République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle et Sciences des Données (DIASD)

Master 1 Systèmes Informatiques Intelligents

Projet TP Méta-heuristique

Job Shop Scheduling Problem x Particle Swarm Optimization

Présenté par:
Moussaoui Sarah
Koroghli Hayat
Fekir Chehla Nabila
Groupe 3

Responsable du TP:
Naila Houacine

Table des Matières

Liste des Figures

Liste des Tableaux

Liste des Algorithmes

Introduction Générale

Contexte

L'optimisation de la planification et de l'ordonnancement des tâches représente un défi majeur dans les environnements industriels. Le *Job Shop Scheduling Problem* (JSSP), ou problème d'ordonnancement d'atelier, figure parmi les problèmes les plus complexes dans ce domaine. Il s'agit de déterminer une séquence optimale d'opérations à exécuter sur un ensemble de machines, en respectant des contraintes strictes d'ordre et de ressources, dans le but de minimiser le temps total d'exécution (makespan). En raison de sa nature combinatoire et de sa complexité NP-difficile, le JSSP demeure un problème difficile à résoudre, surtout lorsque la taille de l'instance augmente.

Parallèlement, les métaheuristiques, inspirées de phénomènes naturels, se sont imposées comme des alternatives performantes aux méthodes exactes, souvent limitées en temps de calcul. Parmi elles, l'algorithme *Particle Swarm Optimization* (PSO), inspiré du comportement collectif des essaims, a montré son efficacité dans divers problèmes d'optimisation continue. Cependant, son application à des problèmes discrets comme le JSSP nécessite des adaptations spécifiques.

Problématique

Comment adapter et optimiser l'algorithme PSO afin qu'il soit capable de résoudre efficacement un problème discret qui est le JSSP, et comment se positionne-t-il en termes de performance par rapport à d'autres méthodes telles que la recherche en profondeur (DFS) ?

Objectifs

L'objectif principal de ce travail est d'explorer l'application de l'algorithme PSO à un problème discret complexe, le JSSP. Pour cela, plusieurs sous-objectifs sont poursuivis :

- Adapter le fonctionnement du PSO à la nature discrète du problème JSSP.
- Mettre en œuvre un PSO optimisé en intégrant différentes stratégies de génération de solutions et de mise à jour des particules.
- Réaliser un réglage des hyperparamètres à l'aide de techniques comme le *grid search*.
- Comparer les performances du PSO avec celles de la méthode de recherche en profondeur (DFS) et de l'algorithme génétique sur les mêmes instances.

Ce mémoire est structuré en trois chapitres :

1. Chapitre 1 : Présentation de l'algorithme PSO et du probleme JSSP
2. Chapitre 2 : Détails de la modélisation du problème et conception
3. Chapitre 3 : Implémentation, réglages et comparaison des résultats

Une conclusion générale viendra clôturer ce travail en résumant les principaux apports et en ouvrant sur des perspectives d'amélioration.

Chapter 1

Géneralités

1.1 Introduction

Dans le domaine de l'optimisation, les métaheuristiques sont largement employées pour résoudre des problèmes complexes où les méthodes exactes sont inefficaces. L'objectif de ce rapport est d'étudier en détail l'algorithme PSO appliqué au problème JSSP, en mettant en avant son inspiration, son fonctionnement et ses performances. L'analyse portera sur les principes théoriques, l'implémentation de l'algorithme ainsi que sur une évaluation expérimentale.

1.2 Job Shop Scheduling Problem (JSSP)

1.2.1 Présentation du problème

Le *Job Shop Scheduling Problem* (JSSP) est l'un de sproblémes les plus connus en ordonnancement. Il consiste à planifier n jobs sur m machines, chaque job étant composé d'une séquence d'opérations à exécuter dans un ordre précis. Chaque opération doit être traitée sur une machine spécifique pendant une durée connue. L'objectif est généralement de minimiser le *makespan*, c'est-à-dire le temps total nécessaire pour achever tous les jobs, défini comme :

$$C_{\max} = \max_i \{C_i\}$$

où C_i est le temps d'achèvement du job i.

1.2.2 Contraintes

Les principales contraintes associées au JSSP sont :

- Chaque job est constitué d'une séquence ordonnée d'opérations.
- Une machine ne peut traiter qu'une seule opération à la fois.
- Une opération ne peut pas être interrompue une fois commencée (non-préemption).
- Les opérations doivent respecter les délais d'enchaînement imposés par l'ordre des machines pour chaque job.

1.2.3 Complexité et recours aux métaheuristiques

Le JSSP est un problème *NP-difficile*, ce qui signifie que le nombre de solutions possibles croît de manière exponentielle avec la taille du problème. Plus précisément, pour n jobs et m machines, le nombre de séquences possibles peut atteindre jusqu'à $(n!)^m$. Par conséquent, les méthodes exactes deviennent rapidement impraticables à mesure que l'instance du problème s'agrandit. Pour faire face à cette complexité, des approches comme les métaheuristiques sont privilégiées.[1]

1.3 Optimisation du Job Shop Scheduling Problem à travers l'utilisation des Metaheuristiques

Plusieurs approches ont été proposées pour l'optimisation du JSSP, et les metaheuristiques se montrent particulièrement efficaces. Ces méthodes de recherche permettent d'explorer de grands espaces de solutions et d'obtenir des résultats de plus en plus proches de la solution optimale, tout en réduisant le temps d'exécution. Elles sont particulièrement adaptées aux problèmes complexes et NP-difficiles. Dans le cadre de notre travail, nous allons citer l'algorithme *Particle Swarm Optimization (PSO)*, qui est à la fois efficace et simple à implémenter, et qui a fait ses preuves pour résoudre des problèmes d'optimisation de type JSSP.

1.3.1 Particle Swarm Optimization (PSO)

Inspiration

L'optimisation par essaim de particules (PSO - Particle Swarm Optimization) est une technique d'optimisation inspirée du comportement collectif des oiseaux en vol ou des bancs de poissons. Ce concept repose sur l'idée que chaque individu de l'essaim (ou particule) ajuste sa trajectoire en fonction de sa propre expérience et de celle des autres membres du groupe. En permettant aux particules de partager les informations entre elles, cette méthode permet une convergence rapide vers des solutions optimales.

Figure 1.1: Swarm d'oiseaux en recherche de routes migratoires

Fonctionnement

Dans l'algorithme PSO, chaque particule représente une **solution candidate** au problème d'optimisation. Elle est caractérisée par :

- Une **position** : solution spécifique dans l'espace de recherche
- Une **vitesse** : vecteur déterminant sa direction de déplacement

Mise à jour de la position : La position $\mathbf{x}_i(t+1)$ de la particule i à l'itération $t+1$ est calculée par :

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{1.1}$$

où $\mathbf{v}_i(t+1)$ est la vitesse mise à jour.

Mise à jour de la vitesse : La vitesse $\mathbf{v}_i(t+1)$ est ajustée selon :

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1 r_1(\mathbf{p}_i - \mathbf{x}_i(t)) + c_2 r_2(\mathbf{g} - \mathbf{x}_i(t)) \tag{1.2}$$

avec :

- w : coefficient d'inertie
- c_1, c_2 : coefficients d'apprentissage
- $r_1, r_2 \sim U(0,1)$: variables aléatoires
- $\mathbf{p}_i$: meilleure position locale (*best local*)
- $\mathbf{g}$: meilleure position globale (*best global*)

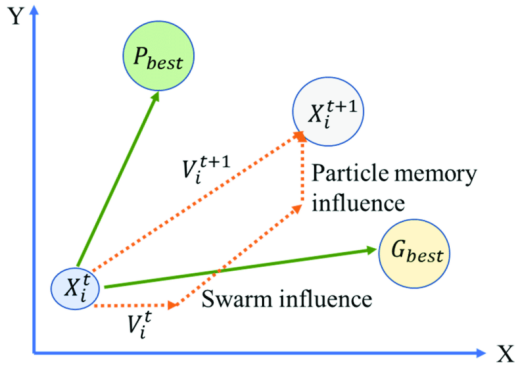


Figure 1.2: Mise à jour d'une particule

Équilibre exploration/exploitation
Le PSO maintient un compromis optimal entre

- **Exploration** (diversification) :
  - Permise par le terme d'inertie ($w\mathbf{v}_i(t)$)
  - Favorise la découverte de nouvelles régions
- **Exploitation** (intensification) :
  - Guidée par $\mathbf{p}_i$ et $\mathbf{g}$
  - Affine les solutions prometteuses

Mise à jour de la position et de la vitesse
Le principe de mise à jour de la position et de la vitesse des particules est au cœur de l'algorithme PSO. À chaque itération, la position et la vitesse des particules sont mises à jour en utilisant les relations suivantes :

$$\text{vitesse}_i^{t+1} = w \cdot \text{vitesse}_i^t + c_1 \cdot r_1 \cdot (\text{best}_i - \text{position}_i^t) + c_2 \cdot r_2 \cdot (\text{best}_g - \text{position}_i^t)$$

$$\text{position}_i^{t+1} = \text{position}_i^t + \text{vitesse}_i^{t+1}$$

Paramètres clés du PSO

Le PSO repose sur plusieurs paramètres essentiels qui influencent son efficacité :

- **Facteur d'inertie** (w) : Ce paramètre détermine l'influence de la vitesse des particules à l'itération précédente sur leur mouvement actuel, permettant ainsi de contrôler l'exploration et l'exploitation.
- **Coefficients d'accélération** (c_1, c_2) : c_1 représente l'attraction des particules vers leur meilleure position locale, tandis que c_2 les oriente vers la meilleure position globale trouvée. L'équilibre entre ces deux coefficients est essentiel pour optimiser la recherche.

- **Taille de la population** : Le nombre de particules dans la population a un impact direct sur la qualité des solutions obtenues. Une population trop réduite peut restreindre l'exploration de l'espace de recherche, tandis qu'une population trop grande peut ralentir le processus d'optimisation.
- **Nombre d'itérations** : Le nombre d'itérations détermine la durée du processus d'optimisation. Un nombre insuffisant d'itérations peut entraîner une solution suboptimale, tandis qu'un nombre trop élevé peut ralentir le processus et entraîner une stagnation, où les particules cessent de trouver de nouvelles solutions améliorées.

Étapes de l'algorithme PSO

1. Initialiser l'essaim avec des positions et vitesses aléatoires.
2. Évaluer la fonction objectif pour chaque particule.
3. Mettre à jour les meilleurs personnels (*pbest*) et le meilleur global (*gbest*).
4. Mettre à jour les vitesses et positions selon les équations définies.
5. Répéter jusqu'à la convergence ou atteinte d'un critère d'arrêt.

Pour le pseudo-code (voir Annexe 3)

Avantages du PSO

- Facile à implémenter avec peu de paramètres.
- Convient aux problèmes d'optimisation à haute dimension.
- S'adapte aux espaces de recherche complexes et non linéaires.

Applications du PSO

Le PSO est largement utilisé dans divers domaines, notamment :

- L'optimisation de fonctions mathématiques.
- Le réglage de paramètres en apprentissage automatique.
- La planification de trajectoires en robotique.
- Les problèmes d'ordonnancement, comme le Job Shop Scheduling Problem (JSSP).

Cette méthode est particulièrement adaptée aux problèmes de recherche dans des espaces complexes et multidimensionnels où les solutions doivent être approximées plutôt qu'exactes, comme c'est le cas pour le Job Shop Scheduling Problem (JSSP). En minimisant le temps d'exécution total (makespan) d'une série de tâches, PSO aide à générer des ordonnancements optimaux qui respectent les contraintes du problème. [2]–[4]

1.4 conclusion

Le Job Shop Scheduling Problem (JSSP) constitue un défi majeur en optimisation combinatoire en raison de sa complexité intrinsèque et de ses nombreuses contraintes. Les méthodes exactes, bien qu'efficaces pour les petites instances, deviennent rapidement inapplicables dès que la taille du problème augmente. Dans ce contexte, les métaheuristiques telles que le PSO (Particle Swarm Optimization) apparaissent comme des alternatives prometteuses. Leur capacité à explorer efficacement de vastes espaces de recherche permet d'obtenir des solutions satisfaisantes en un temps raisonnable. Ce chapitre a posé les bases nécessaires à la compréhension du JSSP

et introduit le PSO comme méthode d'optimisation pertinente pour ce type de problème. Les chapitres suivants se concentreront sur l'implémentation concrète de cet algorithme, son adaptation au JSSP, et l'évaluation de ses performances.

Chapter 2

Conception

2.1 Introduction

Dans ce chapitre, nous présentons les différentes étapes qui nous ont permis de concevoir une solution pour le problème d'ordonnancement des tâches dans un atelier (Job Shop Scheduling Problem). Nous commençons par modéliser le problème pour le rendre compréhensible par un programme informatique. Ensuite, nous expliquons comment nous avons adapté l'algorithme PSO (Particle Swarm Optimization) à notre cas, en utilisant des échanges (swaps) pour travailler avec des données discrètes. Enfin, nous introduisons des améliorations pour rendre l'algorithme plus performant, et nous présentons aussi une méthode de recherche classique (DFS) combinée avec des techniques d'élagage pour limiter le temps de calcul.

2.2 Modélisation du probléme

Un ordonnancement (Schedule) est une liste chronologique de tuples représentant l'ordre d'exécution des jobs sur les machines. Chaque tuple contient :

- **Job ID** : l'identifiant du job.
- **Machine ID** : l'identifiant de la machine utilisée.

Exemple :

$$\text{Schedule} = \left[\begin{array}{cccc} (1,0) & (1,1) & (0,0) & (0,1) \end{array} \right]$$

Interprétation :

- Exécution du Job 1 sur la Machine 0.
- Puis du Job 1 sur la Machine 1.
- Suivi du Job 0 sur la Machine 0.
- Enfin du Job 0 sur la Machine 1.

Clarifications supplémentaires :

- La séquence dans la liste du job reflète l'ordre de passage sur les machines (problème de flow shop ou job shop selon les contraintes).
- Le temps d'exécution d'un job sur une machine est préservé dans une structure afin de calculer le makespan et tracer le diagramme de Gantt.

2.3 Algorithme Swap-Based PSO

La principale limitation de l'algorithme **Particle Swarm Optimization (PSO)** est qu'il est conçu pour des problèmes continus. Plusieurs variantes ont été proposées pour l'adapter aux problèmes discrets et combinatoires, dont le **Swap-Based PSO (SPSO)**.

2.3.1 Principe de Fonctionnement

Le SPSO s'inspire du PSO standard, mais remplace les opérations arithmétiques continues par des **opérations discrètes**, Chaque particule est caractérisée par :

- **Une position ($\mathbf{X_i}$)** : Une solution condidate (ex : Schedule = $\begin{bmatrix} (1,0) & (1,1) & (0,0) & (0,1) \end{bmatrix}$).
- **Une vitesse ($\mathbf{V_i}$)** : Une séquence de swaps (échanges) à appliquer pour modifier la position.

Mise à jour de la vitesse et position dans SPSO

Les nouvelles positions sont calculées en appliquant des swaps basés sur :

- La meilleure solution personnelle (**pBest**)
- La meilleure solution globale (**gBest**)

Mise à jour de la vitesse

La vitesse d'une particule dans le Swap-Based PSO est mise à jour selon trois composantes :

$$V_i(t+1) = \omega V_i(t) \oplus c_1(pBest_i \ominus X_i(t)) \oplus c_2(gBest \ominus X_i(t)) \tag{2.1}$$

où :

- ω est le coefficient d'inertie (généralement $\in [0.1, 0.9]$)
- c_1, c_2 sont les coefficients d'apprentissage
- $\ominus$ représente l'opérateur de différence entre permutations
- $\oplus$ est l'opérateur de combinaison de swaps

Composantes de la vitesse

1. **Terme d'inertie** ($\omega V_i(t)$) :
   - Conserve une partie des swaps précédents
   - Exemple : si $\omega = 0.6$, on conserve 60% des swaps aléatoirement
2. **Influence personnelle** ($c_1(pBest_i \ominus X_i(t))$) :
   - Calcule les swaps nécessaires pour passer de X_i à $pBest_i$
   - Applique une fraction de ces swaps selon c_1
3. **Influence sociale** ($c_2(gBest \ominus X_i(t))$) :
   - Calcule les swaps pour atteindre $gBest$
   - Applique une fraction de ces swaps selon c_2

Mise à jour de la position

La nouvelle position est obtenue en appliquant la vitesse mise à jour :

$$X_i(t+1) = X_i(t) \otimes V_i(t+1) \tag{2.2}$$

où $\otimes$ applique séquentiellement chaque swap de $V_i(t+1)$ à $X_i(t)$.

2.3.2 Initialisation des particules

Trois stratégies d'initialisation ont été utilisées :

- **Aléatoire** : toutes les particules sont générées sans règle spécifique, assurant une grande diversité.
- **SPT** : toutes les particules suivent la règle du plus court temps de traitement, offrant des solutions initiales de bonne qualité.
- **Hybride** : moitié SPT, moitié aléatoire, pour combiner qualité initiale et diversité des solutions.

2.3.3 Pseudo-code du PSO basé sur les séquences de swaps

Algorithm 1 PSO basé sur les séquences de swaps (SPSO) avec mise à jour probabiliste de la vélocité

for chaque particule i **do**
  Initialiser la position $X_i(0)$ avec un ordonnancement faisable aléatoire (ou via SPT ou hybride)
  Initialiser la vélocité $V_i(0)$ avec une séquence de swaps aléatoire respectant les contraintes
  Définir le meilleur personnel $pBest_i \leftarrow X_i(0)$
end for
Définir le meilleur global $gBest \leftarrow \arg\min_i \{fitness(pBest_i)\}$
for chaque itération t **do**
  for chaque particule i **do**
    $V_i(t+1) \leftarrow \emptyset$
    for chaque swap s dans $V_i(t)$ **do**
      if rand() $< \omega$ **then**
        Ajouter s à $V_i(t+1)$
      end if
    end for
    for chaque swap s dans $pBest_i \ominus X_i(t)$ **do**
      if rand() $< c_1$ **then**
        Ajouter s à $V_i(t+1)$
      end if
    end for
    for chaque swap s dans $gBest \ominus X_i(t)$ **do**
      if rand() $< c_2$ **then**
        Ajouter s à $V_i(t+1)$
      end if
    end for
    Mettre à jour la position : $X_i(t+1) \leftarrow X_i(t) \otimes V_i(t+1)$
    if $fitness(X_i(t+1)) < fitness(pBest_i)$ **then**
      $pBest_i \leftarrow X_i(t+1)$
    end if
    if $fitness(X_i(t+1)) < fitness(gBest)$ **then**
      $gBest \leftarrow X_i(t+1)$
    end if
  end for
end for
return $gBest$ comme le meilleur ordonnancement trouvé

2.4 Algorithme Swap-Based PSO amélioré

Le PSO basé sur les séquences de swaps présente un inconvénient majeur : la stagnation prématurée des particules autour d'un optimum local. Afin de pallier ce problème dans le cadre de ce travail, plusieurs stratégies et fonctions ont été introduites pour favoriser l'exploration et maintenir la diversité de la population. Les principales améliorations apportées sont les suivantes :

Initialisation équilibrée par cluster de charge machine

Une fonction d'initialisation personnalisée a été mise en place afin d'améliorer la qualité des solutions initiales dans le cadre du PSO. Elle vise à générer une séquence d'opérations équilibrée en termes de charge machine, ce qui permet d'orienter l'algorithme vers des zones plus prometteuses de l'espace de recherche dès le départ.

Le principe est le suivant :

- Pour chaque job, la première opération non encore planifiée est considérée comme disponible.
- Ces opérations disponibles sont regroupées en *clusters* de taille maximale fixe, tout en évitant de placer deux opérations affectées à la même machine dans un même cluster.
- Les clusters sont ensuite triés et insérés dans la séquence en priorisant les opérations affectées aux machines les moins chargées, et avec des temps d'exécution plus courts.

L'objectif est de répartir la charge de travail de manière plus homogène entre les machines dès la génération de la solution initiale. Cela permet d'éviter les séquences déséquilibrées qui entraîneraient de longues périodes d'inactivité pour certaines machines, ce qui est courant avec une initialisation purement aléatoire.

Cette stratégie améliore la performance globale du PSO en fournissant des points de départ plus compétitifs, réduisant ainsi le nombre d'itérations nécessaires pour atteindre une solution de bonne qualité.

Remplacement partiel par diversification aléatoire et perturbation contrôlée

Lorsqu'une stagnation est détectée, une portion des particules les moins performantes (environ un quart) est remplacée. Deux approches sont utilisées :

- Avec une probabilité de 70 %, la particule est entièrement remplacée par une nouvelle solution générée aléatoirement.
- Sinon, une nouvelle solution est créée en appliquant un nombre limité d'échanges (swaps) à la meilleure solution globale actuelle. Ces perturbations sont contraintes à respecter la validité de la séquence pour chaque job, garantissant que la solution reste faisable.

Cette technique permet d'introduire de la diversité dans la population tout en conservant une orientation vers les bonnes solutions, facilitant ainsi la sortie d'un minimum local.

Amélioration par Paramètres Adaptatifs

Afin d'améliorer l'exploration et l'exploitation au cours de l'algorithme, des paramètres adaptatifs ont été introduits. Cette approche permet de favoriser une exploration plus large au début de l'algorithme, et une exploitation plus ciblée à mesure que l'algorithme progresse. L'adaptation des paramètres dépend de la progression de l'itération, ce qui permet à l'algorithme de s'ajuster à l'évolution de la recherche.

Les principaux paramètres ajustés sont les suivants :

- **L'inertie w :** diminué de manière quadratique au fur et à mesure de l'évolution de l'itération, ce qui permet une exploration large au début et une exploitation plus ciblée à la fin. La formule utilisée est :

$$w_{\text{current}} = w_{\max} - (w_{\max} - w_{\min}) \cdot \text{progress}^2$$

où $w_{\max}$ et $w_{\min}$ sont respectivement les bornes supérieure et inférieure de w, et *progress* est la fraction de l'itération actuelle par rapport à l'itération maximale.

- **Paramètre c_1 (cognitif) :** Ce paramètre, qui contrôle l'influence de la mémoire individuelle, diminue au fil du temps selon une progression exponentielle. La formule est la suivante :

$$c_{1,\text{current}} = c_{1,\max} - (c_{1,\max} - c_{1,\min}) \cdot \text{progress}^{1.5}$$

Cette réduction permet de commencer par une exploration plus large avant de se concentrer sur les meilleures solutions trouvées.

- **Paramètre c_2 (social) :** Contrairement à c_1, le paramètre c_2 est ajusté pour favoriser une exploitation rapide des solutions au fur et à mesure de la progression de l'algorithme. La formule utilisée est :

$$c_{2,\text{current}} = c_{2,\min} + (c_{2,\max} - c_{2,\min}) \cdot \text{progress}^{0.7}$$

Cela permet un renforcement de l'exploitation à mesure que l'algorithme avance.

Amélioration par ajout de mutations

Lorsqu'une stagnation est détectée, ajouter des mutations pour diversifier la population et favoriser l'exploration de nouvelles solutions.

Algorithm 2 Mutation aléatoire sur une particule

```
function MUTATE(X_i, prob_mut)
    if rand() < prob_mut then
        Sélectionner aléatoirement deux opérations op_1 et op_2 dans X_i
        Vérifier la faisabilité du swap (op_1, op_2)
        if le swap est faisable then
            Échanger op_1 et op_2 dans X_i
        end if
    end if
    return X_i
end function
```

Ces ajustements permettent à l'algorithme de privilégier l'exploration au début, tout en se concentrant progressivement sur l'exploitation des meilleures solutions au fur et à mesure que l'itération avance.

2.5 Algorithme DFS et Optimisation par Bound & Branch et Pruning

Dans le cadre du problème *Job Shop Scheduling* (JSSP), nous avons utilisé une méthode de recherche *Depth-First Search* (DFS) pour explorer toutes les solutions possibles. Toutefois, nous avons constaté que la complexité du problème rend cette approche inefficace pour des instances de taille plus grande. DFS est une méthode déterministe, ce qui signifie qu'elle suit un parcours systématique de toutes les options possibles, mais elle peut devenir très lente lorsqu'il y a un grand nombre de possibilités à explorer.

Le JSSP étant un problème NP-difficile, une implémentation DFS standard génère un arbre de décision dont la taille croît de manière exponentielle. Pour illustrer ce phénomène, prenons le cas d'une instance avec 5 jobs et 5 machines : l'espace de recherche théorique atteint $5!^5$ combinaisons (environ 24×10^9). Nous avons donc dû mettre en place des mécanismes d'optimisation pour réduire cet espace de recherche.

Pour ce faire, nous avons implémenté trois techniques clés :

- **Bound (Élagage par borne inférieure)** : La méthode `lower_bound` que nous avons implémentée calcule une estimation optimiste du makespan en prenant en compte :
  - La charge restante des opérations non planifiées pour chaque job, c'est-à-dire la somme des durées des opérations restantes pour les jobs encore incomplets.
  - Le makespan actuel, qui est comparé à la charge restante pour déterminer si la branche peut produire une solution meilleure que la meilleure trouvée jusqu'à présent.

  Si la borne inférieure estimée (calculée via `lower_bound`) est supérieure ou égale au meilleur makespan trouvé jusqu'à présent, la branche est élaguée (abandonnée) car elle ne peut plus produire de solution meilleure.
- **Branch (Sélection intelligente des branches)** :
  Nous avons introduit un tri des jobs selon le critère *most constrained first*, priorisant ainsi l'exploration des séquences les plus prometteuses. Ce choix s'est avéré crucial pour accélérer la convergence.
- **Pruning (Élagage dynamique)** :
  Nous avons inclus cette méthode pour vérifier systématiquement si le `end_time` calculé dépasse le `best_makespan` courant. Ce mécanisme permet d'éliminer immédiatement les solutions sous-optimales.

pour le pseudo code (voir Annexe 4)

2.6 Conclusion

Dans ce chapitre, nous avons expliqué comment le problème d'ordonnancement a été modélisé et comment nous avons utilisé l'algorithme PSO adapté aux problèmes de type Job Shop. Nous avons vu plusieurs façons d'initialiser les solutions, dont l'approche aléatoire, SPT et une version hybride. Pour améliorer les résultats, nous avons ajouté des idées comme une meilleure répartition des tâches dès le départ, des perturbations pour éviter les blocages, et des paramètres qui s'adaptent au fil du temps.
Enfin, une méthode de recherche plus classique (DFS) a aussi été mise en place, avec des techniques pour éviter d'explorer trop de solutions inutiles. Tout cela a permis de rendre notre approche plus efficace et plus intelligente.

Chapter 3

Implémentation de la solution

3.1 Introduction

Dans ce chapitre, nous allons chercher à obtenir les meilleurs résultats possibles en testant les différents algorithmes cité dans le chapitre précedent. Pour commencer, nous effectuerons une recherche des meilleurs paramètres afin de trouver les bonnes valeurs pour chaque méthode. Ensuite, nous testerons l'algorithme PSO basé sur les échanges (swap), avec plusieurs façons de démarrer les solutions : de manière aléatoire, avec la règle SPT, ou avec une combinaison des deux. Nous appliquerons aussi une version améliorée du PSO, avec des optimisations pour améliorer les performances. Enfin, nous comparerons ces résultats avec ceux obtenus grâce à l'algorithme DFS, combiné avec la méthode Branch and Bound.

3.2 Environnement expérimental

3.2.1 Configuration matérielle et logicielle

Les tests ont été exécutés sur une machine avec les spécifications suivantes :

- **Processeur :** Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 MHz, 4 cœur(s), 8 processeur(s) logique(s)
- **Mémoire :** 8 Go de RAM
- **Système d'exploitation :** Microsoft Windows 11 Professionnel
- **Compilateur :** Python 3.10.10

3.2.2 Description du dataset utilisé

Le dataset Taillard, proposé par Éric Taillard en 1993, est une collection de benchmarks standards pour tester et comparer les performances des algorithmes de résolution du *Job Shop Scheduling Problem (JSSP)*.

Caractéristiques du dataset

- **Nom du fichier :** sous la forme `dataXX_YY.txt`, où :
  - `XX` représente le nombre de jobs (ex. 15, 20, 30, 50, 100)
  - `YY` représente le nombre de machines
- **Type de problème :** JSSP classique avec une séquence stricte d'opérations sur différentes machines

Structure des données

- **Instances :** 80 instances classées par combinaisons *jobs* × *machines* (ex. 15×15, 15×20), chaque combinaison ayant 10 instances.
- **Représentation :** chaque instance est composée de deux matrices $n \times m$:
  - **Matrice des machines** : indique l'ordre des machines pour chaque opération
  - **Matrice des durées** : indique la durée de traitement de chaque opération

Exemple d'entrée

Matrice des machines :

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Matrice des durées :

$$\begin{bmatrix} 5 & 8 & 3 \\ 7 & 6 & 2 \\ 4 & 5 & 9 \end{bmatrix}$$

Critères de sélection des instances

- **Variété et complexité :** couvre différents cas d'échelle (petite à grande), représentatif de scénarios réels.
- **Benchmark reconnu :** standard académique utilisé pour comparer des algorithmes (PSO, Branch Bound, etc.).

3.3 Implémentation de l'algorithme Swap-Based PSO

3.3.1 Sélection des paramètres du PSO via Grid Search

Les paramètres de l'algorithme SPSO, affichés dans le tableau ci-dessous, ont été sélectionnés après l'exécution d'un algorithme de *Grid Search* visant à optimiser les performances pour différentes tailles de problèmes (nombre de jobs et de machines) :

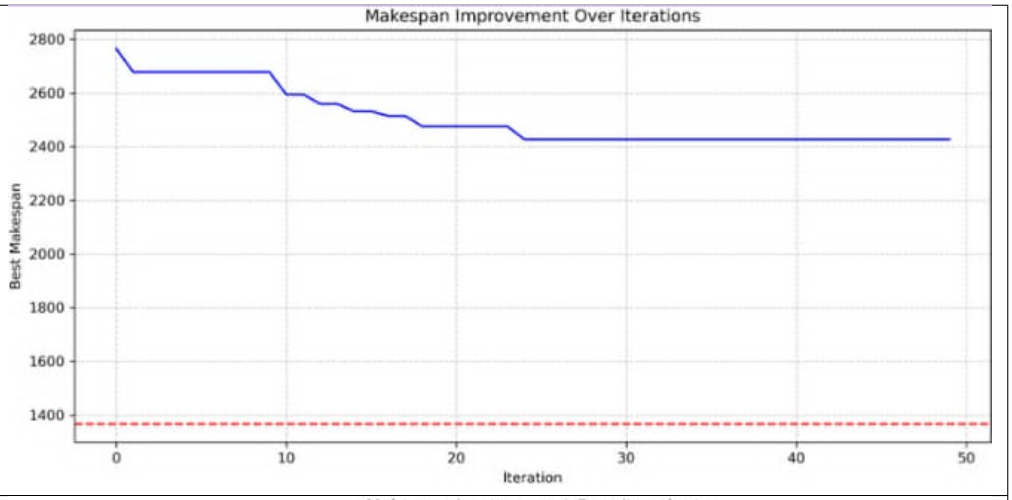
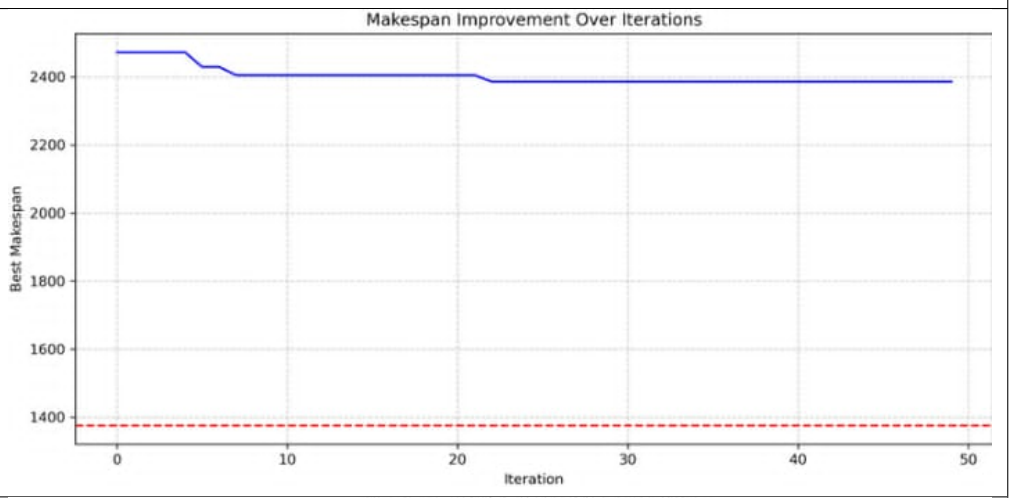
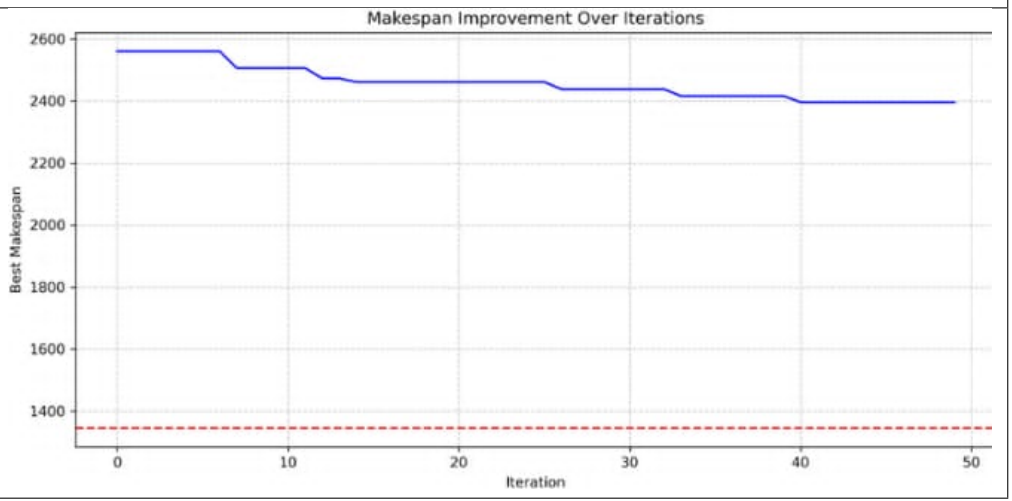
Méthode d'initialisation	Nb particules	Nb itérations	w	c1	c2
random	30	100	0.7	0.9	0.6
SPT	30	100	0.7	0.9	1
Hybride (SPT + random)	50	100	0.7	1	0.6

Table 3.1: Paramètres utilisés pour chaque méthode d'initialisation

3.3.2 Tests et résultats

Nous visualisons la convergence de l'algorithme à travers le graphique suivant, pour 20 jobs et 15 machines.

Table 3.2: Courbes de convergence du SPSO

Initialisation avec random	
Initialisation avec SPT	
Initialisation hybride	

3.4 Implémentation de l'algorithme Swap-Based PSO amélioré

3.4.1 Sélection des paramètres du PSO via Grid Search

Les paramètres de l'algorithme SPSO, affichés dans le tableau ci-dessous, ont été sélectionnés après l'exécution d'un algorithme de *Grid Search* visant à optimiser les performances pour différentes tailles de problèmes (nombre de jobs et de machines) :

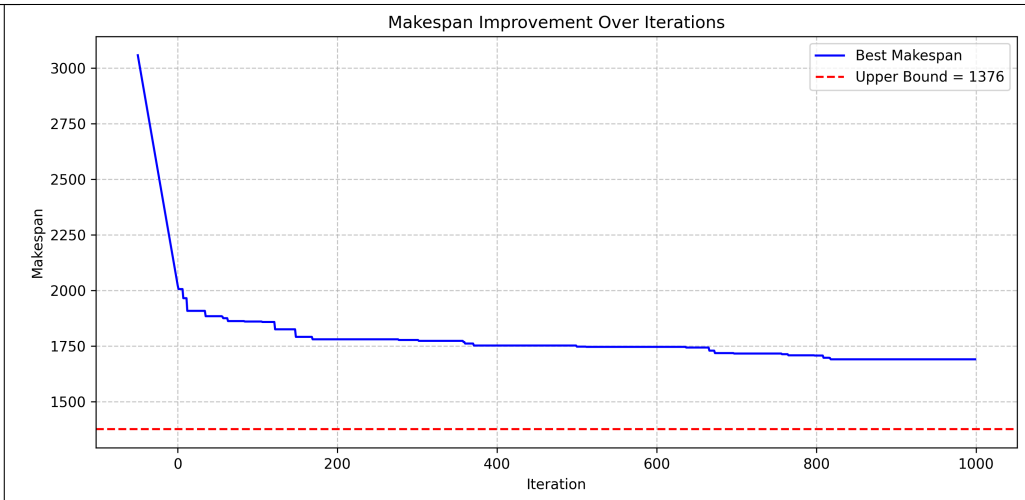
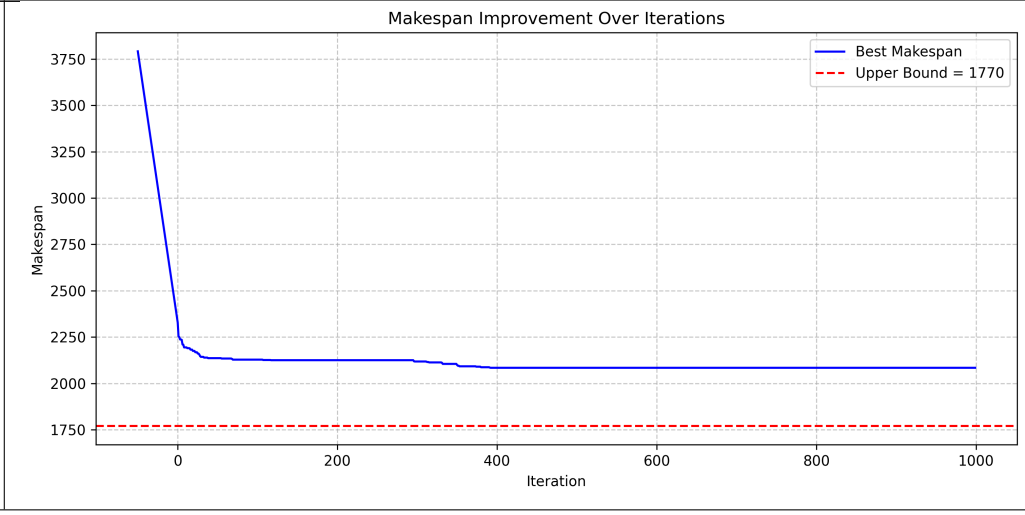
Nb jobs machines	Nb particules	Nb itérations	w	c1	c2	Taux mutation
(20 15)	30	1000	(0.7 0.2)	(0.9 0.5)	(0.8 0.6)	0.5
(30 15)	80	1000	(0.8 0.2)	(0.9 0.5)	(0.9 0.6)	0.6
(50 15)	100	1000	(0.5 0.2)	(0.8 0.4)	(0.9 0.6)	0.7

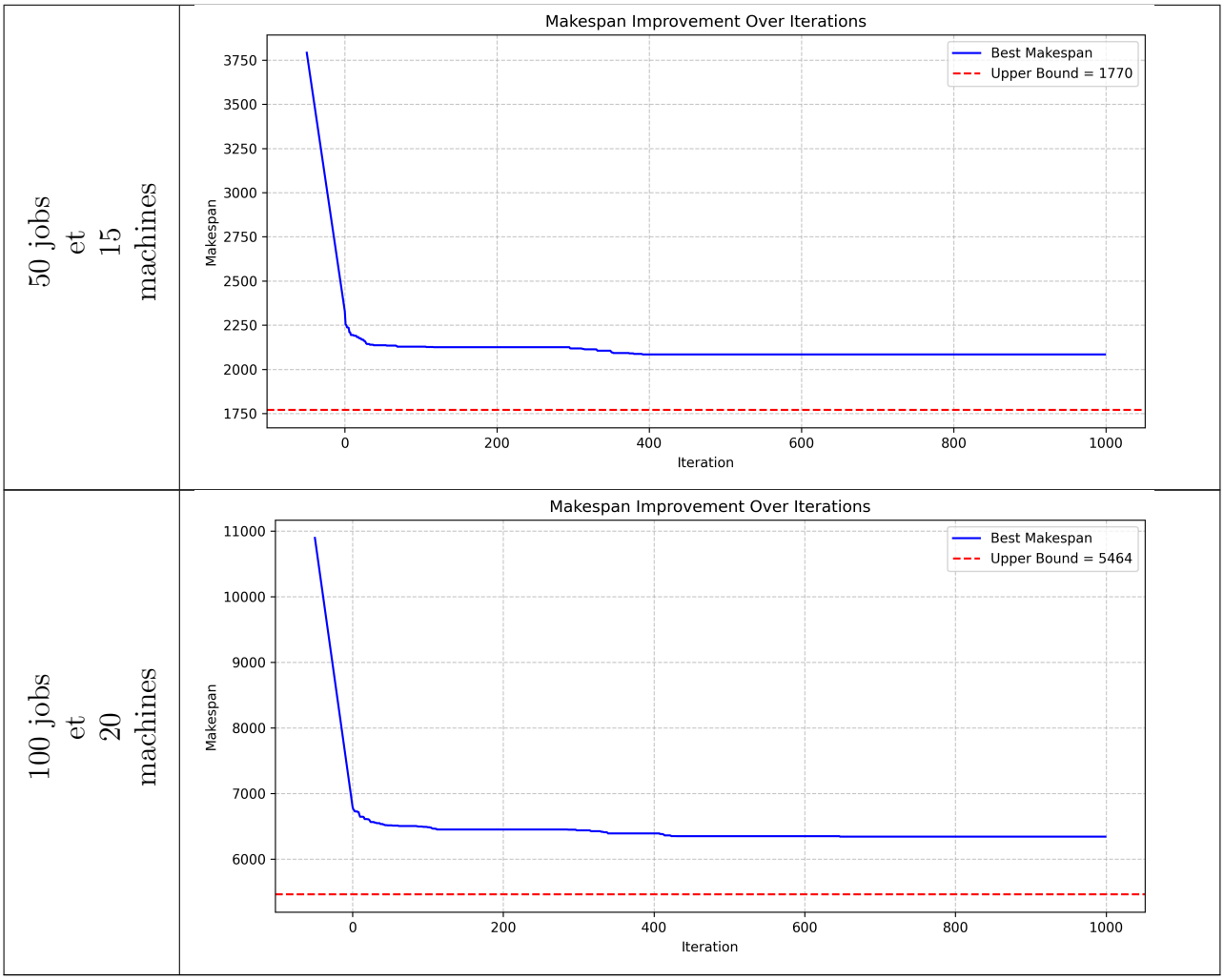
Table 3.3: Paramètres PSO utilisés selon la taille du problème

3.4.2 Tests et résultats

Nous visualisons la convergence de l'algorithme à travers le graphique suivant, pour différents nombres de jobs et de machines.

Table 3.4: Courbes de convergence

20 jobs et 15 machines	
30 jobs et 15 machines	



Le diagramme de Gantt suivant correspond à l'instance de 30 jobs et 15 machines. Il montre l'ordonnancement trouvé par l'algorithme après optimisation. Pour visualiser d'autres instances, veuillez consulter l'annexe A [lien].

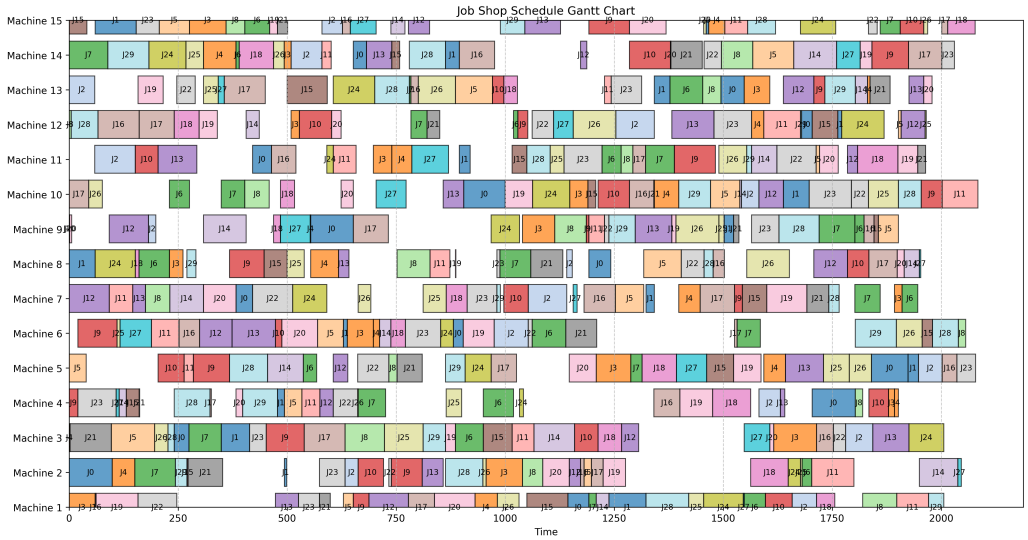


Figure 3.1: Diagramme de Gantt après optimisation

3.5 Implémentation de l'algorithme Depth-First Search (DFS)

Le tableau ci-dessous présente les temps d'exécution de l'algorithme pour différentes configurations d'instances, en fonction du nombre de jobs et de machines.

Table 3.5: Temps d'exécution pour différentes tailles d'instances JSSP

Taille (Jobs × Machines)	**Temps (secondes)**
2×2	0.001
3×3	0.0185
4×4	7.80
5×5	18 000

3.5.1 Tests et résultats:

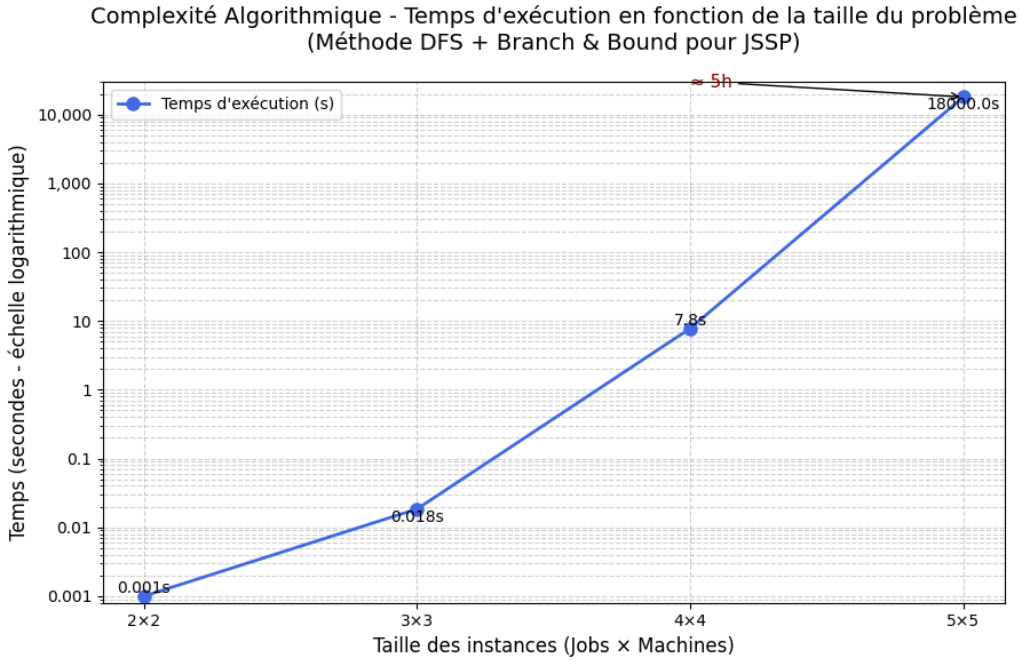


Figure 3.2: Temps d'execution de DFS en fonction de la taille du problème

Analyses des résultats:

- Dès une taille de 4×4, le temps d'exécution atteint plusieurs secondes, ce qui montre déjà une croissance rapide de la complexité.
- Pour une instance de taille 5×5, le temps dépasse les 5 heures, ce qui est clairement inacceptable dans un contexte opérationnel ou industriel.
- Même avec l'ajout de la stratégie *Branch and Bound*, l'élagage reste insuffisant pour contenir l'explosion combinatoire.
- Le nombre de combinaisons possibles croît de façon exponentielle avec le nombre de jobs et de machines, rendant impossible l'exploration exhaustive de toutes les solutions.

Ces résultats montrent que l'approche DFS, bien qu'elle est utile pour de petites instances, ne peut pas être utilisée efficacement pour des problèmes de taille moyenne ou grande dans le cadre du JSSP.

3.6 Analyse des résultats et discussion

- **Discussion des performances des algorithmes**
  La version améliorée de l'algorithme *Swap-Based PSO* a montré de meilleures performances que la version initiale. Elle réduit significativement le makespan et converge plus rapidement vers des solutions de qualité. L'algorithme *DFS* (Depth-First Search), bien qu'efficace pour de très petites instances, devient rapidement inefficace pour les cas plus complexes, avec des temps d'exécution très élevés et des solutions de moins bonne qualité.

- **Analyse des éventuels points faibles et pistes d'amélioration**
  La version améliorée du PSO reste sujette à une certaine stagnation, bien que celle-ci soit moins rigide. Pour y remédier, il serait pertinent d'ajouter des mécanismes adaptatifs sur les paramètres du PSO et de favoriser davantage la diversité des particules. Quant au DFS, sa complexité exponentielle rend son application peu réaliste pour les instances moyennes à grandes ; une hybridation avec des heuristiques pourrait être envisagée.

- **Comparaison des résultats des algorithmes**
  - *DFS* donne des résultats exacts sur de très petits problèmes, mais devient inutilisable pour des tailles réalistes à cause du temps d'exécution.
  - *Swap-Based PSO* (version initiale) donne des résultats raisonnables, mais souffre de convergence prématurée vers un optimum local.
  - *Swap-Based PSO amélioré* présente une meilleure stabilité, une meilleure capacité à échapper à la stagnation, et une amélioration globale de la qualité des solutions et du temps de convergence.

3.7 Conclusion

À travers ce chapitre, nous avons exploré plusieurs approches pour résoudre le problème d'ordonnancement. Nous avons commencé par le PSO basé sur les swaps, en testant différentes méthodes d'initialisation comme l'aléatoire, la règle SPT, ou une combinaison des deux. Ensuite, nous avons appliqué une version améliorée du PSO, en ajoutant des optimisations pour améliorer la qualité des solutions et accélérer la convergence. Enfin, nous avons testé l'algorithme DFS avec la méthode Branch and Bound, pour évaluer sa performance sur des petites instances.

Les résultats obtenus montrent que les méthodes basées sur le PSO sont plus efficaces pour les instances de taille moyenne à grande, surtout la version améliorée. En revanche, l'approche DFS, bien qu'exacte, devient vite impraticable à cause de son temps de calcul très élevé. Cette analyse nous permettra de mieux orienter le choix de la méthode selon la taille et la complexité du problème à résoudre.

Conclusion Genérale

Conclusion générale

Dans ce projet, nous avons étudié le problème d'ordonnancement d'atelier (JSSP), un problème NP-difficile qui consiste à organiser des tâches sur plusieurs machines tout en respectant des contraintes strictes. Ce type de problème devient très difficile à résoudre efficacement lorsque sa taille augmente, ce qui limite l'utilisation des méthodes exactes. Pour surmonter cette difficulté, nous avons utilisé une métaheuristique appelée PSO (Particle Swarm Optimization).

Nous avons d'abord implémenté une version classique du PSO, en testant plusieurs méthodes d'initialisation des particules : aléatoire, SPT (Shortest Processing Time), et une approche hybride combinant les deux. Cela nous a permis d'analyser l'impact de la qualité initiale des solutions sur les performances de l'algorithme.

Ensuite, pour améliorer les résultats obtenus, nous avons proposé une version enrichie du PSO. Cette version introduit une phase de **clustering** lors de l'initialisation, afin de mieux répartir les particules dans l'espace de recherche. De plus, nous avons ajouté des **mutations** déclenchées en cas de stagnation pendant l'exécution, pour favoriser l'exploration et éviter que l'algorithme ne reste bloqué dans une solution locale.

Les résultats obtenus ont montré que le PSO, notamment dans sa version améliorée, est capable de fournir de bonnes solutions au JSSP, tout en étant rapide et adaptable.

Pour aller plus loin, il serait intéressant de combiner cette approche avec d'autres techniques d'optimisation ou d'explorer des stratégies plus avancées d'exploration locale. Tester cette méthode sur des instances plus grandes permettrait aussi de mieux évaluer sa robustesse face à des problèmes de taille industrielle.

Chapter 4

Annexe

4.1 Pseudo-code de l'algorithme PSO classique

Algorithm 3 Particle Swarm Optimization (PSO)

Initialize swarm of N particles with random positions and velocities
Set parameters: inertia weight w, acceleration coefficients c_1, c_2
for each particle i in the swarm **do**
    Evaluate fitness of particle: $f(x_i)$
    Set personal best: $pbest_i = x_i$
end for
Set global best: $gbest = \arg\min f(pbest_i)$
while stopping criterion not met **do**
    for each particle i **do**
        Generate random numbers $r_1, r_2 \in [0, 1]$
        Update velocity:

$$v_i = w \cdot v_i + c_1 \cdot r_1 \cdot (pbest_i - x_i) + c_2 \cdot r_2 \cdot (gbest - x_i)$$

        Update position:

$$x_i = x_i + v_i$$

        Evaluate fitness: $f(x_i)$
        if $f(x_i) < f(pbest_i)$ **then**
            Update personal best: $pbest_i = x_i$
        end if
        if $f(pbest_i) < f(gbest)$ **then**
            Update global best: $gbest = pbest_i$
        end if
    end for
end while
Return global best solution $gbest$

4.2 Pseudo-code du solveur DFS avec Branch and Bound pour le JSSP

Algorithm 4 DFS et Branch and Bound

Attributs :
  `jssp` : Instance du problème JSSP
  `best_schedule` : Meilleur planning trouvé
  `best_makespan` : Meilleur makespan trouvé
  `nodes_explored` : Nombre de nœuds explorés

Méthode `solve()`
  Initialiser le planning avec `jssp.initialize_schedule()`
  Appeler `DFS(jobs_restants, planning_vide, makespan = 0)`
  Retourner `best_schedule` et `best_makespan`

Méthode `lower_bound(remaining_jobs, current_makespan)`
  Calculer la borne inférieure comme étant :
    max(`current_makespan`, temps total restant du job le plus long)
  Retourner cette borne

Méthode `DFS(remaining_jobs, current_schedule, current_makespan)`
  Incrémenter `nodes_explored`
  Estimer la borne inférieure : `estimated_lb = lower_bound(...)`
  Si `estimated_lb` $\geq$ `best_makespan` **alors** Retourner (élagage)
  Si `remaining_jobs` est vide :
    Si `current_makespan` $<$ `best_makespan`, mettre à jour `best_makespan` et `best_schedule`
    Retourner
  Trier `remaining_jobs` par charge restante décroissante
  Pour chaque job **dans** `remaining_jobs` :
    Si le job est terminé, continuer
    Calculer `start_time = max(fin_op_précédente, dispo_machine)`
    Calculer `end_time = start_time + durée_opération`
    Si `end_time` $\geq$ `best_makespan`, continuer (élagage)
    Planifier l'opération sur la machine
    Avancer à l'opération suivante du job
    Appeler récursivement `DFS(...)`
    Backtrack : annuler la planification de l'opération et restaurer l'état précédent

Bibliography

[1] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, "Scheduling in computer and manufacturing systems," *European Journal of Operational Research*, vol. 127, no. 3, pp. 490–510, 2000.

[2] J. Kennedy and R. Eberhart, "Particle swarm optimization," *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948, 1995.

[3] M. Clerc, *Particle Swarm Optimization*. ISTE, 2006.

[4] R. Eberhart and Y. Shi, "Comparison between genetic algorithms and particle swarm optimization," *International Conference on Evolutionary Programming*, pp. 611–616, 1998.