faire partie de la population est plus haute (Blum & Roli, 2003). Dans les algorithmes évolutifs, un individu n'est pas nécessairement une solution mais peut être une solution partielle, un ensemble de solutions ou un composant pouvant être transformé en une solution. Souvent, la taille de la population est fixe et pour créer la population suivante, on garde les meilleurs individus de la population courante (Blum & Roli, 2003).

- ***Algorithme (Blum & Roli, 2003)***

$P \leftarrow GenerateInitialPopulation$
$Evaluate(P)$
$while(stop\ criterion\ not\ met)do$
    $P' \leftarrow Recombine(P)$
    $P'' \leftarrow Mutate(P')$
    $Evaluate(P'')$
    $P \leftarrow Select(P'' \cup P)$
$endwhile$

- ***Fonctionnement***

Avant de rentrer dans la boucle, un ensemble d'ordonnancements *P* est généré. Les individus de la population sont ici des ordonnancements. Pour chacun des ordonnancements, le nombre de changements d'outils est calculé afin de savoir lesquels sont de bonne qualité. Comme la plupart des métaheuristiques, l'EC utilise des techniques d'intensification et de diversification

(Hertz & Kobler, 2000). Tout d'abord, l'utilisation de la population permet d'explorer un large espace de recherche (Blum & Roli, 2003). Pour l'intensification, deux techniques sont souvent utilisées. La première applique une recherche locale à chaque ordonnancement de la population (l'algorithme est alors appelé algorithme mémétique) afin d'identifier rapidement les bonnes zones de l'espace de recherche (Blum & Roli, 2003). Le second utilise l'opérateur de recombinaison afin de recombiner les parties des ordonnancements qualifiées de bonnes. Chaque ordonnancement peut être recombiné soit avec un autre ordonnancement de la population, soit avec les ordonnancements présents dans son voisinage et ainsi créer un ordonnancement *descendant* (Hao & Solnon, 2014). Un descendant est créé sur base des informations croisées de ses parents ou sur base des informations statistiques de la population. Si une recombinaison donne un ordonnancement infaisable, il faut gérer le cas soit en le rejetant, soit en utilisant une stratégie de pénalité, soit en le réparant (Blum & Roli, 2003). Une fois qu'une recombinaison a été appliquée à tous les ordonnancements, on obtient une nouvelle population P' (Hertz & Kobler, 2000). Un opérateur de mutation est également utilisé dans un but de diversification, celui-ci applique une petite perturbation aléatoire sur chaque ordonnancement (Blum & Roli, 2003). Après ces actions de recombinaison et de mutation, la nouvelle population P'' est de nouveau évaluée pour connaitre les ordonnancements avec le *fitness* le plus intéressant. La population résultant d'une itération correspond à l'union entre P et P'' (Blum & Roli, 2003). Il existe évidemment différents cas plus spécifiques d'EC mais nous n'irons pas plus loin dans la description de cette métaheuristique.

3.3.4.7 L'optimisation par colonie de fourmis ou Ant Colony Optimization (ACO)

- ***Caractéristiques***

Cette métaheuristique se base sur les colonies de fourmis qui essaient de trouver le chemin le plus court entre leur nid et la nourriture. Un ensemble de X fourmis effectuera un certain nombre de fois des tours en apprenant à chacun des tours quel chemin est le plus intéressant (Blum & Roli, 2003).

- ***Algorithme (Blum & Roli, 2003)***

```
while(stop criterion not met)do
        ScheduleActivities
                AntBasedSolutionConstruction
                PheromoneUpdate
                DaemonActions    %optional
        end ScheduleActivities
endwhile
```

- ***Fonctionnement***

Pour former un ordonnancement, chaque fourmi va choisir quel job rajouter à l'ordonnancement partiel en fonction de la quantité de phéromones attribuée au job (Blum, 2005). A chaque itération, un job est ajouté à l'ordonnancement partiel. Le job ajouté doit être faisable et celui-ci sera choisi selon une règle de probabilité qui se base notamment sur une information privée de la fourmi (sa mémoire interne) ainsi que sur la valeur de *phéromone* et la valeur heuristique de ce job (Blum, 2005 ; Dorigo, Di Caro, Gambardella, 1999). Ainsi, quand une fourmi a le choix entre plusieurs jobs, elle choisira avec une probabilité plus grande le job sur lequel la concentration de phéromone est la plus élevée et/ou l'attractivité du job est la plus grande. Le choix du prochain job est basé sur une probabilité de transition telle que pour un ordonnancement partiel s, il existe un ensemble de job C pouvant être ajouté à s (Hao & Solnon, 2014). Dès lors, la fourmi va choisir le job $i \in C$ avec une probabilité :

$$p_s(i) = \frac{[\tau_s(i)]^\alpha * [\eta_s(i)]^\beta}{\sum_{j \epsilon C}[\tau_s(j)]^\alpha * [\eta_s(j)]^\beta}$$

où $\tau_s(i)$ est le facteur phéromonial associé au job i et $\eta_s(i)$ est le facteur heuristique associé à ce même job. α et β sont des paramètres permettant de jouer avec l'influence de ces deux facteurs (Hao & Solnon, 2014).

Une particularité importante est que chaque fourmi garde en mémoire l'ordonnancement construit (Blum, 2005). Une fois que la fourmi a complètement construit son ordonnancement, elle va faire deux choses : elle va diminuer les traces de phéromones avant de les mettre à jour (Hao & Solnon 2014). Diminuer les traces de phéromones permet de ne pas rester piégé dans un minimum local ou dans une zone de qualité médiocre, pour faire cela, un processus d'évaporation de phéromones est mis en place (Blum, 2005). Pour chaque job i de l'ordonnancement s,

$$\tau_s(i) \leftarrow \tau_s(i) * (1 - \rho) \qquad \text{avec } \rho \epsilon [0; 1] \text{ (Hao \& Solnon, 2014)}$$

Grâce à cela, l'intensité des phéromones diminue avec le temps et cela permet de favoriser l'exploration d'autres espaces de recherche (Blum, 2005). Après, les valeurs de phéromones des différents jobs de la solution sont mises à jour en fonction de la qualité de la solution (Blum, 2005).

$$\tau_s(i) \leftarrow \tau_s(i) + \frac{Q}{f(s)}$$

où Q est une constante propre au modèle.

Chaque fourmi exercera un certain nombre de tours avant que la métaheuristique ne s'arrête. Eventuellement, des procédures supplémentaires peuvent être implémentées, les D*aemonActions*. Celles-ci peuvent correspondre par exemple à une recherche locale qui sera appliquée sur les solutions construites par les fourmis. Cependant, cette étape est optionnelle (Blum et Roli, 2003).

Ce chapitre a principalement présenté ce qu'était une métaheuristique et a illustré plusieurs d'entre-elles. Le chapitre suivant s'attardera à la recherche locale itérative, l'Iterated Local Search (ILS), qui est la métaheuristique qui sera utilisée pour résoudre le Job Sequencing and Tool Switching Problem.

CHAPITRE IV

ITERATED LOCAL SEARCH

Ce cinquième chapitre sera consacré à l'Iterated Local Search (ILS) dont les différentes parties seront détaillées de manière théorique.

4.1 FONCTIONNEMENT GÉNÉRAL

La recherche locale itérative se base sur le fonctionnement de la recherche locale mais réalise une perturbation une fois qu'un optimum local est trouvé. En effet, le point faible de la recherche locale est que quand un optimum local est trouvé, on reste bloqué sur celui-ci (Lourenço, Martin, & Stützle, 2010 ; Hoos & Stützle, 2005). Pour répondre à cette faiblesse, on peut utiliser la recherche locale itérative. Après avoir trouvé un optimum local, l'algorithme va chercher à s'échapper de ce minimum local en faisant une perturbation afin d'explorer d'autres voisinages (Stützle, 1998b).

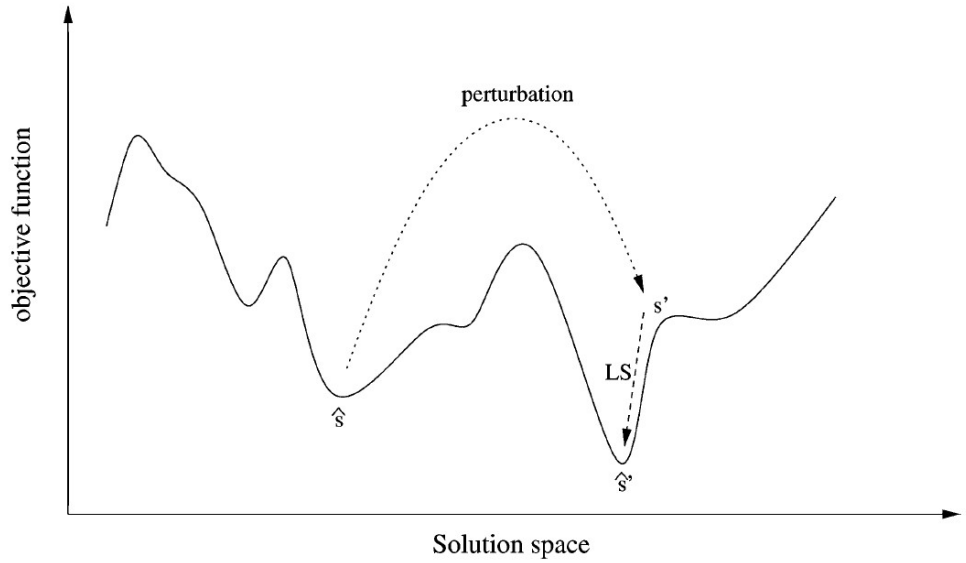

Figure 4 : Apport de la perturbation (Blum & Roli, 2003, p. 284)

Dans la figure 4, on voit que ŝ est un minimum local mais que cet optimum n'est pas global. Grâce à l'Iterated Local Search, une perturbation va être appliquée sur cet optimum local et un nouveau voisinage, celui de s', sera exploré. Sur cette nouvelle solution courante, une recherche locale sera effectuée et un nouvel optimum, cette fois-ci global, sera trouvé : $\hat{s}'$.

L'algorithme de la recherche locale itérative est présenté ci-dessous (Blum & Roli, 2003).

$s_0 \leftarrow GenerateInitialSolution$
$s^* \leftarrow LocalSearch(s_0)$
$while\ (stop\ criterion\ not\ met) do$
    $s' \leftarrow Perturbation(s^*, history)$
    $s^{*\prime} \leftarrow LocalSearch(s')$
    $s^* \leftarrow ApplyAcceptanceCriterion(s^*, s^{*\prime}, history)$
$endwhile$

Cette métaheuristique est composée de plusieurs parties pouvant chacune être optimisée afin d'obtenir un algorithme général fonctionnant de manière optimale. Ces parties sont au nombre de quatre : générer une solution initiale, appliquer une recherche locale, appliquer une perturbation, sélectionner une solution courante (critère d'acceptation). Dans la suite de ce chapitre, chacune des parties sera détaillée.

4.2 GÉNÉRER LA SOLUTION INITIALE

L'algorithme de recherche locale itérative doit commencer depuis une solution initiale. Cette solution initiale a de l'importance (Lourenço et al., 2010). En effet, on peut choisir cette solution initiale en partant d'une solution aléatoire. Cependant, on peut aussi démarrer d'une solution qui est elle-même générée par une heuristique. Partir d'une telle solution offre l'avantage de partir d'une solution qui est dans la plupart des cas meilleure qu'une solution choisie au hasard (Jain et al., 2000). De ce fait, on trouvera plus rapidement une solution optimale qu'en partant d'une solution de départ aléatoire (Lourenço et al., 2010 ; Stützle, 1998a). Afin de générer cette solution, on peut utiliser une heuristique gloutonne (Lourenço et al., 2010). Ainsi, une heuristique gloutonne combinée avec une recherche locale permet de trouver des solutions de meilleures qualités en moins d'itérations et en un temps de computation plus faible (Lourenço et al., 2010). L'heuristique gloutonne NEH, décrite par Nawaz, Enscore et Ham (1983), fait partie des plus populaire. Elle fonctionne de la sorte :

1) $S_i = \sum_{j=1,\ j\neq i}^{n} t_{ij}\ \forall i\ \in N$
   où t_{ij} correspond au nombre de changements d'outils entre le job i et le job j.
2) Classer les jobs par ordre décroissant de S_i
3) Prendre les 2 premiers jobs et les agencer afin de minimiser le nombre de changements
4) Pour les jobs classés de la 3ème à la dernière position : insérer le job dans l'ordonnancement partiel à la position qui minimise le nombre de changements de cet ordonnancement.

Dans le cadre d'un temps fixe de computation, il est intéressant d'avoir une bonne solution de départ afin de ne pas perdre de temps ou des itérations que l'on pourrait éviter (Jain et al., 2000).

Si le temps de calcul n'a pas une grande importance, le choix de la solution de départ est moins préjudiciable (Lourenço et al., 2010).

4.3 LA RECHERCHE LOCALE

Une recherche locale est une technique qui va explorer un voisinage. Le voisinage est déterminé en fonction de l'opérateur qui au départ d'une solution permet d'atteindre des solutions dites voisines. Les méthodes de descentes sont très populaires au sein des recherches locales (Marmion, 2011). Il s'agit d'une méthode qui cherche dans un voisinage déterminé la meilleure solution de ce voisinage. Elle part d'une solution et va tester toutes les solutions du voisinage en question dans le but de ne garder que la meilleure. Autrement dit, cette recherche locale va explorer les solutions dites proches de la solution courante. En partant d'une solution s, la méthode de descente consiste à choisir un s' tel que $s' \in N(s)$ et si $f(s') \leq f(s)$, remplacer s par s'. Ensuite, il faut répéter cela jusqu'à ce que pour tous les voisins s' de s, $f(s) \leq f(s')$. Alors, la recherche locale est finie et s est l'optimum local (Marmion, 2011). Dans les méthodes de descentes, il existe différentes techniques pour choisir le voisin s' : les plus connues sont la méthode Hill Climbing, la méthode First Improvment Hill Climbing et la méthode Netcrawler (Marmion, 2011).

La méthode Hill Climbing va explorer tout le voisinage afin de trouver la meilleure solution. On a donc à faire à une exploration exhaustive du voisinage (Marmion, 2011). Dans $N(s)$, le s' choisi est celui qui a la meilleure valeur de fonction objective.

La méthode First Improvement Hill Climbing va pour sa part garder la première solution améliorant la meilleure solution actuelle. Dans bien des cas, cette technique ne va donc pas explorer tout le voisinage étant donné qu'elle arrêtera l'exploration dès le moment où elle aura trouvé une solution améliorante (Marmion, 2011). Dès que pour $s' \in N(s)$, $f(s') < f(s)$, s devient s' et un autre voisinage va être exploré.

La méthode Netcrawler va quant à elle explorer le voisinage et s'arrêter dès qu'elle rencontre une solution dont la valeur de la fonction objective est au moins égale à celle de la solution courante (Marmion, 2011). Dès que pour $s' \in N(s)$, $f(s') \leq f(s)$, s devient s' et le voisinage de s va être exploré.

Il existe beaucoup de recherches locales différentes. Très souvent, la méthode employée est celle de la descente associée à une des trois techniques pour choisir le voisin. Mais ce qui varie

de l'une à l'autre, c'est la manière dont le voisinage à explorer est construit, cela dépend de l'opérateur appliqué sur la solution courante.

4.4 LA PERTURBATION

Grâce à la recherche locale, un optimum local est trouvé. Or, cet optimum n'est souvent que local et pas global et on est coincé dans cet optimum local. Avec la perturbation, on peut s'échapper de cet optimum local (Lourenço et al., 2010). On va modifier une solution, déterminée par le critère d'acceptation, afin de pouvoir recommencer une recherche locale sur base d'une nouvelle solution courante.

La perturbation ne doit ni être trop grande ni être trop faible. En utilisant une perturbation trop grande, on trouvera une nouvelle solution courante qui sera trop éloignée de la solution sur laquelle la perturbation a été appliquée (Lourenço et al., 2010). Ce genre de perturbation reviendrait à redémarrer d'une solution aléatoire. Par contre, en étant trop faible, la perturbation ne permettra pas de s'éloigner assez de la solution déterminée par le critère d'acceptation et on retombera sur cette dernière (Lourenço et al., 2010) ou dans le voisinage venant d'être exploré.

La perturbation est caractérisée par sa force, aussi appelée ordre. Il s'agit du nombre de composants de la solution que l'on va modifier (Lourenço et al., 2010) ou en d'autres mots le nombre d'éléments qui varient d'une solution à l'autre (den Besten, Stützle, & Dorigo, 2001). Il est conseillé d'avoir une perturbation d'un ordre supérieur à celui de la recherche locale afin que la recherche locale ne puisse pas annuler cette perturbation. (Lourenço et al., 2010 ; Stützle, 1998b). Par exemple, si on a une recherche locale d'ordre 2, deux jobs changent de place d'un ordonnancement à l'autre. Pour éviter de retomber sur un ordonnancement déjà visité, la perturbation doit être au minimum d'ordre 3, c'est-à-dire que trois jobs doivent bouger de place dans l'ordonnancement entre la solution acceptée par le critère d'acceptation et la solution après perturbation. Utiliser la mémoire peut aussi être utile afin de s'assurer de ne pas retomber sur une possibilité que l'on a déjà étudiée (Lourenço et al., 2010). De plus, jouer avec la force de la perturbation est intéressant (Lourenço et al., 2010). Il est par exemple possible d'intensifier l'exploration en utilisant une perturbation de force faible (mais pas trop), surtout si la perturbation est appliquée sur la meilleure solution trouvée jusqu'à présent.

Comme pour les recherches locales, il existe beaucoup de possibilités de perturbation.

4.5 LE CRITÈRE D'ACCEPTATION

Le critère d'acceptation détermine la solution sur laquelle la perturbation sera appliquée. Ainsi, une fois que la recherche locale a fini l'exploration d'un voisinage, il sera question de déterminer quelle solution deviendra la solution courante et sera retenue pour la suite (Lourenço et al., 2010). Le critère d'acceptation a donc de l'importance car il détermine la solution sur laquelle va se baser la perturbation. De plus, ce critère peut jouer un rôle en favorisant plutôt une intensification ou une diversification (Lourenço et al., 2010). Lourenço et al. (2010) ont présenté quatre critères d'acceptation.

Le premier *BETTER* va sélectionner comme solution courante la meilleure solution trouvée jusqu'à présent (Stützle, 1998a). Il s'agit par exemple de l'ordonnancement ayant le plus faible nombre de changements parmi tous les ordonnancements visités parmi tous les voisinages explorés.

$$BETTER(s^*, s^{*\prime}, history) = \begin{cases} s^{*\prime} & si\ \ C(s^{*\prime}) < C(s^*) \\ s^* & sinon \end{cases}$$

avec s^* étant la meilleure solution trouvée jusqu'à présent et $s^{*\prime}$ étant la meilleure solution trouvée dans le dernier voisinage.

Ce premier critère favorise l'intensification. A l'inverse, on peut aussi utiliser une technique de diversification, Random Walk (RW), qui garde toujours le dernier optimum local trouvé (Lourenço et al., 2010). Il s'agit par exemple de garder le meilleur ordonnancement du dernier voisinage exploré par la recherche locale.

$$RW(s^*, s^{*\prime}, history) = s^*$$

Le troisième critère d'acceptation est basé sur le recuit simulé. Ici, on garde l'optimum local s'il est améliorant mais s'il ne l'est pas, il sera malgré tout gardé avec une probabilité définie (Lourenço et al., 2010 ; Stützle, 1998a). Dans cette situation, on accepte dans certains cas une solution de moins bonne qualité.

$$SMC(s^*, s^{*\prime}, history) = \begin{cases} s^{*\prime} & si\ \ C(s^{*\prime}) < C(s^*) \\ s^{*\prime} & si\ \ C(s^{*\prime}) > C(s^*) \quad avec\ une\ probabilité\ e^{\frac{C(s^*)-C(s^{*\prime})}{T}} \\ s^* & sinon \end{cases}$$

Un quatrième critère consiste à sélectionner comme solution courante une toute nouvelle solution, s, choisie aléatoirement ou construite via une heuristique si après un certain nombre d'itérations, aucune solution améliorante n'a été trouvée (Lourenço et al., 2010).

$$RESTART(s^*, s^{*\prime}, history) = \begin{cases} s^{*\prime} & si \;\; C(s^{*\prime}) < C(s^*) \\ s & si \;\; C(s^{*\prime}) \geq C(s^*) \quad et \quad i - i_{last} > i_r \\ s^* & sinon \end{cases}$$

avec s qui est la nouvelle solution générée aléatoirement ou via une heuristique ; i_{last} étant la dernière itération où il y a eu une amélioration de la meilleure solution et i_r étant un paramètre qui dit après combien d'itérations sans amélioration l'algorithme redémarrera d'une nouvelle solution.

4.6 LE CRITÈRE D'ARRÊT

Une fois entré dans la boucle, l'algorithme ILS va procéder à des perturbations suivies de recherches locales jusqu'à ce que le critère d'arrêt soit atteint. Ce critère d'arrêt est généralement le temps de computation ou le nombre d'itérations (Marmion, 2011).

4.7 L'ILS VS LES AUTRES MÉTAHEURISTIQUES

Pour clôturer le chapitre dédié à l'Iterated Local Search, les différentes métaheuristiques présentées au chapitre 3 seront comparées à celle décrite dans ce chapitre-ci.

Il y a tout d'abord le recuit simulé, la recherche taboue et la recherche locale guidée. Ces trois métaheuristiques pourraient être imbriquées dans l'ILS et jouer le rôle de recherche locale. Le recuit simulé peut être comparé à une ILS dans laquelle il n'y aurait pas de recherche locale et dont le critère d'acceptation serait LSMC (voir le point 4.5.) La recherche taboue imbriquée dans l'ILS pourrait être utile étant donné qu'elle utilise fortement sa mémoire et permettrait de mieux balancer intensification et diversification.

La méthode ACO ainsi que la méthode GRASP sont toutes deux constructives contrairement à l'ILS qui se base sur la perturbation. On peut ainsi dire que la perturbation remplace la phase de construction pour générer une nouvelle solution courante (en supposant qu'une recherche locale soit imbriquée dans l'algorithme ACO).

La méthode de l'algorithme évolutionniste et la méthode ACO sont basées sur une population, elles génèrent sur base de plusieurs solutions une nouvelle solution. Par contre, l'ILS ne se base que sur une solution !

La recherche dans des voisinages variables est la métaheuristique qui peut se rapprocher le plus de l'ILS. En effet, si on considère comme critère d'acceptation *BETTER* et qu'on fait varier la force de la perturbation, la VNS est fort semblable à notre métaheuristique. La différence est

que le but de l'ILS est d'aller de solution optimal locale en solution optimal locale alors que l'idée de la VNS est de changer systématiquement de voisinage durant l'exploration.

Ce chapitre a décrit de manière théorique les parties caractérisant l'ILS ainsi que les attributs majeurs qui la différencient des autres métaheuristiques. Le chapitre suivant expliquera les différentes expérimentations faites pour chacune des parties.

CHAPITRE V

IMPLÉMENTATION D'UNE RECHERCHE LOCALE ITÉRATIVE POUR LE SSP

Ce chapitre sera dédié à l'implémentation de la métaheuristique Iterated Local Search. Pour chacune des quatre parties principales, les différentes méthodes et approches testées seront décrites. Chaque partie a été pensée et implémentée de manière simple dans un premier temps avant d'être améliorée par la suite. En fin de chapitre, l'ILS avec les composants sélectionnés sera présentée et les raisons des choix posés seront argumentées.

5.1 SOLUTIONS INITIALES EXPÉRIMENTÉES

L'ordonnancement initial n'a pas fait l'objet d'une attention particulière lors des premiers tests. Le code se basait sur un ordonnancement initial qui avait été rentré au préalable par l'utilisateur. Typiquement, s'il s'agissait d'ordonnancer dix jobs, l'ordonnancement initial était le suivant : [1,2,3,4,5,6,7,8,9,10]. Par la suite, trois techniques générant des solutions initiales ont été implémentées.

La première générait un ordonnancement initial aléatoire. Cette technique, souvent mentionnée dans les articles traitant de l'ILS, a comme point fort de ne pas utiliser beaucoup de temps computationnel (Lourenço et al., 2010). Cependant, le fait que cet ordonnancement initial soit aléatoire ne donne aucune garantie au niveau de sa qualité. Or, le fait d'avoir un ordonnancement de bonne qualité a un impact sur le nombre d'itérations nécessaires avant d'arriver à une solution de qualité égale. En effet, en partant d'une solution s_0 de bonne qualité, on arrivera plus vite à un optimum local (Jain et al., 2000). Cependant, si le temps n'est pas un facteur important, cette technique aléatoire est à considérer.

La seconde technique implémentée sera décrite très brièvement car elle ressemble très fortement à la troisième qui est celle employée dans l'algorithme final. Cette deuxième technique a été présentée par Tang et Denardo (1988) et est appelée *shortest edge heuristic*. Cette méthode se base sur une heuristique gloutonne pour générer la solution initiale s_0. La différence avec la troisième méthode est que le premier job de l'ordonnancement, dans ce cas-

ci, est choisi aléatoirement. Pour les jobs de la position 2 à N, ils sont agencés de la même manière que selon la 3ème technique.

Cela nous amène à la troisième technique employée pour générer la solution initiale. Afin de retirer le caractère randomisé du premier job, une autre alternative a été mise en place. Elle consiste à choisir pour la première position de l'ordonnancement un job nécessitant un grand nombre d'outils. L'idée selon laquelle cette alternative pouvait être intéressante sera expliquée dans le point 5.7.2. Donc, le premier job est soit le job employant le plus d'outils (s'il n'y en a qu'un seul), soit il est choisi au hasard parmi les jobs utilisant le plus d'outils (s'il y a plusieurs jobs employant le même nombre d'outils). Pour les autres jobs de l'ordonnancement, ils seront choisis via une heuristique gloutonne également. Pour cette heuristique gloutonne, un job sera inséré à la fin de l'ordonnancement partiel initial si ce job est celui partageant le plus d'outils identiques avec le dernier job de l'ordonnancement partiel (Hao & Solnon, 2014). Un pseudocode de l'algorithme de solution initiale se trouve ci-dessous ainsi que les explications correspondantes.

$p \leftarrow 1$
$J' \leftarrow PickJobsWithTheMostTools(J)$
$j \leftarrow PickAtRandom(J')$
$s_0(p) \leftarrow j$
$J \setminus \{j\}$
$while(J\ is\ not\ empty)do$
    $j \leftarrow MostCommonToolsWithLastJob(J)$
    $p \leftarrow p + 1$
    $s_0(p) \leftarrow j$
    $J \setminus \{j\}$
$endwhile$

Au départ, (i) on initialise la variable de position p à 1 et un ensemble reprenant le(s) job(s) nécessitant le plus d'outils est créé : J'. Etant donné que tous les jobs de ce nouvel ensemble utilisent le même nombre d'outils, (ii) le premier job j de l'ordonnancement partiel s_0 sera choisi aléatoirement parmi J'. Avant de rentrer dans la boucle, le job j va être supprimé de l'ensemble J. Ensuite, (iii) parmi les jobs de l'ensemble J, celui ayant le plus d'outils en commun avec le job occupant la position p de l'ordonnancement sera sélectionné. Si plusieurs jobs correspondent, le job qui sera ajouté est le dernier job évalué. La prochaine étape (iv) consiste à incrémenter p et à insérer le job j à la position p de l'ordonnancement partiel s_0. Enfin, (v) le job j est enlevé de l'ensemble J. Ensuite, on retournera à l'étape (iii) du début de la boucle tant que l'ensemble J n'est pas vide.

5.2 RECHERCHES LOCALES EXPÉRIMENTÉES

Les recherches locales implémentées peuvent se ranger en deux catégories en fonction de l'opérateur appliqué sur la solution courante : soit le *swap*, soit l'*insert*. Ces recherches locales se sont inspirées de nombreux articles comme notamment celui de Ergun et Orlin (2006) ou celui de Gagné, Sioud, et Gravel (2014) ou celui de Stützle (1998a).

5.2.1 L'opérateur *swap*

L'opération *swap* consiste à inverser la position de deux jobs dans l'ordonnancement.

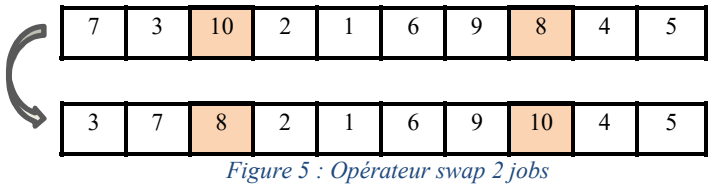

Figure 5 : Opérateur swap 2 jobs

Dans la figure 5, le job 10 est inversé avec le job 8.

Différents *swap* ont été mis en place :

- Dans un premier temps, un *swap* travaillant sur uniquement des jobs voisins a été envisagé. Seuls les jobs voisins d'un ordonnancement étaient inversés (figure 6).

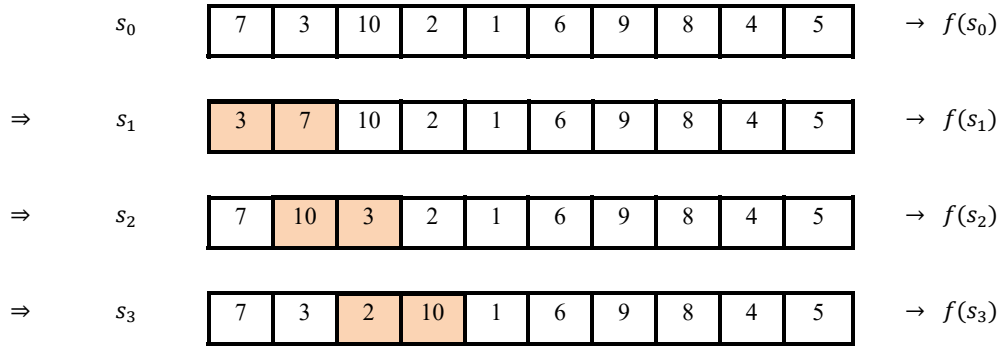

Figure 6 : Opérateur swap voisin

Dans le cas où tous les jobs voisins sont inversés, $n-1$ inversions sont possibles depuis la solution courante s_0 (n étant le nombre de jobs).

- Cependant, ce *swap* sur uniquement les jobs voisins ne permet pas d'accéder à beaucoup de voisins de l'ordonnancement courant. De plus, ce *swap* ne change pas fortement l'ordonnancement car seuls les jobs voisins changent de place. Afin de remédier à cela,

toutes les possibilités de *swap* entre deux jobs, n'étant pas nécessairement voisins, ont été réalisées.

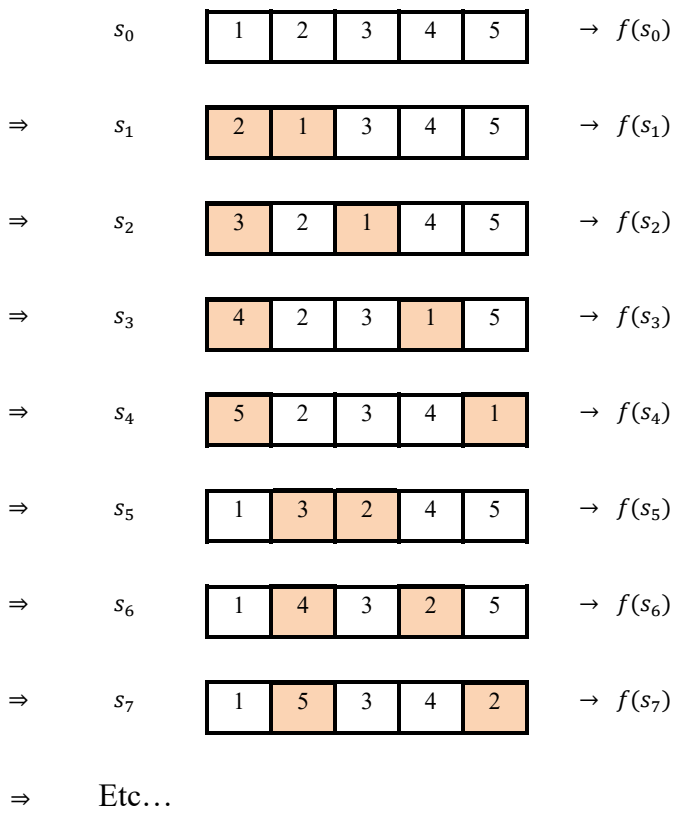

Figure 7 : Opérateur swap exhaustif

Ainsi, dans la figure 7, le job 1 est inversé avec les $n-1$ autres jobs de l'ordonnancement courant, puis c'est le job 2 qui est inversé avec les $n-2$ autres jobs et ainsi de suite jusqu'à ce que le job *N-1* soit inversé avec le job *N*. Donc, pour un ordonnancement de n jobs, si tous les *swap* possibles sont faits, il y a $\frac{N*(N-1)}{2}$ ordonnancements possibles (sans compter l'ordonnancement courant).

- Quand le nombre de jobs est très élevé, le nombre d'ordonnancements possibles devient important si toutes les possibilités de *swap* entre deux jobs sont faites. C'est pourquoi, trois procédures de *swap* plus allégées ont aussi été implémentées. L'une va choisir un job aléatoirement et inverser sa position dans l'ordonnancement avec celles de tous les autres. $N-1$ ordonnancements sont donc possibles pour ce cas-là.
- La seconde va inverser X paires de jobs. En fonction du nombre de jobs, un nombre X de paires de jobs sera déterminé et chaque paire de jobs sera formée aléatoirement en veillant à ce qu'il n'y ait pas de doublons entre les X paires. Ensuite, pour chaque paire,

les jobs seront inversés. Dans ce cas-ci, au départ d'une solution courante, il y aura X ordonnancements possibles dans le voisinage en question.

- Enfin, une troisième procédure plus allégée aussi consiste à choisir aléatoirement X jobs qui seront inversés avec tous les autres jobs. Le nombre de jobs, X, qui seront inversés dépend du nombre total de jobs. Pour X jobs à inverser avec tous les jobs possibles, le nombre d'ordonnancements possibles est égal à $(N-1)+(N-2)+\cdots+(N-X)$.

Les tests sur ces différentes procédures *swaps* ont montré qu'inverser chaque job avec tous les autres jobs possibles donnait de meilleurs résultats. En effet, cela permettait d'avoir la certitude de trouver l'optimum local du voisinage définit par une inversion de deux jobs. En effet, le *swap* qui exerçait une inversion sur uniquement les jobs voisins explorait un voisinage restreint (Stützle, 1998a). Cela avait l'avantage d'avoir un temps de computation très faible mais le fait que le voisinage soit très petit obligeait de procéder à des perturbations très régulièrement. De plus, ces perturbations ne devaient pas être trop fortes pour pouvoir explorer tous les voisinages possibles. Les recherches locales via les *swaps* où un ou plusieurs jobs étaient choisis aléatoirement reposent trop sur le caractère randomisé. En effet, la probabilité que l'on ne tombe pas sur l'optimum local est élevée or on ne reviendra plus dans ce voisinage une fois que l'on aura fait une recherche locale dessus et qu'une perturbation aura été faite. Nous privilégierons donc des recherches locales explorant beaucoup de solutions même si cela demande un temps computationnel plus important.

A contrario, une recherche locale d'une force plus importante a aussi été implémentée. Il s'agit d'une recherche locale où trois jobs sont inversés. Celle-ci consiste à inverser la place de trois jobs les uns avec les autres en testant toutes les possibilités de combinaisons et d'inversions. Ainsi, une permutation de trois jobs entraine la création de cinq nouvelles possibilités d'ordonnancements. Cependant, cette recherche locale est fort large et sur de grands ordonnancements, le temps de calcul devient vite trop important. Or, on peut passer un temps raisonnable à explorer un voisinage mais il ne faut pas passer non plus trop de temps au sein d'un voisinage car rien n'assure qu'une solution de bonne qualité s'y trouve.

5.2.2 L'opérateur *insert*

L'opération *insert* consiste à choisir un job et à insérer ce job à une position autre que la sienne dans l'ordonnancement. De ce fait, une partie des jobs de l'ordonnancement sera décalée vers la gauche ou vers la droite afin de laisser une place à ce job. Comme pour l'opération *swap*, il est possible de soit choisir un seul job aléatoirement et l'insérer à toutes les places possibles, soit choisir X jobs aléatoirement et un à un, les insérer à toutes les positions possibles de

l'ordonnancement, soit tous les jobs peuvent être pris un à un et insérés à toutes les positions de l'ordonnancement.

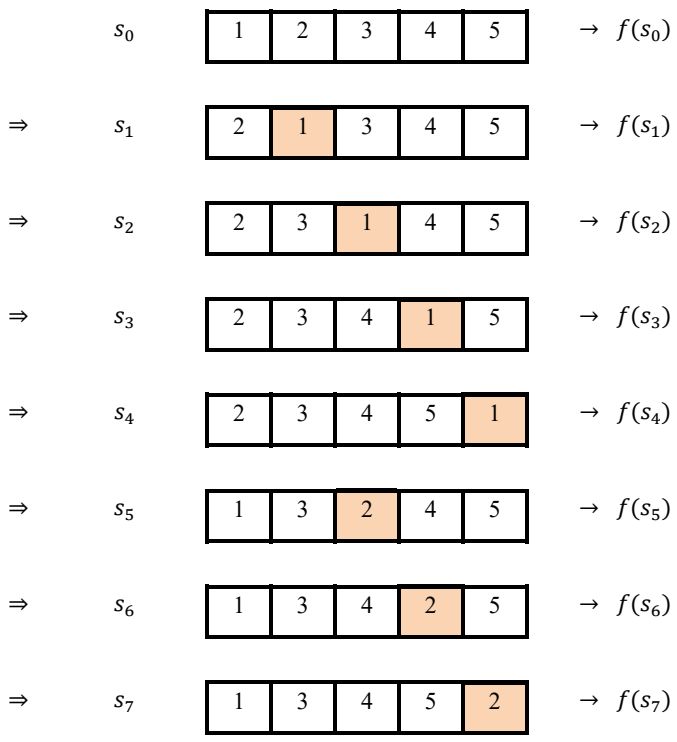

Figure 8 : Opérateur insert exhaustif

La figure ci-dessus montre la recherche locale *insert*, sur base d'une solution courante s_0, dans le cas où tous les jobs sont insérés un à un à toutes les places de l'ordonnancement. Pour les mêmes raisons que pour la procédure *swap*, insérer tous les jobs un à un à toutes les positions de l'ordonnancement a été privilégié.

5.2.3 L'opérateur *block-grouping*

Cette recherche locale, inspirée de l'article de Paiva et Carvalho (2017), procède de la sorte : sur base d'un ordonnancement, elle va déterminer pour chaque outil le(s) bloc(s) de job(s) ayant besoin de l'outil en question. Pour un outil donné, un bloc de jobs est un ensemble de jobs ordonnancés les uns à la suite des autres dans l'ordonnancement utilisant cet outil. Dans une matrice outils-jobs, comme présenté dans la figure 9, un bloc de jobs est entouré de jobs n'utilisant pas l'outil en question. Un bloc de job(s) doit être composé d'au minimum un job et peut compter jusqu'à n jobs.

Jobs / Outils	1	3	4	5	2
1	1	1	0	1	0
2	1	0	0	1	0
3	0	1	0	0	1
4	0	1	1	0	0
5	0	0	1	0	0
6	1	0	1	1	0
7	0	1	0	0	1
	Capacité de la machine = 4				

Figure 9 : Matrice outils-jobs pour un ordonnancement d'un problème 5 jobs, 7 outils et une capacité de 4

Dans cet exemple d'ordonnancement, pour l'outil 1, il y a deux blocs : le premier est composé des jobs 1 et 3 car ils utilisent tous les deux l'outil 1 et se suivent dans l'ordonnancement ; le second bloc est composé uniquement du job 5 car il utilise bel et bien l'outil 1. Les outils 2, 3, 6 et 7 comptent aussi 2 blocs. Par contre, les outils 4 et 5 comptent uniquement 1 bloc.

Pour chaque outil, quand il y a plus d'un bloc, chaque bloc va être déplacé soit avant chacun des autres blocs, soit après chacun des autres blocs, soit il ne bougera pas. Pour savoir si un bloc sera déplacé avant ou après un autre bloc, on évalue le nombre de changements d'outils si le déplacement du bloc était réalisé.

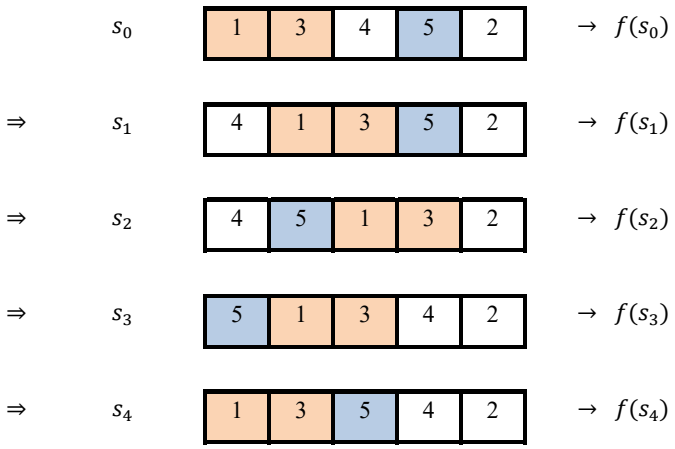

Figure 10 : Opérateur block-grouping pour l'outil 1

L'ordonnancement courant, s_0, est donné à la première ligne de la figure 10. Les jobs de couleur orange correspondent aux jobs appartenant au premier bloc et le job en bleu correspond au job appartenant au second bloc si on se focalise sur l'outil 1 (voir figure 9). Les ordonnancements s_1 et s_2 viennent d'un déplacement du bloc 1 avant et après le bloc 2 respectivement. Les ordonnancements s_3 et s_4 viennent d'un déplacement du bloc 2 avant et après le bloc 1 respectivement.

Evidemment, il faut également tester les ordonnancements quand on se focalise sur les six autres outils du problème et tester les déplacements des blocs de jobs de ces outils. Cette

technique a pour but de minimiser les changements d'outils en faisant se suivre au maximum les jobs utilisant les mêmes outils (avoir des « 1 » qui se suivent dans la matrice outils-jobs). L'algorithme du *block-grouping*, inspiré de celui de Paiva et Carvalho (2017), est affiché ci-après.

$$
\begin{aligned}
&for\ (all\ tools) do\\
&\quad for\ (all\ 1blocks\ i) do\\
&\quad\quad for\ (all\ 1blocks\ j \mid j \exists\ and\ j \neq i) do\\
&\quad\quad\quad move \leftarrow \{before(i,j), after(i,j), no\_move\}\\
&\quad\quad\quad perform(move)\\
&\quad\quad endfor\\
&\quad endfor\\
&endfor
\end{aligned}
$$

L'algorithme présenté ci-dessus est légèrement différent de celui de Paiva et Carvalho (2017). Pour chaque outil, il identifie les blocs constitués des jobs se suivant dans l'ordonnancement qui utilisent l'outil en question. Ces blocs de jobs se suivant dans l'ordonnancement et utilisant un outil sont appelés les *1blocks*. Si plusieurs blocs existent, pour chaque bloc i déplacé avant ou après chaque bloc j, le nombre de changements du nouvel ordonnancement s' est calculé et si $f(s') < f(s)$, alors le déplacement est validé. Plusieurs blocs peuvent donc être bougés entre l'entrée dans la boucle et la sortie de celle-ci car chaque déplacement validé est effectué.

5.2.4 L'opérateur 2-opt, 3-opt, k-opt, algorithme de Lin-Kernighan

Il existe encore bien d'autres types de recherches locales. En laissant libre cours à notre imagination, on peut encore en trouver plusieurs. Il faut bien entendu les tester afin de les valider ou les réfuter et s'assurer de toujours bien les combiner avec les perturbations. Il parait cependant important d'en aborder encore quatre qui sont très populaires mais qui n'ont pas été implémentées.

La première est appelée recherche locale 2-opt (Croes, 1958). Elle va considérer deux arcs choisis aléatoirement et va casser chacun des deux arcs. Ensuite, elle va recréer un nouvel arc entre un des deux nœuds d'un des deux arcs cassés et un des deux nœuds de l'autre arc cassé. Il faut s'assurer qu'une fois l'autre arc lui aussi recomposé, on ait toujours bien un seul ordonnancement contenant tous les jobs une seule fois à une seule place distincte.

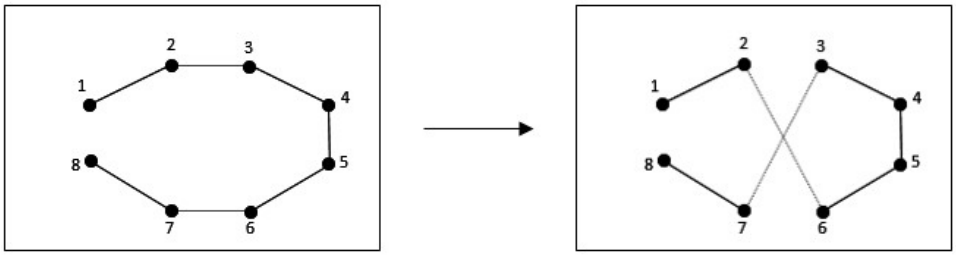

Figure 11 : Opérateur 2-opt (schéma)

Dans la figure 11, les arcs entre les jobs 2 et 3 et entre les jobs 6 et 7 ont été cassés pour reformer deux nouveaux arcs : l'arc 2-6 et l'arc 3-7. Ci-dessous se trouve l'ordonnancement initial s_0 et l'ordonnancement après cette recherche locale 2-opt sur ces deux arcs décrits ci-dessus. Ensuite, d'autres arcs sont choisis afin de procéder à une nouvelle 2-opt et ainsi de suite.

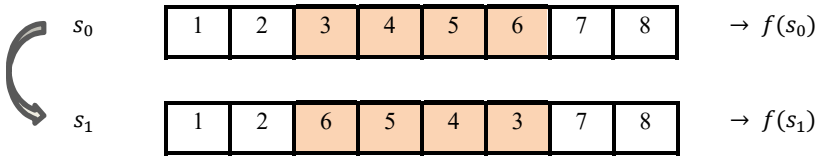

Figure 12 : Opérateur 2-opt (ordonnancement)

Cette recherche locale est souvent appliquée au problème du voyageur de commerce. La raison pour laquelle cette recherche locale n'a pas été implémentée est que cette recherche locale a été jugée d'une force élevée. De ce fait, la perturbation qui devait y être associée devait encore être plus forte que cette recherche locale. Dans le point 5.3., nous verrons que cette technique a été testée plutôt comme perturbation.

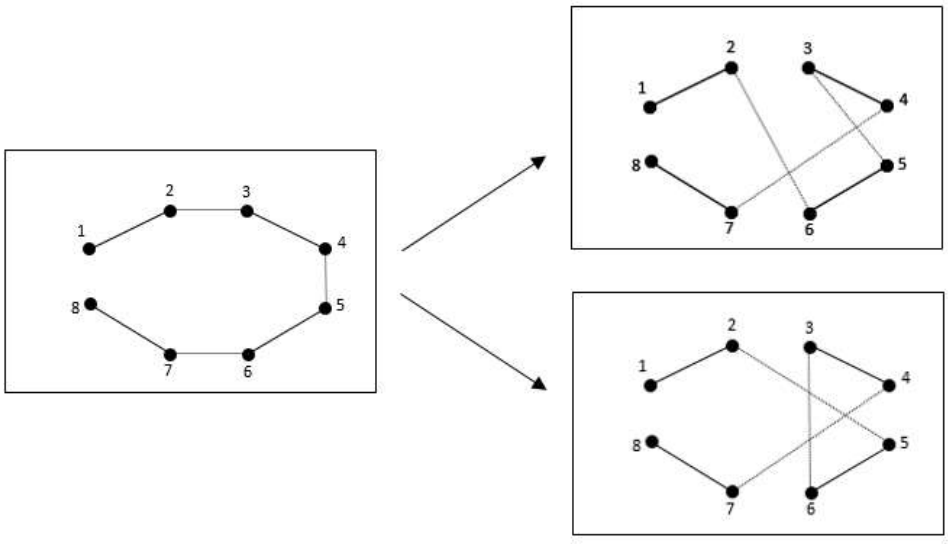

Figure 13 : Opérateur 3-opt (schéma), 2 possibilités de recombinaisons

La figure ci-dessus présente une recherche locale 3-opt (Lin, 1965). Elle va casser trois arcs pour en reformer trois nouveaux et changer ainsi l'ordonnancement. Dans la figure 13, les arcs 2-3, 4-5 et 6-7 sont cassés et deux possibilités de reformation de trois arcs sont présentées : soit

2-6, 5-3 et 4-7, soit 2-5, 6-3 et 4-7 (voir figure 14). Il existe d'autres possibilités de reformation d'arcs pour la 3-opt. Comme pour la 2-opt, par la suite d'autres arcs peuvent être choisis pour être cassés et reformés.

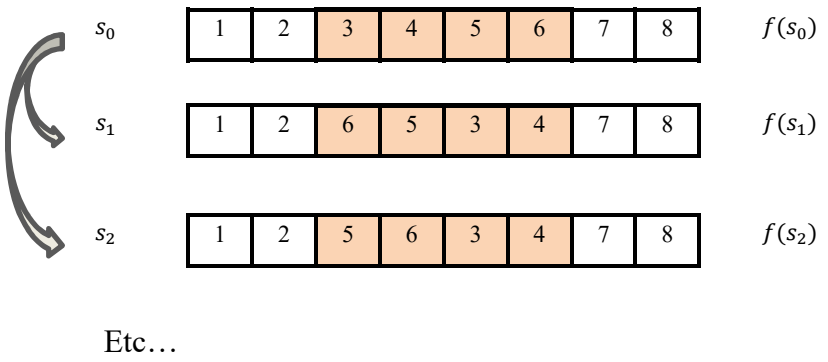

Figure 14 : Opérateur 3-opt (ordonnancement), 2 possibilités de recombinaisons

Dérivé de la 2-opt et de la 3-opt, il est aussi possible de casser un nombre k d'arcs et de faire différentes reformations d'arcs par la suite. Cette méthode est appelé k-opt mais a un temps computationnel qui devient très vite important.

La dernière recherche locale a été présentée par Lin et Kernighan (1973). Cet algorithme va casser et reformer un certain nombre d'arcs. Cette technique est adaptative, elle choisira à chaque étape combien d'arcs elle juge nécessaire de casser et reformer.

5.3 PERTURBATIONS EXPÉRIMENTÉES

La perturbation permet de s'échapper d'un minimum local. Etant donné que les recherches locales privilégiées explorent de larges voisinages, la perturbation mise en place doit être assez forte que pour ne pas retomber dans le voisinage exploré. Les deux premières perturbations étaient intuitives et ont mené à des résultats peu concluants. Voici les deux premières perturbations réalisées sur un optimum local :

5.3.1 Perturbation $ABCD \rightarrow CDAB$

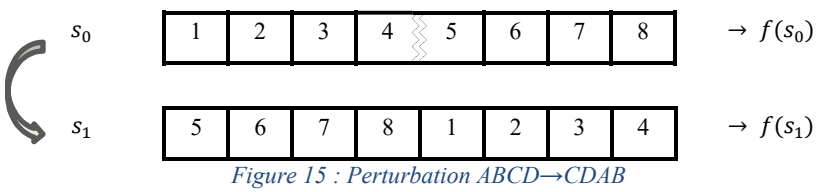

Figure 15 : Perturbation ABCD→CDAB

Cette perturbation scinde en deux parties l'ordonnancement. Le découpage se fait autant que possible au milieu de l'ordonnancement (pour les ordonnancements impairs, la cassure se fait

à la position entière supérieure au milieu). Ensuite, les deux parties sont inversées afin de donner le nouvel ordonnancement. Cette perturbation était efficace pour de petites instances mais beaucoup moins pour de plus grandes instances.

5.3.2 Perturbation $ABCD \rightarrow DCBA$

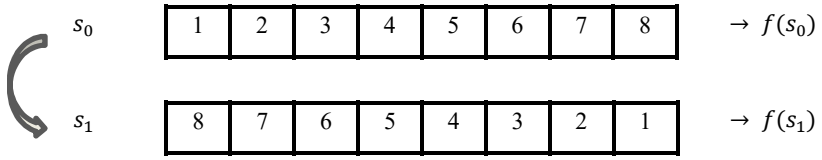

Figure 16 : Perturbation ABCD →DCBA

Cette perturbation inverse l'ordonnancement. Pour chaque job $j \in J$, si j occupe la position $p \in N$, il bougera à la position $N - p + 1$ dans le nouvel ordonnancement. Cependant, cette perturbation s'est révélée non efficace car selon le principe de symétrie, un ordonnancement a le même nombre de changements que son ordonnancement inversé (Ghiani, Grieco, Guerriero, 2010).

5.3.3 Perturbation swap 0.2*N random job

La perturbation suivante va inverser la position de plusieurs jobs pris deux à deux. Le nombre de jobs dont la position sera inversée n'est pas facile à déterminer. Etant donné le fait que trouver le nombre idéal de jobs à inverser demanderait beaucoup de temps et un nombre de tests très important, ce pourcentage idéal a été repris de l'article de López-Ibáñez, Dubois-Lacoste, Cáceres, Birattari, et Stützle (2016) qui utilise une proportion de jobs inversés valant $0.2 * n$. Ceci biaise un peu l'utilisation de cette perturbation car selon la recherche locale, il faudrait sans doute ajuster cette proportion. Dans le cas où l'on a dix jobs, deux seront choisis aléatoirement et chacun sera inversé avec un autre job. On définit au préalable les paires de jobs à inverser afin de ne pas avoir de doublons. Dans la figure 17, les jobs 3 et 7 ont été tirés.

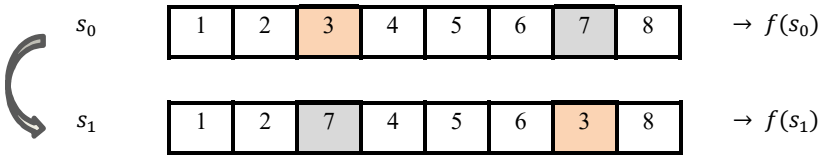

*Figure 17 : Perturbation swap 0.2*N random job*

Evidemment, pour que cette perturbation soit efficace et permette bien de s'échapper d'un minimum local, il faut que le nombre de jobs soit au moins égal à 13 afin que la force de la perturbation (égale à 3 car on arrondit) soit plus grande que la force de la recherche locale (si on fait un *swap* sur tous les jobs possibles : force égale à 2).

5.3.4 Perturbation *shake*

La perturbation suivante est inspirée de l'article de Gunawan, Lau, et Lu (2015) dans lequel une perturbation *shake* est effectuée. Dans un ordonnancement donné, une séquence de jobs est choisie. Le nouvel ordonnancement est reformé de la sorte : les jobs avant la séquence prélevée sont placés au début du nouvel ordonnancement partiel, les jobs après la séquence prélevée viennent s'ajouter à la suite dans cet ordonnancement partiel et pour finir la séquence prélevée est placée à la fin. Pour définir cette dernière, il faut deux paramètres : *cons* et *post*. *Cons* détermine la longueur de la séquence prélevée alors que *post* identifie la position du premier job de cette séquence. A chaque itération, *cons* est incrémenté de 1 alors que le paramètre *post* est incrémenté du paramètre *cons*.

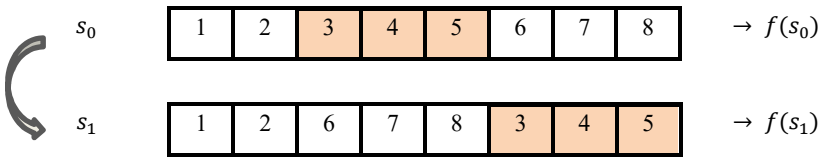

Figure 18 : Perturbation shake

Pour la figure 18, *post* est égal à 3 et *cons* vaut aussi 3. Leurs nouvelles valeurs, qui seront utilisées pour la perturbation suivante, sont : *post* égal à 6 et *cons* égal à 4.

Si la séquence qui doit être prélevée dépasse des bornes de l'ordonnancement, la séquence prélevée commence à *post* jusque n puis recommencera à 1 jusqu'à $cons - (n - post) - 1$. De même pour mettre à jour la valeur de post, si $post + cons > n$, alors la nouvelle valeur de post sera $cons - (n - post)$. Pour sa part, *cons* sera incrémenté tant qu'il est plus petit ou égal à $0.75 * n$. Dans le cas contraire, il sera réinitialisé.

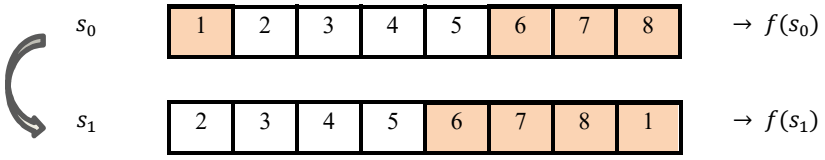

Figure 19 : Perturbation shake (cas spécial)

Pour la figure 19, *post* vaut 6 et *cons* est égal à 4. Pour la perturbation suivante, *post* vaudra 2 et *cons* sera égal à 5.

5.3.5 Perturbation $ABCD \rightarrow CADB$ ou $ABCD \rightarrow BDAC$

La perturbation suivante est une perturbation qui scinde en quatre parties un ordonnancement et en reforme un nouvel en assemblant les parties selon l'un des deux modèles ci-dessous :

(1)

Figure 20 : Perturbation : modèle ABCD → CADB

(2)

Figure 21 : Perturbation : modèle ABCD → BDAC

Les quatre parties sont déterminées via trois nombres différents $\in \{1,2,\dots n\}$ tirés aléatoirement. Ces nombres correspondent à la position à laquelle l'ordonnancement sera scindé. La découpe s'effectue toujours entre la position précédant le nombre et la position du nombre aléatoire. Par exemple, si le nombre 4 est tiré, la découpe s'effectuera entre la position 3 et la position 4. Si la position « 1 » est tirée, c'est un cas spécial ! La découpe s'effectuera alors après le job occupant la position 1 et cela se répercute sur la découpe suivante si c'est la position « 2 » qui est le 2ème nombre aléatoire. Dans la figure 22, le modèle $ABCD \rightarrow CADB$ est appliqué et les trois nombres aléatoires sont : 1, 4, 6.

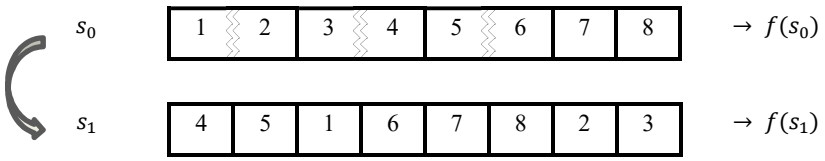

Figure 22 : Perturbation ABCD →CADB

5.3.6 Perturbation 2-opt

La dernière perturbation sélectionne une séquence de l'ordonnancement et sans la changer de position dans l'ordonnancement, inverse cette séquence. Pour déterminer la séquence, deux positions sont choisies aléatoirement et tous les jobs occupant une position entre la première et la deuxième position tirées font partie de la séquence (incluant les deux jobs placés à ces positions).

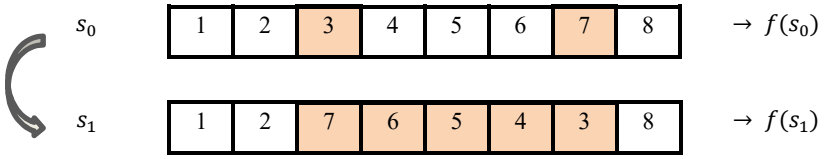

Figure 23 : Perturbation 2-opt

Dans la figure 23, les deux positions choisies aléatoirement sont 3 et 7.

5.4 CRITÈRES D'ACCEPTATION EXPÉRIMENTÉS

Le critère d'acceptation va déterminer l'ordonnancement sur lequel la perturbation va être appliquée.

Dans le chapitre 4, quatre techniques avaient été décrites : *BETTER*, *RW*, *LSMC* et *RESTART*. En pratique, trois techniques différentes ont été testées. La technique *BETTER*, qui sélectionnait le meilleur ordonnancement trouvé jusqu'à présent. La seconde, *RW*, sélectionnait le meilleur ordonnancement du dernier voisinage visité. Et la troisième méthode testée n'a pas été présentée dans le chapitre 4. Cette dernière sélectionnait le dernier ordonnancement visité et était donc basée sur un critère plus aléatoire. Ce critère d'acceptation a l'avantage d'être fortement diversifiant. Les deux autres méthodes, *LSMC* et *RESTART*, n'ont pas été implémentées. Pour la première, différents paramètres doivent être déterminés : T, la température et le paramètre indiquant comment la température baisse. Pour la seconde, le paramètre i_r doit aussi être déterminé. Calculer la valeur de ces différents paramètres n'est pas aisé car ils doivent aussi pouvoir varier de problème en problème en fonction de la taille de ceux-ci. Or, calculer ces paramètres aurait demandé beaucoup de temps et c'est la raison pour laquelle nous nous sommes focalisés uniquement sur les trois critères d'acceptation indiqués précédemment.

5.5 PROCÉDURES POUR LA PARTIE TOOLING

5.5.1 Heuristique intuitive

Au début des tests de l'Iterated Local Search, une heuristique intuitive avait été implémentée en lieu et place de la KTNS (voir point 5.5.2). Implémenter la KTNS n'était pas facile et allait prendre du temps afin d'être bien réalisée. C'est pourquoi une autre heuristique a été mise en place. Celle-ci est plus basique mais permettait, pour des instances de taille raisonnable, de pouvoir déjà tester l'ILS. Sur base d'un ordonnancement donné, son principe est le suivant :

$$
\begin{aligned}
&forall\ (j\ in\ jobs) do\\
&\quad forall(t\ in\ tools) do\\
&\quad\quad if(t\ \in {}^{c}T_j\ \ AND\ \ |T_j| < Cap\ \ AND\ \ t \in T_{j+1})\ then\\
&\quad\quad\quad add\ tool\ t\ on\ the\ machine\ for\ job\ j\\
&\quad\quad elif(t\ \in {}^{c}T_j\ \ AND\ \ |T_j| < Cap\ \ AND\ \ t \in T_{j-1})\ )\ then\\
&\quad\quad\quad add\ tool\ t\ on\ the\ machine\ for\ job\ j\\
&\quad\quad end-if\\
&\quad end-do\\
&end-do
\end{aligned}
$$

où $|T_j|$ correspond au nombre d'outils nécessaires pour le job j ; ${}^{c}T_j$ est l'ensemble d'outils non nécessaires pour le job j ; T_{j+1} correspond à l'ensemble des outils nécessaires pour le job suivant j dans l'ordonnancement et T_{j-1} correspond à l'ensemble des outils nécessaires pour le job précédent j dans l'ordonnancement.

Cette heuristique a été utile afin de pouvoir avancer dans la recherche des autres parties de l'ILS. Cependant, quand les instances devenaient plus grandes, il est apparu évident que cette heuristique ne suffirait plus et une procédure KTNS a été implémentée.

5.5.2 Procédure « Keep the Tool Needed Soonest »

La recherche locale itérative permet de tester des ordonnancements afin de trouver celui qui est optimal. Cependant, pour CHAQUE ordonnancement déterminé, il faut encore déterminer les outils qui seront présents sur la machine pour l'exécution de chacun des jobs. En effet, le SSP est un problème d'ordonnancement mais aussi un problème de *tooling* pour lequel il faut déterminer quels outils placer sur la machine. Les outils nécessaires pour l'exécution d'un job doivent bien entendu être chargés sur la machine en priorité mais il faut aussi déterminer quels outils non-nécessaires il est intéressant de garder sur la machine. C'est pourquoi, pour chaque ordonnancement testé, une procédure, Keep the Tool Needed Soonest (KTNS), va être appliquée sur l'ordonnancement.

Le principe fondamental de la KTNS est le suivant : si le nombre d'outils pour un job est inférieur à la capacité de la machine, il faut garder les outils du job précédent dans l'ordonnancement qui seront utilisés au plus tôt par le(s) job(s) suivant(s) (Tang et Denardo, 1987). Cette procédure va permettre de déterminer quels outils il est plus intéressant de garder sur la ligne de production et quels outils il faudrait échanger avec ceux qui ne sont pas présents sur la machine et qui doivent l'être pour l'exécution du job. La KTNS est donc utile quand $|T_j| < C$ et que l'on doit déterminer quels outils non-nécessaires garder.

Tang et Denardo (1987) ont démontré qu'il y avait un lien entre le SSP et le problème du voyageur de commerce. En effet, le SSP peut être semblable au TSP si, pour chaque job, le nombre d'outils présents sur la machine est égal à la capacité de celle-ci. La procédure KTNS permet de remplir cette condition en déterminant quels outils non-utiles garder sur la ligne de production. Dès lors, les deux problèmes sont semblables ; les distances entre les villes correspondent aux nombres de changements d'outils entre deux jobs.

Jobs / Outils	7	1	4	2	3	8	5	6
1	1	0	0	1	0	1	0	0
2	1	0	0	0	0	0	1	0
3	0	1	0	0	0	0	0	1
4	0	1	1	0	0	0	0	0
5	0	0	1	0	1	1	0	0
6	1	0	0	0	0	0	1	0
	Capacité de la machine = 3							

Figure 24 : Matrice outils-jobs pour 8 jobs, 6 outils et une capacité de 3 avant KTNS

L'exemple de la figure 24 est basé sur un ordonnancement donné de 8 jobs et 6 outils avec une capacité de 3. Pour cet ordonnancement, on peut voir que peu de jobs ont un nombre d'outils présents sur la machine qui équivaut à la capacité de cette dernière. Seul le job 7 a un nombre d'outils équivalent à la capacité de la machine. Par contre, pour le job 1, un outil supplémentaire pourrait être chargé sur la machine. Cet outil ne serait pas utilisé pour la fabrication du job 1 mais permettrait de limiter le nombre de changements. Pour minimiser ce nombre de changements, il est ainsi intéressant de garder sur la machine un des outils du job 7 : soit l'outil 1, soit l'outil 2, soit l'outil 7. Parmi ces trois outils, c'est l'outil 1 qui sera utilisé le plus tôt. Il sera en effet utile pour le job 2 (à la position 4) alors que les deux autres outils ne seront nécessaires que pour le job 5 (à la position 7).

La procédure KTNS se base sur deux éléments clés (Tang & Denardo, 1987) :

- Aucun outil ne sera rajouté sur la machine à moins qu'il ne soit nécessaire pour la réalisation du job courant.
- Si un outil doit être inséré sur la machine, les outils qui resteront sur la machine sont les outils dont on aura besoin en premier lieu pour les prochains jobs.

Cependant, plusieurs autres subtilités sont aussi à prendre en compte pour mettre en place la procédure KTNS. Ainsi, il faut ajouter les outils adéquats au premier job de l'ordonnancement dans le cas où celui-ci n'utilise pas un nombre d'outils égal à la capacité de la ligne de production. Les outils à rajouter, appartenant à ${}^{c}T_j$, seront ceux nécessaires aux jobs occupant la position la plus proche du premier job. Cependant, si plusieurs outils appartenant à un même job pouvaient être ajoutés mais violent la contrainte de capacité, parmi ces outils, ceux étant utilisés au plus tôt par les jobs suivants seraient choisis pour être ajoutés au premier job. Pour les autres jobs, on garde les outils du job précédent qui seront utilisés le plus rapidement par les prochains jobs. Pour le(s) dernier(s) job(s), il arrive parfois qu'il y ait plusieurs outils qui

puissent être ajoutés sans qu'on sache le(s)quel(s) choisir. Dans ce cas, les outils qui seront gardés sur la machine pour ce(s) dernier(s) job(s) seront choisis aléatoirement. La figure 25 présente la matrice jobs-outils après la procédure KTNS. Les outils soulignés sont ceux qui ont été gardés sur la machine alors qu'ils sont inutiles pour la réalisation du job courant. On peut voir que pour le job 5, n'importe lequel des trois outils utilisés par le job 8 peut être gardé, cela n'a pas d'importance. Pour le job 6, deux des trois outils du job 5 peuvent être gardés sur la machine. Là encore, on peut choisir les deux outils aléatoirement. Cet ordonnancement requiert donc neuf changements au total.

Jobs / Outils	7	1	4	2	3	8	5	6
1	1	<u>1</u>	<u>1</u>	1	<u>1</u>	1	<u>1</u>	<u>1</u>
2	1	0	0	0	0	0	1	<u>1</u>
3	0	1	0	0	0	0	0	1
4	0	1	1	<u>1</u>	<u>1</u>	<u>1</u>	0	0
5	0	0	1	<u>1</u>	1	1	0	0
6	1	0	0	0	0	0	1	0
	Capacité de la machine = 3							

Figure 25 : Matrice outils-jobs pour 8 jobs, 6 outils et une capacité de 3 après KTNS

5.6 PROCÉDURE CHECK

Une procédure de mémoire a été mise en place. Celle-ci retient tous les ordonnancements visités. L'utilité d'une telle procédure est que s'il arrive qu'un ordonnancement ait déjà été visité et le soit à nouveau (si la perturbation n'a pas été assez forte par exemple), la KTNS ne sera pas appliquée à cet ordonnancement, il ne sera pas considéré et la recherche locale pourra continuer son exploration sans avoir perdu trop de temps. Normalement, cette procédure n'intervient pas très souvent sauf pour les problèmes de petites tailles. Dans ces problèmes, la recherche locale combinée à la perturbation peut donner des voisinages qui parfois se superposent les uns avec les autres. En effet, il y a moins de solutions possibles et les solutions sont dès lors plus rapidement visitées.

Dans le point suivant, la recherche locale itérative sélectionnée va être présentée. Le pseudocode ainsi que les explications relatives aux choix posés seront argumentées.

5.7 LA RECHERCHE LOCALE ITÉRATIVE SÉLECTIONNÉE

Dans les points précédents de ce chapitre, les différentes techniques expérimentées ont été mises en évidence. Cette partie va présenter l'ILS implémentée. Pour trouver l'ILS offrant les meilleurs résultats, différentes combinaisons des quatre parties composant la recherche locale itérative ont été testées. Seule la meilleure combinaison va être exposée.

5.7.1 Algorithme – explications

$sol_{initiale} \leftarrow generate_{sol_{initiale}}$
$step \leftarrow 1$
$repeat$
    $if(step > 1)do$
        $if(step\ pair)do$
            $CADB(ordbest)$
        $else$
            $BDAC(ordbest)$
        $end - if$
    $end - if$

    $if(step\ impair)do$
        $swap(ord)$
        $repeat$
            $swap(ordbest)$
        $until(no\ more\ improvement)$
    $end - if$

    $if(step\ pair)do$
        $insert(ord)$
        $repeat$
            $insert(ordbest)$
        $until(no\ more\ improvement)$
    $end - if$
    $step \leftarrow step + 1$
$until(tps_{computation} > \max\_time\ )$

La solution initiale est générée, comme dit au point 5.1., via une heuristique gloutonne. Le premier job de l'ordonnancement est le ou un des jobs utilisant le plus d'outils. Par la suite, le job ayant le plus d'outils en commun avec le dernier job de l'ordonnancement partiel est le job qui sera rajouté à la fin de l'ordonnancement partiel.

Ensuite, on entre dans une boucle que l'on ne quittera pas avant que le critère d'arrêt soit atteint. Dans le cas présent, ce critère d'arrêt est une limite de temps. En effet, une limite de temps permet de se comparer à une marche aléatoire. Il est par exemple possible de voir après X

secondes qui de l'ILS ou de la recherche aléatoire est la meilleure. Si le but était de se comparer à un autre algorithme, il aurait été intéressant d'utiliser comme critère d'arrêt un nombre d'itération défini. Or, se comparer aux résultats d'une autre métaheuristique est compliqué. En effet, les machines utilisées pour les tests sont rarement les mêmes, la qualité d'implémentation de l'algorithme n'est sûrement pas optimale étant donné que ce travail est réalisé dans le cadre d'étude de gestion et les critères d'arrêts utilisés ne sont pas les mêmes dans les différents travaux déjà réalisés sur le sujet.

Dans cette boucle, deux chemins sont possibles :

- Soit la procédure *swap* sera utilisée comme recherche locale. Il s'agit d'inverser deux jobs d'un ordonnancement mais de le faire de manière exhaustive. Ce qui veut dire que toutes les positions seront inversées entre-elles. Cependant, il y a deux subtilités :
    - Le *swap(ord)* ne va pas agir sur un voisinage mais sur le dernier ordonnancement. Ainsi, deux jobs seront échangés ; sur base du dernier ordonnancement, deux nouveaux jobs seront échangés et ainsi de suite jusqu'à ce que les $\frac{N*(N-1)}{2}$ permutations soient faites. Pour choisir les jobs à inverser, la suite logique des positions est suivie (voir figure 26).

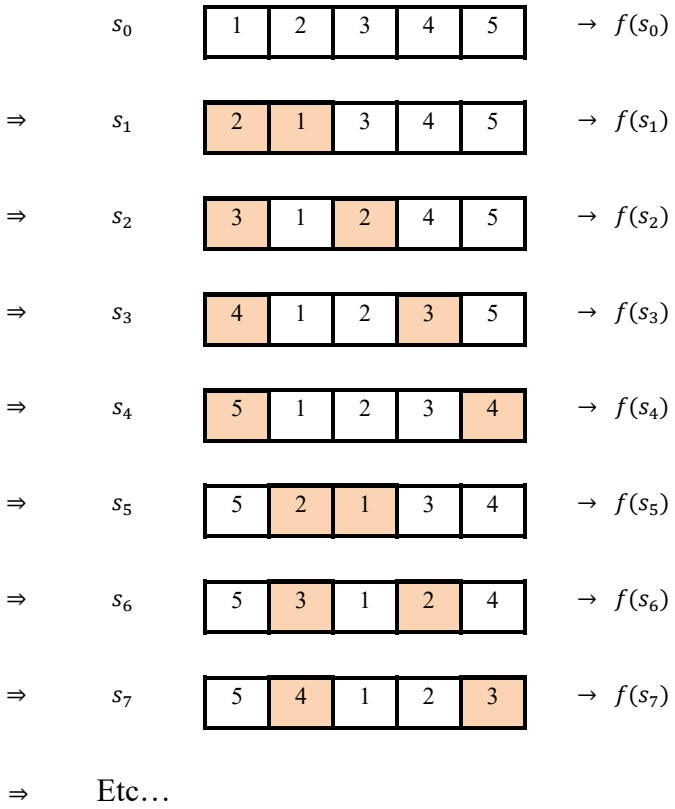

Figure 26 : Opérateur swap(ord) exhaustif

- Le *swap(ordbest)* va échanger deux jobs du meilleur ordonnancement trouvé jusqu'à présent et dès qu'un ordonnancement meilleur ou égal est trouvé, cet ordonnancement sera le nouvel ordonnancement courant de ce *swap(ordbest)*. Pour choisir le meilleur voisin, la technique *Netcrawler* est employée. Ces permutations vont cependant continuer sans recommencer à la position 1 quand une nouvelle meilleure solution est trouvée. Ainsi, comme pour le *swap(ord)*, $\frac{N*(N-1)}{2}$ permutations seront réalisées. Ce *swap(ordbest)* va être réalisé en boucle jusqu'à ce qu'il n'y ait plus d'améliorations entre la meilleure solution avant de faire un *swap(ordbest)* et après l'avoir fait.

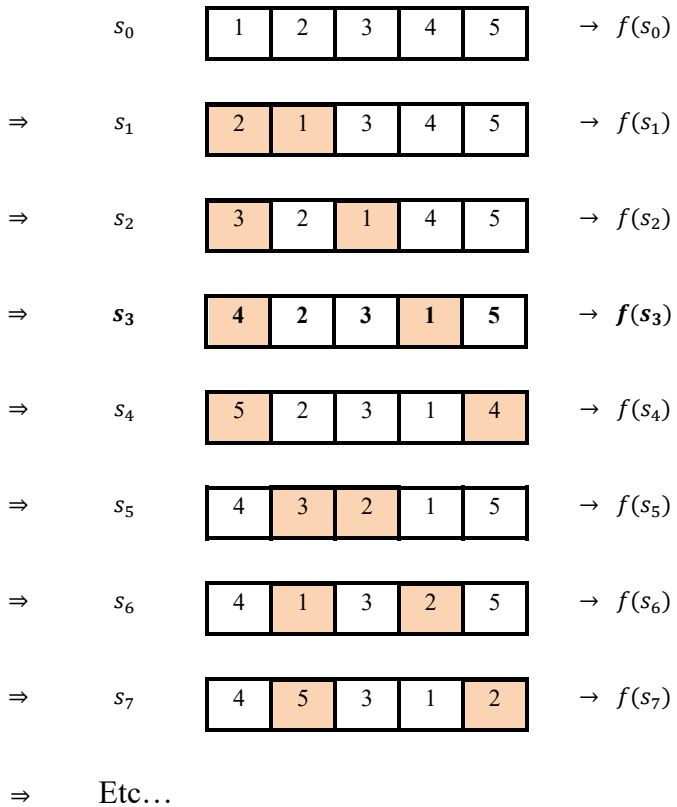

Figure 27 : Opérateur swap(ordbest) exhaustif

Dans la figure 27, la méthode de descente utilisée est illustrée. Jusqu'à s_3, le job 1 est inversé avec les autres jobs de l'ordonnancement en se basant sur l'ordonnancement s_0. L'ordonnancement s_3 a une valeur meilleure ou égale pour la fonction objective : $f(s_3) \leq f(s_0)$. Par la suite, l'ordonnancement s_3 sera celui sur lequel les inversions vont se baser.

- Soit la procédure *insert* va être utilisée pour la recherche locale. Dans ce cas-là, chaque job de l'ordonnancement sera inséré à chaque place de l'ordonnancement. Il y a deux *insert* différents :
    - L'*insert(ord)* va explorer un voisinage de manière exhaustive. Sur base de l'ordonnancement courant après perturbation, elle va explorer son voisinage complètement en insérant chaque job à chaque position afin de voir si un des ordonnancements améliore l'optimum. Pour choisir le meilleur voisin, la technique *Hill Climbing* est choisie. Cet opérateur est illustré dans la figure 8.
    - L'*insert(ordbest)* va explorer le voisinage de la meilleure solution trouvée jusque-là mais en s'arrêtant dès qu'un ordonnancement meilleur ou égal est trouvé (technique *Netcrawler*). Si cela arrive, l'*insert(ordbest)* redémarrera depuis cette nouvelle meilleure solution et l'insertion se poursuivra en suivant la suite logique job-position auquel l'*insert(ordbest)* était arrivé. Cet *insert(ordbest)* va être réalisé en boucle jusqu'à ce qu'il n'y ait plus d'améliorations entre la meilleure solution avant de faire un *insert(ordbest)* et après l'avoir fait.

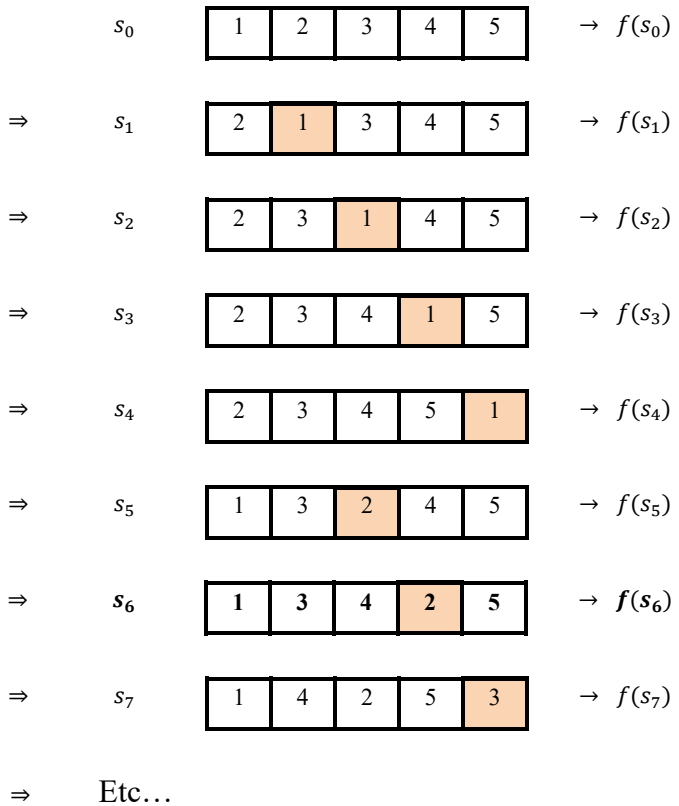

Figure 28 : Opérateur insert(ordbest) exhaustif

Dans la figure 28, jusqu'à l'ordonnancement s_6, la fonction objective n'est ni améliorée ni égalée. Les inversions se basent donc sur s_0 . Pour l'ordonnancement s_6 , $f(s_6) \leq f(s_0)$ et l'ordonnancement s_6 devient l'ordonnancement sur lequel les inversions suivantes vont se baser.

Pour les *swaps* ou les *inserts*, à CHAQUE ordonnancement, pour tester si $f(s') \leq f(s)$ la procédure KTNS est appliquée.

Une fois sortie de la boucle de la recherche locale, le critère d'acceptation, *BETTER*, va sélectionner la meilleure solution trouvée jusqu'à présent et va appliquer une perturbation dessus. Cette perturbation consiste à scinder en 4 parties le meilleur ordonnancement et à en recomposer un nouvel comme expliqué dans le point 5.3.5. En fonction du chemin pris (une fois l'un, une fois l'autre) la nouvelle solution courante sera recomposée soit selon le schéma CADB soit selon le schéma BDAC.

5.7.2 Argumentation des choix pris

Pour la solution initiale, deux choix ont été réellement envisagés : soit la générer grâce à l'heuristique gloutonne avec comme premier job un job choisi aléatoirement (dernière colonne du tableau 2), soit avec pour premier job un de ceux utilisant le plus d'outils (colonne centrale du tableau 2). Il faut savoir, que la KTNS n'entre pas en ligne de compte pour la construction de la solution initiale. Cependant, le choix de placer comme premier job un de ceux utilisant le plus d'outils s'est basé sur le fonctionnement de la procédure KTNS. Nous avons supposé qu'avoir un job avec beaucoup d'outils en première position permettrait de faciliter la procédure KTNS. Quand cette procédure devra être appliquée sur s_0, pour le job en première position, elle ne devra pas charger beaucoup d'outils en prévision des jobs futurs.

Un test a été réalisé afin d'analyser uniquement la qualité de la solution initiale et celle-ci était fort proche que l'on utilise l'une ou l'autre technique. Ce test a été réalisé sur les 16 groupes de données (chacun contenant 10 problèmes qui ont chacun été exécuté 20 fois). Ces groupes de données seront présentés au point 6.1. Dans le tableau 2, les moyennes pour chaque groupe de données sont indiquées.

Groupe de données	Sol. Initiale aléatoire	Sol. Initiale générée avec une heuristique	
		1er job avec le plus d'outils	1er job choisi aléatoirement
datA1	18,07	**14,07**	14,1
datA2	14,965	**12,1**	12,11
datA3	12,775	11,02	**10,925**
datA4	11,41	10,305	**10,275**
datB1	39,895	**31,475**	31,61
datB2	31,34	**25,84**	26,22
datB3	26,365	**22,775**	23,29
datB4	22,91	**20,77**	21,065
datC1	150,97	**119,67**	122,505
datC2	129,26	**104,57**	106,99
datC3	104,73	**86,98**	88,37
datC4	76,3	**66,915**	68,27
datD1	291,825	**235,975**	236,385
datD2	261,275	**212,625**	212,875
datD3	224,72	**184,105**	184,965
datD4	176,52	151,825	**150,58**

Tableau 2 : Comparaison des solutions initiales en fonction de l'heuristique utilisée

La seconde technique, celle plaçant comme premier job un de ceux utilisant le plus d'outils a été privilégiée car elle donne des résultats légèrement meilleurs comme on peut le constater dans le tableau 2. Cependant, pour chaque instance, moins de solutions initiales différentes existent. Les résultats du tableau 2 se trouvent sur le drive ainsi que l'implémentation de la procédure permettant de les générer.

La raison du choix de la recherche locale et de la perturbation s'explique car ils vont de pair. La recherche locale effectuée, dans les deux cas, applique une intensification forte et une diversification forte. En effet, le fait que tout le voisinage n'est pas exploré, car on se base sur la technique Netcrawler qui conserve un ordonnancement dès que celui-ci est au moins aussi bon que le meilleur trouvé jusque-là, permet de visiter beaucoup de voisinages différents. De plus, on réitère cette recherche locale tant qu'elle améliore le meilleur ordonnancement trouvé. De ce fait, beaucoup de voisinages sont explorés et une diversification est effectuée. Par contre, le fait de toujours se baser sur la meilleure solution trouvée jusqu'à présent montre le caractère d'intensification de cette recherche locale. Le critère d'acceptation, *BETTER*, intensifie aussi fort le mécanisme. Par contre, la perturbation choisie est très forte. En effet, elle découpe en quatre parties la solution optimale avant de la recomposer. On ne repart donc pas d'une solution aléatoire mais on est loin de la solution trouvée précédemment. Cette forte diversification est obligatoire pour éviter de retomber dans un des nombreux voisinages visités au préalable. Ainsi, l'*insert(ord)* et le *swap(ord)* permettent d'explorer des voisinages tout à fait nouveaux (diversification) alors que l'*insert(ordbest)* et le *swap(ordbest)* permettent de pousser à fond l'exploration du meilleur ordonnancement trouvé jusque-là (intensification). Le

fonctionnement du *swap(ordbest)* et de l'*insert(ordbest)* sont similaires car ces deux opérateurs se basent sur la meilleure solution trouvée jusqu'à présent et utilisent la technique Netcrawler pour sélectionner le voisin. L'*insert(ord)* se base sur la solution après perturbation et va utiliser la technique *Hill Climbing* pour sélectionner le meilleur voisin. Ce qui nous amène au *swap(ord)*. Celui-ci se base au départ sur la solution après perturbation et puis, continuellement sur le dernier ordonnancement visité. Ce qui est tout à fait particulier ! Une autre ILS avait été implémentée avec un *swap(ord)'* qui fonctionnait comme l'*insert(ord)* : l'opérateur *swap(ord)'* se basait alors tout le temps sur la solution après perturbation et utilisait une technique *Hill Climbing* pour sélectionner le voisin. Cependant, l'ILS configurée avec ce *swap(ord)'* n'a pas été retenue après l'avoir comparée à l'ILS présentée au point 5.7.1. Les résultats du tableau 3 comparent ces deux ILS en se basant sur les problèmes du groupe de données datD (voir point 6.1 pour plus d'informations sur ces données). La comparaison a été effectuée sur ce groupe de données car ses instances sont plus grandes et plus compliquées. Chaque instance a été effectuée dix fois. Les implémentations de l'ILS présentée au point 5.7.1 et de l'ILS dérivée (avec le *swap(ord)'*) sont disponibles sur le drive (ainsi que leurs résultats respectifs).

Groupe de données	ILS implémentée			ILS dérivée			écart
	S*	S	σ	S*	S	σ	
datD1	204,8	207,08	**1,429**	**204,1**	**206,03**	2,079	0,51%
datD2	**178,5**	**180,28**	**1,298**	179,8	181,48	1,513	-0,67%
datD3	152,4	**153,65**	**1,181**	**152**	154,23	1,762	-0,38%
datD4	**120**	**121,53**	**1,243**	120,1	121,55	1,295	-0,02%

Tableau 3 : Comparaison de l'ILS implémentée avec une ILS dérivée de celle-ci

En analysant le tableau 3, il n'est pas possible de dire avec certitude que l'une des deux ILS est meilleure que l'autre. En effet, les moyennes de l'ILS implémentée ont beau être meilleures au premier regard pour trois des quatre groupes de données mais quand on prête attention aux écarts-types, on ne peut affirmer qu'une des ILS surperforme l'autre. Dès lors, nous avons eu recours aux boîtes à moustaches afin de pouvoir visualiser si l'une des deux ILS était meilleure que l'autre. Malheureusement, il n'a pas été possible de dire avec certitude quelle recherche locale itérative performait le mieux. En effet, pour chaque groupe de problèmes, les médianes obtenues via une des ILS tombaient dans l'intervalle entre le 1er et le 3ème quartile de l'autre ILS. Nous avons aussi eu recours au test des rangs de Wilcoxon. Mais de nouveau, il n'a pas été possible d'affirmer que l'une ou l'autre ILS était meilleure. Pour ce test, l'hypothèse nulle était la suivante : les deux échantillons suivaient la même loi de distribution. Si cette hypothèse n'était pas rejetée, il ne serait pas possible d'affirmer si l'une ou l'autre ILS était meilleure. Pour trois des quatre groupes de données, l'hypothèse nulle n'a pas pu être rejetée. Etant donné

que ni les boîtes à moustaches ni le test des rangs de Wilcoxon ne permettaient de choisir avec certitude une des deux ILS, nous avons choisi celle qui offrait plus souvent des résultats moyens meilleurs et qui avait des écarts-types plus faibles. Ces analyses (via les boîtes à moustaches et via le test des rangs de Wilcoxon) se trouvent dans le drive.

Tout comme les recherches locales varient (*swap(ord)*, *swap(ordbest)*, *insert(ord)*, *insert(ordbest)*), les perturbations aussi. Cela permet d'être sûr d'avoir de nouveaux ordonnancements après perturbation. En effet, il y a deux perturbations, différentes au niveau de la reformation du nouvel ordonnancement courant, et chacune sélectionne aléatoirement trois jobs afin de faire le découpage et de reformer un nouvel ordonnancement.

5.8 TRAITEMENT PARALLÈLE

Pour évaluer la qualité de l'Iterated Local Search mise en place, cette dernière a été appliquée à différents problèmes. Chaque problème a certaines caractéristiques et cela permet ainsi de vérifier que l'ILS fonctionne bien quelles que soient les conditions. Pour chacun des problèmes, l'ILS a été appliquée vingt fois afin de pouvoir dégager une moyenne. Dans le chapitre 6, consacré à l'analyse des résultats, les données utilisées seront présentées avec plus de précision. Pour exécuter plusieurs fois différents problèmes sans perdre de temps, il est intéressant d'utiliser le calcul parallèle. En important le module « mmjobs », il est possible d'échanger de l'information entre des modèles tournant conjointement. L'ILS décrite au point 5.7.1 est implémentée dans un modèle appelé *slave* alors qu'un autre modèle appelé *master* est lui aussi créé. Le modèle *master* va exécuter le modèle *slave*. Le modèle *master* fait donc appel au modèle *slave* et va changer certains de ses paramètres. Dans ce cas-ci, les paramètres correspondront au problème auquel on se trouve et à l'itération d'exécution d'un problème (qui sera exécuté vingt fois au total). Le modèle *master* fait donc tourner le modèle *slave* en parallèle.

Ce chapitre a non seulement présenté les différentes techniques qui avaient été testées pour chacune des parties de l'Iterated Local Search mais a aussi proposé un algorithme. Cet algorithme a été obtenu après plusieurs expérimentations. Les résultats que ce-dernier donne pour un ensemble d'instances données seront présentés dans le chapitre 6.

CHAPITRE VI

ANALYSE DES RÉSULTATS

Ce chapitre va présenter les résultats obtenus quand on applique l'ILS au Job Sequencing and Tool Switching Problem. L'implémentation de l'Iterated Local Search se trouve dans le drive. Les résultats obtenus seront comparés dans un premier temps aux résultats obtenus par Catanzaro et al. (2015). Ensuite, ceux-ci seront comparés à une méthode aléatoire ainsi qu'à l'ILS mise en place par Paiva et Carvalho (2017). Pour clore ce chapitre, des pistes d'amélioration de la métaheuristique présentée seront abordées.

6.1 PRÉSENTATION DES DONNÉES

Les expérimentations ont été réalisées sur un ensemble de données fournies par Catanzaro et al. (2015). Celles-ci sont présentes sur le drive. Ces données sont divisées en quatre grands groupes (datA, datB, datC, datD) en fonction du nombre de jobs et d'outils utilisés par chacun. Pour chaque groupe, il existe aussi quatre sous-groupes ayant chacun une capacité différente d'outils pouvant être chargés sur la machine. Vous pouvez retrouver ces seize sous-groupes sur le drive.

Données	# jobs	# outils	Capacité
datA1	10	10	4
datA2	10	10	5
datA3	10	10	6
datA4	10	10	7
datB1	15	20	6
datB2	15	20	8
datB3	15	20	10
datB4	15	20	12
datC1	30	40	15
datC2	30	40	17
datC3	30	40	20
datC4	30	40	25
datD1	40	60	20
datD2	40	60	22
datD3	40	60	25
datD4	40	60	30

Tableau 4 : Présentation des données

Comme on peut le voir dans le tableau 4, il y a donc en tout 16 groupes. Chacun des groupes est composé de 10 problèmes ; il y a donc 160 instances. Pour pouvoir analyser les résultats, l'ILS a tourné vingt fois sur chaque instance.

Les implémentations ont été faites sur *Fico Xpress Ive* 8.5 (version 1.24.24) dont le langage de programmation associé est *Mosel* 4.8.3. Les expériences computationnelles ont été menées sur un ordinateur HP avec les caractéristiques suivantes:

- processeur : Intel Core i5-7200U CPU@ 2.5GHz, 2712 MHz, 2 cœurs, 4 processeurs logiques
- mémoire RAM de 8,00 Go
- système d'exploitation Windows 10 (Version 10.0.17134)

6.2 COMPARAISON DES RÉSULTATS AVEC CATANZARO ET AL. (2015)

Les notations employées dans les tableaux suivants vont être décrites :

- S^* équivaut à une moyenne de 10 valeurs, chacune étant la valeur minimale trouvée pour chaque problème.
- S équivaut à une moyenne de 10 valeurs, chacune étant la moyenne des 20 valeurs obtenues pour chaque problème.
- T correspond soit au temps moyen pour atteindre une solution (en secondes) soit il correspond au critère d'arrêt (en secondes).
- σ correspond à une moyenne de 10 valeurs, chacune étant l'écart-type d'un des problèmes.

Groupe de données	ILS implémentée				Best known solutions
	S*	S	T	σ	S
datA1	**12,5**	**12,5**	10	0	**12,5**
datA2	**10,8**	**10,8**	10	0	**10,8**
datA3	**10,1**	**10,1**	10	0	**10,1**
datA4	**10**	**10**	10	0	**10**
datB1	**26,5**	26,895	30	0,354923473	26,9
datB2	**21,7**	21,795	30	0,151076347	22
datB3	**19,7**	19,71	30	0,030779351	19,8
datB4	**19,2**	**19,2**	30	0	**19,2**
datC1	**101,1**	102,775	90	1,061210581	102
datC2	**85,2**	86,74	90	1,119918138	85,9
datC3	**69**	69,92	90	0,579595246	69,4
datC4	**52,2**	53,53	90	0,781608219	53,6
datD1	204,8	206,865	180	1,46155888	**203,2**
datD2	**178,5**	180,15	180	1,352811129	179
datD3	**151,9**	153,65	180	1,213646035	152,5
datD4	**119,5**	121,425	180	1,418523609	120,9

Tableau 5 : Comparaison des résultats de l'ILS et des résultats de Catanzaro et al. (2015)

Dans les données fournies par Catanzaro et al. (2015), pour chaque jeu de données, une *best known value* est donnée. La *best known solution* du tableau 5 équivaut à une moyenne des 10 valeurs d'un groupe de données, chaque valeur étant la *best known value* de chaque problème. Nous nous comparons donc à cette *best known solution* dans un premier temps. Pour obtenir ces solutions, les formulations décrites par Catanzaro et al. (2015) ont été implémentées en Mosel 3.4.3 sur FICO Xpress Optimizer (librairie v25.01.05). Les tests ont été effectués sur un IMac Core i7, 3.5GHz, équipé d'une mémoire RAM de 16GB avec comme système d'exploitation Mac OS X (Darwin version 10.9.1). Cependant, les temps de computation ne sont pas inclus dans les données. Nous comparerons donc uniquement les nombres de changements pour chaque ensemble de données.

On peut voir dans le tableau 5 que pour les groupes de données datA, datB et aussi datC4, les valeurs moyennes fournies par l'ILS (S) sont de qualités meilleures ou égales par rapport aux meilleures solutions connues. Pour les groupes de données datA, datB, datC, datD2, datD3 et datD4, l'ILS fournit au moins une solution de qualité meilleure ou égale aux meilleures solutions connues. Cependant, pour les groupes de données datC1, datC2, datC3 et datD, l'ILS trouve en moyenne des résultats légèrement moins bons que la meilleure solution connue. L'ILS a cependant permis de trouver pour 51 instances des nouvelles meilleures solutions. Ces résultats se trouvent dans un fichier *Excel* sur le drive.

On voit que pour les instances plus grandes, l'ILS obtient des résultats qui s'éloignent un peu des meilleures solutions déjà existantes.

C'est en se basant sur les meilleures solutions existantes, fournies par Catanzaro et al. (2015), que le critère d'arrêt a été défini. Ayant une idée de la valeur que l'algorithme devait retourner, différentes expérimentations ont été faites afin de trouver un temps, le plus faible possible, qui permettait d'avoir une solution de bonne qualité. En effet, le compromis temps de computation – qualité de la solution est compliqué à résoudre. Dans le cas de ces tests, on peut voir que les temps de computation ne sont pas très élevés tandis que la qualité de la solution est meilleure ou proche de la meilleure solution connue.

6.3 COMPARAISON DES RÉSULTATS AVEC UNE RECHERCHE ALÉATOIRE

Après s'être comparé avec les meilleures solutions trouvées par Catanzaro et al. (2015), l'algorithme ILS va être comparée avec une méthode de recherche aléatoire. Le code de cette recherche aléatoire se trouve dans le drive. Cette méthode génère aléatoirement des

ordonnancements. Pour chacun de ceux-ci, la procédure KTNS est appliquée et si l'ordonnancement a un nombre de changements plus faible, il est stocké en mémoire comme le meilleur ordonnancement. Une fois le critère d'arrêt (identique à celui de l'ILS) atteint, la recherche locale ressort le meilleur ordonnancement trouvé et le nombre de changements associé. Les valeurs des critères d'arrêts sont : 90 et 180 secondes pour respectivement datC et datD. L'ILS va en effet être comparée à la recherche aléatoire uniquement pour les groupes de données datC et datD. Pour les groupes datA et datB, l'ILS trouve déjà des résultats de qualités meilleures ou égales aux meilleures solutions déjà existantes. Par contre, pour les groupes datC et datD, les résultats moyens sont légèrement en dessous de ceux déjà existants. Cette comparaison avec une recherche aléatoire va permettre de montrer l'apport d'une méthode approximée.

Groupe de données	ILS implémentée				Recherche aléatoire				écart
	S*	S	T	σ	S*	S	T	σ	
datC1	**101,1**	**102,775**	90	1,061	123,8	127,115	90	1,438	-23,68%
datC2	**85,2**	**86,74**	90	1,120	105,2	108,33	90	1,340	-24,89%
datC3	**69**	**69,92**	90	0,580	82,8	85,985	90	1,218	-22,98%
datC4	**52,2**	**53,53**	90	0,782	61,6	63,185	90	0,842	-18,04%
datD1	**204,8**	**206,865**	180	1,462	253,4	257,965	180	1,941	-24,70%
datD2	**178,5**	**180,15**	180	1,353	223,3	227,575	180	1,793	-26,33%
datD3	**151,9**	**153,65**	180	1,214	190,7	194,655	180	1,760	-26,69%
datD4	**119,5**	**121,425**	180	1,419	149,4	152,615	180	1,380	-25,69%

Tableau 6 : Comparaison des résultats de l'ILS avec une recherche aléatoire

On peut voir que l'ILS surperforme pour chaque groupe de problèmes datC et datD la recherche aléatoire. Ce résultat était le résultat attendu et a bel et bien été confirmé. Tous les résultats et le tableau comparatif sont disponibles sur le drive.

La dernière colonne du tableau 6 présente le gain apporté par l'ILS par rapport à la marche aléatoire. Ce gain a été calculé de la sorte :

$$Gain = 100 * ((S_{ILS} - S_{RW})/S_{ILS})$$

En moyenne, la métaheuristique ILS implémentée dans le cadre de ce mémoire obtiendra un ordonnancement demandant 24.12% moins de changements. Ce gain est évidemment établi sur base du critère d'arrêt et des instances choisies dans le tableau 6. En laissant tourner l'algorithme de recherche aléatoire très longtemps, S_{RW} trouverait probablement une solution proche de S_{ILS}.

6.4 COMPARAISON DES RÉSULTATS AVEC L'ILS DE PAIVA ET CARVALHO (2017)

Après s'être comparé avec une méthode aléatoire, l'algorithme de recherche locale itérative va maintenant être confronté à l'algorithme présenté par Paiva et Carvalho (2017). Il aurait

également été possible de se comparer à l'algorithme génétique basé sur un apprentissage renforcé dynamique de Ahmadi, Goldengorin, Süer et Mosadegh (2018). Cependant, l'algorithme de Paiva et Carvalho (2017) offre des résultats de pointe et utilise une Iterated Local Search comme métaheuristique. Leur algorithme de recherche locale itérative se base sur une recherche locale de *block-grouping*. Ce *block-grouping* est associé à une autre recherche locale : le *swap*. Pour la perturbation, ils ont choisi d'utiliser le *swap 0.2*N random* (présenté au point 5.3.3). Plus de détails sur leur algorithme sont disponibles dans l'article de Paiva et Carvalho (2017). Ces derniers ont aussi testé leur algorithme sur les instances fournies dans l'article de Catanzaro et al. (2015). Ils ont réalisé leur implémentation en C++ et ont utilisé le compilateur g++ 4.4.1. Leurs tests ont été réalisés sur un ordinateur Intel i5 @3.2 GHz (4 cœurs) avec une mémoire RAM de 8 GB sous le système d'exploitation Ubuntu 15.10.

Groupe de données	ILS implémentée				ILS de Paiva et Carvalho (2017)				écart
	S*	S	T	σ	S*	S	T	σ	
datA1	**12,5**	**12,5**	10	0,000	**12,5**	**12,5**	0,09	0	0,00%
datA2	**10,8**	**10,8**	10	0,000	**10,8**	**10,8**	0,09	0	0,00%
datA3	**10,1**	**10,1**	10	0,000	**10,1**	**10,1**	0,05	0	0,00%
datA4	**10**	**10**	10	0,000	**10**	**10**	0,04	0	0,00%
datB1	**26,5**	26,895	30	0,355	**26,5**	**26,5**	1,09	0	1,47%
datB2	**21,7**	21,795	30	0,151	**21,7**	**21,7**	1,38	0	0,44%
datB3	**19,7**	19,71	30	0,031	**19,7**	**19,7**	1,121	0	0,05%
datB4	**19,2**	**19,2**	30	0,000	**19,2**	**19,2**	0,714	0	0,00%
datC1	101,1	102,775	90	1,061	**98,9**	**99,09**	99,471	0,15	3,59%
datC2	85,2	86,74	90	1,120	**82,5**	**82,82**	137,084	0,184	4,52%
datC3	69	69,92	90	0,580	**66,6**	**66,78**	172,586	0,112	4,49%
datC4	52,2	53,53	90	0,782	**51,3**	**51,47**	137,789	0,104	3,85%
datD1	204,8	206,865	180	1,462	**198**	**198,36**	488,661	0,268	4,11%
datD2	178,5	180,15	180	1,353	**173,6**	**174,05**	706,126	0,223	3,39%
datD3	151,9	153,65	180	1,214	**146,2**	**146,68**	1069,98	0,24	4,54%
datD4	119,5	121,425	180	1,419	**115,2**	**115,42**	1518,8	0,199	4,95%

Tableau 7 : Comparaison des résultats de l'ILS implémentée avec l'ILS de Paiva et Carvalho (2017)

On peut voir dans le tableau 7 que leur algorithme fournit de meilleurs résultats de manière générale. Seuls les résultats moyens (S) de datA et datB4 sont équivalents. Par contre, la meilleure solution trouvée, S^*, est identique pour datA et datB. Pour les groupes de données datC et datD, leur algorithme permet de trouver de meilleures solutions aussi bien en terme de meilleurs résultats trouvés qu'en terme de résultats moyens. En moyenne, pour les groupes de données datC et datD, leur algorithme détermine un résultat avec 4.18% moins de changements que l'ILS implémentée dans ce mémoire. De plus, leur algorithme fournit des résultats avec un écart-type très faible ! Cependant, à partir de datC et de datD, le temps computationnel mis par leur algorithme est plus important. Par contre, leur temps computationnel nécessaire pour datA et datB est beaucoup plus faible que le nôtre. L'ILS présentée dans le cadre de ce mémoire n'est

pas capable d'obtenir de tels résultats en un temps de calcul aussi si court. Ce tableau comparatif est présent sur le drive.

Etant donné le temps plus long mis par leur algorithme pour les instances datC et datD, d'autres expérimentations ont été menées. Elles avaient pour but de comparer les résultats des deux ILS pour les instances datC et datD avec comme critère d'arrêt le temps de computation moyen mis par leur algorithme.

Groupe de données	ILS implémentée			ILS de Paiva et al. (2017)			écart
	S*	S	T	S*	S	T	
datC1	101,5	102,25	99	**98,9**	**99,09**	99,471	3,09%
datC2	86	86,5	137	**82,5**	**82,82**	137,084	4,25%
datC3	69,5	69,75	172	**66,6**	**66,78**	172,586	4,26%
datC4	53,5	53,6	138	**51,3**	**51,47**	137,789	3,97%
datD1	205,2	205,7	488	**198**	**198,36**	488,661	3,57%
datD2	179,3	180,1	706	**173,6**	**174,05**	706,126	3,36%
datD3	152,7	153,45	1070	**146,2**	**146,68**	1069,98	4,41%
datD4	118,6	119,1	1518	**115,2**	**115,42**	1518,8	3,09%

Tableau 8 : Comparaison de l'ILS implémentée avec l'ILS de Paiva et Carvalho (2017) avec pour critère d'arrêt le temps moyen mis par leur algorithme

Le critère d'arrêt a beau avoir augmenté, on peut voir dans le tableau 8 que les résultats obtenus par l'ILS de Paiva et Carvalho (2017) restent de meilleure qualité. Leur ILS permet d'obtenir 3.75% moins de changements par rapport à l'ILS implémentée dans le cadre de ce travail. Ces données sont cependant à interpréter avec beaucoup de précaution car l'ILS n'a été testée que deux fois par instance pour les 80 instances. C'est la raison pour laquelle les écart-types n'ont pas été indiqués.

6.5 PISTES D'AMÉLIORATIONS

Dans le point 6.4, deux tableaux sont présentés. Les résultats moyens de l'ILS implémentée dans ce travail sont comparés dans le tableau 9 en fonction du critère d'arrêt.

Groupe de données	ILS implémentée			ILS implémentée			écart
	S*	S	T	S*	S	T	
datC1	101,5	**102,25**	99	**101,1**	102,775	90	-0,51%
datC2	86	**86,5**	137	**85,2**	86,74	90	-0,28%
datC3	69,5	**69,75**	172	**69**	69,92	90	-0,24%
datC4	53,5	53,6	138	**52,2**	**53,53**	90	0,13%
datD1	205,2	**205,7**	488	**204,8**	206,865	180	-0,57%
datD2	179,3	**180,1**	706	**178,5**	180,15	180	-0,03%
datD3	152,7	**153,45**	1070	**151,9**	153,65	180	-0,13%
datD4	**118,6**	**119,1**	1518	119,5	121,425	180	-1,95%

Tableau 9 : Comparaison des résultats obtenus par l'ILS en fonction du temps de computation

En lisant le tableau 9, nous nous apercevons que malgré l'augmentation importante du temps de calcul accordée à la recherche locale itérative, elle n'améliore que peu ou pas du tout les solutions trouvées précédemment avec le même algorithme. En moyenne, 0.45% moins de changements sont nécessaires quand l'ILS a comme critère d'arrêt le temps moyen de Paiva et Carvalho (2017) par rapport au critère d'arrêt défini de base. Il est bon de rappeler que les résultats obtenus en un temps de computation plus long font l'objet d'une moyenne d'uniquement deux valeurs pour chacune des 80 instances. Ces constations nous ont malgré tout amenés à mettre en doute la qualité de la recherche locale itérative implémentée. Trois raisons peuvent expliquer ces résultats moins performants.

La première raison est que l'algorithme atteint très vite (de l'ordre de 180 secondes pour datD) une solution de qualité satisfaisante mais que pour avoir une solution de très bonne qualité, il faudrait laisser tourner l'algorithme encore plus longtemps. Ainsi, peut-être qu'en laissant tourner très longtemps (une nuit voir un jour complet) l'algorithme implémentée dans ce travail, celui-ci retournerait une solution proche de la solution trouvée par Paiva et Carvalho (2017). En effet, on peut constater que même pour les premières instances, leur algorithme est plus performant en ce qui concerne la rapidité d'exécution. Etant donné que le temps computationnel augmente de manière exponentielle, notre algorithme a peut-être besoin d'être lancé avec un critère d'arrêt très grand pour les instances moyennes et larges. Cette piste de réponse n'est pas une réelle amélioration car le but d'une métaheuristique est malgré tout d'avoir un temps de computation acceptable. Cependant, dans les industries, on pourrait lancer l'algorithme le vendredi soir en quittant l'usine et revenir le lundi matin en ayant alors l'ordonnancement optimal des tâches pour la semaine à venir si un tel temps de computation permettait de trouver une solution proche de l'optimum trouvé jusqu'à présent.

Il se peut que l'algorithme implémenté soit plus lent ou que les conditions techniques ne soient pas optimales par rapport à celles mises en œuvre par Paiva et Carvalho (2017). En effet, le code derrière l'algorithme n'est certainement pas implémenté de manière optimale. Etant en étude d'ingénieur de gestion, l'objectif de ce mémoire n'était pas centré sur la performance du code car notre niveau de connaissances et de compétences en programmation reste limité. De plus, le langage de programmation utilisé, *Mosel*, n'est pas le langage de programmation le plus rapide. Il serait intéressant d'utiliser un langage plus rapide comme C ou C++. Ce langage a d'ailleurs été utilisé par Paiva et Carvalho (2017) pour leur implémentation. Enfin, la puissance des machines n'est pas comparable. Paiva et Carvalho (2017) utilisent un ordinateur ayant une meilleure performance en termes de fréquence et celle-ci possède 4 cœurs.

L'autre raison qui peut expliquer ces résultats moins performants concerne l'algorithme qui est fort déterministe. En effet, différents éléments poussent souvent l'algorithme vers les mêmes solutions. Ainsi, la solution initiale choisie est peu diversifiée. Si on avait choisi de construire la solution initiale via l'heuristique avec le premier job choisi aléatoirement, plus de solutions initiales différentes auraient résulté de cet algorithme. Les solutions initiales seraient alors en moyenne moins bonnes mais seraient plus diversifiées. Ensuite, le fait d'explorer les voisinages de manière non-exhaustive nous fait peut-être passer à côté de solutions optimales. En effet, mis à part l'*insert(ord)*, les autres recherches locales se basent soit sur le dernier ordonnancement visité soit sur la technique *Netcrawler* pour choisir le prochain voisin. Or ces techniques n'explorent absolument pas de manière étendue les voisinages. De ce fait, il pourrait être intéressant d'explorer de manière exhaustive chaque voisinage avec la technique du *Hill Climbing*. Cette technique a été envisagée mais entraine un temps computationnel fort important. Il faudrait donc trouver un moyen d'associer cette recherche locale avec une perturbation dirigeant la recherche dans des voisinages de bonnes qualités afin de ne pas perdre de temps à explorer des voisinages à faible potentiel. Ensuite, il y a aussi le critère d'acceptation qui a un caractère déterministe. Il sélectionne continuellement la meilleure solution trouvée jusqu'à présent. Or, s'il n'y a pas eu d'amélioration de la solution lors de la recherche locale, le critère d'acceptation sélectionnera la même solution que celle qui a déjà subi précédemment la perturbation. Heureusement, la perturbation ne renforce pas ce caractère déterministe. Elle essaie au contraire d'orienter l'exploration vers de nouveaux voisinages. Avec le critère d'acceptation RW, l'ILS ne donnait pas de meilleurs résultats. Cependant, il existe un critère d'acceptation qui mériterait d'être tester étant donné qu'on se trouve devant une ILS un peu trop déterministe : le critère d'acceptation *Backtrack*. Ce critère d'acceptation fonctionne pour β itérations comme RW (sélectionner le meilleur ordonnancement du dernier voisinage visité) mais si durant ces β itérations, aucune solution améliorante n'est trouvée, il fonctionnera comme *BETTER* en sélectionnant la meilleure solution trouvée jusqu'à présent (den Besten et al., 2001 ; Gunawan et al., 2015). Si ce critère d'acceptation était envisagé, la perturbation $ABCD \rightarrow CADB$ pourrait être utilisée pour la meilleure solution trouvée jusqu'à présent mais une autre perturbation plus légère pourrait être appliquée sur la meilleure solution du dernier voisinage.

Ce chapitre a permis d'analyser les résultats obtenus avec l'Iterated Local Search implémentée dans le cadre de ce travail. On peut voir qu'elle est plus performante qu'une recherche aléatoire mais aussi qu'elle améliore souvent la meilleure solution trouvée par Catanzaro et al. (2015). En se comparant aux résultats obtenus par Paiva et Carvalho (2017), on voit que notre ILS permet d'obtenir des résultats proches de leurs solutions. Les écarts, en moyenne de 4.18%, se font ressentir pour les instances de tailles moyennes et grandes. Enfin, étant donné que l'ILS implémentée sous-performe leur ILS, une remise en question a été effectuée et a permis de dégager des pistes d'amélioration.

CONCLUSION

Ce mémoire a abordé le Job Sequencing and Tool Switching Problem. Dans un premier temps, le problème d'ordonnancement de jobs et de gestion d'outils a été décrit. Ensuite, les méthodes de résolutions exactes existantes ont été présentées. Le point faible principal de ces méthodes concerne la durée de temps de calcul. Le chapitre 3 s'est quant à lui focalisé sur les méthodes de résolutions approchées en insistant sur les métaheuristiques. Après une description théorique de la métaheuristique utilisée, l'Iterated Local Search, l'algorithme de cette dernière a été présenté. Enfin, les résultats renvoyés par cette métaheuristique ont été analysés.

Les résultats obtenus avec l'algorithme de recherche locale itérative se sont avérés être de qualité satisfaisante. En effet, ceux-ci sont nettement meilleurs que ceux qui auraient été obtenus via une méthode de recherche aléatoire sur un laps de temps défini. Les résultats sont également de qualité meilleure ou comparable à ceux de Catanzaro et al. (2015). De plus, l'algorithme a permis de trouver 51 nouvelles meilleures valeurs pour les 160 instances sur lesquelles il a été testé. Cette comparaison se base sur les meilleures valeurs qui étaient connues jusque-là et qui ont été fournies par Catanzaro et al. (2015). Par contre, l'algorithme se montre moins performant que celui mis au point par Paiva et Carvalho (2017) pour des instances de grandes tailles. Cependant, les résultats obtenus s'écartent seulement de 4.18% par rapport à ceux de Paiva et Carvalho (2017) pour les groupes de données datC et datD. On peut donc juger ces résultats comme corrects car ce mémoire a été réalisé dans le cadre d'étude de gestion et l'implémentation de l'algorithme n'est probablement pas optimale. Résoudre le problème d'ordonnancement de jobs et de gestion d'outils avec l'Iterated Local Search permet aussi d'obtenir une réponse en un temps raisonnable. En effet, les méthodes de résolutions exactes mettent beaucoup plus de temps pour résoudre ces problèmes. Or, le temps computationnel est un facteur important ! Dans le secteur industriel, un temps de calcul un peu plus important est parfois toléré. Ainsi, faire le planning de la fabrication de produits pour une semaine peut être calculé durant le weekend alors que dans d'autres secteurs, le temps est beaucoup plus précieux. On peut constater que les temps de computation de l'algorithme présenté sont relativement faibles et que l'ILS pourrait donc être appliquée à d'autres problèmes dans lesquels le temps serait une donnée plus critique. Le compromis temps de calcul-qualité de la solution est toujours compliqué car nous souhaiterions avoir les deux. Dans certaines situations, il est possible

d'accepter un temps plus long en contrepartie de meilleurs résultats alors que dans d'autres cas, le temps prime sur la qualité. On peut également constater que l'ILS ne donne pas toujours la même solution à un problème. Un écart-type, quoique relativement faible, existe et ne permet pas toujours d'obtenir la meilleure solution si on lance l'algorithme sur une seule itération. Ceci résulte du caractère moins prévisible caractérisant les métaheuristiques.

Ce travail a appliqué la recherche locale itérative sur le SSP. Cependant, cette métaheuristique pourrait être appliquée à d'autres problèmes comme par exemple le problème du trafic aérien au-dessus des aéroports. Dans ce genre de problème, les temps de calcul seraient encore plus importants car des décisions doivent être prises en une poignée de secondes. Appliquer l'ILS à d'autres problèmes (problème du trafic aérien, problème du voyageur de commerce,…) ne devrait pas représenter une trop grande difficulté étant donné que certaines parties sont communes entre les problèmes. Les caractéristiques des problèmes varient mais le corps de la procédure pourrait rester le même. Cependant, la procédure ILS pourrait être modifiée et/ou configurée en fonction de besoins plus spécifiques.

Travailler sur les métaheuristiques m'a permis d'apprendre beaucoup de notions de recherche opérationnelle. Ainsi, les notions de voisinage, de recherche locale, de diversification et d'intensification pour ne citer qu'elles m'ont ouvert à d'autres méthodes de résolutions de problèmes combinatoires. De plus, le problème du SSP m'a aussi permis d'étudier un problème concret rencontré par les industries. Dans le cadre de ce mémoire, plusieurs simplifications ont été faites mais il me semble désormais possible de considérer ce problème en englobant plus de contraintes (par exemple des outils non-uniformes). Enfin, travailler sur ce mémoire m'a permis d'améliorer mon raisonnement en matière de programmation. En effet, plus de la moitié du temps passé sur ce mémoire l'a été sur le logiciel *FICO Xpress Ive*.

Pour clôturer ce mémoire, je reviendrai sur certaines pistes d'améliorations possibles. En effet, la métaheuristique présentée a certaines limites. Ainsi, celle-ci est un peu trop déterministe et pourrait être améliorée en permettant à l'algorithme de visiter des voisinages de manière plus aléatoire en fonction des itérations. Ensuite, un langage de programmation plus rapide, tel que C ou C++, pourrait aussi améliorer la performance de l'ILS implémentée. Enfin, l'Iterated Local Search pourrait évoluer afin de non pas se baser sur une solution mais sur un ensemble de solutions (multi-start ILS). Une méthode de recherche locale itérative basée sur une population d'individus associée à une liste taboue pourrait améliorer les résultats. Cela engendrerait un algorithme probablement plus lourd mais qui pourrait permettre aux individus de la population

de s’entraider en explorant simultanément des voisinages différents. La possibilité d’améliorer l’algorithme ILS pour le SSP reste donc bien présente.

BIBLIOGRAPHIE

Adjiashvili, D., Bosio, S., & Zemmer, K. (2015). Minimizing the number of switch instances on a flexible machine in polynomial time. *Operations Research Letters, 43(3)*, 317-322. doi:10.1016/j.orl.2015.04.001

Ahmadi, E., Goldengorin, B., Süer, G. A., & Mosadegh, H. (2018). A hybrid method of 2-TSP and novel learning-based GA for job sequencing and tool switching problem. *Applied Soft Computing, 65*, 214-229. doi:10.1016/j.asoc.2017.12.045

Al-Fawzan, M. A., & Al-Sultan, K. S. (2003). A tabu search based algorithm for minimizing the number of tool switches on a flexible machine. *Computers & Industrial Engineering, 44(1)*, 35-47. doi:10.1016/S0360-8352(02)00183-3

Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2012). Solving the tool switching problem with memetic algorithms. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM, 26(2)*, 221-235. doi:10.1017/S089006041100014X

Ambite, J. L., & Knoblock, C. A. (2001). Planning by Rewriting. *Journal of Artificial Intelligence Research, 15*, 207-261.

Bard J. F. (1988). A Heuristic for Minimizing the Number of Tool Switches on a Flexible Machine. *IIE Transactions, 20(4)*, 382-391, doi:10.1080/07408178808966195

Ben Ismail, S. (2012). Majeure Informatique - INF413 – C5 : Introduction à l'optimisation combinatoire [Présentation Powerpoint]. Telecom Bretagne.

Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, 2(4), 353-373. doi:10.1016/j.plrev.2005.10.001

Blum, C., & Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3), 268-308. doi:10.1145/937503.937505

Catanzaro, D., Gouveia, L., & Labbé, M. (2015). Improved integer linear programming formulations for the job Sequencing and tool Switching Problem. *European Journal of Operational Research*, *244*, 766-777. doi:10.1016/j.ejor.2015.02.018

Černý, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications, 45(1)*, 41-51. doi:10.1007/BF00940812

Chaves, A. A., Senne, E. L. F., & Yanasse, H. H. (2012). Uma nova heurística para o problema de minimização de trocas de ferramentas. *Gestão & Produção, 19(1)*, 17-30. doi:10.1590/S0104-530X2012000100002

Crama, Y. (1997). Combinatorial optimization models for production scheduling in automated manufacturing systems. *European Journal of Operational Research, 99(1),* 136-153. doi:10.1016/S0377-2217(96)00388-8

Crama, Y., Kolen, A. W. J., Oerlemans, A. G., & Spieksma, F. C. R. (1994). Minimizing the number of tool switches on a flexible machine. *International Journal of Flexible Manufacturing Systems, 6(1),* 33-54. doi:10.1007/BF01324874

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research, 6(6)*, 791-812. doi:10.1287/opre.6.6.791

den Besten, M., Stützle, T., & Dorigo, M. (2001). Design of iterated local search algorithms: An example application to the single machine total weighted tardiness problem. In Boers, E. J. W., & al., *Applications of Evolutionary Computing: Proceedings of EvoWorkshops* (Vol. 2037, pp. 441-451). Berlin, Heidelberg : Springer

Djellab, H., Djellab, K., & Gourgand, M. (2000). A new heuristic based on a hypergraph representation for the tool switching problem. *International Journal of Production Economics, 64(1)*, 165-176. doi:10.1016/S0925-5273(99)00055-9

Dorigo, M., Caro, G. D., & Gambardella, L. M. (1999). Ant algorithms for discrete optimization. *Artificial Life, 5(2)*, 137-172. doi:10.1162/106454699568728

Ergun, Ö., & Orlin, J. B. (2006). Fast neighborhood search for the single machine total weighted tardiness problem. *Operations Research Letters, 34(1)*, 41-45. doi:10.1016/j.orl.2005.01.010

Fathi, Y., & Barnette, K. W. (2002). Heuristic procedures for the parallel machine problem with tool switches. *International Journal of Production Research, 40(1)*, 151-164. doi:10.1080/00207540110076115

Feo, T. A., & Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization, 6(2)*, 109-133. doi:10.1007/BF01096763

Gagné, C., Sioud, A., & Gravel, M. (2014). Recherches dans le voisinage pour la résolution d'un problème de flowshop flexible et hybride avec temps de réglages dépendants de la séquence. *MOSIM 2014, 10ème Conférence Francophone de Modélisation, Optimisation et Simulation*. Nancy, France.

Gendreau, M., & Potvin, J.-Y. (Eds). (2010). Handbook of Metaheuristics (2e éd.), vol. 146, *International Series in Operations Research and Management Science*. New-York, NY : Springer.

Ghedira K. (2007). *Optimisation combinatoire par métaheuristiques : Origines, concepts et éléments de base, algorithmes canoniques et étendus (vol.20)*. In *Sciences et technologies*. Paris : Technip.

Ghiani, G., Grieco, A., & Guerriero, E. (2010). Solving the job sequencing and tool switching problem as a nonlinear least cost hamiltonian cycle problem. *Networks, 55(4)*, 379-385. doi:10.1002/net.20341

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research, 13(5)*, 533-549. doi:10.1016/0305-0548(86)90048-1

Glover, F., & Laguna, M. (1997). *Tabu search*. Boston: Kluwer Academic Publishers.

Gray, A. E., Seidmann, A., & Stecke, K. E. (1993). A synthesis of decision models for tool management in automated manufacturing. *Management Science, 39(5)*, 549-567. doi:10.1287/mnsc.39.5.549

Gunawan A., Lau H.C., & Lu K. (2015). An Iterated Local Search Algorithm for Solving the Orienteering Problem with Time Windows. In: Ochoa G., Chicano F. (eds) *Evolutionary Computation in Combinatorial Optimization. EvoCOP 2015. Lecture Notes in Computer Science, vol 9026*. Springer, Cham. doi:10.1007/978-3-319-16468-7_6

Hao J.K., & Solnon C. (2014). Méta-heuristiques et intelligence artificielle. In P. Marquis, O. Papini & H. Prade (Eds.), Algorithmes pour l'intelligence artificielle (pp.1-19). Toulouse: Cépaduès.

Hertz, A., & Kobler, D. (2000). A framework for the description of evolutionary algorithms. *European Journal of Operational Research 126*, 1–12.

Hertz, A., & Widmer, M. (1996). An improved tabu search approach for solving the job shop scheduling problem with tooling constraints. *Discrete Applied Mathematics*, 65(1), 319-345. doi:10.1016/0166-218X(95)00040-X

Hoos, H. H., & Stützle, T. (2005). *Stochastic local search: Foundations and applications*. San Francisco: Morgan Kaufmann.

Jain, A. S., Rangaswamy, B., & Meeran, S. (2000). New and "Stronger" job-shop neighbourhoods: A focus on the method of nowicki and smutnicki (1996). *Journal of Heuristics, 6(4)*,457-480. doi:10.1023/A:1009617209268

Johnson D. S., & McGeoch L. A. (1997). The travelling salesman problem: A case study in local optimization. In: E.H.L. Aarts and J.K. Lenstra (eds.), *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, England (pp. 215–310).

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220(4598)*, 671-680. doi:10.1126/science.220.4598.671

Laporte, G., Salazar-González, J. J., & Semet, F. (2004). Exact algorithms for the job sequencing and tool switching problem. *IIE Transactions*, 36(1), 37-45. doi:10.1080/07408170490257871

Lin, S. (1965). Computer solutions of the traveling salesman problem. *The Bell System Technical Journal 44 (10)*, 2245-2269. doi: 10.1002/j.1538-7305.1965.tb04146.x

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research, 21(2)*, 498-516. doi:10.1287/opre.21.2.498

López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., & Stützle, T. (2016). The irace package: iterated racing for automatic algorithm configuration. *Operations Research Perspectives, 3*, 43–58. doi: 10.1016/j.orp.2016.09.002 .

Lourenço, H.R., Martin, O., & Stützle, T. (2010). Iterated Local Search: Framework and Applications. In M. Gendreau & J.Y. Potvin (Eds.) *Handbook of Metaheuristics*, 2nd. Edition. Vol.146., *International Series in Operations Research & Management Science* (pp. 363-397). New York: Springer.

Marmion, M.-E. (2011). *Recherche locale et optimisation combinatoire : De l'analyse structurelle d'un problème à la conception d'algorithmes efficaces (Doctoral dissertation).* Lille: Université de Lille 1.

Michallet J. (2013). *Problèmes de tournées de véhicules périodiques avec contraintes de sécurité ou de qualité de service (Doctoral dissertation.* Troyes : Université de technologie de Troyes.

Mladenović, N., & Hansen, P. (1997). Variable neighborhood search. *Computers and Operations Research, 24(11)*, 1097-1100. doi:10.1016/S0305-0548(97)00031-2

Nawaz, M., Enscore, E. Jr., & Ham, I. (1983). A heuristic algorithm for the m-machine n-job flow-shop sequencing problem. *Omega, 11(1),* 91-95

Nowicki, E., & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6), 797-813. doi:10.1287/mnsc.42.6.797

Paiva, G. S., & Carvalho, M. A. M. (2017). Improved heuristic algorithms for the job sequencing and tool switching problem. *Computers and Operations Research, 88*, 208-219. doi:10.1016/j.cor.2017.07.013

Phan, R. (2013). *Méthodes exactes et approchées par partition en cliques de graphes* (Master's thesis). Université Blaise Pascal, Clermont-Ferrand II.

Salonen, K., Raduly-Baka, C., & Nevalainen, O. S. (2006). A note on the tool switching problem of a flexible machine. *Computers & Industrial Engineering, 50(4),* 458-465. doi:10.1016/j.cie.2004.11.002

Skiena, S. (1990). *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica* (pp. 196-198). Reading, MA: Addison-Wesley.

Stützle T. (1998a). Applying iterated local search to the permutation flow shop problem. *Technical Report AIDA-98-04*, University of Technology, Darmstadt.

Stützle, T. (1998b). *Local search algorithms for combinatorial problems - analysis, improvements, and new applications* (Doctoral dissertation). Technische Universität Darmstadt, Darmstadt.

Stützle, T. (2006). Iterated local search for the quadratic assignment problem. *European Journal of Operational Research, 174(3)*, 1519-1539. doi:10.1016/j.ejor.2005.01.066

Sylejmani, K. (2013). *Optimizing trip itinerary for tourist groups* (Doctoral dissertation). Department of Computer Engineering, University of Prishtina (Kosovo). doi:10.13140/RG.2.1.3390.2325

Tang, C. S., & Denardo, E. V. (1988). Models arising from a flexible manufacturing machine, part I: Minimization of the number of tool switches. *Operations Research, 36(5)*, 767-777. doi:10.1287/opre.36.5.767

Voudouris, C., & Tsang, E.P.K. (1995). Guided Local Search. *Technical report CSM-247, Department of Computer Science*, University of Essex, pp. 166.

Yang, X. S., Deb, S., & Fong, S. (2014). Metaheuristic Algorithms: Optimal Balance of Intensification and Diversification. *Applied Mathematics & Information Sciences, 8(3)*, 977-983. doi: 10.12785/amis/080306

ANNEXES

ANNEXE 1 : FICHIERS PRÉSENTS SUR LE DRIVE

« data formulation de tang et denardo.xlsx »
Reprend les données du problème (Feuil1) et les résultats (Feuil2).

« formulation de tang et denardo.mos »
Contient l'implémentation de la formulation de Tang et Denardo (1988)

« ILS modifiee - slave.mos »
Contient l'implémentation de l'ILS dérivée de celle présentée au point 5.7.1

« Iterated Local Search - master.mos »
Permet d'exécuter les fichiers « *Iterated Local Search - slave.mos* » et « *ILS modifiee - slave.mos* » un certain nombre de fois pour différents problèmes.

« Iterated Local Search - slave.mos »
Contient l'implémentation de l'ILS présentée au point 5.7.1

« Ordonnancement initial - master.mos »
Permet d'exécuter le fichier « *Ordonnancement initial - slave.mos* » un certain nombre de fois pour différents problèmes.

« Ordonnancement initial - slave.mos »
Contient l'implémentation des trois procédures testées permettant de générer une solution initiale

« Procedure KTNS.mos »
Contient l'implémentation de la procédure KTNS quand un ordonnancement est rentré par l'utilisateur

« random search - master.mos »
Permet d'exécuter le fichier « *random search - slave.mos* » un certain nombre de fois pour différents problèmes.

« random search - slave.mos »
Contient l'implémentation de la procédure de recherche aléatoire.

« dat%%.xlsx »

Chaque « *dat%%.xlsx* » présente dans les dix premières feuilles les données de dix problèmes.

La feuille 11 reprend les résultats quand l'ILS est appliquée sur ces problèmes.

La feuille 12 contient les résultats quand la procédure de recherche aléatoire est appliquée aux problèmes.

La feuille 13 montre les résultats obtenus par l'ILS quand elle a comme critère d'arrêt le temps moyen de computation de l'algorithme de Paiva et Carvalho (2017).

Les feuilles 14, 15, 16 contiennent les solutions initiales quand l'algorithme appliqué consiste, respectivement, à générer une solution initiale aléatoire, à générer une solution initiale via une heuristique avec comme premier job celui qui emploie le plus d'outils, à générer une solution initiale via une heuristique avec un premier job choisi aléatoirement.

La feuille 17 reprend les résultats obtenus quand l'ILS dérivée est appliquée sur le groupe de données datD.

« Résultats.xlsx »

La feuille 1 reprend le tableau comparant les résultats obtenus avec l'ILS et les résultats de Paiva et Carvalho (2017)

La feuille 2 reprend le tableau comparant les résultats obtenus avec l'ILS et les meilleures solutions trouvées par Catanzaro et al. (2015)

La feuille 3 reprend le tableau comparant les résultats obtenus avec l'ILS et les résultats obtenus avec la recherche aléatoire

La feuille 4 reprend le tableau comparant les résultats obtenus avec l'ILS (critère d'arrêt de base) et les résultats obtenus par l'ILS avec comme critère d'arrêt le temps moyen de computation de Paiva et Carvalho (2017)

La feuille 5 reprend le tableau comparant les solutions initiales obtenues en fonction de la procédure de solution initiale choisie

La feuille 6 reprend le tableau comparant les résultats de l'ILS implémentée dans le cadre de ce mémoire et les résultats de l'ILS dérivée (avec le *swap(ord)'*). Elle présente aussi les boîtes à moustaches de ces résultats pour les groupes de données datD.

« Test des rangs de Wilcoxon.xlsx »

La feuille 1 reprend les résultats de l'ILS implémentée et de l'ILS dérivée pour les groupes de données datD.

Les quatre autres feuilles présentent les résultats des tests du signe et des tests de Wilcoxon signé.