UCL Université catholique de Louvain

Louvain School of Management (LSM)

LOUVAIN School of Management

A Metaheuristic for the Job Sequencing and Tool Switching Problem

An Ant-Colony Optimization Algorithm

Mémoire réalisé par
Daussogne Quentin

Promoteur(s)
Catanzaro Daniele

Lecteur
Quesada José Miguel

Année académique 2017-2018
Master en Ingénieur de gestion

Chaussée de Binche, 151 - 7000 Mons, Belgique www.uclouvain.be/lsm

Résumé

Ce mémoire donne une résolution métaheuristique du problème d'ordonnancement et de changement d'outils sur une machine flexible (*Job Sequencing and Tool Switching Problem*). La métaheuristique utilisée est l'algorithme Ant Colony Optimization.

Tout d'abord, nous nous sommes concentrés sur le problème, sa définition, sa complexité. Nous avons alors déjà pu aborder dans les chapitres suivants les méthodes exactes qui permettent de résoudre ce problème. Ensuite, nous avons défini ce qu'était une heuristique qui permet d'obtenir une solution approchée de l'optimum. Dans ce chapitre, nous nous sommes concentrés principalement sur les heuristiques qui allaient être utilisées par après dans le cadre de l'implémentation de l'algorithme Ant-Colony. Ensuite, nous avons résumé le fonctionnement d'un bon nombres de métaheuristiques afin de mettre en avant les différentes possibilités pour résoudre des problèmes combinatoires comme celui qui nous intéresse dans ce mémoire. Enfin, nous nous sommes concentrés sur l'algorithme des fourmis dont nous avons implémenté la variante Ant Colony System. Le but était d'arriver à obtenir un résultat plus satisfaisant qu'une méthode aléatoire sur un court laps de temps. Ensuite, nous avons proposé des pistes d'amélioration de l'algorithme pour obtenir des meilleurs résultats dans cette même durée.

Table des matières

Liste des tables 3
Liste des figures 5
Introduction 7
Chapitre 1 : The job Sequencing and tool Switching Problem (SSP) 9
1.1. Contexte 9
1.2. Les paramètres du problème 9
1.3. L'objectif et la complexité 11
Chapitre 2 : La méthode exacte 15
2.1. Définition 15
2.2. Application d'une méthode exacte au SSP 15
2.2.1. Notation 16
2.2.2. La formulation selon Tang et Denardo (Tang & Denardo, 1988) 16
2.2.3. La formulation selon Laporte et al. (Laporte et al., 2004) 18
2.2.4. Travaux postérieurs 19
Chapitre 3 : Les méthodes heuristiques 21
3.1. Définition 21
3.2. K-opt Strategies 22
3.3. Shortest-edge heuristic 23
Chapitre 4 : Les méthodes métaheuristiques 25
4.1. Définition 25
4.2. Métaheuristiques appliquées au SSP 27
4.2.1. *Tabu Search* (*TS*) 27
4.2.2. L'*Iterated Local Search (ILS)* 30
4.2.3. L'*Evolutionary Computation (EC)* 31
4.3. Autres métaheuristiques 34
4.3.1. La *Basic Local Search : Iterative Improvement* 34
4.3.2. La *Simulated Annealing* (*SA*) 35
4.3.3. Explorative Local Search Methods 37
Chapitre 5 : L'algorithme ACO 43
5.1. L'origine de l'algorithme 43
5.2. Le fonctionnement de l'algorithme 44

5.3. The ant-colony optimization metaheuristic 47
5.4. Variantes de l'algorithme ACO 50
Chapitre 6 : Application de l'algorithme ACO au problème de JSTSP 53
6.1. Mise en place d'algorithme 53
6.1.1. L'algorithme *Ant-Colony System* 53
6.1.2. Les paramètres utilisés 56
6.2. Résultats computationnels 56
6.3. Pistes d'amélioration 61
Conclusion 65
Bibliographie 67

Liste des tables

Table 1: Exemple de problème 10
Table 2: Exemple de solution au problème de la table 1 10
Table 3: Explications des différents problèmes testés 57
Table 4: Comparaison des résultats de l'ACS avec ceux de l'ILS de Paiva et al. (2017) pour les instances A et B 58
Table 5: Comparaison des résultats de l'ACS avec ceux d'une méthode aléatoire pour les instances C et D 59
Table 6: Comparaison des écart-types entre l'ACS et la méthode aléatoire pour les instances C et D 60
Table 7: Comparaison des résultats de l'ACS avec modification de q_0 62
Table 8: Comparaison des résultats de la méthode ACS avec ceux de l'ACS avec recherche locale à chaque étape de la construction pour les instances C et D 63
Table 9: Comparaison des résultats de l'ACS lorsque le nombre d'itérations est différent pour les instances C et D 64

Liste des figures

Figure 1: K-opt Strategies (Blazinskas et Misevicius, 2011) 23
Figure 2: Algorithme : Tabu Search (TS) 29
Figure 3: Algorithme : Iterated Local Search (ILS) 31
Figure 4: Algorithme : Evolutionary Computation (EC) 32
Figure 5: Algorithme : Iterative Improvement 35
Figure 6: Algorithme : Simulated Annealing (SA) 36
Figure 7: Algorithme : Greedy Randomized Adaptive Search Procedure (GRASP) 38
Figure 8: Algorithme : Variable Neighborhood Search (VNS) 39
Figure 9: Algorithme : Guided Local Search (GLS) 42
Figure 10 : Sélection du chemin le plus court par une colonie de fourmis 43
Figure 11: Modèle de base d'ACO 44
Figure 12: Fonctionnement de l'algorithme (Blum, 2005) 47

Introduction

Le monde de la *supply chain* aujourd'hui n'est plus le même qu'avant, les chaînes de productions demandent plus de flexibilité et sont de plus en plus complexes. C'est dans ce cadre qu'est apparu le problème qui sera résolu dans ce mémoire.

Il est nécessaire aujourd'hui d'optimiser les *process* et donc d'éviter les coûts inutiles. C'est pour cela que mettre en place un ordonnancement efficace des outils de production est très important. Ce mémoire se concentrera sur un des problèmes qui touchent l'industrie moderne, le *Job Sequencing and Tool Switching Problem*. Après un chapitre explicatif du problème. Nous cheminerons par les différentes méthodes proposées par de nombreux chercheurs afin de résoudre ce type de problème combinatoire complexe allant de la méthode exacte aux métaheuristiques tout en passant par les méthodes heuristiques. Ensuite, nous arriverons à l'application de la méthode approchée qui est l'objectif de ce mémoire : un algorithme de colonie de fourmis (*Ant Colony Algorithm* ou *ACO*) pour résoudre des problèmes combinatoires. Après un chapitre explicatif sur cet algorithme, nous ferons une analyse des résultats de l'implémentation d'une de ses variantes permettant de résoudre le problème de départ ainsi que d'éventuelles pistes d'amélioration proposées (le code ainsi que les analyses des résultats se trouvent sur une *Dropbox* à l'adresse suivante : https://www.dropbox.com/sh/6kfrebnugovvv9q/AADcUSEgFBMmIaF_azZSjEbva?dl=0).

Chapitre 1 : The job Sequencing and tool Switching Problem (SSP)

Avant de mettre en place l'algorithme *ACO*, il faut bien comprendre le problème, sa difficulté, ses paramètres et ses complexités. Ce chapitre expliquera donc dans quel cadre est apparu ce problème, quel est l'objectif d'un tel problème et quels sont les différents enjeux, paramètres et variables.

1.1. Contexte

Désormais, il est de plus en plus souvent demandé aux industries d'être flexibles. Cela demande nécessairement de devoir s'adapter en termes d'équipements, ou de procédures de production. Cependant, les entreprises essaient toujours de limiter les coûts et les investissements qui peuvent être évités. De ce fait, beaucoup d'industries utilisent aujourd'hui des machines dites « flexibles ».

Ces machines peuvent accomplir différentes tâches sans devoir faire de grands changements entre deux opérations. Ces machines ont un dépôt ayant une capacité maximale à ne pas dépasser dans lequel sont stockés les différents outils dédiés à une opération. Il faudra donc planifier la production au mieux pour ne pas accomplir trop de changements d'outils si le nombre d'outils disponibles pour accomplir tous les « jobs » est supérieur à la capacité de la machine. Changer des outils signifie interrompre la ligne de production et engendre donc un coût entre chaque opération. (Crama, 1997; Crama, Moonen, Spieksma, & Talloen, 2007; Tang & Denardo, 1988 ; Gray et al., 1993; Paiva & Carvalho, 2017)

1.2. Les paramètres du problème

Ce problème est composé d'un ensemble d'opérations ou de jobs devant être exécutés $J=\{1,....n\}$ ainsi que d'un ensemble d'outils disponibles $T=\{1,....t\}$. A chaque job est dédié un ensemble d'outils ne dépassant pas la capacité C de la machine. Cet ensemble est noté T_k et représente les outils nécessaires à l'accomplissement de la tâche k appartenant à J.

La solution de ce problème sera une séquence de job φ de l'ensemble J et une proposition de programmation des outils à chaque moment de la production. Pour ce problème, nous considérons que le coût des changements d'outils est identique, peu importe le *switch* effectué, que les outils ont la même taille et que chaque tâche n'intervient qu'une fois dans la chaîne de production. (Crama, Kolen, Oerlemans et Spieksma, 1994; Ghiani, Grieco, & Guerriero, 2010; Laporte, Salazar-Gonzáles, & Semet, 2004; Tang & Denardo, 1988 ; Belady, 1966)

Table 1: Exemple de problème

Jobs	1	2	3	4	5	6	7	8	9	10
Outils	4	3	8	1	1	1	3	6	4	1
	6	4	10	5	2	5	4	9	5	3
	7	9		10	5	8	7	10	7	9
							8			

Magazine capacity = 4

Table 2: Exemple de solution au problème de la table 1

φ	7	9	1	2	10	8	3	4	6	5
Outils	3	3	3	3	1	1	1	1	1	1
	4	4	4	4	3	6	6	5	5	2
	7	5	6	6	6	9	8	8	8	5
	8	7	7	9	9	10	10	10	10	8

Magazine capacity = 4

La table 1 est un exemple de problème de *SSP* avec 10 jobs et 10 outils et le tableau 2 est une solution possible au problème avec la séquence φ de jobs. le plan de changements d'outils est représenté par une matrice binaire $S^{\varphi}_{m,n}$. Dans cette matrice, nous suivons l'ordre des jobs et les colonnes sont remplies selon que l'outil est utilisé ou non. S'il est utilisé, nous aurons $S^{\varphi}_{i,j} = 1$, où i est l'outil chargé pour le job j, ce sera égal à 0 dans le cas contraire. Le nombre de changements est calculé par le nombre d'inversions entre deux jobs tel que $S^{\varphi}_{i,j-1} = 0$ et $S^{\varphi}_{i,j} = 1$ $(\forall i \in T, \mathrm{j} \in \{2, \dots, n\}$, l'outil i étant placé dans la machine entre les deux jobs j-1 et j. Le placement des outils pour le premier job est le coût de chargements et doit être compté comme changements d'outils. Par exemple, si la capacité est de 4, il faudra placer 4 outils au départ et

donc il y aura 4 changements de base. L'explication de cette matrice sera expliquée à partir d'un exemple dans le point suivant lorsque sera abordée la procédure permettant de remplir le dépôt d'outils efficacement.

Dans la table 2, il s'agit d'une proposition de solution avec un ordonnancement φ sur la première ligne. Les autres lignes correspondent aux outils chargés dans la machine à chaque opération. Les numéros d'outils entourés sont des outils non nécessaires à l'opération de cette étape-là. Pour désigner quels seront ces outils, on utilise la méthode *KTNS* (*Keep Tools Needed Soonest*). Par exemple, l'outil 3 est nécessaire à l'exécution du job 7 mais pas à celui du job 9 et 1, cependant, c'est l'outil le plus rapidement utilisé ensuite. Cela permet de limiter le nombre de changements et d'éviter de retirer les outils qui vont être utilisés le plus rapidement. (Tang & Denardo, 1988). La méthode *KTNS* est expliquée au point suivant. Le coût de changement pour cet exemple est de 12. On compte le chargement des outils avant la mise en route de la production dans ce coût. Donc il y a 4 outils chargés et ensuite, 8 changements.

1.3. L'objectif et la complexité

L'objectif du problème *SSP* (aussi appelé *JSTSP*) sera donc de limiter les coûts de changement entre les opérations afin d'optimiser la chaîne de production. Il est possible de le diviser en deux sous-problèmes (Tang & Denardo, 1988) :

- Le problème de séquençage (*SP* ou *Sequencing Problem*) qui consiste à ordonnancer convenablement les différents « jobs » afin de minimiser le nombre de changements d'outils. Le *SP* est un problème d'optimisation combinatoire NP-Difficile ;
- Le problème d'outils (*TP* ou *Tooling Problem*) qui consiste à minimiser dans l'ordonnancement le nombre de chargements et de déchargements d'outils de la machine. Le problème *TP* peut être résolu simplement avec la méthode de *Keep Tools Needed Soonest* (KTNS).

Dans un problème d'optimisation combinatoire, nous allons chercher à atteindre un objectif qui peut être un nombre entier, un sous-ensemble, une permutation ou une structure de graphe. Un problème d'optimisation combinatoire *P=(S,f)* se définit par un ensemble de variables $X=\{x_1,\ldots, x_n\}$, des domaines variables $D_1,\ldots,D_n$, des contraintes et une fonction objective *f* qui doit être minimisée où $f : D_1 \times \ldots \times D_n \rightarrow \mathbb{R}^+$. L'ensemble des solutions possibles sera $S = \{s = \{(x_1 v_1), \ldots, (x_n v_n)\} | v_i \in D_i,\ s$ satisfait toutes les contraintes$\}$. S est habituellement

appelé un espace de recherche ou un espace de solution. Pour résoudre ce type de problème, il faut trouver une solution $s^* \in S$ avec une valeur de la fonction objective qui est $f(s*) \leq f(s) \quad \forall s \in S$. La solution s^* est une solution optimale globale de et nous avons l'ensemble S^* inclus dans S qui est l'ensemble des solutions globales (Papadimitriou et Steiglitz, 1982). Des exemples de problèmes combinatoires sont le *TSP* (*travelling Salesman Problem*), le *QAP* (*Quadratic Assignment Problem*), …

La procédure *Keep Tools Needed Soonest*:

La procédure *KTNS* est primordiale dans le cadre du problème de *Job Sequencing and Tool Switching Problem* car elle permet de combler les vides dans le dépôt d'outils avec des outils non nécessaires pour le job *j* afin de minimiser le nombre de changements pour les jobs suivants. Cette méthode n'est utile dans le cas du *SSP* que parce qu'il s'agit de résoudre des problèmes où la capacité du dépôt n'est pas toujours atteinte par les outils nécessaires à l'exécution d'un job ($|Tj| < C$). Cette méthode sera donc utilisée pour la partie *Tooling Problem*.

L'objectif de la procédure sera donc de charger pour le job *j* de l'ordonnancement φ, les outils nécessaires à l'exécution du job mais également les outils non nécessaires qui interviendront le plus vite dans les jobs $j+1$ à $j+n$. Pour optimaliser ce chargement d'outils, Tang et Denardo, dans la procédure KTNS qu'ils présentent en 1988, remplissent tout d'abord au maximum de sa capacité le dépôt de la machine lors de l'exécution du premier job avec les outils les plus rapidement utilisés par la suite. La procédure commencera réellement à partir du deuxième job de la séquence.

Il y a donc deux « règles ». Il faudra toujours garder dans le dépôt les outils inutiles qui seront utilisés le plus rapidement dans la séquence et il ne faut pas provoquer le chargement d'un outil à l'avance sans qu'il soit nécessaire directement.
Crama et al. (1994) ont démontré que la procédure de *KTNS* proposait une solution optimale en essayant de minimiser les *0-Blocks.* Par exemple :

$$P = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

$P_{i,j}$ est une matrice de permutation où *i* représente la ligne des outils *i* et *j* représente le job *j*. Lorsque $P_{i,j} = 1$, cela signifie que l'outil *i* est chargé sur la machine pour le job *j*. La capacité du dépôt est de 3. Le but de la manœuvre sera de repérer les blocs de 0, c'est-à-dire, les suites de 0 dans une ligne ayant à chaque extrême un 1. Ici, il y aura donc comme 0-*block* {(1,2)}, {(1,5)}, {(2,3)(2,4)}, {3,3),(3,4),(3,5)}, {(4,2),(4,3)}, {(5,5)}.
Ces blocs suivent les conditions suivantes :

1. $1 < j \leq j + k < N$
2. $P_{i,j} = P_{i,j+1} = \cdots = P_{i,j+k} = 0$
3. $P_{i,j-1} = P_{i,j+k+1} = 1$

Nous savons que la capacité *C* à ne pas dépasser est de 3, donc pour continuer, il faut déterminer à chaque job *j* la capacité restante pour pouvoir charger des outils supplémentaires :

$$c_j = C - \sum_{i=1} P_{i,j}$$

Dans notre exemple, c_j ne sera pas nul pour le job 2, 3 et 5. Donc il sera possible de charger des outils pour ces trois jobs. Pour terminer la procédure, il s'agira de charger l'outil qui sera utilisé le plus rapidement. Dans la matrice, nous remplacerons donc dans les 0-*block*, les 0 par des 1 sans toutefois dépasser la capacité maximale d'outils autorisée. On obtiendra donc la matrice d'outils *O* suivante :

$$O = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Dans notre problème, il est possible que le ou les dernier(s) job(s) ai(en)t une capacité disponible. Pour y remédier, il s'agira de garder en stock les outils précédemment utilisés.
Ensuite pour calculer le nombre de changements, il suffit de calculer la différence entre chaque colonne et de diviser par 2. On obtient alors le nombre de changements sans compter le coût de chargement de base.

La propriété de symétrie

Ghiani, Grieco, & Guerriero nous permettent de ne pas oublier une propriété importante du problème de *JSTSP*, la propriété de symétrie :

> Pour chaque séquence de jobs, son nombre de changements minimum sera le même que l'inverse de cette séquence. Cette propriété a permis notamment de réduire le nombre d'ordonnancements à tester lors de la résolution du problème par des méthodes exactes. (Ghiani et al., 2010 ; Ghiani et al., 2007)

Pour résoudre ce problème de *SSP*, il existe plusieurs méthodes. Dans le chapitre 2, nous aborderons les méthodes exactes avec programmations linéaires. Dans le chapitre 3, nous expliquerons les différentes méthodes heuristiques permettant d'obtenir une solution approchée et ensuite nous aborderons les méthodes métaheuristiques. Au travers de ces chapitres, nous comprendrons pourquoi faire le choix d'une métaheuristique plutôt que la méthode exacte, quels en sont les avantages et les inconvénients.

Chapitre 2 : La méthode exacte

Dans ce chapitre sera expliqué en quoi consiste une méthode exacte. Pour la comprendre au mieux, nous allons l'appliquer au problème *JSTSP* avec les premières formulations proposées et évoquer les travaux postérieurs. Ce sera l'occasion d'aborder les résultats qu'ont donnés ces différentes méthodes.

2.1. Définition

Les méthodes exactes permettent de trouver la meilleure solution possible, la solution optimale (Dorigo et Stützle, 2004). Elles sont généralement utilisées pour des problèmes de petites tailles car elles prennent du temps à trouver une solution vu qu'elles passent en revue toutes les possibilités. Lorsqu'il s'agit d'une optimisation combinatoire, il existe plusieurs méthodes exactes permettant de résoudre un problème. Il y a les algorithmes de *branch and bound*, l'optimisation linéaire en nombres entiers ou la programmation par contraintes par exemple.
Lorsque le nombre de paramètres ou le problème est trop grand ou trop complexe, qu'il s'agit d'un problème NP-difficile, il est plus courant d'utiliser des méthodes approchées, c'est-à-dire des heuristiques ou métaheuristiques. Il est possible cependant d'utiliser une méthode exacte mais les temps de calcul computationnels sont trop grands pour être optimum. C'est pour cela qu'il est parfois plus utile d'utiliser une métaheuristique. (Blum et Roli, 2003, Dorigo et Stützle, 2004)

2.2. Application d'une méthode exacte au SSP

Jusque maintenant, il y a eu beaucoup de recherches développant des approches heuristiques pour solutionner le *SSP* mais peu se sont concentrées sur les approches exactes.
Tout d'abord, Tang et Denardo (1988) ont proposé la formulation de ce problème en un programme linéaire pour résoudre de manière exacte un *SSP* et pour proposer une heuristique approximant sa solution optimale. Cependant, les deux approches n'étaient pas assez performantes. C'est pour cela que Laporte, Salazar-Gonzáles et Semet (2004) ont proposé une formulation d'un programme linéaire d'entier (*ILP* ou *Integar Linear Programming*) et un algorithme de *branch and bound* pour résoudre le *SSP*. Cette formulation est performante pour

un problème de moins de 10 jobs avec une heure de temps de computation mais est incapable de résoudre un problème plus grand. En revanche, le *branch and bound* donna des meilleurs résultats pour 25 jobs avec un même temps de computation.

Ghiani, Grieco et Guerriero (2010) ont également proposé une méthode exacte pour résoudre le *SSP* en se basant sur un modèle non linéaire du *TSP* (*Travelling Salesman Problem* ou Problème du Voyageur de Commerce). En améliorant l'algorithme *KTNS* pour le *Tooling Problem* et en développant un algorithme *branch and cut*, ils ont réussi à surpasser la proposition de Laporte en permettant l'utilisation de leur algorithme sur 45 jobs sur une même durée.

2.2.1. Notation

Nous avons un ensemble $J = \{1,2,\dots,n\}$ de *n* jobs à exécuter dans une machine flexible, ces jobs $j \in J$ pouvant être exécutés par des outils *m* faisant partie de l'ensemble $T = \{1,2,\dots,m\}$. L'ensemble J_t sera l'ensemble des outils $t \in T$. Le complément de J_t sera ${}^{C}J_t$. Pour exprimer la capacité du dépôt de la machine, nous aurons $C \geq \max_j \{|T_j|\}$

2.2.2. La formulation selon Tang et Denardo (Tang & Denardo, 1988)

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

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

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

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

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

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

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

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

Dans la formulation de Tang et Denardo, L'ensemble *K* est l'ensemble des *n* positions dans la séquence de jobs devant être déterminées. La variable de décision u_j^k est égale à 1 lorsque le

job *j* est exécuté en position *k*, 0 sinon. Pour savoir si un outil *t* est utilisé à la position *k*, la variable de décision v_t^k est égale à 1, 0 sinon. Enfin, la dernière variable de décision de cette formulation est w_t^k et sera égale à 1 lorsque l'outil *t* est utilisé pour le job exécuté en *k* mais pas en *k*-1.

Nous pouvons alors expliquer le modèle de Tang et Denardo. Tout d'abord, la fonction objective (1a) sert à minimiser le nombre total de changements d'outils. La somme $\sum_{t\in T} v_t^1$ sera le nombre d'outils à charger au départ dans la machine lors de l'exécution du premier job. Les contraintes (1b) et (1c) sont des contraintes d'affectation. Respectivement, à chaque position *k* dans *K*, il y aura un job *j* assigné et pour chaque job *j*, il y aura une position *k* assignée. Ces contraintes sont présentes pour que tous les jobs soient utilisés et que toute la séquence soit remplie. La contrainte (1d) permet d'éviter d'assigner un job *j* nécessitant l'outil *t* à la position *k* si l'outil n'est pas présent dans le dépôt à cette position. La contrainte (1e) permet de ne jamais dépasser la capacité maximale. Les inégalités peuvent devenir des égalités s'il n'y a pas besoin d'enlever les outils non utilisés (Laporte et al., 2004). Avec la contrainte (1f), le modèle comptera un changement d'outil à partir du moment où un outil est utilisé en *k* mais pas en *k*-1.

Cette formulation démontre certains inconvénients que Laporte, Salazar-Gonzáles et Semet (2004) ont tenté d'effacer avec une autre formulation. En effet, le programme linéaire de Tang et Denardo affichait une solution optimale et faisable avec ces différentes valeurs pour les variables de décision :

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

Ils ont alors proposé un programme linéaire faisant un lien entre le problème du TSP et le sous-problème *SP*.

2.2.3. La formulation selon Laporte et al. (Laporte et al., 2004)

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

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

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

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

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

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

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

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

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

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

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

Laporte et al. ont commencé en introduisant un job fictif 0 pour marquer le début et la fin de la procédure. Évidemment, T_0, l'ensemble d'outils pour le job 0 est vide. Pour résoudre ce problème en se rapprochant du *TSP*, ils ont décidé de créer un cycle hamiltonien démarrant au job 0 et y revenant à la fin de l'exécution lorsque tous les jobs ont été « visités » une fois. Le poids entre chaque job du circuit correspondra au nombre de changements entre les deux jobs. Pour comprendre la formulation ci-dessus, il faut comprendre $J_0 = J \cup \{0\}, J_i = J \setminus \{i\}$ pour tout $i \in J$, et $J_0^i = J_0 \setminus \{i\}$ pour tout $i \in J$ et G est le graphe complet possédant un ensemble J_0 de vecteurs. La variable x_{ij} est une variable de décision binaire qui est égale à 1 si le job *j* suit directement le job *i* dans l'ordonnancement pour tout *i* et *j* dans l'ensemble J_0 avec $i \neq j$. Ensuite, la seconde variable de décision, y_j^t, est également binaire et sera égale à 1 si l'outil $t \in T$ est dans le dépôt de la machine à l'exécution du job *j*, 0 sinon, pour tout $j \in J_0, t \in T$. Enfin, z_j^t est la dernière variable de décision étant égale à 1 quand l'outil $t \in T$ est inséré dans le dépôt à l'exécution de *j,* pour tout $j \in J_0, t \in T$.

Maintenant que les différentes variables de décisions sont définies, voici les détails du modèle. La fonction objective (2a) minimise le nombre de changements total comme pour Tang et Denardo. Les deux contraintes suivantes (2a) et (2b) sont des contraintes d'affectation comme dans la formulation précédente. La contrainte (2d) empêche le cas des sous-tours et la contrainte

(2e) est une contrainte de capacité. La contrainte (2f) permet d'éviter que dans l'ordonnancement optimal le job *j* suive le job *i* si, au même moment, l'outil *t* requis pour l'exécution de *j* n'est pas chargé dans la machine ou n'est pas inséré au moment du job *i*. La contrainte (2g) est une contrainte complémentaire à (2f) lors de la transition du job fictif 0 au premier job de l'ordonnancement. Les contraintes (2h) et (2i) assure qu'un changement d'outil est exécuté seulement quand l'outil est nécessaire immédiatement dans l'exécution d'un job.
Cette formulation dépasse celle de Tang et Denardo. Cependant, les résultats des expériences computationnelles sont toujours peu satisfaisants. Leur méthode permet de résoudre des problèmes de 25 outils en 1h de computation. (Laporte, Salazar-Gonzáles, & Semet, 2004).

2.2.4. Travaux postérieurs

Tout d'abord, Ghiani et al. (2010) ont poursuivi les travaux de Laporte et al. en proposant un problème non linéaire de cycle hamiltonien. Pour les auteurs de l'article, le modèle de Laporte représente la seule formulation exacte du SSP. Ils ont proposé une solution qui permet d'obtenir un résultat optimal pour 45 jobs en développant un algorithme de *branch and cut*.

Basé sur ces nombreux travaux, Catanzaro, Gouveia et Labbé (2015) ont proposé un modèle linéaire d'entier (*ILP*) plus performant avec un plus grand nombre de problèmes résolus et un temps de computation plus rapide. Cependant il y a un écart plus important entre les solutions. L'inconvénient majeur des méthodes exactes est la difficulté d'obtenir des résultats valables pour un nombre de données élevé. De plus, les temps de computation sont parfois trop grands et donc il est parfois préférable d'utiliser des métaheuristiques qui permettent d'approcher la solution optimale même si on ne l'atteint pas (Blum et Roli, 2003 ; Papadimitriou et Steiglitz, 1982 ; Nemhauser et Wolsey, 1988). Ces différentes métaheuristiques seront expliquées dans le chapitre 4 après que nous ayons abordé les heuristiques.

Chapitre 3 : Les méthodes heuristiques

Dans ce chapitre sera abordée l'approche heuristique d'un problème combinatoire. Il commencera par une brève définition de ce qu'est une heuristique et ensuite, nous développerons de manière théorique les heuristiques qui seront utilisées dans le cadre de la mise en place d'un *Ant Colony System* pour le problème de *JSTSP*.

3.1. Définition

Une heuristique est une méthode de calcul qui permet d'obtenir rapidement une solution réalisable qui peut ne pas être optimale ou exacte dans le cadre de problème d'optimisation difficile.

Les heuristiques sont utilisées pour des optimisations combinatoires, en théorie des graphes, en théorie de la complexité des algorithmes et en intelligence artificielle. Les heuristiques sont utilisées pour obtenir une solution approchée ou pour arriver plus rapidement à une solution exacte. Les heuristiques sont adaptées à des problèmes spécifiques mais elles peuvent simplement utiliser des principes généraux. (Papadimitriou et Steiglitz, 1982)

Pour résoudre des problèmes d'optimisation combinatoire, beaucoup d'algorithmes ont été développés. Outre les algorithmes complets abordés au chapitre 2, il existe des algorithmes approximatifs. Contrairement aux méthodes exactes, l'utilisation de ces heuristiques permet d'obtenir des résultats satisfaisants mais pas spécialement optimaux en un temps de computation inférieur. Nous pouvons encore diviser cette catégorie en deux méthodes distinctes, les méthodes constructives et les méthodes de recherche locale :

- Les algorithmes constructifs construisent une solution en rajoutant au fur et à mesure de l'algorithme des éléments de solution. La résolution commence donc avec un ensemble vide qui se remplira pendant l'exécution de l'algorithme jusqu'à ce que la solution soit complète. Par exemple, pour le *SSP*, ce sera à partir du moment où l'on a un ordonnancement comprenant tous les jobs à effectuer. Dans cette catégorie, il y a les *Traveling SP heuristics, Block minimization heuristics, Greedy heuristics* (heuristiques gloutonnes) et *Interval heuristics.* Pour en savoir plus sur ces heuristiques, vous pouvez lire les écrits de Crama et al. (1994) ;

- Les méthodes de recherche locale, par contre, partiront d'une solution et remplaceront itérativement cette solution par une meilleure dans le voisinage de la solution de l'itération précédente. Vu que l'algorithme donnera des solutions dans un voisinage déterminé, nous parlerons de minimum local.

Les méthodes constructives sont les plus rapides mais donnent des résultats de qualité inférieure. (Blum et Roli, 2003)

Dans le cadre de l'implémentation de l'algorithme *ACO* au dernier chapitre, nous allons utiliser une méthode heuristique de recherche locale et une constructive. La suite de ce chapitre sera donc dédiée à l'explication de ces 2 heuristiques.

3.2. K-opt Strategies

L'heuristique *K-opt* est une méthode très efficace pour la résolution de problèmes comme celui du voyageur de commerce. C'est une recherche locale qui démarre donc d'une solution initiale donnée à améliorer.

La plus connue et utilisée pour le *TSP* est la *2-opt*, la méthode globale de cette heuristique est expliquée en détail dans les écrits de Crama et al. (1994). Tout d'abord, il s'agira de sélectionner deux jobs *i* et *j* dans la séquence initiale dont l'échange aboutit en une séquence améliorant le résultat. La séquence renvoyée par l'heuristique se note σ, s'il n'y a pas d'amélioration, nous garderons la séquence de départ.

Il existe deux options, soit on choisit de stopper l'heuristique à la première amélioration ou alors nous laissons la méthode parcourir toutes les possibilités et choisir la meilleure. Si l'heuristique *K-opt* est combinée avec une autre méthode, s'arrêter à la première amélioration permettra d'alléger l'algorithme.

Pour une méthode *2-opt*, il s'agit de supprimer deux arcs et de les remplacer par deux autres arcs. D'une manière plus simple, il s'agit juste d'inverser la séquence comprise dans le segment entre les deux arcs. La méthode plus poussée, *3-opt,* supprimera dans la même logique trois arcs qui seront remplacés par trois autres. La complexité de cet algorithme sera de $O(N^kM)$. Pour notre problème, vu qu'il conviendra d'exécuter la méthode KTNS sur chaque séquence de l'algorithme, la complexité deviendra $O(N^{k+1}M)$. Ces méthodes seront utilisées dans le cadre

de la *Local Search* dans l'algorithme des fourmis qui sera développé dans les deux derniers chapitres.

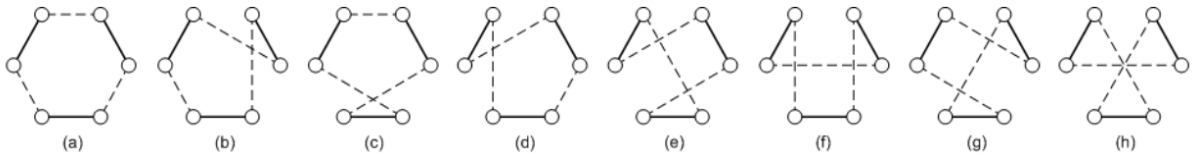

Figure 1: K-opt Strategies (Blazinskas et Misevicius, 2011)

Dans les différents schémas de la figure 1, nous pouvons voir plusieurs mouvements *2-opt* (a,b,c,d) ainsi que des mouvements *3-opt* pour le e,f,g,h.

3.3. Shortest-edge heuristic

Cette méthode est un algorithme glouton utilisé par Tang et Denardo (1988). Il s'agit simplement de construire la solution initiale en choisissant à chaque fois le job suivant le plus proche. Dans notre cas, ce sera l'arc qui demandera le moins de changements d'outils. Il s'agit d'une méthode qui construit une solution deux à deux et donc elle ne se concentre absolument pas sur les choix futurs ou sur une mémoire. Sa complexité est de $O(N^2 logN)$.

Cette méthode très simple sera utilisée dans l'algorithme *ACS* mis en place pour résoudre le *JSTSP*.

Des méthodes heuristiques découlent les métaheuristiques qui seront expliquées en détail dans le chapitre suivant et qui sont des méthodes mêlant des règles heuristiques avec des méthodes stochastiques.

Chapitre 4 : Les méthodes métaheuristiques

Dans ce chapitre, nous aborderons les méthodes métaheuristiques qui sont une alternative à la méthode exacte et qui permettent d'obtenir des résultats parfois moins optimaux mais offrant certains avantages lorsqu'elles sont utilisées sur des problèmes combinatoires de grande taille. Il est important de décrire les différentes métaheuristiques existantes afin de pouvoir comprendre le fonctionnement de celles-ci et leurs différentes approches pour résoudre un problème combinatoire. Certaines des métaheuristiques expliquées dans ce chapitre ont d'ailleurs déjà été appliquées au problème du *JSTSP*.

4.1. Définition

Durant les 30 dernières années, les métaheuristiques sont apparues. Ces algorithmes approximatifs combinent des méthodes heuristiques pour explorer efficacement les solutions possibles. Le terme « métaheuristique » par Glover (1986) est composé de deux mots grecs :

- « Heuristique » dérivé du mot *heuriskein* qui signifie « trouver »
- « Meta » qui signifie « au-delà, dans un niveau supérieur »

Avant que ce terme soit utilisé, le terme d'heuristique moderne était utilisé (Reeves, 1993). Nous pouvons faire une liste non exhaustive des métaheuristiques existantes : *Ant Colony Optimization* (*ACO*), *Evolutionnary Computation* (*EC*) incluant les *Genetic Algorithms* (*GA*), *l'Iterated Local Search* (*ILS*), *Simulated Annealing* (*SA*) et la *Tabu Search* ((*TS*).

Il est difficile de donner une définition claire et acceptée de tous pour le terme métaheuristique. Beaucoup de chercheurs ont tenté de donner leur définition propre. Il y a celle d'Osman et Laporte (1996), Voß, Martello, Osman et Roucairol (1999) et Stützle (1999) par exemple, dont Blum et Roli (2003) ont pu en résumer certaines des propriétés générales :

- Stratégies guidant la recherche ;
- L'objectif est d'explorer efficacement l'espace de recherche pour trouver une solution optimale ;
- Les techniques vont de la simple recherche locale à des procédés de recherche complexe ;
- Elles sont approximatives et non-déterministes ;

- Elles mettent en place des mécanismes qui permettent de ne pas être enfermé dans un seul endroit de l'espace de recherche (il n'est pas possible de s'enfermer dans un optimum local) ;
- Les concepts de base permettent un niveau de description abstrait ;
- Elles ne sont pas spécifiques à un seul problème.

Les différents algorithmes peuvent être classés selon certaines caractéristiques :

1) Nous pouvons les classer en fonction de leur origine, c'est-à-dire qu'ils seront soit inspirés de la nature ou non. Nous aurons alors les algorithmes *ACO* et *GA* pour les premiers et l'*ILS*, la *TS* pour les non-inspirés de la nature ;
2) Nous pouvons distinguer les algorithmes s'ils sont basés sur une population ou sur la recherche d'un seul point. Est-ce que l'algorithme se base sur une population (*population*-based) ou sur une solution à chaque instant (*trajectory method*) ? Les algorithmes qui fonctionnent sur base d'une seule solution sont des méthodes de trajectoire, c'est le cas des algorithmes de recherche locale (*Tabu Search, Iterated Local Search, Variable Neighborhood Search*). On parle de trajectoire car ils décrivent une trajectoire dans l'espace de recherche durant l'exécution de la métaheuristique. Un algorithme basé sur la population décrira au contraire l'évolution d'un ensemble de points dans l'espace de recherche ;
3) Il est possible également de classifier les métaheuristiques selon qu'elles aient une fonction objective dynamique ou statique. Par exemple, la *Guided Local Search* (*GLS*) modifie sa fonction objective durant l'exécution de l'algorithme ;
4) Ensuite, nous pouvons émettre une classification selon que l'algorithme considère un seul voisinage ou non ;
5) Enfin, il est possible de les classifier en fonction des méthodes d'utilisation de la mémoire. Est-ce que l'algorithme utilise une mémoire court-terme ou long-terme ? La mémoire à court terme permet de mémoriser des solutions précédentes alors que la long-terme retiendra plutôt la valeur de paramètres à propos de la recherche. (Blum et Roli, 2003)

Il y a deux termes importants dans une recherche de métaheuristique : la diversification et l'intensification. Comme on le sait, une métaheuristique est un algorithme stochastique et donc il agira avec une partie aléatoire. Cependant, durant les itérations d'un algorithme, il est possible de faire évoluer la solution vers une autre en favorisant soit la diversification soit

l'intensification. La diversification permettra de visiter des solutions « plus lointaines », d'explorer des zones ayant moins de probabilité d'être visitées alors que l'intensification prônera de passer par des chemins déjà connus par l'algorithme. (Blum et Roli, 2003)

Dans la suite de ce chapitre, nous expliquerons les différentes métaheuristiques à partir de l'article de Blum et Roli (2003) mais également l'ouvrage de Dreo, Petrowski, Siarry et Taillard (2003) ainsi que des ouvrages de référence pour chaque métaheuristique. Tout d'abord, nous commencerons par les métaheuristiques qui ont déjà été appliquées au *SSP* et ensuite, nous expliquerons les autres métaheuristiques existantes.

4.2. Métaheuristiques appliquées au SSP

4.2.1. *Tabu Search* (*TS*)

La recherche taboue est la métaheuristique la plus utilisée pour les problèmes combinatoires. Les principes de base de cette méthode ont été donnés par Glover (1986 et 1977) et l'algorithme est décrit de manière exhaustive par Glover et Laguna (1997). Dans cette méthode, l'historique de recherche sera utilisé pour s'échapper des minima locaux et pour implémenter une stratégie d'exploration.

L'algorithme simple applique une recherche locale avec meilleure amélioration et utilise une mémoire à court terme pour ne pas s'enfermer dans un minimum local et pour éviter d'être coincé dans un cycle. On utilise la mémoire à court terme pour enregistrer dans une *tabu list* les solutions les plus récemment visitées afin de ne plus faire les mêmes mouvements. En utilisant cet algorithme, le voisinage de la solution actuelle comportera donc des solutions n'appartenant pas à la *tabu list* mais plutôt à un ensemble que nous nommerons *allowed set*. A chaque itération, on sélectionnera dans ce *set* la meilleure solution qui deviendra la solution actuelle et qui deviendra le point de départ à l'itération suivante. Cette solution est évidemment ajoutée à la *tabu list* à la place d'une ancienne solution qui est supprimée de cette liste selon une règle *FIFO (First In First Out)*. (Glover et Laguna, 1997 ; Blum et Roli, 2003) La recherche tabou peut être considérée comme une méthode dynamique vu que l'ensemble des solutions disponibles est modifié à chaque itération (Stützle, 1999b). L'algorithme s'arrête lorsqu'une condition de terminaison est remplie ou si l'ensemble *allowed set* est vide, cela arrive seulement au cas où la *tabu list* interdit de visiter d'autres solutions. Il est cependant possible d'éviter

l'arrêt de la recherche en permettant l'utilisation de la solution la moins récente de la *tabu list*. (Blum et Roli, 2003 ; Dreo et al., 2003)

Cet algorithme permet de ne pas retourner sur une solution déjà visitée et cela empêche les cycles infinis. En effet, il permet aussi d'utiliser des solutions moins bonnes pour éviter ce cycle. La longueur *l* de la liste tabou (ou *tabu tenure*) permet de contrôler la mémoire. Plus *l* sera grand, plus l'espace de recherche sera grand alors que s'il est petit, on se concentrera sur des petites surfaces de recherche. Cette longueur peut varier au fur et à mesure des itérations. Dans certains algorithmes , la *tabu tenure* augmente lorsqu'il faut augmenter la *diversification* et diminue s'il n'y a plus d'améliorations afin d'augmenter l'*intensification*.

L'utilisation de la mémoire court-terme pour implémenter une liste de solution complète n'est pas pratique et gérer ces listes est inefficient. De ce fait, il sera plus utile de stocker des *attributs* de la solution. Ce sont des composants de la solution, des mouvements ou des différences entre les solutions. Pour chaque attribut, il y aura une *tabu list* dédiée. Les attributs et les listes dédiées forment les conditions *tabu* qui permettent de déterminer le voisinage et l'*allowed set*. Cette technique est bien plus efficace mais elle engendre une perte d'informations car si on interdit un attribut parmi plusieurs types, cela pourrait probablement priver la recherche de plus d'une solution et il est donc aussi probable de ne pas visiter une solution de bonne qualité qui aurait pu améliorer la recherche par après. Pour contourner ce problème, on peut définir un critère d'aspiration (*aspiration criteria*) qui permettra de placer une solution dans l'ensemble autorisé malgré que la condition *tabu* l'en empêche. Ce critère d'aspiration déterminera donc également les conditions d'aspiration qui permettront de remplir l'*allowed set*. Généralement, ces conditions se résument à garder des meilleures solutions que la meilleure solution actuelle. (Blum et Roli, 2003 ; Glover et Laguna, 1997)

L'utilisation de *tabu list* n'est pas le seul moyen de se servir de l'historique de recherche et de la mémoire. Généralement, cela se matérialisera par l'utilisation d'une mémoire court-terme mais on peut également utiliser une mémoire long-terme pour mémoriser toute la procédure de recherche. C'est particulièrement utile pour mettre en place un guidage stratégique de l'algorithme. Cette utilisation de la mémoire à long terme s'accorde selon 4 principes (Blum et Roli, 2003):

- Le caractère récent : la mémoire mémorise chaque solution ou attribut de l'itération la plus récente ;

- La fréquence : l'algorithme garde en mémoire le nombre de fois où une solution ou un attribut sont apparus dans l'exécution de l'algorithme ;
- La qualité : ce principe permet de garder en mémoire les bons composants des solutions ;
- L'influence : cette propriété va regarder les choix qui ont été faits pendant la recherche et pourra indiquer lesquels ont été les plus critiques.

$s \leftarrow$ GenerateInitialSolution()

InitializeTabuLists(TL_1 , . . . , TL_r)
$k \leftarrow 0$
while termination conditions not met **do**

$AllowedSet(s, k) \leftarrow \{s' \in N(s) \mid s$ does not violate a tabu condition, or it satisfies at least one aspiration condition$\}$

$s \leftarrow$ ChooseBestOf ($AllowedSet(s, k)$)

UpdateTabuListsAndAspirationConditions()

$k \leftarrow k+1$

endwhile

Figure 2: Algorithme : Tabu Search (TS)

Al-Fawzan and Al-Sultan (2002) ont mis en place un algorithme tabou afin de résoudre le problème du *SSP*. Pour obtenir les voisinages, ils ont utilisé deux méthodes. Ils ont simplement utilisé des *swap* entre deux jobs choisis aléatoirement dans la séquence ou alors ils ont déplacé des segments entiers de la séquence vers une nouvelle position. Comme nous en parlions précédemment, l'utilisation de la mémoire est compliquée à gérer. Dans ce cas, la mémoire long-terme sera utilisée pour pénaliser des solutions en fonction de ces modifications et de la présence des différentes paires de nœuds dans la liste taboue. Pour sélectionner le *swap* ou le déplacement de segments, un mécanisme est mis en place afin d'effectuer une oscillation stratégique entre les deux possibilités.

4.2.2. L'*Iterated Local Search (ILS)*

L'*ILS* est un algorithme simple mais très performant décrit par Stützle (1999a, 1999b), Lourenço, Martin et Stützle (2001 et 2002) et Martin, Otto et Felten (1991). Tout d'abord l'algorithme part d'une solution initiale, effectue une recherche locale jusqu'à trouver un optimum local, ensuite, il effectue une perturbation de la solution et il recommence une recherche locale. Si la perturbation est trop petite, il aura tendance à rester dans le même « bassin » d'attraction et donc il y a de grandes chances de retomber sur le même optimum alors qu'une perturbation trop grande serait équivalente à recommencer la recherche locale à partir d'une solution développée aléatoirement. Il peut être utilisé comme base d'autres algorithmes tels que la méthode *VNS* (expliquée par la suite) mais peut également inclure des autres méthodes dans son fonctionnement.

Dans cet algorithme, il exécute une recherche locale autour de *s* jusqu'à trouver un minimum local, il perturbe la solution trouvée et obtient *s'* à laquelle il applique une recherche locale jusqu'à trouver un minimum local. En fonction d'un critère d'acceptation, il remplace la solution trouvée avant la perturbation par la nouvelle solution et il recommence la perturbation. L'objectif de perturber une solution est de ne pas trop s'en éloigner, sinon on pourrait simplement utiliser un ordonnancement trouvé aléatoirement. Le critère d'acceptation joue un rôle de contrepoids en donnant un feedback à l'action de perturbation en fonction des caractéristiques du nouveau minimum local. (Blum et Roli, 2003)

L'avantage de cet algorithme est qu'il y a beaucoup de libertés dans le choix de la solution initiale, de la perturbation et également des critères d'acceptation (Blum et Roli, 2003) :

- La solution initiale peut être choisie aléatoirement ou suivant une heuristique ;
- La perturbation, quant à elle, n'est généralement pas déterministe pour ne pas tomber dans un cycle. La force de la perturbation est le point le plus important, elle peut être fixe ou variable. Si elle est fixe, cela veut dire que la distance entre deux solutions est constante peu importe la taille du problème. Plus le problème est grand, plus il faudra favoriser une force élevée. Elle peut être variable si on l'adapte en fonction d'un besoin de diversification ou d'intensification. Le mécanisme pour obtenir le résultat de la perturbation peut être aléatoire ou semi-déterministe ;

- Le critère d'acceptation peut simplement être d'accepter une solution seulement en cas d'amélioration ou de toujours accepter la solution proposée. Ce sont des exemples extrêmes évidemment. En effet, il y a plusieurs possibilités pour trouver un équilibre entre les deux. Il est possible d'accepter toutes les solutions si elles sont meilleures mais en ajoutant une probabilité de choisir aussi une solution même si elle est moins bonne.

```
s0 ← GenerateInitialSolution()
s^ ← LocalSearch(s0)

        while termination conditions not met do

        s' ← Perturbation(s^, history)
        s^' ← LocalSearch(s')
        s^ ← ApplyAcceptanceCriterion(s^, s^', history)

        endwhile
```

Figure 3: Algorithme : Iterated Local Search (ILS)

Cette méthode a été implémentée pour le problème du *SSP* par Paiva et Carvalho (2017) et a fourni des résultats plus que satisfaisants.

4.2.3. L'*Evolutionary Computation (EC)*

L'algorithme *EC* est inspiré de la capacité qu'à la nature à s'adapter à son environnement. Ce sont des modèles computationnels de processus évolutionnaires. A chaque itération, chaque individu de la population actuelle se voit appliquer un nombre d'opérateurs pour générer une nouvelle génération. Ces opérateurs s'appellent des *recombination* (recombinaison) ou *cross-over* (croisement) pour recombiner 2 individus ou plus afin de reproduire une nouvelle génération. Il existe d'autres types d'opérateurs de mutation et de modification qui permettent aux individus de s'adapter. Pour mettre en place cet algorithme, il faut faire une sélection des individus, elle sera faite sur base de leurs aptitudes (*fitness*) caractérisées par une valeur d'une fonction objective ou le résultat d'une simulation, … Pour effectuer cette sélection, l'algorithme utilise une probabilité de sélectionner un individu ayant la plus haute aptitude possible. La sélection permet de désigner quels seront les individus parents de la prochaine génération. On effectue donc une sélection naturelle selon la loi du plus fort avec l'adaptation aux changements d'environnement que peuvent supporter seulement les individus ayant les bonnes aptitudes. (Blum et Roli, 2003 ; Dreo et al., 2003)

Il existe plusieurs versions différentes *d'Evolutionary Computation*. Tout d'abord, Fogel (1962) et Fogel, Owens et Walsh (1966) ont proposé l'*Evolutionary Programming (EP)*. Quelques années plus tard, Rechenberg (1973) développe l'*Evolutionary Strategies (ES)*. Enfin, Holland (1975) a développé des *Genetic Algorithms (GAs)*. Les algorithmes génétiques sont communément utilisés pour résoudre les problèmes d'optimisation combinatoire.

$P \leftarrow$ GenerateInitialPopulation()
Evaluate(P)

while termination conditions not met **do**

$P' \leftarrow$ Recombine(P)

$P'' \leftarrow$ Mutate(P')

Evaluate(P'')
$P \leftarrow$ Select($P'' \cup P$)

endwhile

Figure 4: Algorithme : Evolutionary Computation (EC)

Dans l'algorithme *EC*, nous pouvons trouver 3 éléments importants. Le premier est caractérisé par le *P* qui représente la population d'individus. De cette population, l'algorithme générera une population de progéniture grâce aux opérateurs de mutation et de recombinaison. Il sélectionne les individus pour la génération suivante parmi les anciens et la nouvelle population.

Les populations d'individus traités dans l'algorithme ne sont pas spécialement des solutions du problème. En effet, elles peuvent être des solutions partielles, des ensembles de solutions, des objets pouvant être transformés en solutions. Les algorithmes *EC* sont principalement utilisés dans les problèmes combinatoires en représentant des solutions comme des séries de *string/bits* ou des permutations de *n* nombres entiers mais elles peuvent aussi se décliner sous forme d'arbres ou de structures complexes. Dans les algorithmes génétiques, on décline les individus en termes de *génotypes* et les solutions en *phénotypes* afin de pouvoir distinguer les représentations des solutions et les solutions elles-mêmes. (Blum et Roli, 2003 ; Dreo et al., 2003)

Lors de la procédure évolutive, il faut sélectionner les individus entrant dans la population de l'itération suivante. Il y a deux types de sélection :

- Remplacement générationnel : l'algorithme choisit la prochaine génération exclusivement dans les progénitures des individus
- Si c'est possible de le faire, l'algorithme transfère les individus de la population actuelle dans la suivante. A ce moment-là, on cherche une stabilité dans la procédure d'évolution.

Vu que la population change à chaque itération, il faut également se poser la question de la taille de celle-ci. La plupart imposent une taille à la population mais il est possible d'avoir une taille variable. Si c'est le cas, l'algorithme pourrait s'arrêter si la population se compose d'un individu. (Blum et Roli, 2003 ; Dreo et al., 2003)

Le 3e élément de l'*EC* après les individus et le processus d'évolution est la structure de voisinage. C'est une fonction $N_{EC} : I \rightarrow 2^I$ sur l'ensemble des individus. Chaque individu $i \in I$ se verra assigné un voisinage tel que $N_{EC}(i) \subseteq I$ qui peut servir de partenaire de recombinaison pour l'individu *i* pour créer des progénitures. Il y a deux types d'algorithmes *EC* qui ressortent de cette règle. On parlera de populations non structurées lorsqu'un individu peut se recombiner avec n'importe quel individu de la population, dans l'autre cas, on parlera de populations structurées. (Blum et Roli, 2003 ; Dreo et al., 2003)

Ensuite, la génération de nouvelles progénitures nécessite une source d'information. Ce sera un couple de parents (un croisement de deux parents). Il est également possible d'avoir un croisement de plus de deux parents afin de créer un nouvel individu (Eiben et al., 1994). Certains algorithmes utilisent même maintenant des statistiques de la population afin de générer la suivante. (Blum et Roli, 2003 ; Dreo et al., 2003)

Évidemment, il y a des solutions infaisables et donc des individus infaisables. Cela peut arriver après une recombinaison où la progéniture n'est pas possible. Il y a 3 possibilités pour gérer ce problème : (Blum et Roli, 2003 ; Dreo et al., 2003)

1) Rejeter les infaisables
2) Pénaliser les individus infaisables
3) Réparer une solution infaisable

Bien souvent, la première possibilité est assez difficile, il sera donc plus simple de pénaliser une solution en jugeant de la qualité de l'individu.

Comme dans toute métaheuristique abordée jusque maintenant il est question de diversification ou d'intensification. Les algorithmes EC ont des stratégies spécifiques pour diversifier ainsi que pour intensifier la recherche. Lorsqu'ils favoriseront la recherche locale et donc l'intensification, on parlera souvent d'algorithme mimétique. (Moscato, 1989 et 1999) D'autres méthodes utilisent les opérateurs de recombinaison afin de combiner les bonnes parties d'un individu. Ces stratégies sont appelées *linkage learning* ou *building block learning*. Pour les stratégies de diversification, il s'agit de résoudre un des problèmes de l'*EC*, c'est-à-dire que ces métaheuristiques ont tendance à trouver des solutions sous-optimales. Pour éviter cela, il faudra utiliser les opérateurs de mutation pour effectuer une perturbation dans la solution.

Amaya, Cotta et Fernandez-Leiva (2008) ont mis en place un algorithme génétique afin de résoudre le *SSP*. Ils ont utilisé un algorithme mimétique inspiré de la sélection naturelle. Nous pouvons également citer plus récemment Ahmadi, Goldengorin, Süer et Mosadegh (2018) qui ont utilisé une méthode *Dynamic Q-learning technique* pour guider un algorithme génétique. Les résultats sont satisfaisants mais sur un plus grand nombre d'exécutions. En effet, il ne trouve pas systématiquement la meilleure solution comparée à l'ILS de Paiva et Carvalho (2017) qui a produit des résultats beaucoup moins éparpillés.

4.3. Autres métaheuristiques

4.3.1. La *Basic Local Search : Iterative Improvement*

Pour cette métaheuristique, Il s'agira de chercher un mouvement dans la solution actuelle seulement s'il y apporte une meilleure solution. Cette méthode cherchera donc une amélioration dans un voisinage donné et s'arrêtera seulement lorsque l'algorithme aura trouvé un optimum local. (Blum et Roli, 2003)

On notera le voisinage de la solution, $N(s)$. Il y aura dans cet algorithme une fonction d'amélioration qui scannera le voisinage de la solution implémentée au début et choisira la première solution qui sera meilleure que la solution *s*. Le niveau de qualité de la solution sera donc directement lié aux voisinages de *s*, à la solution *s* choisie et à la fonction d'amélioration. Pour des problèmes combinatoires, les résultats ne sont généralement pas satisfaisants. C'est pour cette raison qu'habituellement, on ajoute à cet algorithme des mécanismes pour ne pas s'enfermer dans un minimum local. Les conditions d'arrêt de l'algorithme seront alors plus complexes. Soit on arrêtera l'algorithme après un certain temps, après un nombre d'itérations

atteint ou à partir du moment où on atteint une valeur donnée dans la solution. (Blum et Roli, 2003)

```
s ← GenerateInitialSolution()
repeat
s ← Improve (N(s))
until no improvement is possible
```

Figure 5: Algorithme : Iterative Improvement

4.3.2. La *Simulated Annealing* (*SA*)

La métaheuristique « Simulated Annealing » est parmi toutes celles existantes la plus ancienne. Cet algorithme trouve son origine dans la mécanique statistique et fut le premier à être proposé pour la résolution de problèmes combinatoires. (Kirkpatrick, Gelatt et Vecchi, 1983 ; Cerny, 1985) Pour éviter de s'enfermer dans un minimum local, l'algorithme permet d'obtenir une solution de moins bonne qualité que l'actuelle. Pour choisir cette solution moins bonne, on instaure une probabilité de sélectionner celle-ci. Cette probabilité diminue au fur et à mesure de la recherche. (Blum et Roli, 2003 ; Dreo et al., 2003)

Tout d'abord l'algorithme choisit une solution aléatoirement ou selon une heuristique constructive et initialise une température *T*. A chaque itération, il y aura une solution $s' \in N(s)$ qui sera sélectionnée aléatoirement. Cette solution sera alors acceptée ou non en fonction de *f(s)*, *f(s')* et *T*. La solution *s'* remplacera d'office *s* si $f(s') < f(s)$. Si jamais *f(s)* est toujours supérieur, la règle de probabilité rentrera en jeu, c'est dans ce cas de figure que l'algorithme peut choisir une solution moins bonne. Cette probabilité sera fonction de *T* et $f(s') - f(s)$. La probabilité suit généralement une distribution de Boltzmann : $\exp\left(-\frac{f(s')-f(s)}{T}\right)$. Au début de l'algorithme, la température *T* est élevée et diminue durant la recherche pour finalement, revenir à la métaheuristique d'*Iterative Improvement.* Plus la différence entre la nouvelle solution *s'* et la solution actuelle *s* sera grande, plus la chance de garder *s'* sera faible et évidemment plus *T* sera grand, plus la probabilité d'accepter un changement de solution sera grande. La méthode *Simulated Annealing* est un combiné de deux stratégies : un parcours aléatoire et une amélioration itérative. On parle de « *annealing* » en référence au procédé de recuit (*annealing process*) des métaux et des verres. (Blum et Roli, 2003)

Au début de la recherche, l'algorithme permet de parcourir une grande partie du graphe et au fur et à mesure, l'algorithme prônera une convergence vers un minimum local. Pour que l'algorithme soit le plus performant possible, il faut choisir une bonne valeur de *T* pour chaque itération. (Blum et Roli, 2003 ; Dreo et al., 2003)

La règle pour modifier ce *T* est appelée la « *cooling rule* » en rapport avec le « *annealing process* ». Cette règle détermine l'évolution de la température à chaque itération. En fonction de l'itération, la règle peut varier pour avoir le bon équilibre entre diversification et intensification. Au début de la recherche, il est probable que *T* soit constant ou suive une diminution linéaire pour élargir les possibilités et qu'elles suivent ensuite une loi de probabilité géométrique pour trouver un minimum local (Blum et Roli, 2003 ; Dreo et al, 2003).

Cette métaheuristique peut être décrite comme une procédure dynamique et une chaîne de Markov (Feller 1968) parce que l'algorithme suit une trajectoire dans l'espace et que les successeurs sont choisis selon leur prédécesseur. Aux pages précédentes, il était question de mémoire, dans ce cas-ci, on peut dire que l'algorithme *SA* est un algorithme demandant peu de mémoire.

```
s ← GenerateInitialSolution()
T ← T0
while termination conditions not met do
        s′ ←PickAtRandom(N(s))
        if (f(s′) < f(s)) then
                s←s′ % s′ replaces s
        else
                Accept s′ as new solution with probability p (T, s′, s)
        endif
        Update (T)
endwhile
```

Figure 6: Algorithme : Simulated Annealing (SA)

4.3.3. Explorative Local Search Methods

Dans ce point seront abordées les méthodes de trajectoire les plus récentes. Elles sont au nombre de 4, la *Greedy Randomized Adaptive Search Procedure*, *Variable Neighborhood Search*, *Guided Local Search, Iterated Local Search*. L'*Iterated Local Search* a déjà été expliquée au point précédent.

4.3.3.1. Greedy Randomized Adaptive Search Procedure (GRASP)

La méthode *GRASP* est une métaheuristique mêlant des métaheuristiques constructives et des recherches locales. Elle est divisée en 2 étapes, tout d'abord, on construit une solution et ensuite on l'améliore jusqu'à arriver à satisfaire la condition de terminaison et à ce moment-là, l'algorithme donne la meilleure solution qu'il a trouvée. (Feo et Resende, 1995 ; Pitsoulu et Resende, 2002 ; Dreo et al., 2003)

La phase de construction de solution est composée d'une heuristique constructive dynamique et de randomisation. On considère que la solution *s* correspond à un sous-ensemble d'un ensemble d'éléments (les composants de la solution). La solution est construite en ajoutant un à un les éléments de la solution. Le choix du prochain élément est déterminé suivant une loi de probabilité uniforme parmi les candidats possibles qui sont déterminés selon un critère heuristique donnant un score plus ou moins élevé selon le bénéfice de placer cet élément dans la solution. Cette liste de candidat s'appelle une *restricted candidate list* (*RCL*) et est composée des meilleurs éléments. On parle de méthode dynamique parce qu'à chaque phase de la construction, les valeurs du critère heuristique seront différentes et donc le score sera différent. La longueur α de la liste de candidats déterminera la force du biais heuristique. Si α=1, alors le meilleur élément sera ajouté directement et on se rapprochera de l'heuristique *gloutonne,* Si α=*n*, l'élément à ajouter sera déterminé de manière totalement aléatoire. De ce fait, α est un paramètre critique qui influence l'échantillon de l'espace de recherche (Blum et Roli, 2003). Pitsouli et Resende (2002) ont déterminé les différentes manières de déterminer α. Le plus simple est de le laisser constant tout au long de la phase de construction par exemple.

Lorsque la première phase a permis de construire une solution, la deuxième phase, celle de recherche locale, commence. Cette procédure peut être une des trois métaheuristiques expliquées précédemment (*Iterative Improvement, Simulated Annealing ou Tabu Search*).

Pour que la méthode GRASP soit efficace, il faut remplir deux conditions. La première est que la solution trouvée en première phase soit dans une région prometteuse de l'espace de recherche. La deuxième est que la solution soit dans un bassin d'attraction de différentes solutions de minimum local. Pour satisfaire la première condition, il faut choisir une heuristique constructive efficace et une longueur α pertinente pour la liste des candidats. Pour la deuxième condition, il faut faire en sorte que les deux phases s'accordent correctement. (Blum et Roli, 2003)

Pour l'algorithme GRASP, il n'y aura besoin de la mémoire que pour stocker les données du problème et la solution temporaire. Ce manque d'utilisation de la mémoire explique pourquoi les autres metaheuristiques surclassent souvent cet algorithme. Il donne toutefois rapidement des bonnes solutions en un minimum de temps de computation grâce à sa simplicité.

```
while termination conditions not met do
    s ← ConstructGreedyRandomizedSolution()
    ApplyLocalSearch(s)
    MemorizeBestFoundSolution()
endwhile
```

Figure 7: Algorithme : Greedy Randomized Adaptive Search Procedure (GRASP)

4.3.3.2. Variable Neighborhood Search (VNS)

Cette métaheuristique a été proposée par Hansen et Mladenovic (1999 et 2001). Leur algorithme utilise une stratégie où il y a des changements dynamiques de la structure du voisinage.

Au début de l'algorithme, il faut construire un ensemble de structures de voisinages et générer une solution initiale. Souvent pour choisir ces voisinages, il faut déterminer une séquence $|N_1| < |N_2| < \cdots < |N_{kmax}|$ de voisinages avec une cardinalité croissante. Ensuite, l'algorithme initialise l'indice du voisinage et commence à faire des itérations jusqu'à arriver à une condition de terminaison.

Il y a 3 phases :

- *Shaking* : Durant cette phase, l'algorithme sélectionne aléatoirement une solution *s'* dans le *k*ème voisinage de la solution actuelle *s* ;
- *Local Search* : On utilise *s'* comme point de départ de la recherche locale qui utilisera n'importe quelle structure de voisinage sans restriction de l'ensemble de structures de voisinage N_k avec $k = 1,\ldots,k_{max}$;
- *Move* : A la fin de la recherche locale, on trouve une nouvelle solution *s''* que l'on compare avec *s*. Il y a alors deux cas de figure : soit *s''* est meilleure que *s* et elle remplace alors cette dernière et l'algorithme recommence à k=1, soit *s* est meilleure et donc on recommence la phase *shaking* avec un voisinage différent.

```
Select a set of neighborhood structures Nk , k = 1, . . . , kmax

 s ← GenerateInitialSolution()
while termination conditions not met do

        k←1
        while k < kmax do

        s' ← PickAtRandom(Nk (s))

        s'' ← LocalSearch(s')

                if(f(s'')< f(s)) then

                s ← s''

                k←1

                else

                k←k+1

                endif

        endwhile

endwhile
```

Figure 8: Algorithme : Variable Neighborhood Search (VNS)

L'objectif de la première phase sera de créer une perturbation de la solution afin de pouvoir commencer à partir d'un bon point de départ pour la deuxième phase. Ce *s'* doit se trouver dans

un « bassin » d'attraction d'un minimum local différent de *s* tout en étant peu éloigné de la solution *s*. Le fait de choisir *s'* dans le même voisinage que la solution actuelle donnera probablement une solution avec les mêmes caractéristiques. (Blum et Roli, 2003; Dreo et al., 2003)

Le changement de voisinage permet d'effectuer une diversification et le fait de choisir une cardinalité croissante pour choisir les ensembles de structures de voisinage permet de rendre cette diversification progressive. Une solution qui est localement optimale pour un voisinage donné peut ne pas être un optimum dans un autre voisinage. Cela dépend des propriétés de l'espace de recherche et donc de la structure du voisinage. C'est le concept de *« One Operator, One Landscape »* détaillé par Jones (1995a, 1995b). Selon ce concept, chaque voisinage définit un paysage de recherche avec des propriétés à chaque fois différentes. De ce fait, il est possible que deux paysages donnent un optimum différent. (Blum et Roli, 2003)

4.3.3.3. Guided Local Search (GLS)

Contrairement aux deux précédentes méthodes qui utilisaient des voisinages dynamiques au fur et à mesure des itérations, la *Guided Local Search* modifiera plutôt la fonction objective de manière dynamique pour guider la recherche. Cette méthode nous est donnée par Voudouris et Tsang (1999) et Voudouris (1997).

Le principe de base sera d'influencer la recherche à se déplacer d'un minimum local à un autre en changeant le paysage de recherche. Cet algorithme comme les précédents se composent d'une structure de voisinage, d'un ensemble de solutions et d'une fonction objective. La différence, est que, dans ce cas-ci, c'est la fonction objective *f* qui va être modifiée afin de rendre la solution actuelle moins désirable (en opposition avec la *VNS* qui modifie la structure du voisinage). Cela force l'algorithme à chercher une autre solution. Cette méthode se base sur un mécanisme de caractéristiques de la solution (*solution features*), ce qui signifie que l'algorithme utilisera des caractéristiques ou propriétés permettant de discriminer les solutions. Pour déterminer si une solution *s* possède effectivement une caractéristique, la *GLS* utilise une fonction $I_i(s)$ où *i* est une caractéristique. Si $I_i(s) = 1$, cela signifie que la caractéristique *i* est présente dans la solution, elle sera égale à 0 sinon.
La fonction objective sera modifiée en ajoutant un terme dépendant de *m* caractéristiques :

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

Dans cette équation, p_i est un paramètre de pénalité et λ est un paramètre de régularisation. Le premier permettra de déterminer l'importance des propriétés. Plus il sera grand, plus il sera important et donc plus grand sera le coût d'avoir cette caractéristique dans la solution. Le second permet d'équilibrer la pertinence des caractéristiques par rapport à la fonction objective originale.

L'algorithme commencera donc à partir d'une solution initiale et utilisera une recherche locale jusqu'à atteindre un minimum local. À partir de ce moment-là, le tableau des pénalités **p**= ($p_1, \ldots, p_m$) est mis à jour en incrémentant certaines des pénalités et la recherche locale redémarre et ainsi de suite. Pour déterminer les caractéristiques pénalisées, il faut trouver celles qui ont le maximum d'utilité :

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

Le coût pour la caractéristique *i* est dénoté c_i. Plus le coût est élevé, plus l'utilité sera grande. Le paramètre de pénalité est là pour empêcher qu'une caractéristique avec une utilité trop grande biaise l'algorithme afin de pouvoir utiliser l'historique de recherche efficacement. Il est possible de modifier la phase de mise à jour de la procédure de pénalité en utilisant une règle multiplicative en plus de la règle d'incrémentation appliquée à chaque itération. Cette règle supplémentaire ne sera pas appliquée à la même fréquence, elle sert à lisser le poids des caractéristiques pour éviter que le paysage de recherche soit trop robuste. Les règles de mise à jour sont directement liées avec les données du problème (Blum et Roli, 2003).

```
s ← GenerateInitialSolution()
while termination conditions not met do

        s ← LocalSearch(s, f′)
        for all feature i with maximum utility Util(s, i) do

                pi ← pi + 1

        endfor

Update (f',p)
endwhile
```

Figure 9: Algorithme : Guided Local Search (GLS)

Chapitre 5 : L'algorithme ACO

Maintenant que nous avons pris connaissance des métaheuristiques les plus connues, nous allons aborder la métaheuristique *ACO*. L'algorithme *ACO* (*Ant-Colony Optimization*) est un algorithme constructif, c'est-à-dire que la solution se construit petit à petit de nœud en nœud (Blum et Roli, 2003). Dans les points suivants seront abordés les origines de l'algorithme, son fonctionnement de base, des versions améliorées ainsi que des applications concrètes de l'algorithme à des problèmes combinatoires.

5.1. L'origine de l'algorithme

L'algorithme *ACO* se réfère à un phénomène biologique. En effet, des chercheurs ont constaté que les fourmis et insectes « sociaux » résolvent des problèmes complexes naturellement. Après avoir étudié le comportement des fourmis, la métaheuristique de la colonie de fourmis est apparue. Pour communiquer ensemble, les fourmis utilisent des phéromones qu'elles détectent grâce à leurs antennes. Les fourmis utilisent ces phéromones afin de créer des pistes odorantes pour guider les autres membres de la colonie. Par exemple, ces phéromones vont permettre aux fourmis de prendre le chemin le plus rapide pour atteindre la source de nourriture.

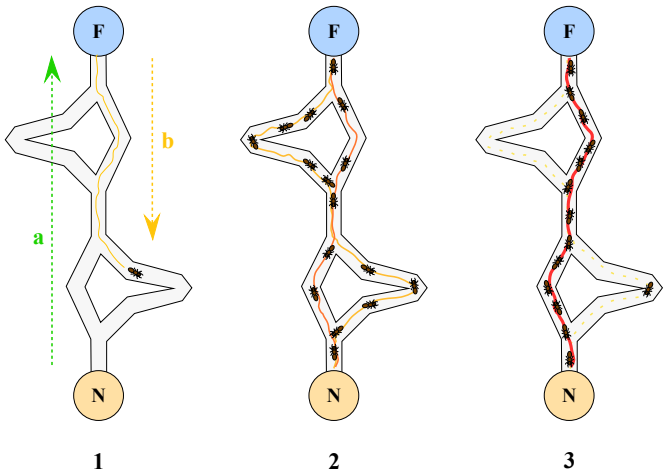

Figure 10 : Sélection du chemin le plus court par une colonie de fourmis[1]

[1] Wikipedia. (2018). Algorithme de colonies de fourmis. Retrieved July 26, 2018 from https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis

Tout d'abord, les fourmis utiliseront aléatoirement les différents chemins possibles tout en déposant des phéromones. Sur le chemin le plus court, les phéromones seront plus fortes vu que la distance est moindre et donc petit à petit, toutes les fourmis effectueront le même chemin. La piste des phéromones sur les mauvais chemins s'évaporera et les chemins plus longs ne seront quasiment plus utilisés (Exemple à la figure 10). Évidemment, si dès le départ, les fourmis choisissent le mauvais chemin, la trace de phéromones risque d'être amplifiée sur le mauvais tracé et donc les fourmis risquent de ne pas passer par le chemin optimal.

Les premiers travaux sur cet algorithme ont été effectués au début des années 90 par Dorigo, Maniezzo et Colorni (Dorigo, 1992 ; Dorigo, Maniezzo et Colorni, 1991 ; Dorigo, Maniezzo et Colorni, 1996).

5.2. Le fonctionnement de l'algorithme

Pour expliquer le fonctionnement de l'algorithme, nous nous sommes basés sur l'article de Blum (2005), l'ouvrage de Dorigo et Stützle (2004) ainsi que le livre de Dreo et al. (2003).Tout d'abord, il faut considérer un exemple avec un modèle de départ. Imaginons donc un nid de fourmis et un foyer où se trouve de la nourriture. Entre ces deux points, il y a deux chemins possibles dont un est plus long que l'autre. Le modèle est donc un graphe $G=(V,E)$ où V sont les deux nœuds et E, les deux chemins. V est composé des noeuds v_s (nid de fourmi) et v_d (endroit où se trouve la nourriture) qui sont reliés par les chemins e_1 et e_2. Vous pouvez représenter le modèle comme dans la figure 11.

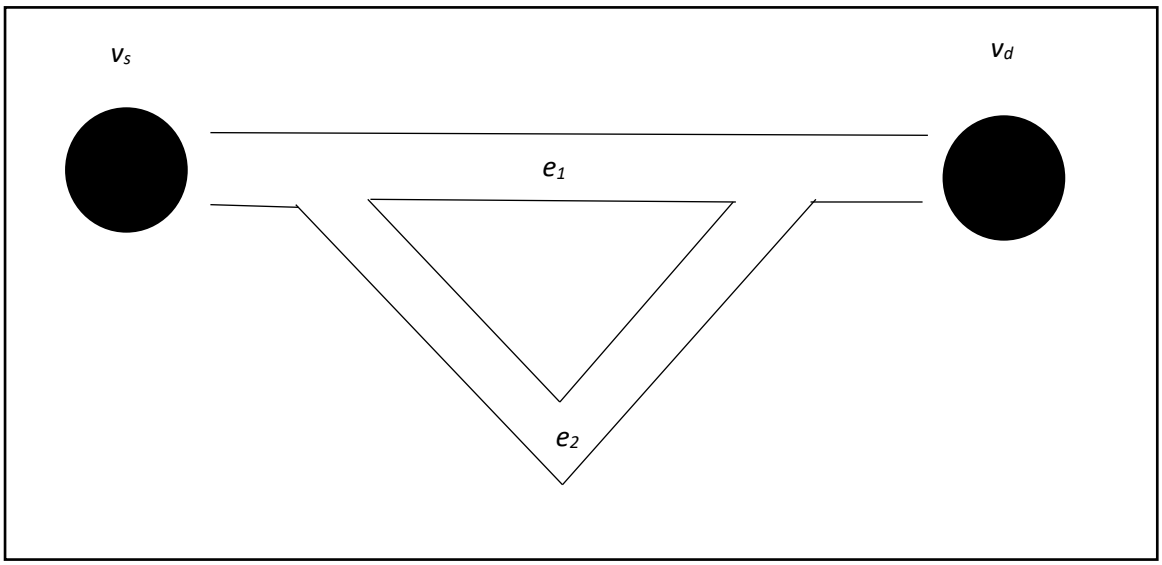

Figure 11: Modèle de base d'ACO

Comme on peut le voir sur le modèle, e_1 a une longueur l_1 plus courte que la longueur l_2 du chemin e_2. Selon l'algorithme des colonies de fourmis, les fourmis déposeront des phéromones sur le chemin qu'elles empruntent. On notera la valeur de la phéromone τ_i en fonction du chemin e_1 ou e_2. Cette valeur de phéromones évoluera en fonction des chemins empruntés par les fourmis.

Cette valeur est importante car elle influencera le choix d'une fourmi de passer par tel ou tel chemin. Ce choix suivra la probabilité suivante :

$$\mathbf{P}_i = \frac{\tau_i}{\tau_1+\tau_2}, \qquad i = 1,2 \tag{5.2.1}$$

Les fourmis déposeront des phéromones durant l'aller et le retour. Les fourmis qui ont emprunté le plus court chemin au début seront aussi celles qui reviendront le plus rapidement au nid. De ce fait, la probabilité d'emprunter ce chemin sera plus haute. Au début, la probabilité d'emprunter les chemins sera la même donc égale à 0.5 , mais la trace de phéromone sur le plus court chemin sera renforcée beaucoup plus fortement que sur le plus long. De plus en plus, les fourmis vont donc choisir le meilleur chemin et renforcer e_1 pendant que sur e_2, la trace de phéromone s'évaporera, réduisant ainsi encore plus la probabilité de choisir ce chemin.

La valeur de la phéromone évoluera donc au fur et à mesure que les fourmis parcourront le graphe selon la formule qui suit :

$$\tau_i \leftarrow \tau_i + \frac{Q}{l_i} \tag{5.2.2}$$

Dans cette formule, Q représente un paramètre positif constant. L'ajout de phéromone dépendra donc de la longueur du chemin. A chaque itération, il y a un nouveau dépôt de phéromones et au début, une évaporation car les phéromones ne sont pas éternelles. Elle se calcule comme suit :

$$\tau_i \leftarrow (1-p) * \tau_i, \qquad i = 1,2 \tag{5.2.3}$$

Le paramètre p permet de réguler cette évaporation et est compris dans l'intervalle [0,1]. Ensuite, après le retour des fourmis dans la colonie, on renforce les traces de phéromones. (Blum, 2005)

Évidemment, ce modèle est trop simpliste et ne peut pas s'appliquer directement à des problèmes d'optimisation combinatoire. Pour expliquer l'algorithme sur un modèle plus complexe, nous prendrons l'exemple du *TSP* (*Travelling Salesman Problem*) résolu selon le premier algorithme *ACO*, *Ant-System (AS)*. (Dorigo, 1992 ; Dorigo et al., 1996)

Tout d'abord, le problème du *TSP* est un problème qui peut être représenté sur un graphe non-dirigé complet $G = (V,E)$. Chaque nœud V est une ville à visiter. Elles sont reliées par des arêtes E ayant chacune un poids déterminé équivalent à la distance entre les deux villes. L'objectif du problème sera d'effectuer le tour le plus court en passant par toutes les villes du graphe une seule fois. Pour appliquer le *Ant-system*, on applique à chaque arête $e_{i,j}$ une valeur de phéromone $\tau_{i,j}$. On ne cherche plus à trouver le plus court chemin entre un nid et un foyer de nourriture mais bien la solution faisable optimisant le problème. Chaque fourmi construit donc une solution possible en commençant par un nœud choisi aléatoirement. Ensuite, la fourmi construira sa route en démarrant à chaque fois du nœud sur lequel elle se trouve et en allant vers un autre qui n'a pas encore été visité. A chaque fois qu'un nœud est traversé, il est rajouté à la solution en construction. La solution en construction est stockée dans la mémoire T. La construction se construira selon la formule de probabilité (5.2.4) qui permet de choisir le nœud suivant n'appartenant pas à T, c'est-à-dire n'étant pas encore visité :

$$\mathbf{P}(e_{i,j}) = \frac{\tau_{i,j}}{\sum_{\{k\in\{1,\dots,|V|\}|v_k\notin T\}} \tau_{i,k}}, \qquad \forall\, j \in \{1,\dots,|V|\}, v_j \notin T \tag{5.2.4}$$

Comme pour l'exemple de départ, après que les fourmis aient fini de construire leur solution, il y aura évaporation et renforcement de la trace de phéromones. Pour l'évaporation, ce sera la formule qui suit avec $\mathcal{T}$ qui est l'ensemble de toutes les valeurs de phéromones :

$$\tau_{i,j} \leftarrow (1-p) * \tau_{i,j}, \qquad \forall \tau_{i,j} \in \mathcal{T} \tag{5.2.5}$$

Après avoir effectué le voyage de retour, la trace de la solution optimale est renforcée comme suit :

$$\tau_{i,j} \leftarrow \tau_{i,j} + \frac{Q}{f(s)} \tag{5.2.6}$$

On définit Q de la même manière que dans la formule (5.2.2), c'est donc une constante positive qui est ensuite divisée par la valeur de la fonction objective de la solution *f(s)*. Comme

précédemment, les fourmis vont voyager *x* fois jusqu'à ce qu'une condition de terminaison soit déterminée (par exemple, un temps maximum ou un nombre d'itérations).

Les résultats donnés par l'algorithme *AS* sont inférieurs à d'autres algorithmes et il a donc connu des évolutions qui seront décrites dans le prochain point.

5.3. The ant-colony optimization metaheuristic

Le fonctionnement de la métaheuristique *ACO* trouve sa version de base dans les recherches de Dorigo, Di Caro et Gambardella (1999). Dorigo en fit avec Stützle une description encore plus précise en 2004. La méthode *ACO* commencera par trouver l'ensemble des composants de la solution du problème combinatoire ainsi que le modèle de phéromone décrit par un ensemble de valeurs de phéromone. Ce modèle est un modèle probabiliste paramétré. En effet, il est utilisé afin de générer des solutions dans l'espace de recherche. Les solutions candidates retenues permettent alors de modifier le modèle de phéromone à chaque itération afin d'influencer l'échantillonnage futur des solutions pour en trouver des meilleures. L'objectif est de concentrer la recherche là où se trouvent les solutions de plus haute qualité. (Blum, 2005, Dorigo et al., 2004)

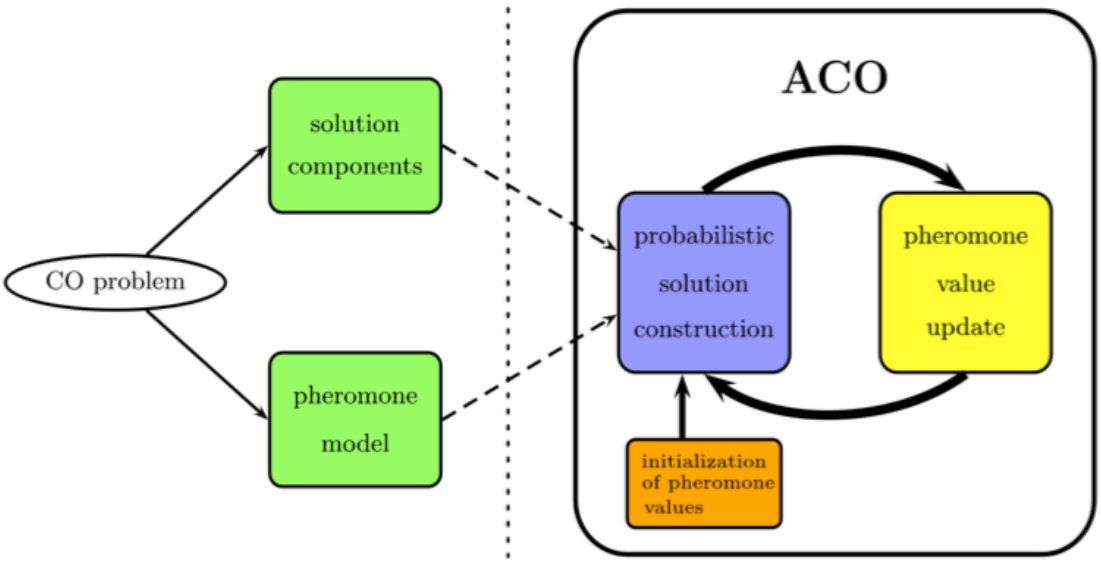

Figure 12: Fonctionnement de l'algorithme (Blum, 2005)

Voici l'algorithme de base :

Algorithme *ACO*

while termination conditions not met **do**

    ScheduleActivities

        AntBasedSolutionConstruction()

        PheromoneUpdate()

        DaemonActions() {optional}

    end ScheduleActivities

end while

Dans l'algorithme, il y aura donc pour commencer une boucle qui indique que l'algorithme recommencera tant qu'il n'atteindra pas une condition de fin (un temps maximum par exemple). Il est composé de trois composants algorithmiques.

Le premier **AntBasedSolutionConstruction()** est une procédure qui construira les solutions. Cette procédure est détaillée comme suit :

Algorithme *Procedure* AntBasedSolutionConstruction() *of ACO*

$s = \langle\rangle$
Determine *N (s)*

    while *N (s)* $\neq \emptyset$ **do**

        $c \leftarrow$ ChooseFrom(*N (s)*)

        $s \leftarrow$ extend s by appending solution component c

        Determine *N (s)*

end while

Tout d'abord, la procédure établit un ensemble vide *s* qui sera la solution donnée par une fourmi. *N(s)* est l'ensemble des composants faisables de la solution. Cet ensemble se videra au fur et à mesure que l'ensemble *s* se remplira. Quand il est vide, la procédure s'arrête. Le composant *c*

est choisi à partir d'une règle suivant la formule de probabilité de choisir le composant *c* appartenant à *N(s)* :

$$\mathbf{p}(c_i|s) = \frac{[\tau_i]^{\alpha}.[\eta(c_i)]^{\beta}}{\sum_{c_j \in N(s)}[\tau_i]^{\alpha}.[\eta(c_i)]^{\beta}}, \qquad \forall c_i \in N(s) \tag{5.3.1}$$

Le premier terme τ_i est comme précédemment la trace de phéromone. La différence avec la formule (5.2.4) est qu'elle prend en compte une valeur heuristique $\eta(c_i)$. De ce fait, la probabilité est moins biaisée vu qu'elle prend en compte la valeur de la solution en plus de la trace de phéromone. Les constantes α et β sont des poids qui permettent de donner une importance plus ou moins grande soit à l'heuristique soit à la trace de phéromones.

Après que chaque fourmi ait sélectionné une solution, il faut mettre à jour la trace de phéromones. Il s'agit de la procédure **PheromoneUpdate()** qui se divise en deux parties. Premièrement, il y aura l'évaporation de la phéromone qui diminue uniformément toutes les valeurs de phéromones. Cela permet de ne pas s'enfermer dans une région sous-optimale en permettant à l'algorithme de visiter de nouveaux endroits de l'espace de recherche. Deuxièmement, il faut renforcer la trace de phéromones. Selon la méthode choisie, l'algorithme renforcera la meilleure solution parmi celles trouvées par les fourmis ou alors toutes ces solutions en augmentant la valeur de la trace de phéromone comme suit :

$$\tau_i \leftarrow (1-p).\tau_i + p. \sum_{\{s \in S_{upd} | c_i \in s\}} w_s.F(s), \quad \forall\, i \in \{1, \dots, n\} \tag{5.3.2}$$

S_{upd} est l'ensemble des solutions qui vont être utilisées pour la mise à jour. Le paramètre *p* est toujours le taux d'évaporation de la phéromone et *F(s)* est une fonction de qualité, c'est-à-dire que lorsque la valeur de la fonction objective *f(s)* est inférieure à celle de *f(s')*, alors cette fonction *F* sera meilleur pour la solution *s* que pour *s'*. A cette fonction, il est possible d'ajouter un poids w_s pour la solution *s*.

S_{upd} est composé de quelques solutions obtenues dans l'itération et que l'on peut noter S_{iter}. Nous aurons aussi une solution temporaire qui est la meilleure jusque-là, c'est la « *Best-so-far solution* », elle se note s_{bs}. En fonction de l'algorithme utilisé, il y aura différentes règles de mise à jour et différents poids de phéromone. Par exemple, pour la méthode *AS*, il faudra modifier la règle de mise à jour (5.3.2) comme ceci :

$$S_{upd} \leftarrow S_{iter} \text{ and } w_s = 1 \ \forall s \in S_{upd} \tag{5.3.3}$$

A ce moment-là, l'algorithme sélectionnera toutes les solutions générées dans l'itération pour la mise à jour des phéromones et le poids de chacune sera de 1. Il existe aussi la règle de mise à jour en fonction de la meilleure solution de l'itération (*IB-update rule*) qui est donnée comme suit :

$$S_{upd} \leftarrow \{s_{ib} = \text{argmax}\{F(s)|s \in S_{iter}\}\} \quad \text{with } w_{s_{ib}} = 1 \qquad (5.3.4)$$

Dans ce cas, il n'y aura que la trace de phéromones de la meilleure solution de l'itération qui sera renforcée. Une autre possibilité est d'utiliser la règle de mise à jour à partir de la meilleure solution actuelle seulement (*BS-update* rule). Ces deux dernières règles augmentent le risque de converger vers une seule solution prématurément. Cependant, ajouter des mécanismes pour éviter ces convergences permet d'obtenir de meilleurs résultats qu'avec la méthode de mise à jour utilisée dans *AS*.

La dernière procédure est la procédure **DaemonActions ()** et est utilisée pour implémenter des actions qui ne peuvent être exécutées par une seule fourmi. Par exemple, cette action peut être une recherche locale sur la solution trouvée précédemment. Toute l'explication de l'algorithme peut se retrouver dans divers ouvrages et articles de Dorigo et Stützle (2004), Dréo et collègues (2003), Blum (2005) et dans d'autres travaux précédents de Dorigo et collègues.

5.4. Variantes de l'algorithme ACO

La méthode originale *AS* n'ayant pas apporté entière satisfaction dans la résolution de problèmes combinatoires, des variantes d'algorithme *ACO* ont été développées. Nous nous concentrerons principalement sur la variante *Ant Colony System (ACS)* car c'est celle qui sera utilisée pour résoudre le *SSP*. Cependant nous pouvons tout de même citer d'autres variantes telles que l'*Elitist Ant System* expliquée par Dorigo en 1992 et Dorigo, Maniezzo et Colorni en 1996. Stüzle et Hoos en 2000 ont proposé la *Max-Min Ant System (MMAS)* qui est la méthode la plus efficace avec l'*ACS*. Il existe aussi la variante *Rank-based Ant-System, Approximate Non-deterministic Tree Search* et *Hyper-cube Framework* dont vous pouvez trouver le fonctionnement dans l'article de Blum (2005) ainsi que dans l'ouvrage de Dorigo et Stützle (2004).

Nous allons donc maintenant expliquer l'*Ant Colony System* proposé par Dorigo et Gambardella (1997a,1997b) détaillé dans l'ouvrage de Dorigo et Stützle (2004).

L'*ACS* diffère de l'*Ant System* par trois points importants :

1) La construction du tour : Il utilise la règle *pseudorandom proportional*. Cette règle permet de choisir avec une probabilité q_0 le meilleur nœud à ajouter à la solution au lieu d'utiliser la simple règle de probabilité (5.3.1) . La fourmi k se trouvant au nœud i ira au nœud j en suivant la règle suivante :

$$j = \begin{cases} argmax_{l \in N_i^k}\{\tau_{il}.\eta_{il}^{\beta}\} & si\ q \leq q_0 \\ J(\beta) & sinon \end{cases} \qquad (5.4.1)$$

La variable q est déterminée aléatoirement et est uniformément distribuée dans l'intervalle [0,1]. *J(β)* fait référence à la règle *random proportionnal* qui utilise la règle de probabilité (5.3.1). Cela veut dire que plus q_0 est élevé, plus les chances seront grandes que l'algorithme donne une solution proche de la meilleure proposée jusqu'à ce moment-là sinon, nous privilégierions d'explorer d'autres tours.

2) La mise à jour globale de la trace des phéromones : Pour mettre à jour la trace de phéromones, nous utiliserons l'équation suivante :

$$\tau_{ij} \leftarrow (1-p).\tau_{ij} + p.\Delta\tau_{ij}^{bs}, \qquad \forall\, i,j \in T^{bs} \qquad (5.4.2)$$

Où $\Delta\tau_{ij}^{bs} = \frac{1}{C^{bs}}$. C^{bs} est le coût du meilleur tour jusque maintenant. La particularité de l'*ACS* est le fait que l'évaporation ainsi que le dépôt des phéromones ne se fait que sur T^{bs}, c'est-à-dire les arcs composant le meilleur tour.

3) La mise à jour locale de la trace des phéromones :

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} + \xi\tau_0 \qquad (5.4.3)$$

ξ et τ_0 sont deux paramètres. Le premier aura une valeur comprise entre 0 et 1, une valeur de 0.1 a été trouvée comme étant une bonne valeur de ξ. Le paramètre τ_0 peut être la valeur de la phéromone au départ de l'algorithme. Cependant, une bonne valeur de τ_0 peut être trouvée en appliquant la formule suivante : $\tau_0 = 1/nC^{nn}$ avec n étant le nombre de job de l'instance et C^{nn} le résultat d'un tour déterminé par une heuristique de *nearest-neighbor*. Dans le cadre de ce travail, nous avons opté pour une *Shortest-edge heuristic.* A chaque fois qu'une fourmi emprunte l'arc *(i,j)*, cet arc sera mis à jour selon cette formule. Elle permet de rendre un arc visité moins désirable afin d'influencer les autres fourmis à emprunter un autre arc pour éviter un comportement de stagnation. La mise à

jour locale peut être utilisée après le passage de chaque fourmi ou alors peut être utilisée après une itération complète.

Maintenant que nous avons abordé le problème, les différentes manières de le résoudre avec une méthode exacte, une heuristique ou une métaheuristique, le prochain chapitre abordera l'implémentation de l'algorithme *ACS* pour le problème du *Job Sequencing and Tool Switching Problem* et les résultats obtenus.

Chapitre 6 : Application de l'algorithme ACO au problème de JSTSP

Le dernier chapitre sera donc consacré à la mise en place de l'algorithme. Tout d'abord, il s'agira d'expliquer comment il a été mis en place, comment est construit l'algorithme. Nous décrirons également les différents paramètres choisis, les différentes méthodes utilisées ainsi que les explications de ces choix. Ensuite, nous aborderons les différents résultats et les améliorations que nous pouvons proposer pour obtenir des meilleurs résultats.

6.1. Mise en place d'algorithme

6.1.1. L'algorithme *Ant-Colony System*

Le problème de *Job Sequencing and Tool Switching Problem* est une variante plus complexe du problème du voyageur de commerce (*TSP*). C'est pourquoi il s'agira d'adapter un algorithme *ACO* utilisé pour le *TSP* à notre problème. Le *JSTSP* diffère car il faut employer la règle de *KTNS* (*Keep Tools Needed Soonest*) afin de remplir la contrainte de capacité et de remplir la capacité maximale du dépôt de la machine.

Dans le cadre d'un algorithme utilisant une méthode de trajectoire telle que l'*Iterated Local Search* par exemple, on démarre d'une solution aléatoire à laquelle on applique la KTNS afin de trouver le coût de la solution. Ensuite, on démarre l'algorithme et on pratique une recherche en modifiant la solution. Dans le cadre d'un algorithme *ACO*, cela sera différent vu que la solution se construit au fur et à mesure de l'exécution de l'algorithme. En effet, il s'agira de calculer à chaque étape de la construction d'une solution, la *KTNS*.

La procédure principale se construira comme suit :

Procedure main *ACO algorithm for JSTSPs*

Set parameters, initialize pheromone trails

while (termination condition not met) **do**

ConstructSolutions

ApplyLocalSearch

UpdateTrails

end
end *ACO algorithm for JSTSPs*

Dans la suite de ce point seront expliquées les différentes procédures et formules mathématiques utilisées afin de trouver la meilleure solution.
Tout d'abord, il faut construire les solutions selon la procédure **ConstructSolutions** :

Procedure ConstructSolutions

s = ⟨⟩
Determine *N (s)*

while *N (s)* ≠ ∅ **do**

c ← ChooseFrom(*N (s)*)
s ← extend *s* by appending solution component *c* Determine *N (s)*

end while

Cette procédure est la même que celle décrite dans le chapitre précédent à laquelle il faudra ajouter les changements de la variante *ACS*. Tout d'abord, nous utiliserons la formule du chapitre précédent (5.4.1) pour déterminer le job *j* à ajouter à la solution. Pour le problème du *JSTSP*, il y aura une différence dans la formule de probabilité (5.3.1) et dans la formule (5.4.1) dans la fonction **ChooseFrom (*N(s)*)**. Les formules différeront de celles utilisées pour l'application d'*ACO* au problème du *TSP* pour la partie heuristique. En effet, dans le problème du voyageur de commerce, on ne fait la différence qu'entre le dernier élément de la solution et le composant *j*. Dans ce problème, vu l'application de la *KTNS*, la solution sera toujours différente en fonction de ce composant *c*. En effet, il faudra exécuter la *KTNS* à chaque étape

de la construction de la solution ainsi que pour chaque ville restant à visiter. Pour simplifier, dans le cas du *TSP*, on aura une matrice de distance 2 à 2 alors que dans le problème du JSTSP, nous aurons une matrice de distance dynamique. C'est pour cela qu'on devra modifier la formule heuristique de base.

Formule heuristique pour le *TSP* :

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{6.1.1}$$

Où d_{ij} est la distance entre i et j.

Formule heuristique pour le *JSTSP* :

$$\eta(c_j) = \frac{1}{(c_j - c)} \tag{6.1.2}$$

Où c est le coût avant l'ajout du composant j à la solution et c_j est le coût comprenant le composant j. Pour calculer ces coûts, on utilisera la procédure *KTNS* expliquée au chapitre 1. Comme cela a été expliqué au chapitre précédent, en fonction de la valeur de q, une valeur aléatoire, l'algorithme choisira le meilleur sommet en fonction de la formule (5.4.1) et utilisera la formule de probabilité (5.3.1) si le meilleur sommet n'est pas choisi. À cause de la procédure *KTNS*, il est possible que la formule heuristique donne une division par 0 au cas où le rajout du job j à la solution n'implique pas de changements d'outils. Si cela arrive, il y a alors deux possibilités. La première sera de donner une valeur de proche de 0 au dénominateur de la formule (6.1.2), ce qui aura pour effet de favoriser la sélection de ce job en augmentant énormément la probabilité de le choisir. La deuxième option sera de remplacer le dénominateur par une valeur telle que 0.5 , cela permettra de rendre le job j deux fois plus désirable pour les fourmis. L'avantage de cette option est que l'algorithme aura plus de chance de le sélectionner mais ce ne sera pas aussi drastique que la première option.

L'algorithme *ACS* choisit un job dans un ensemble de jobs non visités. Afin de limiter le choix à des jobs de bonne qualité, nous avons décidé que l'algorithme sélectionnerait le prochain job parmi les 10 meilleurs de l'ensemble non visités. Cela permet d'avoir des meilleurs résultats directement.

Lorsque toutes les fourmis auront parcouru le graphe, nous appliquerons la procédure suivante, **ApplyLocalSearch** sur toutes les solutions trouvées. La méthode de recherche locale utilisée est la 2-opt abordée au chapitre 3 en ne gardant que la meilleure amélioration. Les temps de computation en testant toutes les possibilités pour ne garder que la meilleure solution sont bien

évidemment supérieurs que de s'arrêter à la première amélioration. Cependant, ces temps restent raisonnables, c'est pourquoi nous avons fait ce choix. La méthode 3-opt a été également testée mais augmentait beaucoup trop les temps de computations en choisissant la meilleure amélioration. Dans le cas où l'on prenait la première amélioration, les résultats étaient à peu près les mêmes que la 2-opt.

Lorsque l'étape de recherche locale est terminée, il s'agira de mettre à jour la trace de phéromones durant la procédure de **UpdateTrails** selon les formules (5.4.2) pour la meilleure solution à ce moment-là de l'exécution de l'algorithme et (5.4.3) pour toutes les solutions des fourmis après recherche locale. Cette trace de phéromone sera réinitialisée toutes les 20 itérations afin de ne pas s'enfermer dans les mêmes chemins. Comme nous l'avons dit au début du chapitre précédent, si les fourmis suivent dès le départ un chemin de mauvaise qualité, il y a un grand risque de ne jamais trouver la solution optimale. Les conditions de terminaison de l'algorithme seront déterminées en fonction de la taille du problème. Soit ce sera un nombre d'itérations, soit ce sera un nombre de secondes.

6.1.2. Les paramètres utilisés

Vu que le problème du *Job Sequencing and Tool Switching Problem* est une prolongation du problème du Voyageur de Commerce (*TSP)*, nous utiliserons les paramètres proposés pour l'*ACS* par Dorigo et Stützle (2004).

Dans ce cas, le β , l'importance heuristique, sera de 2. Le taux d'évaporation pour la mise à jour globale de la phéromone p sera égal à 0,1 tout comme le taux de mise à jour locale, ξ. Nous utiliserons dix fourmis par itération et la valeur de la phéromone de départ τ_0 sera égale à $1/nC^{nn}$ avec le coût C^{nn} qui sera calculé selon la méthode expliquée au chapitre 3, *Shortest-edge heuristic*. Enfin, la valeur de q_0 sera de 0.98. Cela signifie donc qu'il y aura 9.8 chances sur 10 de sélectionner le meilleur sommet pour la construction de la solution.

6.2. Résultats computationnels

L'algorithme a été implémenté sur le logiciel d'optimisation FICO Xpress Optimization en langage mosel et il était exécuté sur un ordinateur avec un processeur i5-4460.

Afin de tester l'algorithme, nous avons utilisé les différentes instances proposées par Catanzaro et al. (2015). Il y a 4 groupes de 4 instances. Ces 4 instances ont les mêmes problèmes si ce n'est que la capacité du dépôt diffère. Donc le problème 1 de l'instance *datA1* sera le même que le problème de l'instance *datA2*. Vous pouvez trouver les informations par instance dans la table 3 où *J* est le nombre de jobs, *T*, le nombre d'outils et *Cap*, la capacité maximale du dépôt. A chaque problème, nous possédons le nombre de changements minimum qui a déjà pu être trouvé. Pour les instances A, il s'agit du minimum global alors que pour les autres instances, il existe des solutions meilleures que celles proposées dans les instances.

Chaque problème de chaque instance a été testé 10 fois. En fonction de l'instance A, B, C ou D, nous avons fait des choix différents au niveau du moyen de comparaison ainsi que des conditions de terminaison de l'algorithme. Pour les instances A et B, nous avons choisi de faire 100 itérations et de comparer les résultats avec ceux de l'*Iterated Local Search* de Paiva et al. (2017) qui se sont basés sur les mêmes instances. Pour les instances C, nous avons opté pour une limitation de temps, l'algorithme tournait pendant 5 minutes (300 secondes) et a été comparé avec une méthode aléatoire qui elle aussi a tourné pendant 5 minutes. Les dernières instances D ont été testées pendant 10 minutes à chaque fois et comparées elles aussi avec une méthode aléatoire.

Table 3: Explications des différents problèmes testés

Récapitulatif des instances	*J*	*T*	*Cap*
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

Instances A et instances B

Pour les instances A, l'algorithme atteint toujours le minimum proposé dans les instances de Catanzaro et al. (2015). Pour les instances B, à part pour 2 problèmes de l'instance *datB1*, nous obtenons toujours un résultat optimal voir un meilleur résultat. Ces résultats se trouvent dans la table 4. Dans ce tableau, le *C* est la moyenne des nombres de changements par problème de l'instance correspondante, *C** est la moyenne des optima trouvés à chaque problème de l'instance correspondante. Le *Δ* est la moyenne des différences de tous les résultats trouvés par rapport à sa meilleure valeur connue pour un problème de l'instance correspondante, le *Δ** représente la moyenne des différences du meilleur résultat trouvé par rapport à la meilleure valeur connue du problème pour chaque problème de l'instance correspondante. *T* est le temps mis par l'algorithme pour exécuter 100 itérations dans le cas de l'*ACS*. Enfin, *I-best* est la valeur moyenne de l'itération à laquelle la meilleure solution de l'algorithme est trouvée.

Table 4: Comparaison des résultats de l'ACS avec ceux de l'ILS de Paiva et al. (2017) pour les instances A et B

	ACS						ILS		
	C	*C**	*T*	*Δ*	*Δ**	*I-best*	*C*	*C**	*T*
datA1	**12,5**	**12,5**	4,32	0	0	1	**12,5**	**12,5**	0,09
datA2	**10,8**	**10,8**	5,30	0	0	1	**10,8**	**10,8**	0,09
datA3	**10,1**	**10,1**	6,01	0	0	1	**10,1**	**10,1**	0,05
datA4	**10**	**10**	6,63	0	0	1	**10**	**10**	0,04
datB1	26,99	26,7	28,94	0,09	-0,2	10,26	26,5	26,5	1,09
datB2	21,77	**21,7**	38,35	-0,23	-0,3	12,78	21,7	**21,7**	1,38
datB3	19,79	**19,7**	44,98	-0,01	-0,1	2,46	19,7	**19,7**	1,12
datB4	**19,2**	**19,2**	50,51	0	0	1,02	**19,2**	**19,2**	0,7

Nous pouvons donc voir que les résultats sont comparables avec la méthode *ILS* de Paiva et al. (2017) au niveau des résultats pour ce qui est de la meilleure solution finale si ce n'est pour l'instance *datB1*. Nous pouvons voir au niveau du temps de computation que l'algorithme *ACS* mis en place dans le cadre de ce mémoire est fortement supérieur. Les deltas montrent pour *datB1,datB2* et *datB3* que l'algorithme trouve un nombre de changements inférieur au nombre de changements donné dans les instances. Nous pouvons également constater que le nombre d'itérations pour trouver la meilleure solution diminue entre *datB1* et *datB4* parce que le fait d'augmenter la capacité maximale du dépôt offre plus de possibilités et plus d'ordonnancements optimaux. L'algorithme a donc plus de chance de trouver un optimum plus rapidement.

L'algorithme *ACS* obtient des moins bons résultats globaux pour *C*, cela veut dire que l'algorithme n'atteignait pas forcément la meilleure solution. L'utilisation d'une recherche locale *2-opt* influence ces résultats. En effet, l'espace de recherche n'est peut-être pas suffisamment large.

Les instances A et B sont des petits problèmes et sont résolus assez facilement. Il sera bien plus intéressant d'analyser les résultats pour les instances C et D qui sont des problèmes plus complexes.

Instances C et instances D

Pour les instances C et D, notre algorithme prend trop de temps de computation pour trouver une solution optimale. Nous avons donc décidé de limiter le nombre de minutes pendant lesquelles l'algorithme était exécuté. De ce fait, pour les problèmes de l'instance C, l'algorithme s'arrêtait après 300 secondes et pour les instances D, après 600 secondes. Les résultats se trouvent dans la table 5.

Table 5: Comparaison des résultats de l'ACS avec ceux d'une méthode aléatoire pour les instances C et D

	ACS							Méthode Aléatoire				
	C	*C**	*T*	*Δ*	*Δ**	*I*	*I-best*	*C*	*C**	*T*	*Δ*	*Δ**
datC1	**108,4**	**106,3**	302,26	6,35	4,3	54,85	21,99	114,77	112,7	300,01	12,77	10,7
datC2	**91,11**	**89,4**	303,58	5,21	3,5	46,5	17,62	97,08	94,9	300,01	11,18	9
datC3	**72,95**	**71,7**	304,39	3,55	2,3	38,33	18,29	78,07	76,4	300,01	8,67	7
datC4	**54,95**	**53,9**	305,52	1,35	0,3	30,33	13,32	57,41	56,3	300,01	3,81	2,7
datD1	**220**	**215,8**	610,02	16,83	12,6	28,77	15,13	255,42	251,7	600,01	52,22	48,5
datD2	**193,9**	**190,7**	612,54	14,86	11,7	24,99	12,33	226,41	223,3	600,01	47,41	44,3
datD3	**164,6**	**161,4**	614,69	12,1	8,9	20,91	10,41	192,76	189,7	600,01	40,26	37,2
datD4	**129,9**	**127,2**	617,03	8,97	6,3	16,7	6,95	151,29	148,2	600,01	30,39	27,3

La colonne *I* est le nombre d'itérations moyen pour chaque instance. Nous pouvons voir que la taille du problème et la capacité du dépôt ont une influence sur le nombre d'itérations *I*. Plus la capacité du dépôt est grande, moins il faudra d'itération avant d'arriver à la meilleure solution. Il faut savoir que dans ces résultats, nous n'arrivons à une solution optimale voire meilleure que celle proposée dans les instances seulement pour les problèmes *datC4*. Pour améliorer ces résultats, il faudrait élargir la recherche locale et augmenter le temps de computation ou plutôt le nombre d'itérations. En effet, pour *datD4* par exemple, avec seulement 10 fourmis et une

moyenne de 16 itérations, il est certainement possible que l'algorithme n'ait pas exploré suffisamment l'espace de recherche que pour pouvoir atteindre l'optimum. Cependant, nous pouvons quand même voir une forte différence entre la méthode aléatoire et l'*ACS* au niveau des résultats. Nous pouvons voir que plus le problème est complexe plus l'écart avec l'optimum connu est grand dans les deux cas. Cependant, l'*ACS* trouve toujours une meilleure solution que la méthode aléatoire en très peu de temps.

Au vu du nombre d'itérations en 10 minutes, il sera nécessaire de tester pour un nombre d'itérations plus grand afin de voir si les résultats s'améliorent ou non. Cela sera testé sur plusieurs problèmes dans le point suivant. Le nombre d'itérations pour les différents problèmes diminue en fonction de la taille du problème. En effet, plus il y a de jobs et d'outils, plus la recherche locale prendra du temps. Cependant, nous pouvons aussi remarquer que la capacité du dépôt de la machine a un impact. La procédure *KTNS* prendra plus de temps à être exécutée pour les problèmes avec une plus grande capacité et donc le nombre d'itérations diminue. La procédure devra, en effet, chercher plus longtemps les outils à charger vu que la capacité est plus grande que la plus haute capacité d'outils que les jobs peuvent avoir. Par exemple, pour *datD4*, nous avons une capacité de dépôt de 30 mais un maximum de 20 outils utilisés par job. Donc la procédure s'applique à chaque job, ce qui n'est pas le cas pour *datD1* où on a une capacité du dépôt de 20 tout comme le nombre d'outils utilisés par job.

Table 6: Comparaison des écart-types entre l'ACS et la méthode aléatoire pour les instances C et D

	ACS			Méthode Aléatoire		
	C	*C**	σ	*C*	*C**	σ
datC1	108,4	106,3	1,2711	114,8	112,7	1,15036
datC2	91,11	89,4	1,0471	97,08	94,9	1,15758
datC3	72,95	71,7	0,9014	78,07	76,4	1,002
datC4	54,95	53,9	0,6713	57,41	56,3	0,75299
datD1	220	215,8	2,2255	255,4	251,7	2,02709
datD2	193,9	190,7	1,9652	226,4	223,3	1,74712
datD3	164,6	161,4	1,8129	192,8	189,7	1,62185
datD4	129,9	127,2	1,5843	151,3	148,2	1,54491

Dans la table 6, nous pouvons voir les écarts-types par rapport à la moyenne des résultats trouvés pour l'*ACS* ainsi que pour la méthode aléatoire. Nous pouvons remarquer que l'écart-type dans les deux cas est faible. Cela ne nous permet donc pas de faire des conclusions sur

base de cette donnée. Cependant nous voyons aisément que l'écart-type des solutions données par l'*ACS* est faible et donc que l'algorithme nous donne des solutions assez équivalentes.

6.3. Pistes d'amélioration

Après avoir exécuté l'algorithme avec les paramètres de base, quelques pistes à tester sont apparues afin de déterminer s'il y avait moyen d'améliorer les résultats que l'algorithme rendait actuellement. Tous les résultats suivants sont à prendre avec recul parce que les instances n'ont pas été testées dix fois à chaque fois comme pour le point précédent mais deux fois. Cependant, les résultats semblent logiques et nous aurions des chiffres sans doute équivalents en l'exécutant un plus grand nombre de fois.

Mise en place d'une recherche locale *3-opt*

La première piste a été de renforcer l'espace de recherche lorsque la fourmi a fini de construire une solution. Donc, nous avons remplacé la *2-opt* par une recherche *3-opt*. La complexité de la recherche locale est donc devenue $O(N^4M)$. Sans aucun doute, implémenter cette recherche locale donnera sûrement des meilleurs résultats dans le cas où nous sélectionnerions la meilleure amélioration et non la première. Cependant, au vu de la complexité, le temps de computation devient très vite trop grand et donc, le choix de s'arrêter à la *2-opt*, est venu naturellement.

Modifier la valeur de q_0

Nous avons vu précédemment que lors de la construction de la solution, l'algorithme sélectionne le meilleur job en fonction d'une probabilité q_0. Nous nous sommes demandé alors ce qu'il se passait si on donnait à l'algorithme plus de « liberté » en diminuant la valeur de q_0. De ce fait, nous avons testé l'algorithme en utilisant 0.5 comme valeur de q_0. Nous avons testé l'algorithme deux fois par problème de chaque instance. Les résultats se trouvent dans la table 7 à la page suivante.

Table 7: Comparaison des résultats de l'ACS avec modification de q_0

	q_0=0,98					q_0=0,5				
	C	*C**	*T*	*Δ*	*Δ**	*C*	*C**	*T*	*Δ*	*Δ**
datC1	**108,35**	**106,3**	302,26	6,35	4,3	118,05	117	303,51	16,1	15
datC2	**91,11**	**89,4**	303,58	5,21	3,5	99,75	98,4	305,08	13,9	12,5
datC3	**72,95**	**71,7**	304,39	3,55	2,3	80,4	79,4	303,54	11	10
datC4	**54,95**	**53,9**	305,52	1,35	0,3	59,5	58,6	304,21	5,9	5
datD1	**220,03**	**215,8**	610,02	16,83	12,6	247,9	244	609,51	44,7	40,8
datD2	**193,86**	**190,7**	612,54	14,86	11,7	223,05	221,2	611,54	44,1	42,2
datD3	**164,6**	**161,4**	614,69	12,1	8,9	188,85	186,4	616,95	36,4	33,9
datD4	**129,87**	**127,2**	617,03	8,97	6,3	147,05	145	616,72	26,2	24,1

Nous pouvons remarquer aisément que les résultats sont meilleurs lorsque q_0 vaut 0.98. En effet, laisser trop de liberté à l'algorithme le fait emprunter des chemins qui peuvent être très loin de l'optimum. La méthode aléatoire est parfois même plus efficace dans ce cas-là. Vu que le problème n'a été testé que 2 fois, les moyennes ne sont pas aussi précises que pour l'exécution de l'algorithme proposé en table 5 mais nous aurions sûrement obtenu des résultats équivalents en l'exécutant le même nombre de fois.

Appliquer une recherche locale à chaque étape de la construction

Afin de trouver une meilleure solution en moins d'itération, il est peut-être possible d'avoir de meilleurs résultats finaux et donc de devoir faire moins d'itérations en appliquant une recherche locale *2-opt* à chaque étape de la construction de la solution par la fourmi. Les temps de computation seront plus longs bien qu'inférieurs à la mise en place d'une recherche large *3-opt*. Cependant, si les résultats sont plus précis, nous aurions tout intérêt à l'appliquer.
Les résultats se trouvent dans la table 8 Nous avons testé cet algorithme deux fois par problème de chaque instance C et D avec la même condition de terminaison que précédemment.

Table 8: Comparaison des résultats de la méthode ACS avec ceux de l'ACS avec recherche locale à chaque étape de la construction pour les instances C et D

	ACS						Recherche locale à chaque étape					
	C	*C**	*T*	*I*	*Δ*	*Δ**	*C*	*C**	*T*	*I*	*Δ*	*Δ**
datC1	108,35	106,3	302,26	54,85	6,35	4,3	**103,5**	**102,9**	316,74	9,2	1,5	0,9
datC2	91,11	89,4	303,58	46,5	5,21	3,5	**87,15**	**86,9**	319,30	7,9	1,25	1
datC3	72,95	71,7	304,39	38,33	3,55	2,3	**70,75**	**70,1**	321,14	6,6	1,35	0,7
datC4	54,95	53,9	305,52	30,33	1,35	0,3	**53,75**	**53,4**	330,89	5,45	0,15	-0,2
datD1	220,03	215,8	610,02	28,77	16,83	12,6	**207,55**	**206,6**	693,31	3,9	4,35	3,4
datD2	193,86	190,7	612,54	24,99	14,86	11,7	**183,45**	**182,2**	687,71	3,4	4,45	3,2
datD3	164,6	161,4	614,69	20,91	12,1	8,9	**155,95**	**155,1**	729,30	3	3,45	2,6
datD4	129,87	127,2	617,03	16,7	8,97	6,3	**124,45**	**123,8**	735,76	2,45	3,55	2,9

Nous remarquons que la qualité des résultats est fortement améliorée pour toutes les instances. Nous pouvons remarquer par ailleurs que le nombre d'itérations diminue fortement. Ces résultats ne sont pas surprenants. La recherche locale prend du temps et vu qu'elle est appliquée à chaque étape de la construction, nous obtenons une bonne solution partielle à chaque fois. C'est pour cela qu'à la fin, la solution est plus proche de l'optimum.
Pour améliorer encore plus cette modification, il faudrait augmenter le nombre d'itérations.

Augmentation du nombre d'itérations pour les instances D

Vu la taille des problèmes des instances D, il semble logique qu'il soit nécessaire d'avoir plus d'itérations que lors des résultats obtenus après 10 minutes. Donc, nous avons testé l'algorithme pour ces instances pour 50 itérations à chaque fois. Comme pour les autres pistes, nous avons testé cela seulement deux fois et non plus dix et cette fois-ci, nous nous sommes concentrés seulement sur les problèmes des instances D. Les résultats sont dans la table 9.

Comme nous pouvons le constater, les résultats généraux (*C*) sont meilleurs et nous obtenons une meilleure solution (*C**). Nous pouvons voir que la distance avec la meilleure valeur connue est moindre également. En augmentant encore le nombre d'itérations, nous obtiendrions peut-être de meilleurs résultats.

Table 9: Comparaison des résultats de l'ACS lorsque le nombre d'itérations est différent pour les instances C et D

	600 secondes					50 itérations				
	C	*C**	*T*	*Δ*	*Δ**	*C*	*C**	*T*	*Δ*	*Δ**
datD1	220,03	215,8	**610,02**	16,8	12,6	216	**215,3**	1103,84	**12,8**	**12,1**
datD2	193,86	190,7	**612,54**	14,9	11,7	189,6	**189**	1250,42	**10,6**	**10**
datD3	164,6	161,4	**614,69**	12,1	8,9	160,5	**159,9**	1492,16	**8**	**7,4**
datD4	129,87	127,2	**617,03**	8,97	6,3	126,25	**125,8**	1882,8	**5,35**	**4,9**

Conclusion

Dans ce mémoire, nous avons donc commencé par expliquer le problème d'optimisation combinatoire du *SSP*. Nous avons abordé les différents travaux effectués pour le résoudre à l'aide d'une méthode exacte ainsi que certains moyens heuristiques pour atteindre une valeur approchée. Ensuite, vu que le sujet de ce mémoire se base sur une métaheuristique, nous avons fait un résumé des métaheuristiques les plus connues. Nous avons également cité les quelques algorithmes métaheuristiques mis en place pour résoudre notre problème.

Nous avons ensuite pu mettre en place un algorithme *ACO* grâce à une explication de l'algorithme au moyen de l'exemple du *TSP*.

Les résultats obtenus permettent de battre une méthode aléatoire sur un court laps de temps. De plus, nous avons utilisé des paramètres du *TSP* en envisageant les deux problèmes sous le même angle. Peut-être serait-il intéressant de tester l'algorithme avec des valeurs de paramètres différents. Cependant cela signifie de tester l'algorithme un très grand nombre de fois.
Nous avons pu remarquer qu'une recherche locale plus puissante et un nombre d'itérations plus grand permettraient d'obtenir de meilleurs résultats. Pour mettre cela en place, il aurait fallu privilégier un langage de programmation plus performant que Mosel. En effet, ici, pour exécuter 50 itérations avec une simple recherche locale *2-opt*, il faut compter entre environ 20 et 30 minutes pour terminer l'exécution sur les problèmes de plus grande taille.

Bibliographie

Articles ou ouvrages :

- 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*. doi: https://doi.org/10.1016/j.asoc.2017.12.045
- Al-Fawzan, M., & Al-Sultan, K. (2002). A tabu search based algorithm for minimizing the number of tool switches on a flexible machine. *Computers & Industrial Engineering*, 44, 35-47.
- Amaya, J. E., Cotta, C., & Fernández-Leiva, A. J. (2008). *A memetic algorithm for the tool switching problem.*
- Belady L.A. (1966). A study of replacement algorithms for virtual storage computers, IBM Syst J l5, 70–101.
- Blazinskas, A., & Misevicius, A., (2011). Combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the travelling salesman problem. Kaunas University of Technology , Department of Multimedia Engineering.
- Blum, C., & Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, *ACM Computing Surveys*, *35*(3), 268-308.
- 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
- 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*(3), 766-777. doi: 10.1016/j.ejor.2015.02.018
- Cerny, V. (1985). A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications,45*(1), 41–51. doi : 10.1007/BF00940812
- 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., Moonen, L. S., Spieksma, F. C. R., & Talloen, E. (2007). The tool switching problem revisited. *European Journal of Operational Research*, *182*(2), 952–957. doi :10.1016/j.ejor.2006.07.028

- Crama, Y., Kolen, A. W. J., Oerlemans, A. G., &, Spieksma, F.C. R. (1994). Minimizing the number of tool switches on a flexible machine. *The International Journal of Flexible Manufacturing System*, *6*, 33–54. doi :10.1080/00207540110060888
- Dorigo, M., & Stützle, T. (2004), *Ant Colony Optimization,* Cambridge: MIT Press.
- Dorigo, M. (1992). Optimization, learning and natural algorithms [in italian]. (Unpublished doctoral dissertation).
- Dorigo, M., Maniezzo, V. & Colorni, A. (1991). Positive Feedback as a Search Strategy (). Technical Report No. 91-016, Politecnico di Milano, Italy .
- Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on systems, man, and cybernetics-part B*, *26*(1), 29-41.
- Dorigo, M., Di Caro, G., & Gambardella L.M. (1999). Ant algorithms for discrete optimization. *Artificial Life*, *5*(2),137–172. doi: 10.1162/106454699568728
- Dorigo, M., & Gambardella, L. M. (1997a). Ant colonies for the traveling salesman problem. *BioSystems*, *43*(2), 73-81. doi: 10.1016/S0303-2647(97)01708-5
- Dorigo, M., & Gambardella, L. M. (1997b). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1), 53-66.
- Dreo J., Petrowski A., Siarry P., & Taillard E. (2003). *Métaheuristiques pour l'optimisation difficile : Recuit simulé, recherche avec tabous, algorithmes évolutionnaires et algorithmes génétiques, colonies de fourmis…* .Paris : Eyrolles.
- Eiben, A. E., Raué , P.-E., AND Ruttkay, Z. (1994). Genetic algorithms with multi-parent recombination. In *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, Davidor, Y., Schwefel , H.-P., & Manner, R., (Eds). Lecture Notes in Computer Science, vol. 866. Springer, Berlin, 78–87.
- Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*. New York : Wiley.
- 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
- Fogel, L. J. (1962). Toward inductive inference automata. In *Proceedings of the International Federation for Information Processing Congress*. Munich, 395–399.
- Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. New York: Wiley.

- 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
- Ghiani, G., Grieco, A., & Guerriero, E. (2007). An exact solution to the TLP problem in an NC machine. *Robotics and Computer-Integrated Manufacturing*, *23*, 645–649. doi :10.1016/j.rcim.2007.02.011
- Glover F. (1986). Future paths for integer programming and links to artificial intelligence. *Computer & Operations Research*, *13*(6), 533–549. doi : 10.1016/0305-0548(86)90048-1
- Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Science*, *8* (1), 156–166. doi :10.1111/j.1540-5915.1977.tb01074.x
- Glover, F., & Laguna, M. (1997). Tabu Search. In *HandBook of combinatorial optimization: Volume 3*. Du D.-Z., & Pardalos P.M. (Eds.) 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
- Hansen, P., & Mladenovic, N. (1999). An introduction to variable neighborhood search. In *Metaheuristics: Advances and trends in local search paradigms for optimization*, Voß, S., Martello, S., Osman, I., & Roucairol, C. (Eds). Kluwer Academic Publishers, Chapter 30, 433–458.
- Hansen, P., & Mladenovic, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research, 130*(3), 449–467. doi: 10.1016/S0377-2217(00)00100-4
- Holland, J. H. (1975). *Adaption in natural and artificial systems*. The University of Michigan Press, Ann Harbor, MI.
- Jones, T. (1995a). Evolutionary algorithms, fitness landscapes and search. Ph.D. thesis, Univ. of New Mexico, Albuquerque, NM.
- Jones, T. (1995b). One operator, one landscape. Santa Fe Institute Tech. Rep. 95-02-025, Santa Fe Institute.
- Kirkpatrick, S., Gelatt, C.D., & Vecchi, M.P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

- Laporte, G., Salazar-Gonzáles, 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
- Lourenço, H. R., Martin, O., & Stützle, T. (2001). A beginner's introduction to Iterated Local Search. In *Proceedings of MIC'2001— Metaheuristics International Conference*. *Vol. 1*. Porto, Portugal, 1–6.
- Lourenço, H. R., Martin, O., & Stützle, T. (2002). Iterated local search. In *Handbook of Metaheuristics*, Glover, F., & Kochenberger, G. (Eds). International Series in Operations Research & Management Science, vol. 57. Kluwer Academic Publishers, Norwell, MA, 321–353.
- Lundy, M., & Mees, A. (1986). Convergence of an annealing algorithm. *Mathematical Programming, 34* (1), 111–124.
- Martin, O., Otto, S.W., & Felten, E. W. (1991). Large-step Markov chains for the traveling salesman problem. *Complex Systems, 5* (3), 299– 326.
- Nemhauser, G. L., & Wolsey, A. L., (1988). Integer and Combinatorial Optimization. New-York: Wiley.
- Moscato, P. (1989). On evolution, search, optimization, genetic algorithms and martial arts: Toward memetic algorithms. Tech. Rep. Caltech Concurrent Computation Program 826, California Institute of Technology, Pasadena, Calif.
- Paiva, G. S., & Carvalho, M. (2017). Improved heuristic algorithms for the job sequencing and tool switching problem. *Computers and Operations Research, 88*, 208-219.
- Moscato, P. (1999). Memetic algorithms: A short introduction. In *New Ideas in Optimization*, Corne , F.G.D.,& Dorigo, M. (Eds). New York: McGraw-Hill.Wiley,
- Osman, I. H. (1993). Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, *41*(4), 421– 451.
- Osman, I. H., & Laporte G. (1996). Metaheuristics: A bibliography. *Annals Operations. Research, 63*(5)*,* 513–623.
- Paiva, G. S., & Carvalho, M. (2017). Improved heuristic algorithms for the job sequencing and tool switching problem. *Computers and Operations Research*, 88, 208-219.
- Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial Optimization algorithms and Complexity.* New-York: Dover Publications.

- Pitsoulis, L.S., & Resende, M.G.C. (2002). Greedy randomized adaptive search procedure. In *Handbook of Applied Optimization*, Pardalos, P., & Resende, M. (Eds.). Oxford University Press, 168–183.
- Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann- Holzboog.
- Stützle, T. (1999a). Iterated local search for the quadratic assignment problem. Tech. rep. aida- 99-03, FG Intellektik, TU Darmstadt.
- Stützle T. (1999b). Local Search Algorithms for Combinatorial Problems—Analysis, Algorithms and New Applications. (Unpublished doctoral dissertation).
- Stützle, T., Hoos, H.H.(2000) MAX -MIN Ant system. *Future Generation Computer Systems,16*(8), 889-914.
- 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.
- Voß, S., Martello, S., Osman, I. H., & Roucairol C. (Eds). (1999). Meta-Heuristics—Advances and Trends in Local Search Paradigms for Optimization. Dordrecht, The Netherlands : Kluwer Academic Publishers.
- Voudouris, C. (1997). Guided local search for combinatorial optimization problems. Ph.D. dissertation, Department of Computer Science, University of Essex, 166.
- Voudouris, C., & Tsang, E. (1999). Guided local search. *European Journal of Operational Research, 113*(2), 469–499

Références internet :

- Wikipedia. (2018). Algorithme de colonies de fourmis. Retrieved July 26, 2018 from https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis