CL02 - CONDITIONNEMENT, MANUTENTION ET ENTREPOSAGE

Rapport de projet

Simon Maillard- Matthieu Pardini – Arthur Pouillery – Victor Rivière

Automne

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

Table des matières

CL02 - CONDITIONNEMENT, MANUTENTION ET ENTREPOSAGE 1
I. Introduction 3
1. Contexte général 3
2. Historique 3
3. Innovations 5
II. Partie modélisation 6
1. Pose du problème 6
2. Présentation des paramètres et variables 7
3. Modèle mathématique de l'article 8
4. Modèle mathématique corrigé et adapté 9
5. Modèle mathématique adapté à CPLEX 10
III. Applications et résultats avec le solveur 11
IV. Partie algorithme 14
1. Présentation de la méthode du coucou 14
2. Présentation de notre code 15
3. Présentation et mode d'emploi de l'exécutable 22
V. Résultats et analyse des performances 25
VI. Conclusion 31
1. Apports du projet 31
2. Analyse critique de l'article et conclusion 31
VII. Bibliographie 32
VIII. Table des illustrations 33
IX. Annexes 34

I. Introduction

1. Contexte général

Nous avons étudié l'article scientifique « Résolution du problème d'équilibrage d'une chaîne de montage robotisé à modèle mixte économe en énergie à l'aide d'un algorithme de recherche de coucou basé sur la mémoire ». Il a été publié par Lakhdar Belkharroubi et Khadidja Yahyaoui en 2022.

Dans les lignes d'assemblages entièrement robotisées, la consommation d'énergie est un nouveau critère important à prendre un compte. La méthode proposée dans cet article a pour objectif de minimiser la consommation totale de la ligne d'assemblage en fonction des affectations des robots et tâches. Ce problème devient très vite compliqué car il faut prendre en compte tous les modèles que la ligne est capable de produire. Il n'existait aucune méthode traitant la consommation d'énergie dans des lignes d'assemblage robotisées qui produisent plusieurs modèles avec une seule configuration. De plus, l'introduction de l'hétérogénéité des modèles et des robots rend le problème plus complexe.

Par conséquent, les auteurs proposent dans cet article un algorithme de recherche du coucou basé sur la mémoire (MBCSA) pour résoudre ce problème. Le principe de la mémoire est utilisé dans ce nouvel algorithme de recherche afin d'échapper aux optimums locaux et de découvrir de nouvelles zones de recherche.

2. Historique

Dans une chaîne de montage, un ensemble de postes de travail sont reliés entre eux à l'aide d'un système capable de déplacer les produits d'un poste de travail à un autre, comme un tapis roulant. Pour obtenir le produit final, un ensemble d'opérations ou de tâches sont effectuées dans chaque poste de travail. Les tâches sont liées entre elles par des relations de précédences et associées à des temps.

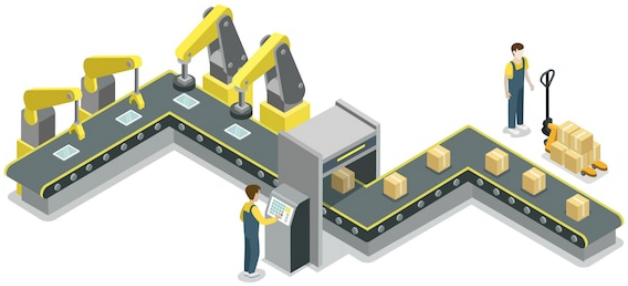

Figure 1 : Exemple ligne d'assemblage robotisée

Ce problème est connu comme le problème d'équilibrage de la ligne d'assemblage (ALBP) (Sivasankaran et Shahabudeen, 2014). Il existe deux versions de l'ALBP, la version simple (SALBP) et la version généralisée (GALBP). La différence entre ces deux versions est que certaines hypothèses du SALBP peuvent être négligées dans le GALBP. La figure 2, que nous avons créée, reprend la différence entre les deux versions et les modèles qui en découlent.

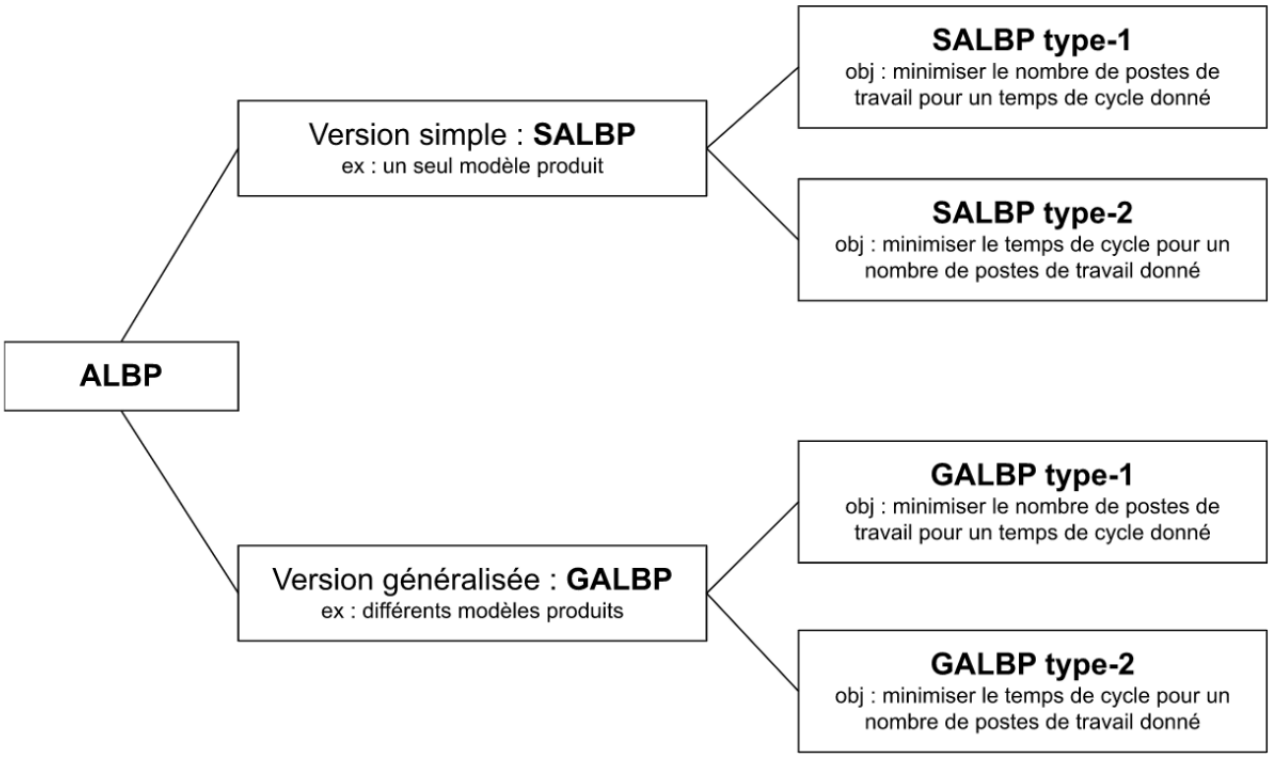

Figure 2 : Graphique des différents problèmes d'équilibrage

Lorsque nous abordons de nouveaux problèmes tels que le mélange de modèles, les contraintes de zonage et l'affectation des robots, l'ALBP devient plus compliqué et le temps de calcul augmente. De nos jours, les fabricants utilisent des lignes d'assemblage mixtes (MiMAL) pour répondre aux demandes des clients. Le problème d'équilibrage de la ligne d'assemblage à modèles mixtes (MiMALBP) est le problème d'affectation des tâches aux postes de travail dans le modèle MiMAL, qui prend en compte tous les modèles à produire sur la ligne. Les tâches d'assemblage peuvent être effectuées manuellement par des humains, automatiquement par des robots, ou en collaboration par des humains et des robots. La robotique est intégrée dans les lignes d'assemblage dans le but de créer des systèmes de fabrications intelligents (Industrie 4.0).

Ainsi, nous en arrivons à la version connue sous le nom de problème d'équilibrage de la ligne d'assemblage à modèle mixte robotique (RMiMALBP). Le RMiMALBP cherche la meilleure affectation des tâches et des robots à un ensemble de postes de travail afin de minimiser ou de maximiser divers critères tels que le temps de cycle, le nombre de postes de travail et l'efficacité de la ligne. Les auteurs de l'article cherchent eux à minimiser la consommation en énergie des robots en cherchant les meilleures affectations.

Il existe certains travaux visant à minimiser la consommation d'énergie dans les lignes d'assemblage robotisées. Cependant, les différences sont déterminées par les caractéristiques du problème. Nous pouvons citer comme exemple la forme de la ligne (droite, en forme de U, parallèle et ligne à deux côtés) ou les ressources à optimiser (nombre de postes de travail, temps de cycle et énergie consommée).

3. Innovations

Le problème de minimisation de l'énergie consommée dans une ligne d'assemblage mixte peut être divisé en deux étapes :

- Recherche de la meilleure affectation des tâches d'assemblage dans les postes de travail
- Trouver la meilleure affectation des robots avec la consommation d'énergie la plus faible.

Cela correspond aux variables de décision du modèle mathématique. Lors de la résolution de ces deux sous-problèmes, tous les modèles du produit doivent être pris en considération afin de trouver une configuration qui satisfait aux exigences de tous les modèles de produits. Il s'agit d'un problème très complexe qui n'a jamais été résolu. Tous les travaux existants sur la minimisation de la consommation d'énergie se concentrent uniquement sur un seul modèle produit dans les lignes d'assemblage robotisées.

En guise de solution à ce problème, les auteurs proposent dans l'article une nouvelle version de l'algorithme de recherche du coucou basée sur le concept de mémoire : la métaheuristique MBCSA (Memory-Based Cuckoo Search Algorithm). Elle est inspirée de la biologie et du comportement de certaines espèces de coucous. Ils y ajoutent un système de mémorisation pour échapper aux optimums locaux et découvrir de nouvelles zones de recherche. Nous verrons que certains problèmes de « petites tailles » peuvent être résolus rapidement par un solveur classique comme CPLEX. Cependant, dès que le problème commence à devenir complexe, le temps de calcul du solveur devient trop grand et l'utilisation de l'heuristique devient obligatoire.

II. Partie modélisation

1. Pose du problème

Tout d'abord, chaque modèle du produit a son propre diagramme de précédence. Par exemple, si nous avons deux modèles A et B, il faut les combiner en un seul et unique modèle C. Ci-dessous un schéma extrait de l'article permettant de visualiser le problème :

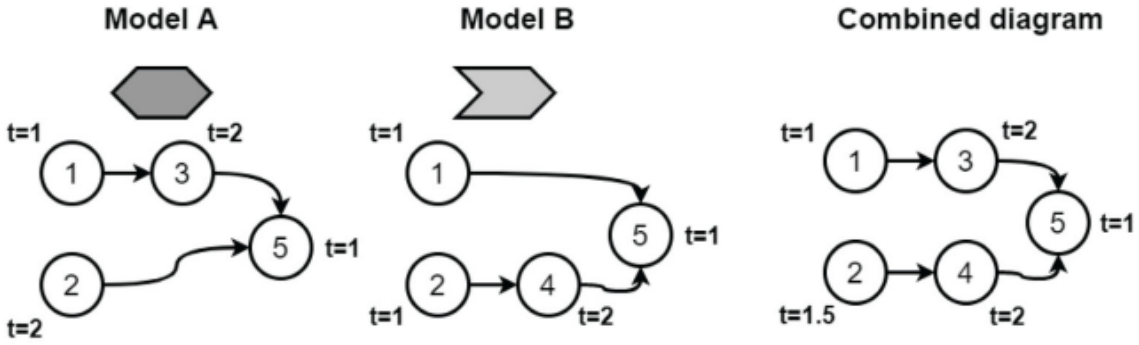

Figure 3 : Diagrammes des prédécesseurs

Nous avons ensuite un ensemble de stations (une station = un robot) et un ensemble de tâches. Les tâches doivent être affectées aux différentes stations en fonction du diagramme combiné de précédence. Le robot avec la plus basse consommation d'énergie selon les tâches de la station lui sera affecté. Chaque tâche a un temps de traitement qui varie selon les modèles. Par conséquent, la consommation d'énergie d'un robot va aussi varier selon les modèles.

L'énergie totale consommée par une station est égale à l'énergie que le robot utilise pour réaliser les tâches qui lui sont assignées additionnées à l'énergie qu'il consomme durant le temps où il n'est pas utilisé (standby). De plus, chaque type de robot possède des caractéristiques différentes avec pour une tâche spécifique un temps de process et une quantité d'énergie consommée. Pour chaque séquence de tâches obtenue, la meilleure affectation de robots avec une consommation d'énergie minimale doit être trouvée, en prenant en compte tous les modèles.

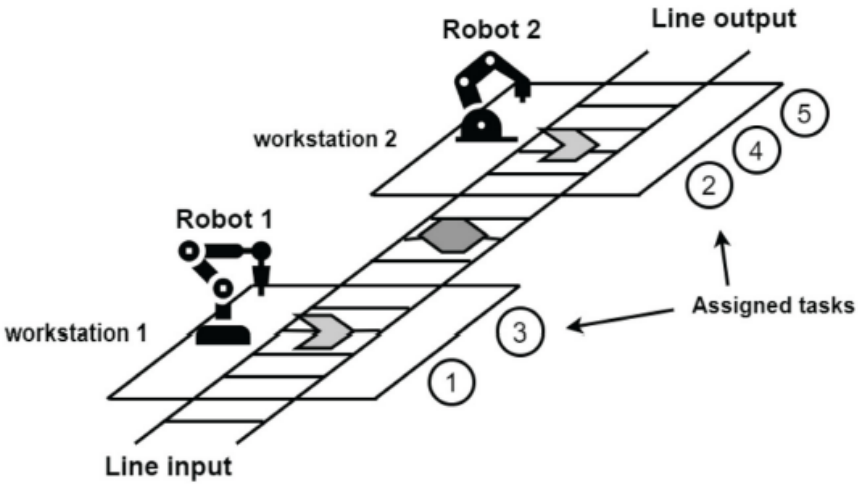

Figure 4 : Schéma ligne d'assemblage robotisée

Dans l'exemple ci-dessus, la ligne d'assemblage est conçue pour assembler deux modèles différents et deux robots différents sont affectés à deux stations pour effectuer un ensemble de tâches en fonction de la combinaison des deux modèles. Cependant, nous ne savons pas si cette configuration est la meilleure en termes d'énergie consommée.

Les hypothèses suivantes sont à prendre en compte pour le modèle :

- La ligne est droite et plusieurs modèles sont assemblés dans une séquence mélangée
- Une tâche ne peut pas être attribuée à différents postes de travail
- L'affectation des tâches est contrainte par des relations de précédence
- La durée de la tâche dépend du robot assigné
- Il existe plusieurs types de robots et il n'y a pas de limite pour chaque type de robot
- Le même type de robot peut être affecté à différents postes de travail
- Un seul robot peut être affecté à un poste de travail
- Les tâches peuvent être effectuées par n'importe quel type de robot
- L'énergie consommée par un robot varie d'un modèle à l'autre
- En se basant sur l'énergie consommée par les robots pour produire chaque modèle, l'énergie moyenne consommée peut être calculée et utilisée pour l'optimisation de la ligne

2. Présentation des paramètres et variables

Cette sous-partie présente les variables données dans l'article. Cependant, nous verrons dans la suite de ce rapport que certaines variables ne seront pas utilisées.

Paramètres :

- N : Nombre de tâches
- i : Indice des tâches
- W : Nombre de stations
- k : Indice des stations
- r : Indice des robots
- t_{ri} : Temps de process du robot r pour accomplir la tâche i
- e_{ri} : Energie consommée par le robot r pour accomplir la tâche i
- e_{sr} : Energie consommée par le robot r par unité de temps durant la période de standby
- P_i : Prédécesseurs de la tâche i
- ECP_{rk} : Energie consommée par le robot r pour réaliser les tâches assignées à la station k
- ECS_{rk} : Energie consommée par le robot r durant la période de standby de la station k

Variables de décisions :

- $X_{ik} = 1$ si la tache i est assignée à la station k

  0 sinon

- $Y_{rk} = 1$ si le robot r est assigné à la station k

  0 sinon

- TEC = Energie totale consommée
- EC_k = Energie consommée par la station k

- CT = Temps de cycle

3. Modèle mathématique de l'article

$$min \quad TEC = \sum_{k=1}^{W} EC_k \tag{1}$$

Subject to:

$$EC_K = ECP_{rk} + ECS_{rk} \tag{2}$$

$$ECP_{rk} = \sum_{i=1}^{N} e_{ri} * X_{ik} \leq for \quad k = 1, 2, \dots, W \tag{3}$$

$$ECS_{rk} = (CT - \sum_{i=1}^{N} t_{ri} * X_{ik}) * es_r \tag{4}$$

$$\sum_{k=1}^{W} k * X_{hk} \leq \sum_{k=1}^{W} k * X_{ik} \quad whereh \in P_i \tag{5}$$

$$\sum_{k=1}^{W} X_{ik} = 1 \quad for \quad i = 1, 2, \dots, N \tag{6}$$

$$\sum_{k=1}^{W} Y_{rk} = 1 \quad for \quad i = 1, 2, \dots, N \tag{7}$$

$$X_{ik} \in \{0, 1\} \tag{8}$$

$$Y_{ik} \in \{0, 1\} \tag{9}$$

Figure 5 : Modèle mathématique des auteurs

Le modèle proposé ne nous convient pas car il comporte des erreurs. La contrainte 2 n'est pas cohérente au niveau des dimensions (k d'un côté, rk de l'autre). Pour la contrainte 5, il faut modifier la formulation avec une matrice de précédence pour facilement la mettre en œuvre. La contrainte 7 n'est pas bien écrite, il s'agit d'une somme sur r et pour tout $k = 1,2 \dots N$. Enfin, le calcul du temps de cycle CT n'est pas précisé. Le modèle mathématique que nous utilisons est disponible dans la sous-partie suivante.

4. Modèle mathématique corrigé et adapté

$$Min \sum_{k=1}^{W} \sum_{r=1}^{R} Y_{r,k} \cdot \left(\left(\sum_{i=1}^{N} e_{r,i} \cdot X_{i,k} \right) + \left(CT - \sum_{i=1}^{N} t_{r,i} \cdot X_{i,k} \right) \cdot es_r \right)$$

$$\sum_{k=1}^{W} k \cdot X_{h,k} \cdot P_{i,h} \leq \sum_{k=1}^{W} k \cdot X_{i,k} \quad \forall i = 1..N, \forall h = 1..N \quad (1)$$

$$\sum_{k=1}^{W} X_{i,k} = 1 \quad \forall i = 1..N \quad (2)$$

$$\sum_{r=1}^{R} Y_{r,k} = 1 \quad \forall k = 1..W \quad (3)$$

$$CT = \max_{k=1..W} \left(\sum_{i=1}^{N} \sum_{r=1}^{R} X_{i,k} \cdot Y_{r,k} \cdot t_{r,i} \right) \quad (4)$$

$$X_{i,k} \in \{0,1\} \quad \forall i = 1..N, \forall k = 1..W \quad (5)$$

$$Y_{r,k} \in \{0,1\} \quad \forall r = 1..R, \forall k = 1..W \quad (6)$$

Figure 6 : Modèle mathématique corrigé

Nous avons remplacé le paramètre P_i par une matrice $P_{i,h}$ avec $h = 1..N$. Cette étape est nécessaire car une tâche peut avoir plusieurs prédécesseurs.

La fonction-objectif correspond à la somme des énergies consommées par les robots pour réaliser leurs tâches et les énergies consommées par les stations durant la période de standby. Cette période est équivalente à la différence entre le temps de cycle et la somme des temps des tâches de la station.

Contrainte 1 :

Cette contrainte a pour but de forcer le respect des relations de précédence.

Contrainte 2 :

Une tâche ne peut être affectée qu'à une unique station.

Contrainte 3 :

Il ne peut y avoir qu'un seul robot par station.

Contrainte 4 :

Le temps de cycle est défini par la station dont la somme des temps de process des tâches est la plus grande.

Contrainte 5 et 6 :

Les variables X et Y sont des variables binaires.

5. Modèle mathématique adapté à CPLEX

Nous avons choisi d'utiliser le logiciel CPLEX pour résoudre le modèle avec les données que les auteurs de l'article nous fournissent. Pour cela, nous avons dû adapter le modèle au niveau des contraintes 4, 5 et 6. Nous avons combiné les matrices des variables X et Y dans une unique matrice à trois dimensions Z.

- $Z_{i,r,k} = 1$ si la tâche i est affectée sur le robot r sur le poste k

  0 sinon

Les autres contraintes ne sont pas modifiées.

$$Min \sum_{k=1}^{W} \sum_{r=1}^{R} Y_{r,k} \cdot \left(\left(\sum_{i=1}^{N} e_{r,i} \cdot X_{i,k} \right) + \left(CT - \sum_{i=1}^{N} t_{r,i} \cdot X_{i,k} \right) \cdot es_r \right)$$

$$\sum_{k=1}^{W} k \cdot X_{h,k} \cdot P_{i,h} \leq \sum_{k=1}^{W} k \cdot X_{i,k} \quad \forall i = 1..N, \forall h = 1..N \quad (1)$$

$$\sum_{k=1}^{W} X_{i,k} = 1 \quad \forall i = 1..N \quad (2)$$

$$\sum_{r=1}^{R} Y_{r,k} = 1 \quad \forall k = 1..W \quad (3)$$

$$CT = \max_{k=1..W} \left(\sum_{i=1}^{N} \sum_{r=1}^{R} Z_{i,r,k} \cdot t_{r,i} \right) \quad (4)$$

$$\left(X_{i,k} = 1 \; ET \; Y_{r,k} = 1\right) \Rightarrow \left(Z_{i,r,k} = 1\right) \quad \forall i = 1..N, \forall r = 1..R, \forall k = 1..W \quad (5)$$

$$X_{i,k} \in \{0,1\} \quad \forall i = 1..N, \forall k = 1..W \quad (6)$$

$$Y_{r,k} \in \{0,1\} \quad \forall r = 1..R, \forall k = 1..W \quad (7)$$

$$Z_{i,r,k} \in \{0,1\} \quad \forall i = 1..N, \forall r = 1..R, \forall k = 1..W \quad (8)$$

Figure 7 : Modèle mathématique corrigé et adapté à CPLEX

III. Applications et résultats avec le solveur

Dans cette partie, nous allons présenter notre modélisation du programme linéaire sur CPLEX et l'application d'un des exemples de problèmes donné par les auteurs. Le script exécuté dans le fichier .mod est disponible en annexe.

Les auteurs de l'article ont utilisé 6 problèmes de dimensions différentes pour tester leur algorithme et le comparer à un algorithme génétique. Nous verrons cela plus en détail dans la partie concernant l'analyse des performances. Intéressons-nous au problème 1, le moins complexe, et à comment nous l'avons adapté à CPLEX.

Pour chaque problème, les auteurs nous donnent :

- Les graphes de précédence des tâches

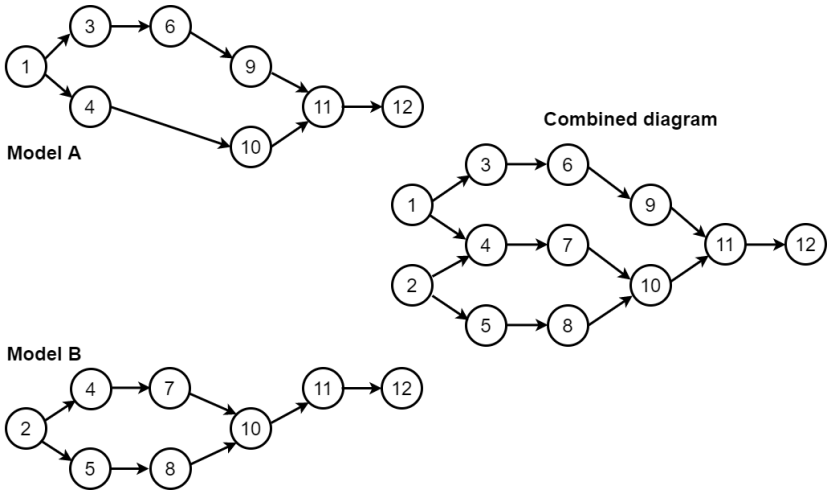

Figure 8 : Graphe des prédécesseurs (problème 1)

- Un document Excel affichant, pour chaque robot :
    - Son temps de traitement, par tâche (en unité de temps)
    - Sa consommation d'énergie pour le traitement, par tâche (kJ)
    - Sa consommation d'énergie au repos (kJ/unité de temps)
- Le nombre de stations disponibles

Avec ces informations, nous avons pu développer le fichier de données (.dat) pour CPLEX. Il fallait transformer le graphe en matrice de précédence. Cette étape est très importante et demande de la rigueur car si la matrice est fausse, le résultat n'aura aucun sens et sera biaisé.

```
P = [
[0 0 0 0 0 0 0 0 0 0 0 0],
[0 0 0 0 0 0 0 0 0 0 0 0],
[1 0 0 0 0 0 0 0 0 0 0 0],
[1 1 0 0 0 0 0 0 0 0 0 0],
[0 1 0 0 0 0 0 0 0 0 0 0],
[0 0 1 0 0 0 0 0 0 0 0 0],
[0 0 0 1 0 0 0 0 0 0 0 0],
[0 0 0 0 1 0 0 0 0 0 0 0],
[0 0 0 0 0 1 0 0 0 0 0 0],
[0 0 0 0 0 0 1 1 0 0 0 0],
[0 0 0 0 0 0 0 0 1 1 0 0],
[0 0 0 0 0 0 0 0 0 0 1 0],
];
```

Figure 9 : Matrice de précédence (problème 1)

Pour le problème 1, CPLEX trouve rapidement une solution optimale. Il s'agit d'une consommation d'énergie de 16,4 kJ et un temps de cycle de 5,5 unités de temps. Cela correspond bien aux résultats des auteurs avec leur heuristique ainsi qu'avec leur algorithme génétique.

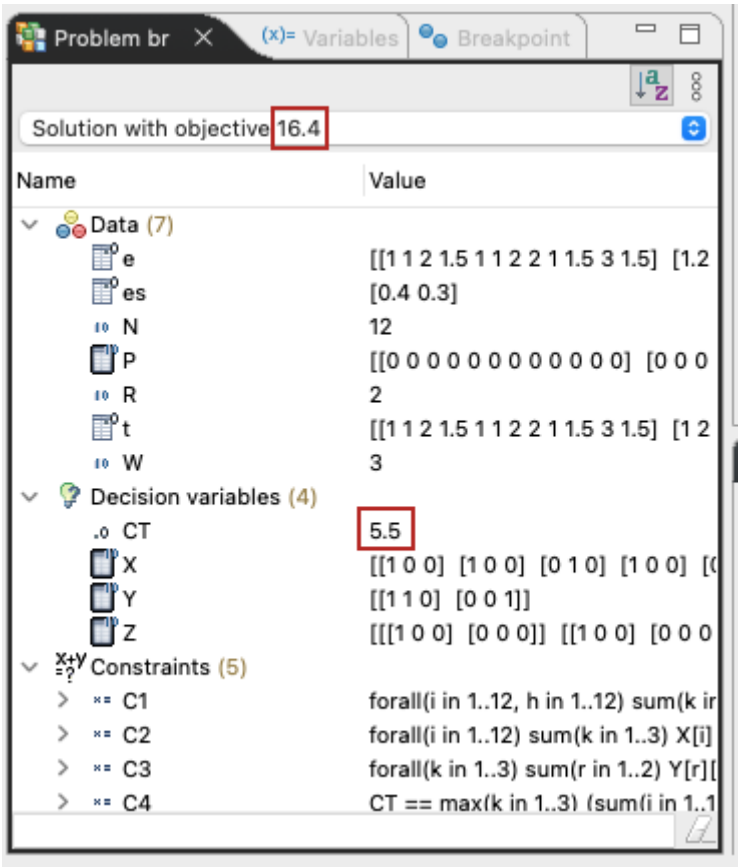

Figure 10 : Résultat problème 1

Les variables de décisions sont les suivantes :

Value for X

1..N (size 12)	1..W (size 3)		
	1	2	3
1	1	0	0
2	1	0	0
3	0	1	0
4	1	0	0
5	0	1	0
6	0	1	0
7	1	0	0
8	0	0	1
9	0	1	0
10	0	0	1
11	0	0	1
12	0	0	1

Figure 11 : Résultat variable de décision X

Value for Y

1..R (size 2)	1..W (size 3)		
	1	2	3
1	1	1	0
2	0	0	1

Figure 12 : Variable de décision Y

Avec les valeurs de variables, nous pouvons déduire les différentes affectations des tâches et robots aux stations :

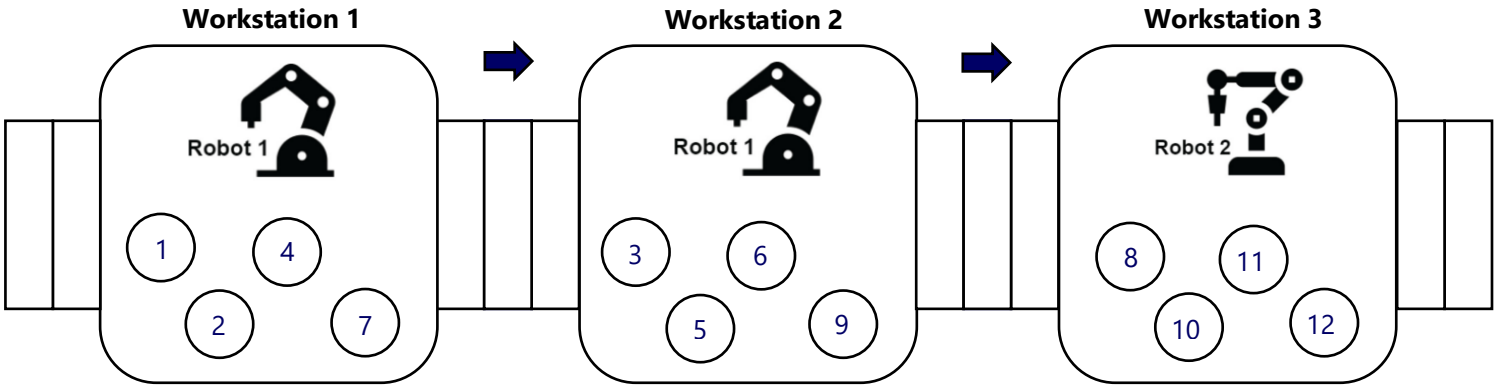

Figure 13 : Résultat des affectations des tâches et robots aux stations (problème 1)

Nous avons également pu résoudre les problèmes 2 et 3, de complexité plus grande. Les limites du solveur sont atteintes pour le problème 4 de 40 tâches, 7 stations et 3 robots. Le solveur tourne sans parvenir à trouver une solution. Pour pallier ce problème, les auteurs ont donc cherché une méthode pour résoudre des problèmes de plus grande taille dans un temps raisonnable. Ainsi, la partie suivante sera consacré au Cuckoo Search Algorithm.

IV. Partie algorithme

1. Présentation de la méthode du coucou

Comme vu dans la partie précédente, le solveur atteint ses limites quand le problème devient trop grand. C'est ainsi que l'algorithme CSA (Cuckoo Search Algorithm) entre en jeu. Nous allons dans un premier temps voir dans cette partie la méthode globale du coucou puis nous verrons l'innovation que les auteurs lui ont apporté avec l'intégration de la notion de mémoire.

La méthode du coucou est une métaheuristique qui essaye d'imiter le phénomène de parasitisme de couvées pratiqué par certaines espèces de coucous. Ce phénomène, est le fait que des coucous pondent des œufs dans le nid d'autres oiseaux afin de celui-ci soit élevé par les hôtes.

Dans notre cas, les œufs seront des solutions réalisables de notre problème. Des œufs seront donc déposés aléatoirement dans des nids (la population) et prendrons la place des anciens œufs s'ils ont un meilleur fitness.

Il arrive que des œufs déposés par des coucous soient repérés par les hôtes des nids. Ainsi nous expulseront ces œufs en définissant une probabilité fixe.

La fonction mémoire de l'algorithme permet de vérifier avant de déposer une solution dans la population que cette solution n'a pas déjà été générée.

La création du voisinage d'une solution, représentée par la création de solutions réalisables générées grâce à une mutation, « swap » dans notre cas (un échange), permet de sélectionner des solutions avec un bon fitness rapidement. Car c'est dans ce voisinage que nous sélectionnerons notre solution qui pourra probablement remplacer un œuf présent dans la solution s'il a un meilleur fitness. Sinon on ne fait rien.

Ensuite on simule que des œufs se sont fait repérer par leurs hôtes en éliminant une proportion des plus mauvaises solutions. Celles-ci sont remplacées par de nouvelles solutions issues de croisement de 2 solutions parents présentes dans la population.

Quand nous avons réalisé toutes les modifications, c'est la fin d'une génération. On recommence le protocole jusqu'à ce que l'on ait généré un nombre de générations défini. Cependant l'algorithme peut aussi prendre fin de manière précoce, si un certain nombre de solutions générées d'affilées sont déjà dans la mémoire, c'est à dire si ces solutions avaient déjà été générées, notre l'algorithme prend fin.

2. Présentation de notre code

Nous avons choisi de reproduire le programme de l'article en utilisant le langage Python. C'est un langage pour lequel nous avions certaines notions, et que nous avons jugé relativement facile à comprendre par tous. En outre, certaines opérations ont été facilitées par l'utilisation de bibliothèques connues (numpy, random...). Nous avons travaillé sur l'environnement de développement Spyder, obtenu avec la distribution Anaconda.

Pour la construction de notre algorithme nous nous sommes appuyés sur le pseudo-code fourni par l'article ainsi que certaines de leurs explications. Nous avons aussi dû nous baser sur un autre article (cf. « An investigation on minimizing cycle time and total energy consumption in robotic assembly line systems », Nilakantan and al. 2015) afin de construire notre fonction d'attribution de robots.

Nous allons partir du pseudo code fournis par l'article pour expliquer notre script. Une décomposition est nécessaire.

Algorithm 2 A memory-based Cuckoo Search Algorithm

1: Ojective funciton: $f(X), X = (X1, X2, ...Xd)$
2: Generate Initial population of N host nests
3: Initialize the memory M, *max_search*, *max_generations*
4: **while** *max_generations* **do**
5: **while** *max_search* **do**
6: Select randomly a nest (say, Y) from the population
7: **if** Y is not in M **then**
8: Reset *max_search* to the initial value
9: Exit from while loop
10: **end if**
11: Decrement *max_search*
12: **if** *max_search* is equal to 0 **then**
13: End the algorithm
14: **end if**
15: **end while**
16: Generate a list of neighbors of Y (say, $NL(Y)$) using the swap mutation with repair mechanism
17: Stock the nest Y and all its neighbors $NL(Y)$ in M
18: Select the best neighbor from $NL(Y)$ as the new cuckoo (say, C)
19: Choose a nest among N (say, H) randomly
20: **if** $f(C) < f(H)$ **then**
21: Replace H by C in the population
22: **end if**
23: A fraction Pa of worst nests are abandoned and new ones are built
24: **while** *max_search* **do**
25: Choose two parents from the reminder of the population
26: Apply the crossover on selected parents with the repair mechanism
27: **if** Generated solution are not in M **then**
28: Reset *max_search* to the initial value
29: Exit from while loop
30: **end if**
31: Decrement *max_search*
32: **if** *max_search* is equal to 0 **then**
33: End the algorithm
34: **end if**
35: **end while**
36: Replace abandoned solutions by generated one
37: Rank the solutions based on the objective value
38: Select the best solution as the best current solution
39: Decrement *max_generations*
40: **end while**

Figure 14 : Pseudo code des auteurs

Ligne 1 – « La fonction-objectif »:

La fonction-objectif est déterminée grâce à un ordonnancement généré pour notre algorithme, bien sûr notre ordonnancement est faisable, c'est-à-dire il respecte notre contrainte de précédence. Ensuite nous allons attribuer à chacune de nos tâches une station avec un robot, nous allons faire cela grâce à une heuristique déterministe donnée par l'article annexe cité précédemment dans le chapitre *3.3. Energy based model-Task and Robot assignment procedure.* Nous pourrons donc déterminer un coût suite à l'affectation.

```
def best_robot(x):
    ''' Fonction déterminant des bonnes affectations tâches/station et robot/station
    à partir d'un ordonnancement des tâches x donné
    Il provient d'un autre article auquel font référence les auteurs'''
    E0=np.sum(np.min(e.T,axis=1))/W
    NRJ=np.zeros((W,R),dtype=float)
    COUNT=np.zeros((W,R),dtype=int)
    ok = False
    while ok != True:
        for w in range(W):
            if w!=0:
                startw=sum(np.max(COUNT[i])for i in range(w))
            else:
                startw=0
            for r in range(R):
                count=0
                E=0
                while E<=E0 and startw+count<T:
                    E+=e[r][x[startw+count]-1]
                    count+=1
                if startw+count<T:
                    NRJ[w][r]=E-e[r][x[startw+count-1]-1]
                    count -= 1
                    COUNT[w][r]=count
                else:
                    NRJ[w][r]=E
                    COUNT[w][r]=count
            if np.max(np.sum(COUNT,axis=0))==T:
                return NRJ,COUNT

        if count==T:
            ok=True
        else:
            E0+=E0step
    return NRJ,COUNT
```

Figure 15 : Fonction d'affectation des ateliers et robots

Best_robot(x) affectant à x, un ordonnancement de tâches, des ateliers et des robots aux ateliers afin de minimiser le fitness.

```
def Cycle_Time(x,robots,nb_taches):
    ''' Calcul de la différence entre le CT max et celui d'une station
    Pour le calcul de l'energie en standby dans la fonction fitness'''
    CT=np.zeros(W)
    for robot in range(W):
        for i in x[sum(nb_taches[:robot]):sum(nb_taches[:robot+1])]:
            CT[robot]=CT[robot]+t[robots[robot]-1][i-1]
    return np.max(CT)-CT
```

Figure 16 : Fonction de calcul du temps de cycle

Cycle_Time(x,robot,nb_taches) permet de calculer le temps de cycle d'une solution et également le coût lorsqu'elle est en Standby.

```
def affectations(x,NRJ,COUNT):
    ''' Fonction traduisant les résultats de la fonction best_robot (NRJ et COUNT)
    en des listes claires concernant les affectations des robots et tâches aux stations'''
    robots = np.zeros(W,dtype=int)
    nb_taches = np.zeros(W,dtype=int)
    for station in range(W):
        nb_taches[station] = np.amax(COUNT[station])
        energie_min = 100000
        for robot in range(R):
            if COUNT[station][robot] == nb_taches[station] and NRJ[station][robot]<energie_min:
                energie_min = NRJ[station][robot]
                robots[station]=robot+1
    return robots,nb_taches
```

Figure 17 : Fonction facilitant la lecture de la solution

Affectation(X,NRJ,COUNT) permet de mettre en forme une solution pour rendre sa lecture facile.

```
def fitness(x):
    '''Fonction qui renvoie le fitness (=consommation d'énergie totale) d'un ordonnancement x donné
    Pour cela il faut également avoir l'affectation des tâches aux stations
    et l'affectation des robots aux stations
    '''
    NRJ,COUNT = best_robot(x)
    robots,nb_taches = affectations(x,NRJ,COUNT)
    fit_process = 0
    fit_standby = 0
    for robot in range(W):
        for i in x[sum(nb_taches[:robot]):sum(nb_taches[:robot+1])]:
            fit_process+=e[robots[robot]-1][i-1]
        fit_standby += Cycle_Time(x,robots,nb_taches)[robot]*es[robots[robot]-1]
    return fit_process+fit_standby
```

Figure 18 : Fonction calculant le fitness

Fitness(x) permet de sortir le coût énergétique d'une solution. Autrement dit, elle calcule le fitness.

Ligne 2 à 3 – « Initialisation »:

```
def hasard_faisable():
    '''Fonction appelée lors de l'initialisation
    Renvoie un ordonnancement faisable selon les relations de précédence'''
    tache_visitee = np.zeros(T,dtype=bool)
    ordonnancement = []
    for ordre in range(T):
        liste_taches_possibles =[]
        for tache in range(T):
            if not tache_visitee[tache]:
                tache_possible = True
                for predecesseur in range(T):
                    if P[tache,predecesseur]==1 and not tache_visitee[predecesseur]:
                        tache_possible=False
                if tache_possible:
                    liste_taches_possibles.append(tache+1)
        indice_aleatoire = random.randint(0,len(liste_taches_possibles)-1)
        ordonnancement.append(liste_taches_possibles[indice_aleatoire])
        tache_visitee[liste_taches_possibles[indice_aleatoire]-1] = True
    return ordonnancement
```

Figure 19 : Fonction générant un ordonnancement faisable

Hasard_faisable( ) est une fonction permettant d'avoir un ordonnancement de tâches construit aléatoirement mais construit en respectant les contraintes de précédence.

```
def initialisation():
    '''Procédure utilisée au début de l'algorithme :
    Remplit une variable globale X avec une population initiale de N nids/oeufs réalisables'''
    global X; X = [] # Population de N nids
    global M; M = [] # Mémoire

    for n in range(N):
        solution = hasard_faisable()
        X.append(list(solution))
```

Figure 20 : Fonction initialisation

Initialisation( ) permet d'initialiser la mémoire de l'algorithme comme étant une liste vide et notre population comme étant une liste remplie de solutions faisables.

Lignes 16 à 22 – « Voisinage et mutation »:

```
def tester_faisable(ordonnancement):
    '''Fonction utilisée par le mécanisme de réparation
    Teste si un ordonnancement donné est faisable selon les relations de précédence
    Si non, renvoie les indices des deux premières tâches à échanger trouvées'''

    resultat = True
    tache_visitee = np.zeros(T,dtype=bool)

    for tache in reversed(ordonnancement):
        tache_actuelle = tache-1
        tache_visitee[tache_actuelle] = True
        for tache2 in range(T):
            if P[tache_actuelle,tache2] and tache_visitee[tache2]:
                resultat = False
                return tache_actuelle+1, tache2+1
    return resultat
```

Figure 21 : Fonction vérifiant si un ordonnancement respecte la contrainte de précédence

Tester_faisable(x) permet de vérifier si un ordonnancement x respecte les contraintes de précédence. Il renvoie soit un True si l'ordonnancement est faisable, sinon les 2 tâches contraignantes.

```
def reparation(x):
    ''' Fonction qui recoit un ordonnancement non faisable et en renvoie un faisable'''
    verif_pred = tester_faisable(x)
    while verif_pred != True :
        x[x.index(verif_pred[1])]=verif_pred[0]
        x[x.index(verif_pred[0])]=verif_pred[1]
        verif_pred = tester_faisable(x)
    return x
```

Figure 22 : Fonction qui rend les ordonnancements violant la contrainte de précédence faisables

Reparation(x) permet d'échange les 2 tâches contraignantes d'un ordonnancement si celui-ci n'est pas faisable.

```python
def generation_voisinage(x):
    '''Fonction qui génère un voisinage de nb_voisins voisins d'une solution (ordonnancement) donnée
    Les voisins correspondent à une mutation swap suivie d'un mécanisme de réparation
    Tous les voisins sont uniques'''
    voisinage = []
    i = 0
    while i < nb_voisins:
        voisin = x[:]

        indice_tache1 = random.randint(0,T-1)
        indice_tache2 = random.randint(0,T-1)
        while indice_tache2 == indice_tache1:
            indice_tache2 = random.randint(0,T-1)
        tache1 = voisin[indice_tache1]
        tache2 = voisin[indice_tache2]
        voisin[indice_tache1] = tache2
        voisin[indice_tache2] = tache1

        voisin = reparation(voisin)

        if not voisin in voisinage:
            voisinage.append(voisin)
            i+=1

    return voisinage
```

Figure 23 : Fonction générant une liste de solutions faisables issues de la mutation d'une unique solution

Generation_voisinage(x) renvoie une liste d'ordonnancements issues de mutations de type « swap » d'un ordonnancement déjà présent dans la population. Toutes les solutions présentes dans la population sont uniques et faisables. Si elles ne sont pas faisables après la mutation elles sont réparées en passant dans les fonctions citées précédemment.

Lignes 23 à 36 – « Sélection et croisement »

```python
def cross_over_ordo(p1,p2):
    '''Fonction qui renvoie les 2 enfants d'un cross-over
    entre deux parents donnés (des ordonnancements)'''
    alea = np.random.randint(1,T//2+1)
    verif_enf1=np.zeros(T,bool)
    verif_enf2=np.zeros(T,bool)
    enf1=[]
    enf2=[]
    for i in range(alea):
        enf1.append(p1[i])
#enf = 1er partie de parent 1 puis on complete enf1 par
#les taches manquante dans l'ordre d'apparition de parent 2
        enf2.append(p2[i])
        verif_enf1[p1[i]-1]=True
        verif_enf2[p2[i]-1]=True
 #enfant 1 egale au debut de parent 1 et se complet par le reste de parent 2
    for i in range(T):
        if verif_enf2[p1[i]-1]==False:
            enf2.append(p1[i])
        if verif_enf1[p2[i]-1]==False:
            enf1.append(p2[i])

    #Mécanisme de réparation :
    enf1 = reparation(enf1)
    enf2 = reparation(enf2)

    return enf1,enf2
```

Figure 24 : Fonction de croisement de solutions

Cross_over_ordo(p1,p2) est une fonction de croisement de 2 solutions sortant 2 nouvelles solutions. On a donc une première solution étant égale à une proportion de la première solution et puis complétée par la seconde. Et inversement pour la seconde nouvelle solution. Ce mécanisme n'implique pas que les solutions sortantes soient faisables. On fait donc appel à la fonction réparation avant de les sortir de la fonction.

```
def selection():
    ''' Sélection : une proportion PA des oeufs (solutions) est abandonnée et remplacée'''
    global max_search
    couples=[]
    X.sort(key=fitness)

    NA = int(PA*N)
    count=NA
    while count !=0:
        if len(couples)==(N-NA)*(N-NA-1):
            return
        while len(couples)<(N-NA)*(N-NA-1) and count!=0:
            alea1,alea2=np.random.randint(0,N-NA),np.random.randint(0,N-NA)
            if alea1!=alea2:
                if (alea1,alea2) not in couples:
                    couples.append((alea1,alea2))
                    couples.append((alea2,alea1))
                    Cr_Ov=cross_over_ordo(X[alea1],X[alea2])
                    for i in range(2):
                        if count>0:
                            if verif_memo(Cr_Ov[i])==False:
                                max_search=max_search_init
                                count-=1
                                X[-count-1]=Cr_Ov[i]
                            else:
                                max_search-=1
                            if max_search==0:
                                return
    return
```

Figure 25 : Fonction de remplacement des pires solutions

Selection() permet d'échanger une proportion *PA* des pires solutions, c'est-à-dire avec un moins bon fitness, de la population. On les échange par des solutions générées à l'aide de solutions faisant partie de la proportion 1-*PA* des meilleures solutions. Les solutions parents sont sélectionnés aléatoirement et les enfants ne doivent pas être déjà dans la mémoire. Si ce n'est pas le cas on change de couple jusqu'à ce qu'on ait changé la proportion *PA* de notre population.

On peut aussi remarquer le paramètre *max_search* qui soit se décrémente de 1 soit reprend sa valeur initiale. Ce paramètre permet que si nous générer un certain nombre de fois (*max_search_init* fois) des solutions dans la mémoire alors on arrête l'algorithme.

Autres lignes « Main »:

```
def main_algo():
    '''Algorithme principal de la heuristique MBCSA tel que présenté dans l'article'''
    global max_search,max_generations
    max_search = max_search_init
    max_generations=max_generations_get
    initialisation()
    Best_x = X[0]
    while max_generations >=0:
        while max_search >=0:
            alea = np.random.randint(0,N-1)
            x=X[alea]
            if not x in M:
                max_search=max_search_init
                break
            else :
                max_search-=1
                if max_search==0:
                    return Best_x
        V=generation_voisinage(x)
        M.append(V)
        M.append(x)
        fitV=[fitness(v) for v in V]
        alea = np.random.randint(0,N-1)
        h=X[alea]
        C=V[np.argmin(fitV)]
        if fitness(h)>fitness(C):
            X[alea]=C
        selection()
        X.sort(key=fitness)
        if fitness(X[0])<fitness(Best_x):
            Best_x=X[0]
        max_generations-=1
    return Best_x
```

Figure 26 : Fonction corps de l'algorithme

Main_algo( ) est la fonction corps de l'algorithme, elle met tout en relation et permet à la population de subir les modifications dans le bon ordre. On remarque le paramètre *max_generation* qui est donc le nombre de fois que nous allons effectuer ce processus sur une population initiale.

La fonction retourne la meilleure solution de sa population finale.

Cependant de notre côté nous ne nous sommes pas arrêtés ici. En effet, d'autres fonctions comme ***main(), paramétrage()*** et d'autres lignes ont été créées afin de facilité le traitement des résultats obtenues grâce à l'algorithme ou encore de rendre l'exécutable plus ergonomique.

3. Présentation et mode d'emploi de l'exécutable

Votre exécutable doit être placé dans un dossier avec les fichiers Excel contenant les caractéristiques des problèmes à traiter.

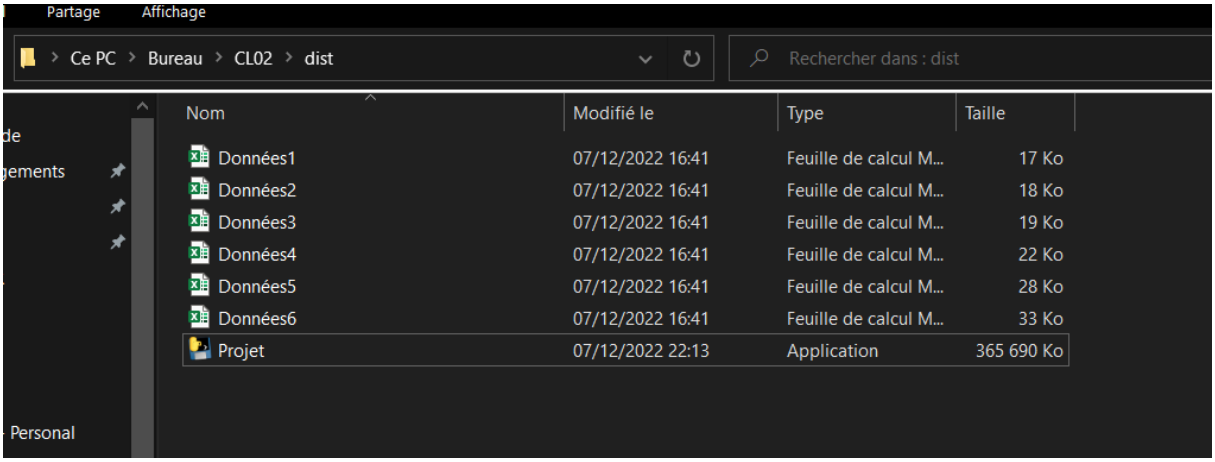

Figure 27 : Dossier comportant les données de problème et l'exécutable

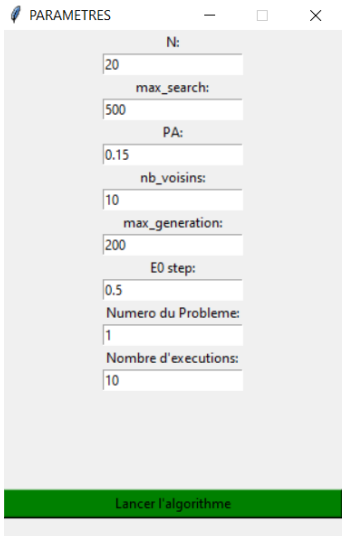

Figure 28 : Fenêtre de dialogue

Après avoir exécuté l'exécutable, attendez que la fenêtre de dialogue s'ouvre.

N représente la taille de votre population, *max_search* définit après combien de solutions générées d'affilée mais étant déjà dans la mémoire on met fin à l'algorithme. *PA* est la proportion de solutions à changer lors de la sélection. *Nb_voisins* est le nombre de voisins à générer lors de la phase de mutation. *Max_generation* est le nombre de fois que nous renouvelons notre population. *E0_step* est un paramètre utile pour l'affectation des robots et des ateliers, il exerce une grande influence sur le temps de traitement mais perd de son importance en fonction de la grandeur de problème. Nous conseillons alors de le dimensionner inférieur à 1 plus le problème est petit et de vous rapprocher de 1 plus le problème est grand, cependant cela reste à la discrétion du programmeur. *Le numéro du problème* sert à traiter le problème caractérisé

par les données du fichier Excel ayant le même numéro. *Nombre d'exécutions* est le nombre de fois que nous exécutons l'algorithme du coucou.

Vous pouvez remplir les cases pour modifier les valeurs des paramètres. L'article propose certaines valeurs de paramètres mais laisse aussi le choix aux programmeurs pour les autres. Des valeurs par défaut sont proposées dans la boite de dialogue afin de résoudre le problème 1.

Used parameters in the MBCSA and the MLCSA.

Problems	Generations	Population's size	Fraction P_a
P1	200	20	0.15
P2	300	30	0.15
P3	300	30	0.15
P4	300	100	0.15
P5	300	100	0.15
P6	300	100	0.15

Figure 29 : Paramètres proposés par l'article

Après avoir rempli les cellules avec les valeurs de vos paramètres et avoir choisi quel problème vous vouliez traiter ainsi que le nombre de fois que vous voulez faire tourner l'algorithme du Coucou.

Appuyez sur « Lancer l'algorithme ».

Une barre de chargement s'affiche sur la console :

Figure 30 : Barre de chargement de la console

A la fin du chargement une boite de dialogue s'affiche nous indiquant que les résultats sont dans un fichier Excel nommé « Resultat.xlsx ». Cliquez sur le bouton « Quitter ».

Accédez aux résultats en ouvrant le tableur Excel « Résultat.xlsx ».

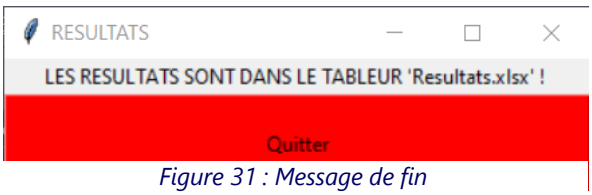

Figure 31 : Message de fin

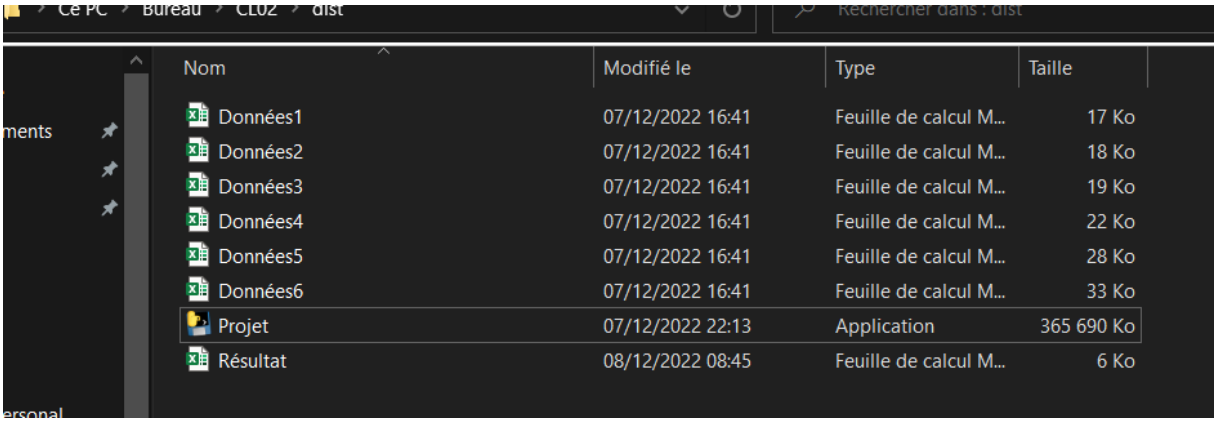

Figure 32 : Répertoire des fichiers

V. Résultats et analyse des performances

Comme nous l'avons vu, l'article propose 6 problèmes de complexité croissante. Les auteurs ont réalisé 9 exécutions pour chaque problème. Ils le comparent avec un algorithme génétique sur 3 critères : la consommation d'énergie (fonction-objectif), le temps de cycle et le temps de calcul sur leur machine.

Nous avons réalisé 10 exécutions de notre algorithme pour chaque problème. 6 paramètres sont nécessaires. Les auteurs en donnent 3 (et quelques indications pour les autres). Nous avons donc d'abord trouvé empiriquement comment bien ajuster tous les paramètres ensembles.

Commençons par étudier les résultats concernant la fonction-objectif :

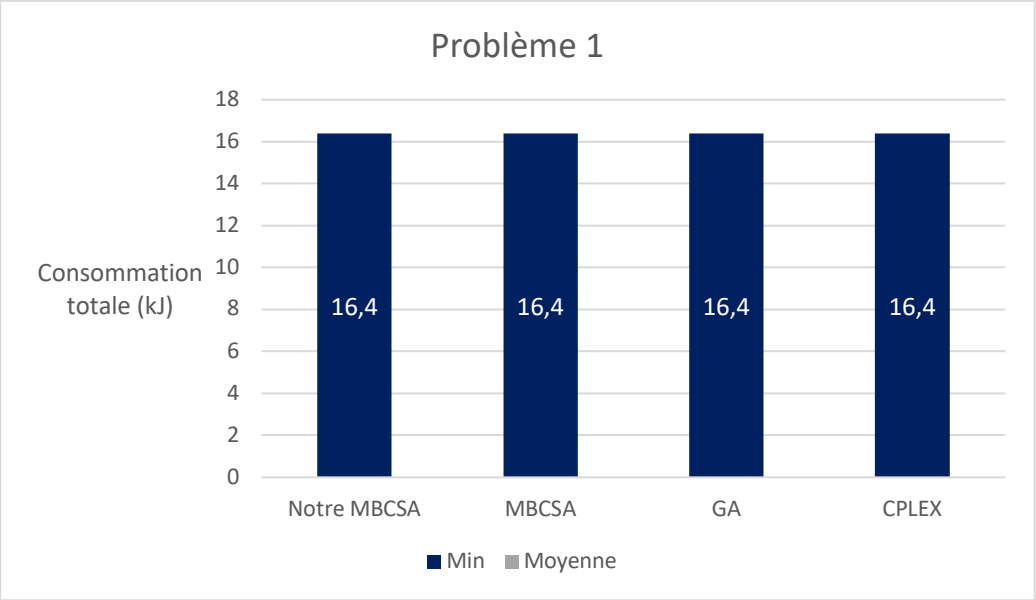

Figure 33 : Résultat problème 1

En abscisse se trouvent les différents algorithmes comparés : Notre algorithme du coucou (MBCSA), celui des chercheurs, leur algorithme génétique (GA) ainsi que notre résultat sur le solveur CPLEX.

La barre grise représente la moyenne des différentes exécutions de l'algorithmes, et celle bleue (ainsi que l'étiquette de données) la meilleure solution trouvée dans l'échantillon.

Pour le problème 1, nous voyons que les 4 algorithmes ont trouvé la solution optimale, à chaque exécution.

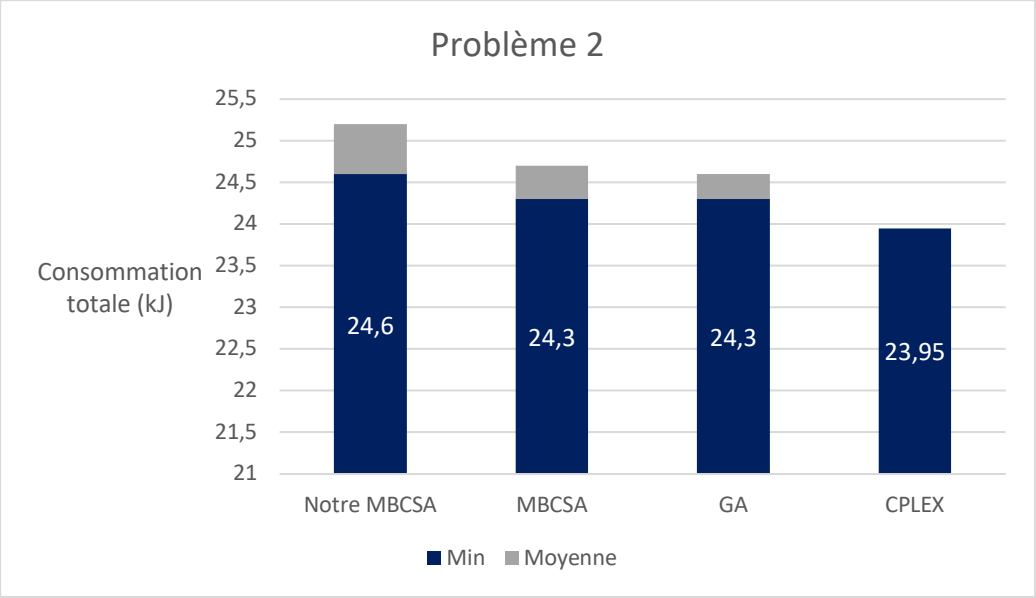

Figure 34 : Résultat problème 2

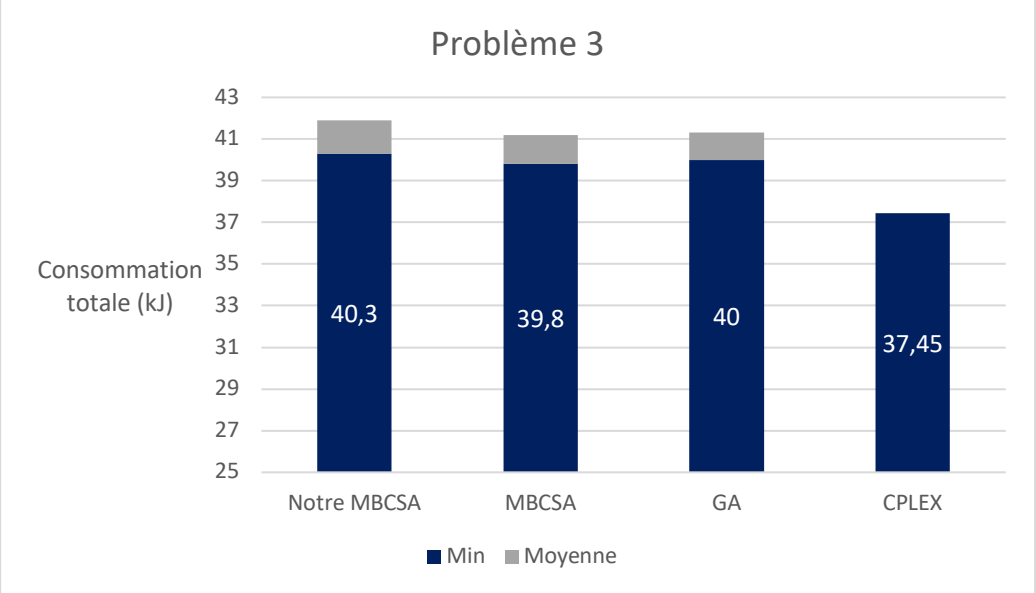

Figure 35 : Résultat problème 3

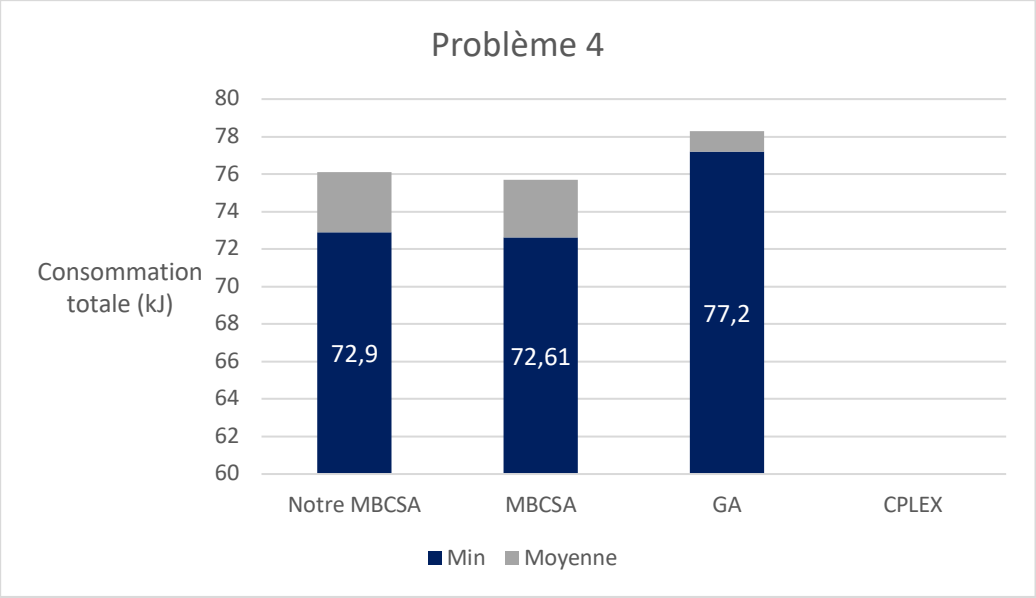

Figure 36 : Résultat problème 4

À partir du problème 4, nous ne connaissons plus la solution optimale, le solveur de CPLEX n'arrivant plus à trouver une solution dans un temps raisonnable.

Du problème 2 à 4, nous trouvons des résultats très proches de ceux des auteurs. Ils sont très légèrement au-dessus, pour la moyenne et le minimum, donc très légèrement moins bons.

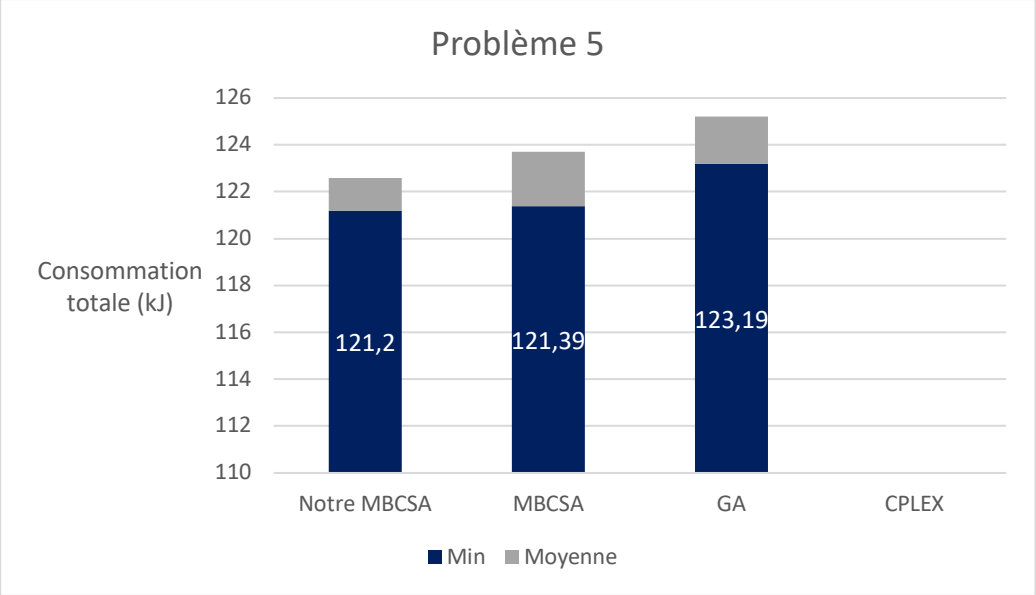

Figure 37 : Résultat problème 5

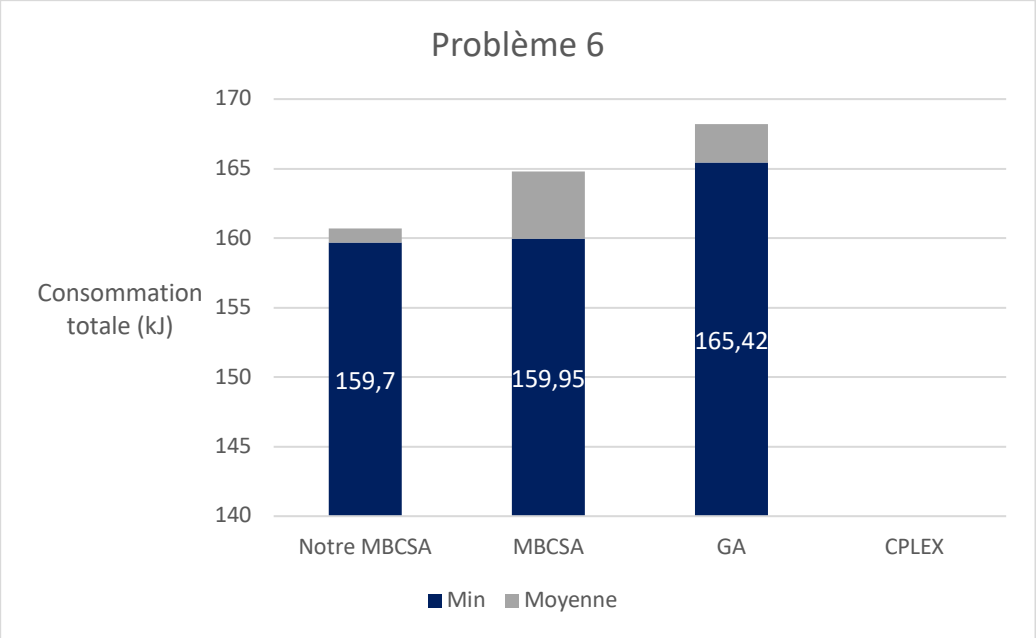

Figure 38 : Résultat problème 6

À partir du problème 5, nous trouvons les meilleurs résultats, autant pour la moyenne des solutions que pour la meilleure.

Étudions maintenant le temps de calcul. Pour les problèmes 1 à 3 :

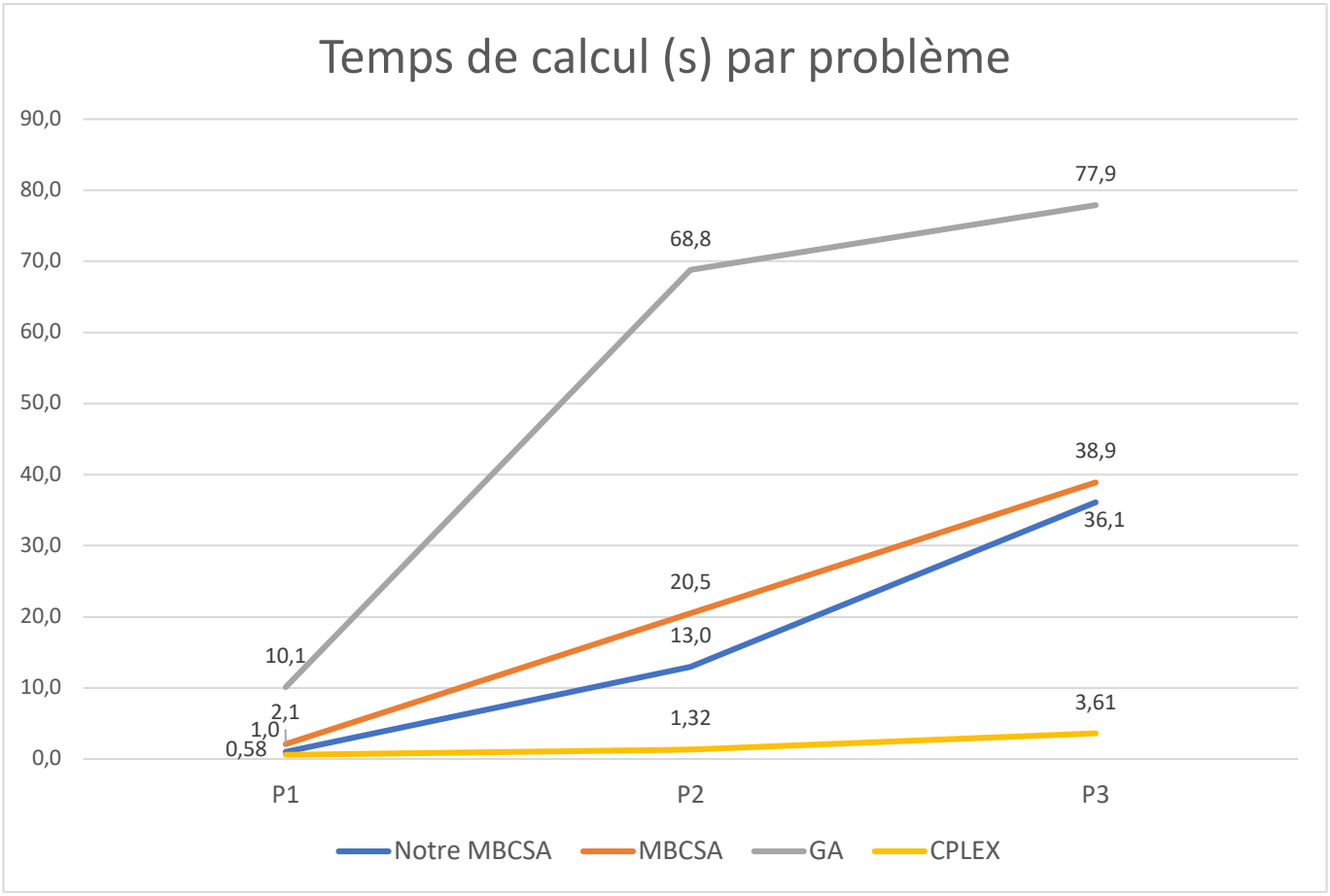

Figure 39 : Temps de calcul en secondes par problème

Notre algorithme est relativement rapide, dans le même ordre de grandeur que le MBCSA.

Cela change à partir du problème 4 :

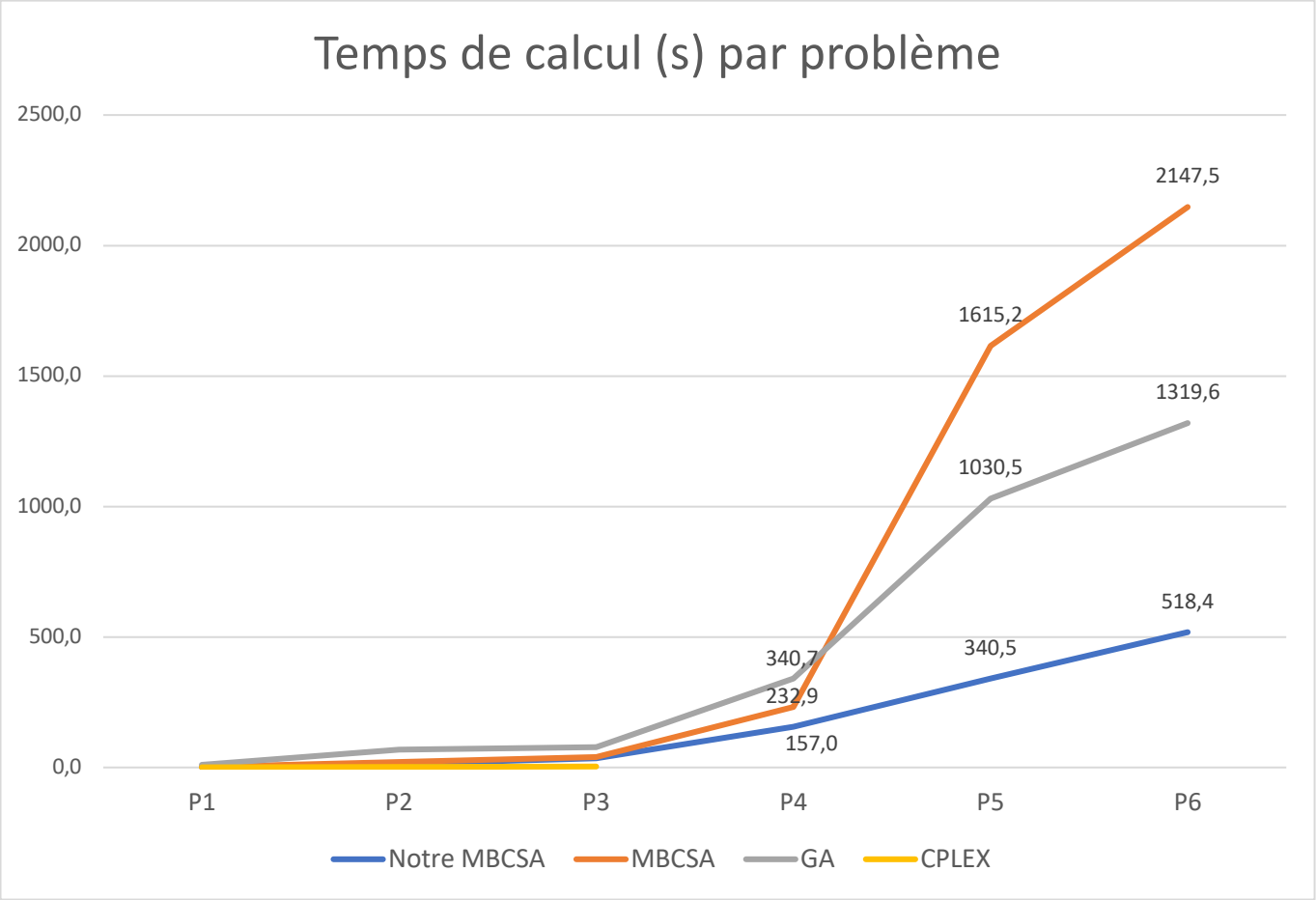

Figure 40 : Temps de calcul en secondes par problème

Les auteurs proposent leur métaheuristique car, comme nous l'avons vu, elle fournit de meilleurs résultats qu'un algorithme génétique. Toutefois, nous voyons que le temps de calcul est plus long que leur algorithme génétique pour des problèmes complexes.

Ces résultats nous surprennent car nous obtenons le record du temps de calcul, avec un temps jusqu'à 4 fois inférieur au MBCSA de l'article. D'autant plus que nous obtenons de meilleures solutions pour ces problèmes-là, et que le langage python n'est pas réputé pour sa rapidité.

Il pourrait y avoir différents facteurs (machines des auteurs, manières de coder les fonctions, paramétrage…)

Vous trouverez en annexe d'autres informations concernant l'analyse de performances.

VI. Conclusion

1. Apports du projet

Ce projet de CL02 a été intéressant pour nous. Il nous a permis de nous familiariser davantage avec la recherche scientifique en Génie Industriel. Nous avons pu comprendre le problème présenté, interpréter et corriger la modélisation mathématique associée, appréhender et répliquer une méthode algorithmique et enfin réaliser un travail rigoureux d'analyse de performances.

Nous avons su reproduire la métaheuristique présentée et trouver des bonnes solutions aux problèmes complexes proposés. Il nous a fallu notamment approfondir notre recherche bibliographique pour comprendre des méthodes d'affectation dans un problème de ligne de production robotisée, lorsque l'objectif à minimiser est la consommation d'énergie.

Au niveau technique, ce fut une opportunité de développer notre aisance en algorithmique, particulièrement avec le langage python. Nous avons également eu la possibilité de travailler avec le solveur CPLEX, auquel nous avions été initiés en travaux pratiques.

Au niveau relationnel, ce travail en équipe a demandé un travail de communication et d'organisation qui contribue également à notre expérience.

2. Analyse critique de l'article et conclusion

Nous relevons des points forts de cet article. Le sujet, et particulièrement l'angle de vue nous semble intéressant. Cela a des applications réelles pour différents systèmes de production robotisée. Le choix de fixer la minimisation de la consommation d'énergie comme fonction-objectif se montre pertinent dans le contexte actuel de tensions sur la production d'énergie. D'autant plus que le temps de cycle pour un produit est naturellement peu élevé avec ce critère (les robots consommant également de l'énergie en stand-by).

Il y a également des points faibles que nous souhaitons souligner. D'abord, l'article comportait quelques erreurs, particulièrement dans la modélisation mathématique proposée. Il semble que les auteurs n'ont pas testé leurs problèmes sur un solveur. De plus, nous estimons que les auteurs auraient pu montrer davantage de clarté et de précision pour la description de leur algorithme et la donnée de leurs paramètres. Enfin, en analysant leurs exécutions, nous voyons que leur temps d'exécution de l'algorithme est relativement long pour les problèmes complexes (35 minutes en moyenne pour le problème 6). Toutefois, l'article évoque cette limite et la volonté d'améliorer l'algorithme.

Comme nous l'avons vu, nous avons pu trouver des solutions avec des temps d'exécutions plus rapides. Il est donc possible d'accélérer cet algorithme sans même changer sa structure globale.

VII. Bibliographie

- BELKHARROUBI, L., KHADIDJA, Y. « Solving the energy-efficient Robotic Mixed-Model Assembly Line balancing problem using a Memory-Based Cuckoo Search Algorithm ». *Engineering Applications of Artificial Intelligence*. Volume 114, Septembre 2022, 105112. Disponible sur : https://www.sciencedirect.com/science/article/abs/pii/S0952197622002494?via%3Dihub

- J. MUKUND NILAKANTANA, GEORGE Q. HUANGB, S.G. PONNAMBALAM. «An investigation on minimizing cycle time and total energy consumption in robotic assembly line systems ». *Journal of Cleaner Production*. Volume 90, 1 Mars 2015, Pages 311-325. Disponible sur: https://www.sciencedirect.com/science/article/abs/pii/S0959652614012219?via%3Dihub

- Source de la figure 1 : https://fr.freepik.com/vecteurs-premium/illustration-isometrique-chaine-production-moderne_6221888.htm

VIII. Table des illustrations

Figure 1 : Exemple ligne d'assemblage robotisée 3
Figure 2 : Graphique des différents problèmes d'équilibrage 4
Figure 3 : Diagrammes des prédécesseurs 6
Figure 4 : Schéma ligne d'assemblage robotisée 6
Figure 5 : Modèle mathématique des auteurs 8
Figure 6 : Modèle mathématique corrigé 9
Figure 7 : Modèle mathématique corrigé et adapté à CPLEX 10
Figure 8 : Graphe des prédécesseurs (problème 1) 11
Figure 9 : Matrice de précédence (problème 1) 12
Figure 10 : Résultat problème 1 12
Figure 11 : Résultat variable de décision X 13
Figure 12 : Variable de décision Y 13
Figure 13 : Résultat des affectations des tâches et robots aux stations (problème 1) 13
Figure 14 : Pseudo code des auteurs 15
Figure 15 : Fonction d'affectation des ateliers et robots 16
Figure 16 : Fonction de calcul du temps de cycle 16
Figure 17 : Fonction facilitant la lecture de la solution 17
Figure 18 : Fonction calculant le fitness 17
Figure 19 : Fonction générant un ordonnancement faisable 17
Figure 20 : Fonction initialisation 18
Figure 21 : Fonction vérifiant si un ordonnancement respecte la contrainte de précédence 18
Figure 22 : Fonction qui rend les ordonnancements violant la contrainte de précédence faisables 18
Figure 23 : Fonction générant une liste de solutions faisables issues de la mutation d'une unique solution 19
Figure 24 : Fonction de croisement de solutions 19
Figure 25 : Fonction de remplacement des pires solutions 20
Figure 26 : Fonction corps de l'algorithme 21
Figure 27 : Dossier comportant les données de problème et l'exécutable 22
Figure 28 : Fenêtre de dialogue 22
Figure 29 : Paramètres proposés par l'article 23
Figure 30 : Barre de chargement de la console 23
Figure 31 : Message de fin 23
Figure 32 : Répertoire des fichiers 24
Figure 33 : Résultat problème 1 25
Figure 34 : Résultat problème 2 26
Figure 35 : Résultat problème 3 26
Figure 36 : Résultat problème 4 27
Figure 37 : Résultat problème 5 27
Figure 38 : Résultat problème 6 28
Figure 39 : Temps de calcul en secondes par problème 29
Figure 40 : Temps de calcul en secondes par problème 30

IX. Annexes

Nos annexes se trouvent dans le dossier du même nom. Il y a :

- Les données des auteurs à partir desquelles nous avons pu tester et comparer notre algorithme
- Les fichiers permettant la modélisation et résolution des problèmes 1 à 4 sur CPLEX
- Le fichier contenant nos résultats détaillés et ayant servi à l'analyse des performances