

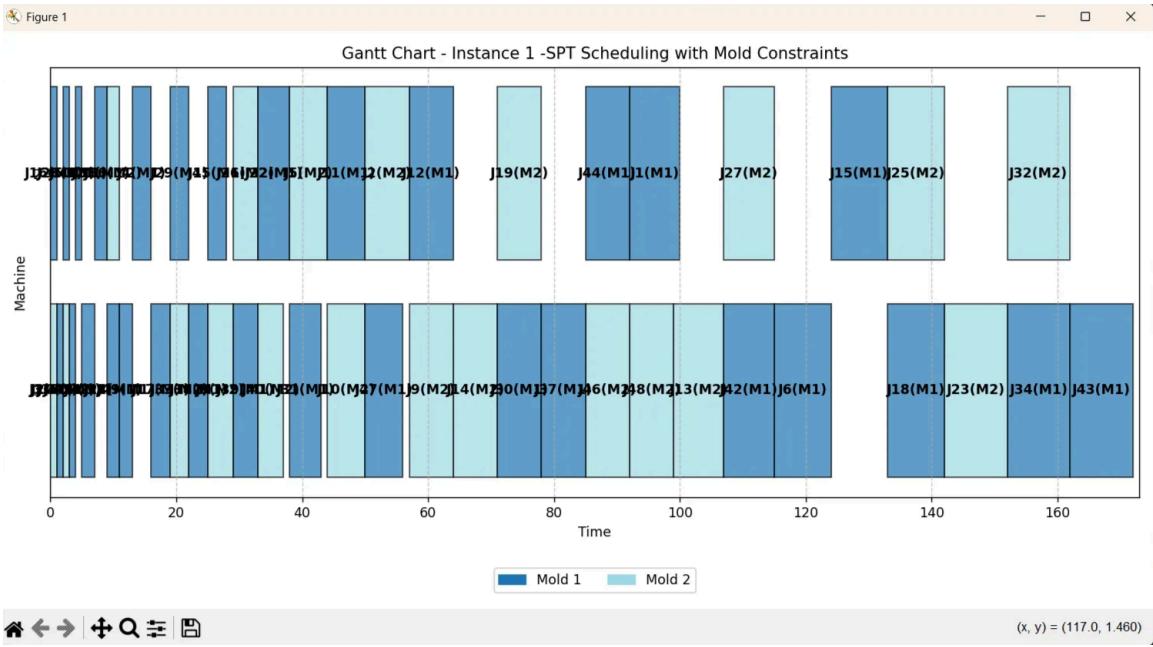


# Rapport : Problème d'optimisation combinatoire à deux machines parallèles identiques avec contraintes de moules.

## Heuristiques :

SPT :

- ⇒ On choisit toujours le job le plus court à traiter en premier.
- ⇒ Parmi les jobs les plus courts, on sélectionne celui dont le moule est disponible le plus tôt.
- ⇒ Elle répète cela jusqu'à ce que tous les jobs soient placés, en essayant de réduire le temps total (makespan).



### LPT :

- ⇒ Cette heuristique traite d'abord les jobs les plus longs, pour essayer de réduire le temps total.
- ⇒ Si deux jobs ont la même durée, elle privilégié celui dont le moule est déjà disponible.
- ⇒ Si les deux moules sont indisponibles, elle choisit celui dont la libération est la plus proche.

### Glouton:

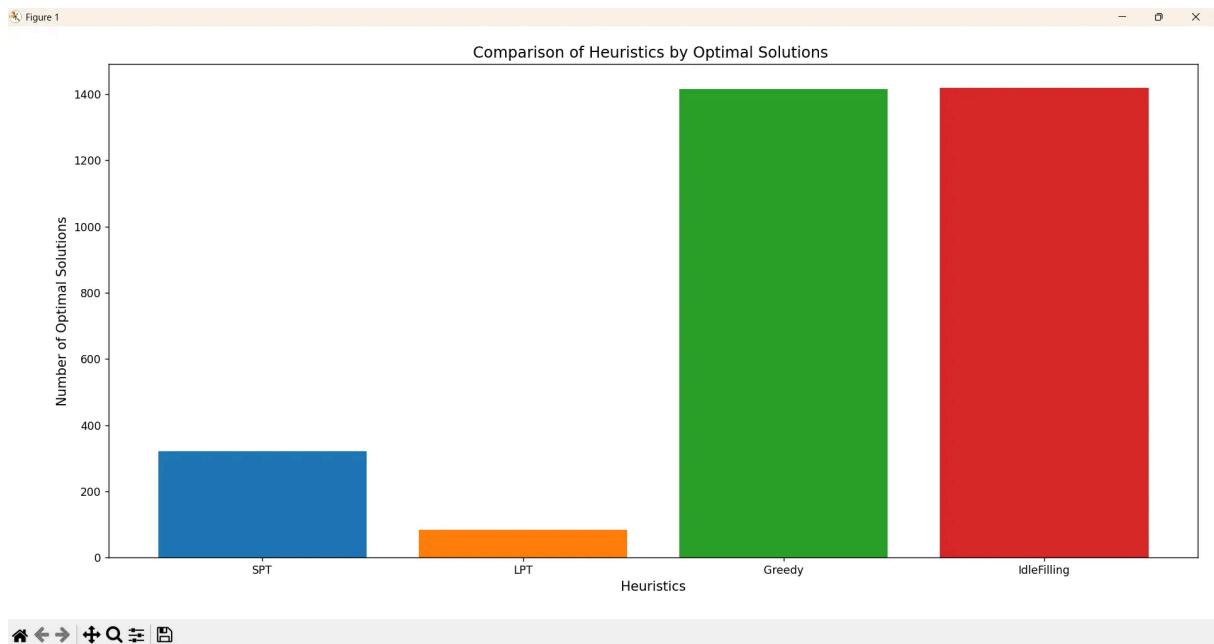
- ⇒ Cette heuristique choisit le job qui finira le plus vite possible.
- ⇒ Elle vérifie chaque job sur chaque machine pour trouver celui qui peut être terminé le plus rapidement.
- ⇒ Si les deux jobs ont exactement le même makespan, il prendra le job qui est en premier dans la liste des jobs restants.

### IdleFillingHeuristic:

- ⇒ Dès que la machine est disponible, on cherche le job qui peut commencer immédiatement (celui dont le moule est disponible).
- ⇒ Si deux jobs peuvent commencer en même temps, elle choisit celui qui se termine le plus tôt.

## La recherche de la meilleure heuristique :

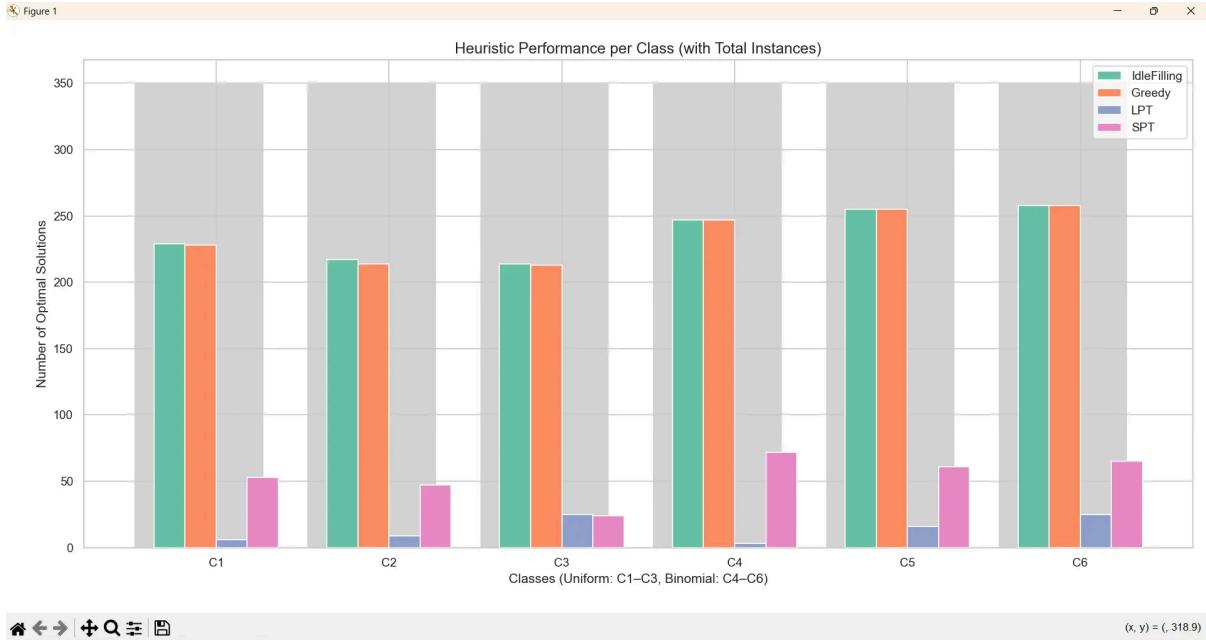
Nombre de solutions optimales par heuristique :



⇒ La performance globale de IdleFillingHeuristic est 66.7% .

⇒ **La méthode Greedy (1400) et Idle Filling (1417) montrent une bonne performance, atteignant environ 66,7% des instances avec des solutions optimales. Cela suggère que ces heuristiques sont efficaces pour les instances les plus courantes, et peuvent s'adapter à des distributions d'exécution variées.**

En revanche, les heuristiques **LPT (73)** et **SPT (319)** ont des résultats significativement plus faibles, particulièrement **LPT**, ce qui pourrait être dû à sa stratégie sous-optimale pour certaines instances.



⇒ L'analyse comparative des quatre métahéuristiques appliquées au problème d'optimisation à deux machines parallèles avec contraintes de moules révèle que les heuristiques IdleFilling et Greedy surpassent systématiquement les approches classiques LPT et SPT en termes de nombre de solutions optimales trouvées.

⇒ Cette supériorité peut s'expliquer par leur capacité à mieux gérer les temps morts et à équilibrer efficacement la charge entre les deux machines tout en respectant les contraintes de moules. Les performances varient selon les classes de problèmes, avec une tendance à l'amélioration pour IdleFilling dans les instances plus complexes, suggérant une meilleure adaptabilité aux contraintes spécifiques du problème étudié.

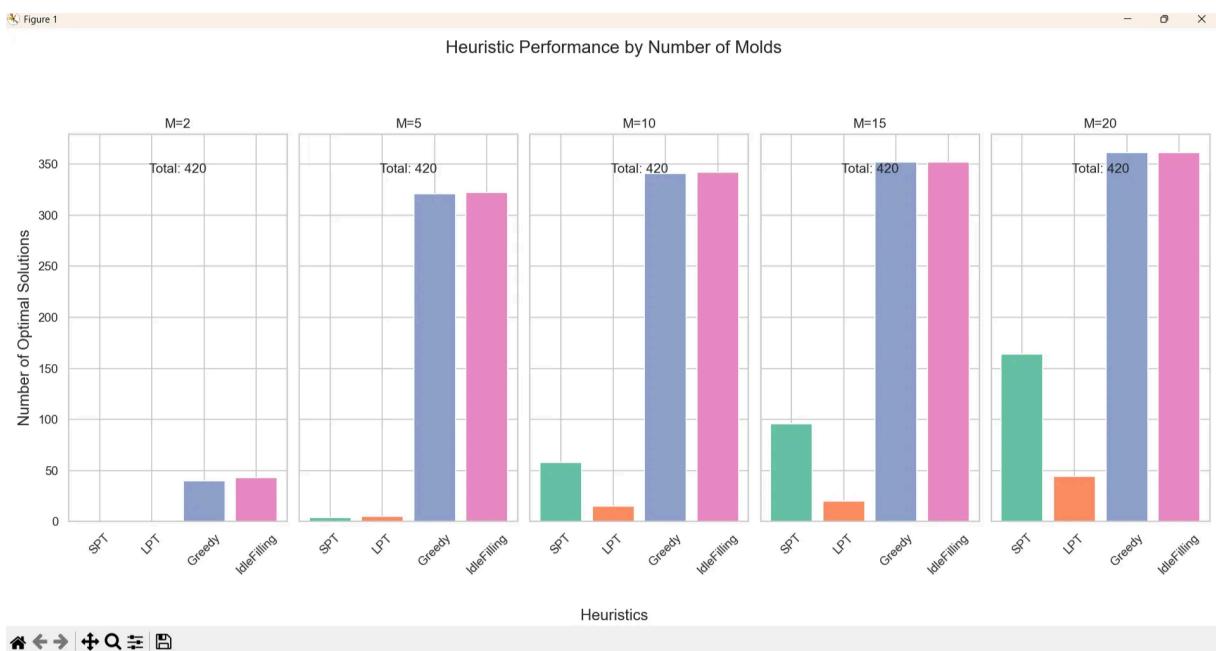
⇒ Ces résultats confirment l'importance d'adapter les heuristiques aux particularités du problème .

⇒ **Comme IdleFillingHeuristic a le nombre des solutions optimales le plus élevé pour des différentes distributions ce qui montre sa robustesse donc on va l'utiliser comme solution de départ dans les heuristiques suivantes ( recherche taboue ,recuit simulé ..).**

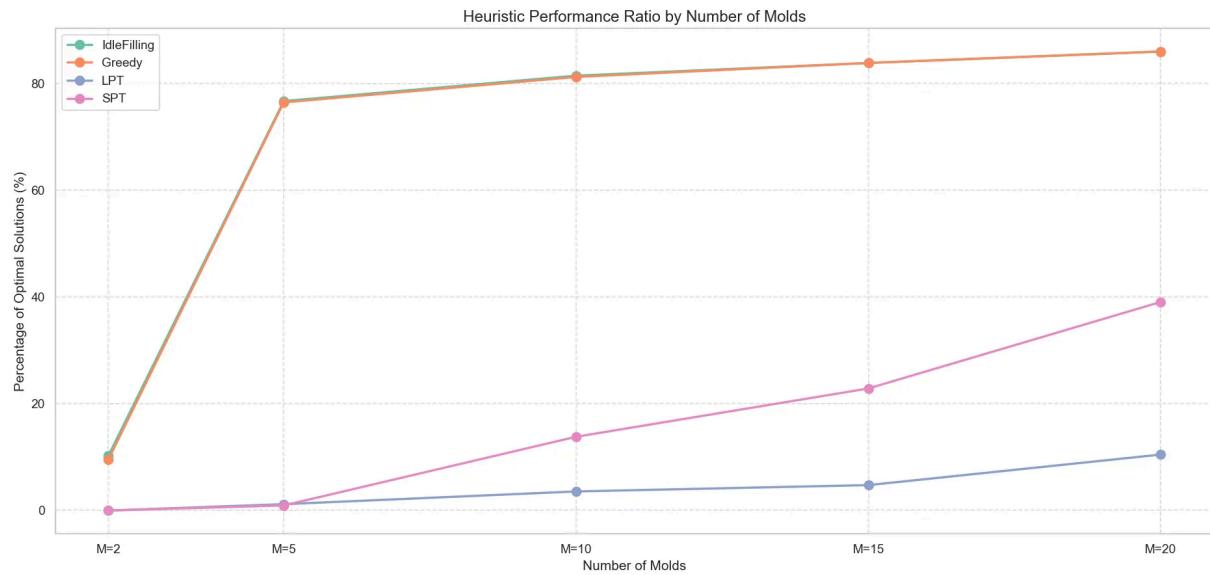
⇒ La mauvaise performance de LPT s'explique par le fait que celui-ci LPT ignore les contraintes de moules c'est à dire la machine reste inactive en attendant que la tache la plus longue prend fin.

⇒ Lorsque la classe est suivie d'une loi uniforme la probabilité d'avoir une des valeurs de makespan est équilibrée donc . Alors que dans la loi uniforme les durées tournent autour de la moyenne donc presque toutes les jobs sont pareils . Cela évite les déséquilibres entre les machines (ex : une machine qui travaille longtemps pendant que l'autre attend (une exécute une tache dont la durée 1 et attend que l'autre finit l'exécution d'une tache dont la durée est 10 parce qu'elle a besoin de moule par exemple.) .

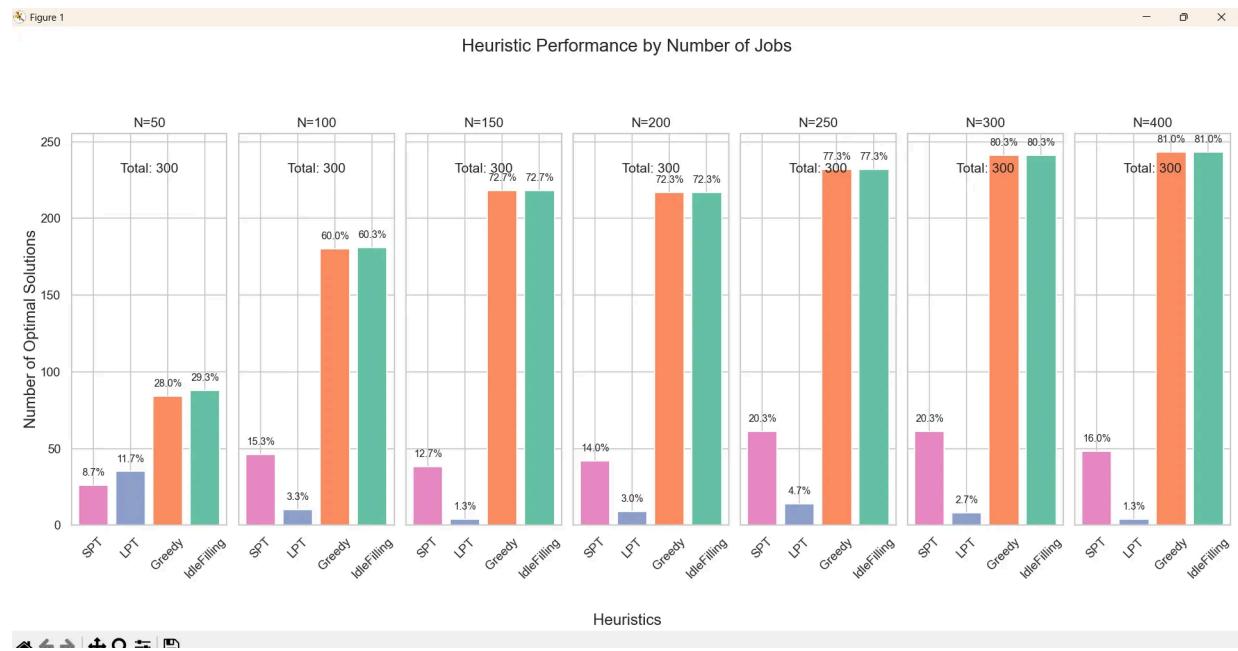
## Heuristics by number of molds :



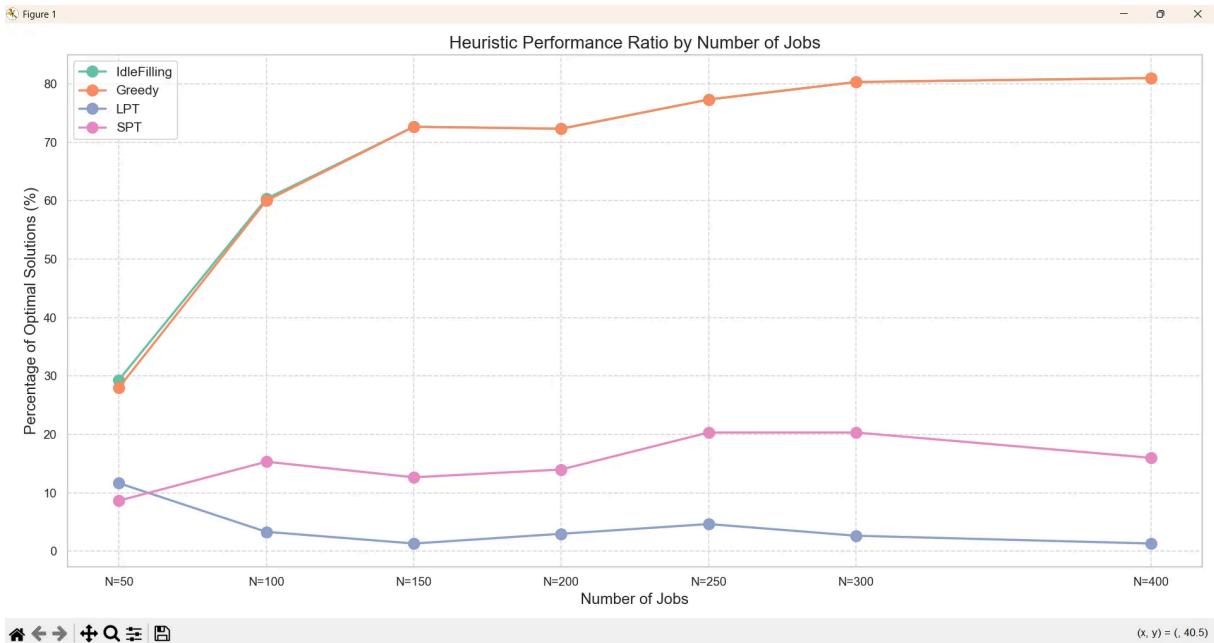
⇒ Plus il y a de moules, moins il y a de conflits car on peut faire fonctionner plus de tâches à la fois car on réduit le temps d'inactivité des machines.



## Heuristics by number of jobs :



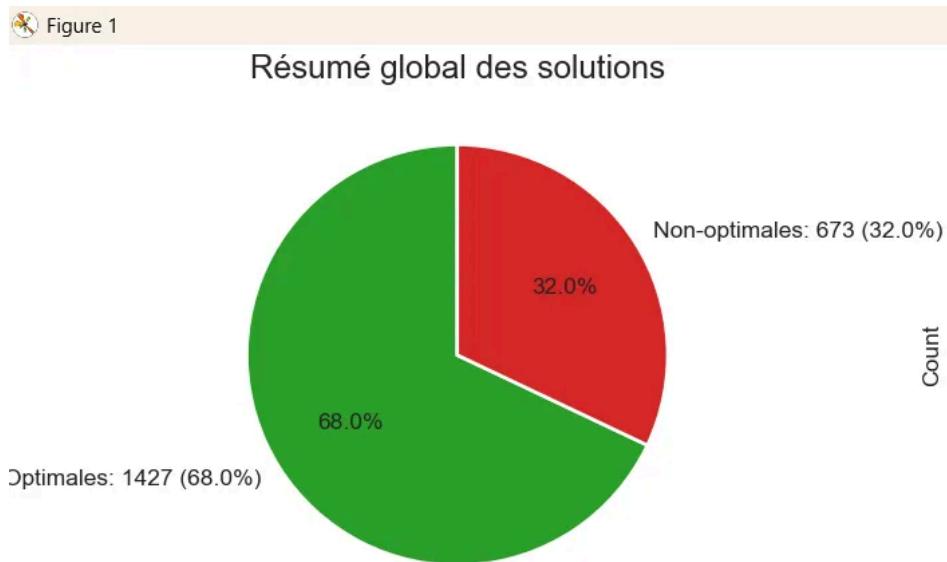
⇒ Les heuristiques choisies sont plus performantes dans les cas où le nombre de jobs est grand .

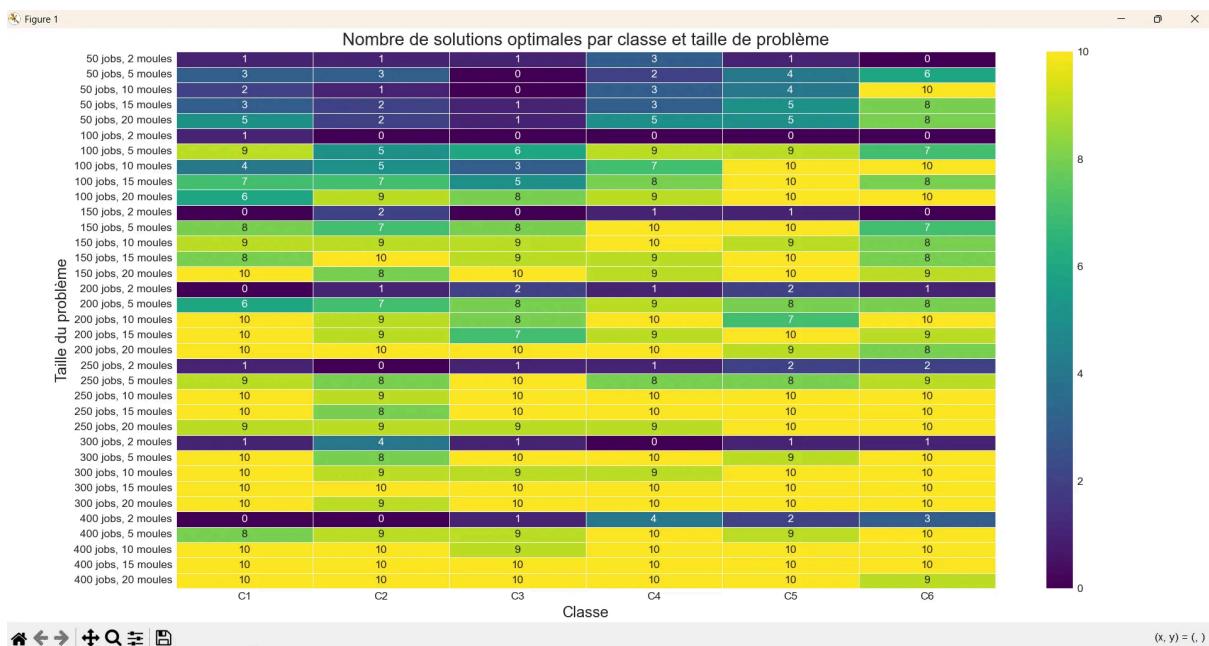


## Métaheuristiques :

**Solution initiale : La solution donnée par la meilleure heuristique .**

## Recherche taboue :





## Paramètres :

**Taille de la liste taboue (tenure taboue) : 10**

**Nb max d'itérations : 100**

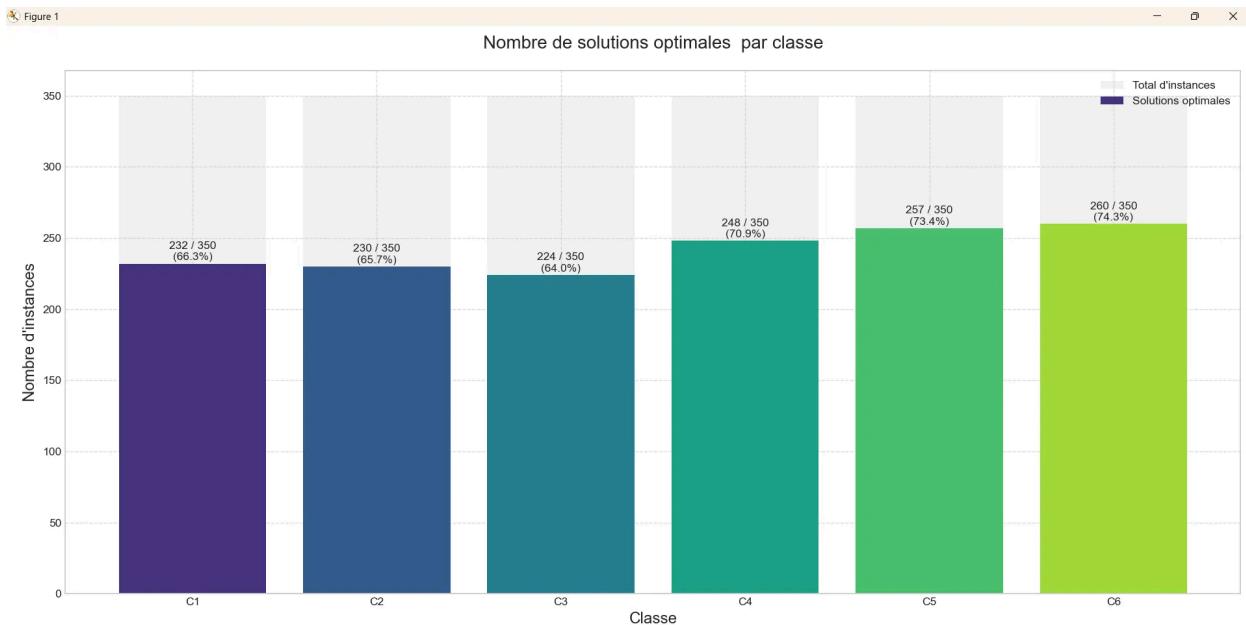
**Sans amélioration : 50 ou 10 selon cas ( grid search nous a donné le nombre**

optimal des itérations sans amélioration selon la classe de l'instance ).

### Voisins générés : 10

**Critère d'arrêt :** On s'arrête soit quand on atteint le nombre maximal d'itérations, soit quand aucune amélioration n'est trouvée pendant un certain temps

## Recuit Simulé :



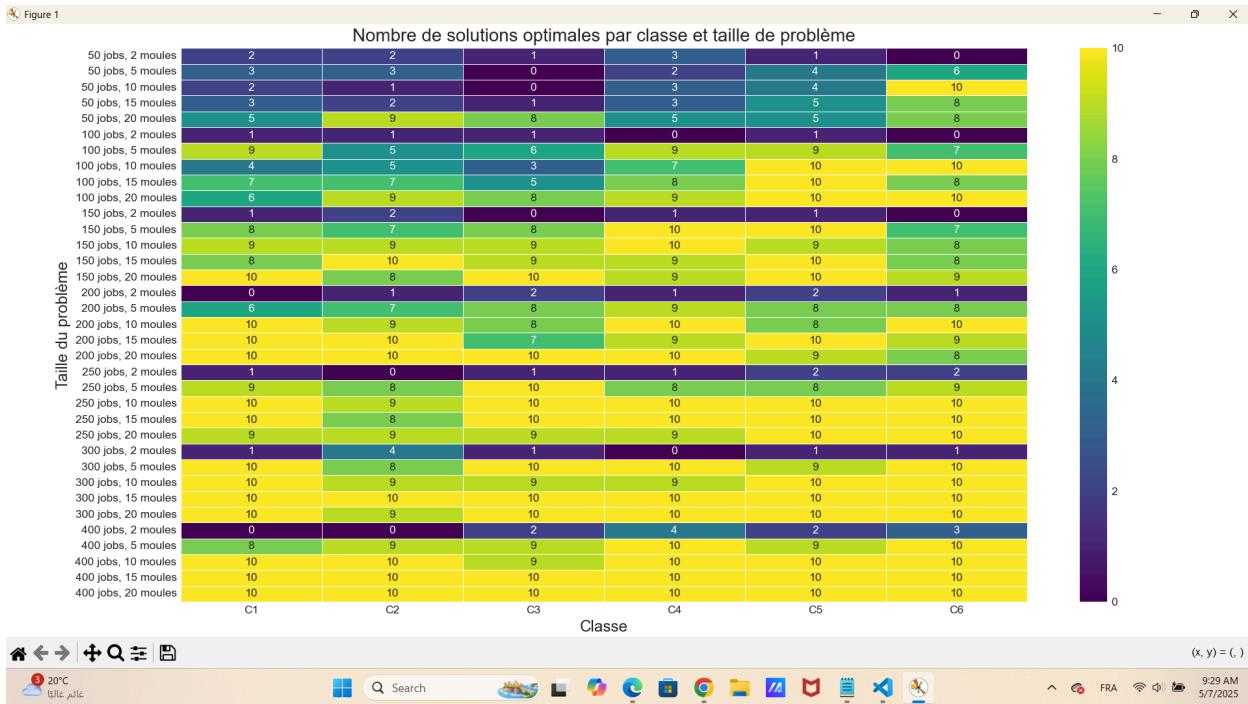
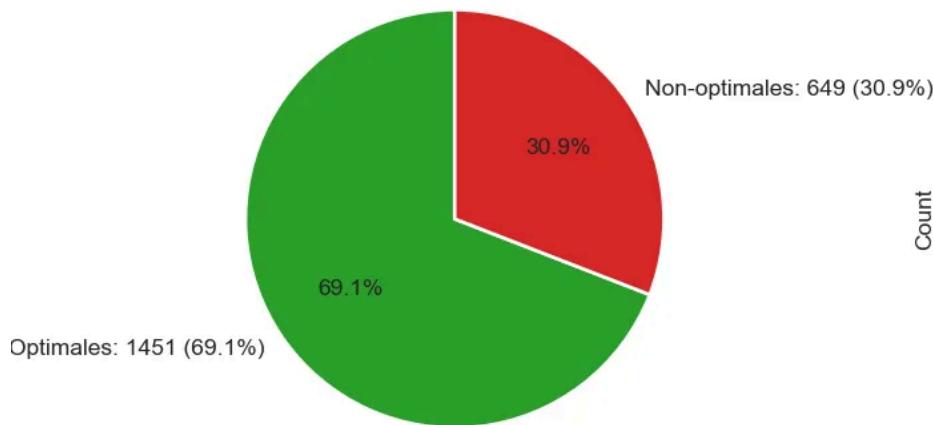


Figure 1

### Résumé global des solutions



### Paramètres :

**Température initiale (initial\_temp)** : 500 à 2000 selon la classe de l'instance et selon la valeur donnée par grid search.

**Taux de refroidissement (cooling\_rate)** : entre 0.9 et 0.97 selon les cas (ralentit

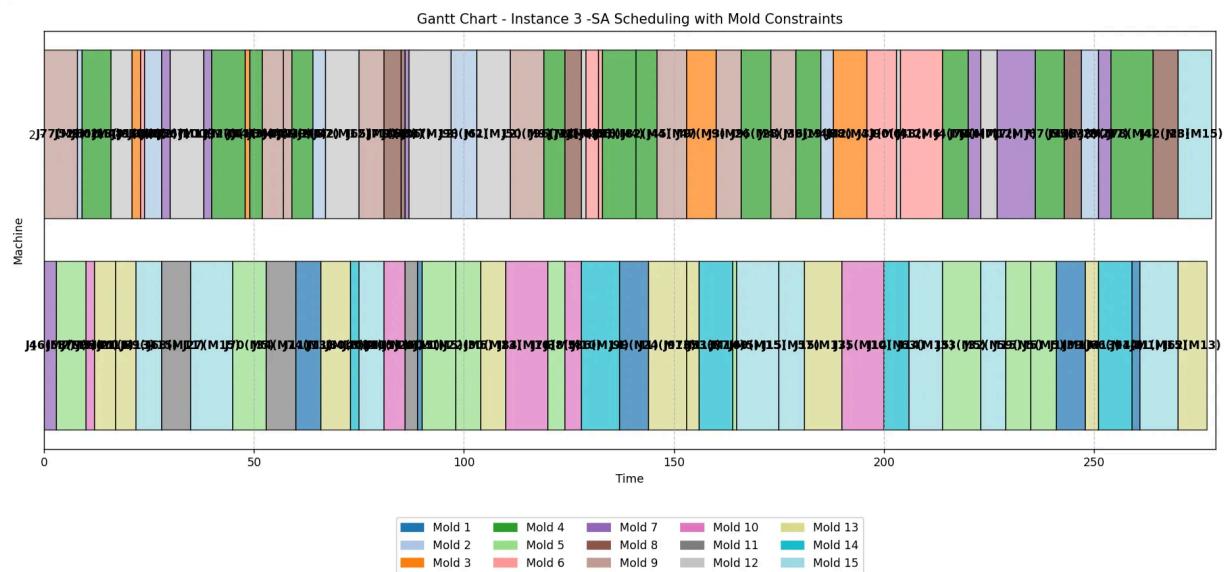
ou accélère la baisse de température).

**Nombre d'itérations par température (max\_iterations) :** 100 à 700 selon instance (définit combien de voisins sont testés avant de refroidir).

### Critère d'arrêt :

La température diminue

**seulement après avoir exploré 100 voisins**. Pendant ces 100 itérations, la température reste constante. Ensuite, on applique  $\text{temp}^* = \text{temp} * 0.95$ .



### Class-specific hyperparameter tuning :

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\nadaa\OneDrive\Desktop\Nada\ING1inf\Semester2\OC\OC_Project - Copy> & C:/Users/nadaa/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/nadaa/OneDrive/Desktop/Nada/ING1inf/Semester2/OC/OC_Project - Copy/NEWFILE.py"
Set up grid search with 192 parameter combinations
Running grid search with 192 combinations, 3 trials each
Running in parallel with 8 processes
Simulated Annealing: 31% | 52/166 [00:06<00:13, 8.28it/s]

```

## Génération de Voisins :

Stratégies :

- ⇒ Échange entre deux jobs sur des machines différentes (swap\_jobs)
- ⇒ Déplacement d'un job vers une autre machine (move\_job)
- ⇒ Échange de deux jobs adjacents sur une même machine (swap\_adjacent\_jobs)
- ⇒ Inversion d'une sous-séquence de jobs sur une même machine (subtour\_reversal)
- ⇒ La validité de chaque voisin généré est vérifiée et le filtrage des doublons est activé.
- ⇒ On a diversifier les stratégies de génération de voisins car lorsqu'on a fait un swap seulement on a été bloqué seulement dans un optimum local d'où l'ajout de plusieurs stratégies.

La qualité de voisins avant et après les stratégies de diversifications :

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Makespan Frequencies Across All Neighbors:
Makespan 224 : 2 neighbors
Makespan 226 : 3 neighbors
Makespan 227 : 2 neighbors
Makespan 232 : 1 neighbors
Makespan 233 : 4 neighbors
Makespan 234 : 2 neighbors
Makespan 236 : 4 neighbors
Makespan 237 : 4 neighbors
Makespan 238 : 11 neighbors
Makespan 239 : 2 neighbors
Makespan 240 : 4 neighbors
Makespan 241 : 5 neighbors
Makespan 242 : 4 neighbors
Makespan 243 : 19 neighbors
Makespan 244 : 11 neighbors
Makespan 245 : 18 neighbors
Makespan 246 : 33 neighbors
Makespan 247 : 31 neighbors
Makespan 248 : 101 neighbors
Makespan 249 : 62 neighbors
Makespan 250 : 39 neighbors
Makespan 251 : 51 neighbors
Makespan 252 : 68 neighbors
Makespan 253 : 168 neighbors
Makespan 254 : 164 neighbors
Makespan 255 : 261 neighbors
Makespan 256 : 497 neighbors
Makespan 257 : 307 neighbors
Makespan 258 : 98 neighbors
Makespan 259 : 344 neighbors
Makespan 260 : 318 neighbors
Makespan 261 : 245 neighbors
Makespan 262 : 250 neighbors

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Makespan 503 : 113 neighbors
Makespan 504 : 73 neighbors
Makespan 505 : 163 neighbors
Makespan 506 : 8 neighbors
Makespan 507 : 17 neighbors
Makespan 508 : 277 neighbors
Makespan 509 : 74 neighbors
Makespan 510 : 127 neighbors
Makespan 511 : 1255 neighbors
Evaluating parameter combinations: 100% | 576/576 [04:17<00:00, 2.23it/s]

💡 Best Parameter Combination:
- initial_temp: 500.0
- cooling_rate: 0.97
- max_iterations: 300.0
- min_temp: 1.0

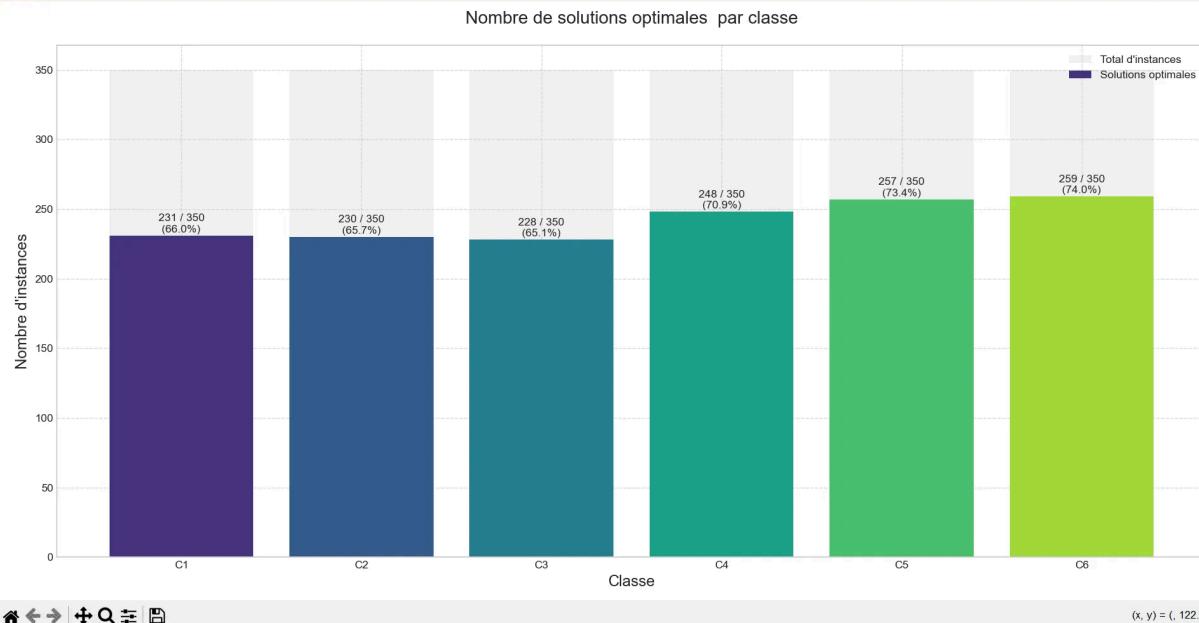
Performance Metrics:
- Mean Makespan: 256.00
✓ Saved best parameters for instance #1 to tuning_results/instance_1/best_sa_parameters.json
PS C:\Users\nadaa\OneDrive\Desktop\Nada\INGIinf\Semester2\OC\OC_Project - Copy> []
Ln 59, Col 59 Spaces: 4 UTF-8 CRLF {} Plain Text ⌂ ⌂ ⌂

```

## Algorithmes génétiques :

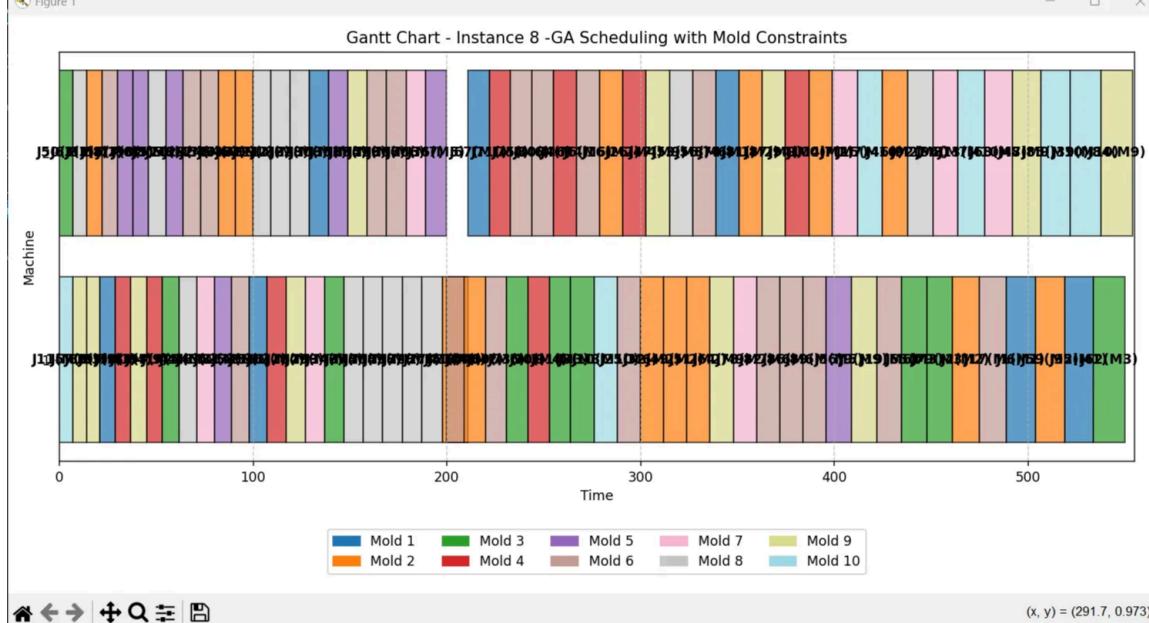
On a considéré que si le gap (solution, lb)  $\leq 1\%$  , celle-là est considérée optimale.

Figure 1

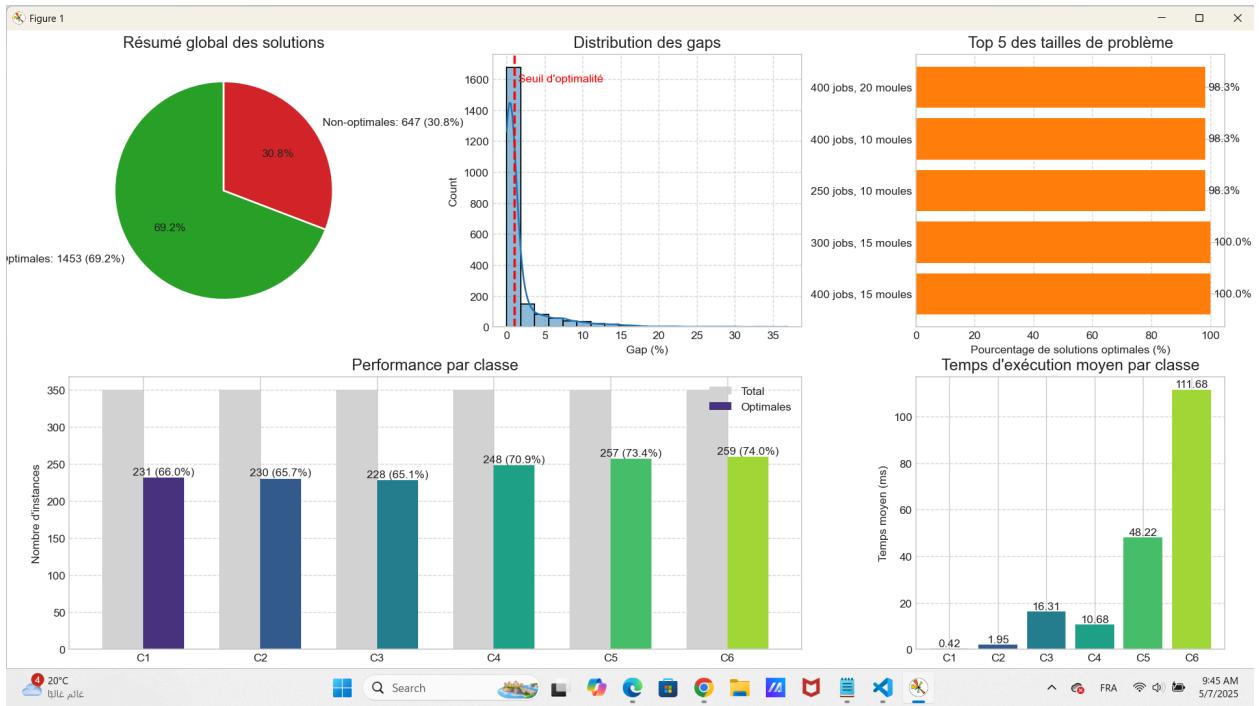

Home Back Forward Search Print Close

(x, y) = (, 122.8)

Figure 1


Home Back Forward Search Print Close

(x, y) = (291.7, 0.973)



### --Population initiale :

50% Recherche taboue .

50% Solutions aléatoires pour améliorer la diversité .

La taille de la population est 10.

On a choisi la formation de 50% de la population initiale par recherche taboue car il a eu un temps d'exécution minimal

<b>Temps d'exécution</b>	
<b>Recherche taboue</b>	vers 4 heures
<b>Recuit simulé</b>	vers 3 jours

### --Méthode de sélection :

La sélection se fait par roulette, où les individus avec une meilleure aptitude ont plus de chances d'être choisis. Cela favorise les bonnes solutions tout en maintenant la diversité.

### Fonction d'adaptation :

**1/Makespan : la probabilité d'être choisi est plus grande plus que le makespan est petit.**

r (entre 0 et total fitness).

**---Critère d'arrêt:**

1. lorsqu'il atteint un nombre maximal de générations (5).

a. une limite de temps (60 secondes)

Croisement :

- croisement à 2 points & on vérifie la validité des contraintes après la génération de chaque enfant.

Mutation :

- Meme stratégies dans la fonction de génération de voisins (swap, insertion).