Approche heuristique pour l'équilibrage de lignes d'assemblage avec minimisation du pic de consommation énergétique

Thiago G. Araujo[1], Zhifei Zheng[2], Matthieu Py[1], Sami Cherif[2],
Rui Sá Shibasaki[2], Laurent Deroussi[1], Nathalie Grangeon[1], Chu-Min Li[2]

[1] Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Étienne, LIMOS, 63000 Clermont-Ferrand, France
[2] Laboratoire MIS UR 4290, Université de Picardie Jules Verne, Amiens, France

thiago.giachetto_de_araujo@uca.fr

Résumé

Dans un monde où les enjeux environnementaux sont de plus en plus importants, l'amélioration de l'efficacité énergétique des systèmes de production est cruciale. Cet article s'intéresse aux problèmes d'équilibrage de lignes d'assemblage, dans lesquels une stratégie possible consiste à minimiser les pics de consommation énergétique générés par l'ordonnancement des tâches choisi, problème dénommé SALB3PM. Pour résoudre ce problème, nous comparons expérimentalement deux méthodes : une méta-heuristique à redémarrages multiples, basée sur une recherche locale exploitant trois types de mouvements ; et une approche de résolution basée sur la Satisfiabilité Maximum (MaxSAT) issue de travaux précédents. Les expérimentations sont menées sur plus d'une centaine d'instances de la littérature et montrent l'efficacité des deux approches pour la résolution de SALB3PM.

Mots-clés

Ordonnancement, SALBP, Énergie, Métaheuristiques, MaxSAT.

Abstract

In a world where environmental concerns are increasingly central, improving the energy efficiency of production systems is crucial. This paper focuses on assembly line balancing problems, where the objective is to minimize the energy consumption peaks generated by task scheduling on assembly lines. This problem is known as the Simple Assembly Line Balancing Problem with Power Peak Minimization (SALB3PM). To solve this problem, we experimentally compare two methods: a multi-start metaheuristic based on local search using three types of moves, and a resolution approach based on Maximum Satisfiability (MaxSAT) derived from previous work. Experiments are conducted on more than a hundred benchmark instances and demonstrate the effectiveness of both approaches for solving SALB3PM.

Keywords

Scheduling, SALBP, Energy, Metaheuristics, MaxSAT.

1 Introduction

Les lignes d'assemblage constituent un système de production important. Elles se caractérisent par un agencement linéaire de postes de travail. Dans ce système, les produits se déplacent séquentiellement d'un poste de travail à l'autre, des tâches spécifiques étant exécutées à chaque poste jusqu'à l'assemblage complet du produit. Ces systèmes peuvent être modélisés sous forme de problèmes d'équilibrage de lignes d'assemblage simples (Simple Assembly Line Balancing Problems, SALBP) [2]. Les variantes SALBP les plus connues ont largement été étudiées dans la littérature [17, 16, 6]. Alors que les conséquences du réchauffement climatique affectent la population mondiale, la réduction des émissions de CO_2 devient impérative. Le rapport de 2024 de l'Agence Internationale de l'Énergie (AIE) a montré que le secteur industriel est responsable de 45% des émissions de CO_2 et de 39% de la consommation totale d'énergie dans le monde [1]. Il est donc essentiel d'améliorer l'efficacité énergétique des systèmes de production. Pour répondre à ces enjeux, de nouvelles variantes de SALBP intégrant des contraintes énergétiques ont récemment été proposées.

Un exemple est le problème SALBP avec minimisation du pic de consommation énergétique (ou simpelement SALB3PM), introduit dans [8]. Dans ce problème, chaque tâche est associée à une consommation d'énergie constante, indépendante du poste de travail. L'objectif est de minimiser le pic de consommation énergétique, c'est-à-dire le maximum de consommation d'énergie sur l'ensemble des unités de temps. Des méthodes exactes ont été proposées pour résoudre SALB3PM dans [8, 14, 18]. Dans l'article original [8], une formulation en Programmation Linéaire en Nombres Entiers (PLNE) a été proposée. Dans [14], un algorithme exact basé sur des appels itératifs à des solveurs de satisfiabilité (SAT) a été introduit pour SALB3PM. Plus récemment, un modèle basé sur la satisfiabilité maximum (MaxSAT) a été proposé dans [18], et deux représentations booléennes différentes ont été testées. Les résultats expérimentaux

montrent que les formulations SAT et MaxSAT sont efficaces pour SALB3PM, surpassant les formulations PLNE existantes. Cependant, à mesure que la taille des instances augmente, ces approches exactes rencontrent des difficultés à résoudre les instances de manière optimale.

Étant donné la difficulté d'obtenir des solutions optimales, des approches métaheuristiques ont été proposées pour une variante de SALB3PM qui ne prend pas en compte les délais entre les tâches, à savoir le SALB3PM-ESD. Pour traiter ce problème, une métaheuristique basée sur une recherche locale évolutive à démarrages multiples (Multi-start Evolutionary Local Search, MS-ELS) a été mise en œuvre dans [11]. En étudiant la même variante que [11], un décodeur basé sur une formulation PLNE pour des heuristiques à base de permutations a été introduit dans [5]. Pour tester leur proposition, les auteurs ont implémenté un algorithme de recuit simulé simple avec le décodeur proposé. Cependant, cette heuristique ne peut pas être utilisée directement pour résoudre le problème SALB3PM classique.

Pour résoudre SALB3PM, on présente dans cet article deux méthodes heuristiques. La première, issue de travaux récents [18], consiste à modéliser le problème SALB3PM sous forme MaxSAT, et à lancer une méthode de résolution incomplète dédié au problème MaxSAT. La seconde, brièvement proposée dans [7], utilise une métaheuristique à redémarrages multiples avec trois différents types de voisinage. Les solutions initiales sont générées à l'aide d'une heuristique constructive basée sur les priorités, avec une fonction de priorité semi-aléatoire. Pour la recherche locale, nous adaptons deux structures de voisinage couramment utilisées dans la littérature, l'insertion de tâche et l'échange de tâches, et nous introduisons un troisième mouvement qui consiste à augmenter le délai des tâches.

L'article est organisé comme suit. La section 2 présente le problème SALB3PM. La section 3 présente la formulation MaxSAT du problème proposée dans [18]. La section 4 présente la métaheuristique proposée. La section 5 propose une étude expérimentale des méthodes présentées. Enfin, on conclut et on discute les travaux futurs dans la section 6.

2 Problème SALB3PM

Une ligne d'assemblage se compose d'un ensemble $\mathcal{M} = \{1, \ldots, m\}$ de postes de travail disposés de manière linéaire. Pour fabriquer un produit, un ensemble $\mathcal{O} = \{1, \ldots, n\}$ d'opérations indivisibles, appelées tâches, doit être exécuté. En raison de contraintes intrinsèques, certaines tâches ne peuvent être exécutées qu'après l'achèvement d'autres tâches. Par conséquent, les tâches doivent respecter des contraintes de précédence. Ces contraintes peuvent être représentées par un graphe de précédence, où chaque tâche est représentée par un nœud, et où un arc (i, j) existe si la tâche i doit être terminée avant que la tâche j ne commence. Cette relation de précédence entre les tâches i et j est notée $i \prec j$.

Exemple 1 *La figure 1 montre un graphe de précédence qui concerne 7 tâches. En particulier, pour que la tâche 6 puisse être exécutée, il faut que ses prédécesseurs directs (tâche 5) et indirects (tâches 1,2 et 4) soient terminés. De manière similaire, la tâche 6 devra être terminée avant que ses successeurs directs (tâche 7) et ses successeurs indirects ne puissent commencer.*

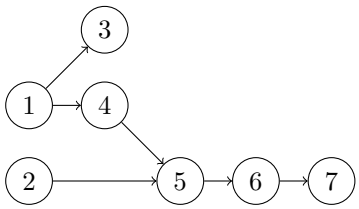


Figure 1: Exemple de graphe de précédence

Dans le problème d'équilibrage de lignes d'assemblage simple (Simple Assembly Line Balancing Problem, SALBP), chaque tâche $j \in \mathcal{O}$ possède un temps de traitement connu t_j, indépendant du poste auquel elle est affectée. Chaque tâche doit être affectée à un et un seul poste de travail de manière à ce que toutes les contraintes de précédence doivent être respectées : si $i \prec j$, alors soit les tâches i et j sont affectées au même poste, soit i est affectée à un poste situé avant celui de la tâche j sur la ligne. Ainsi, l'ensemble des tâches doit être réparti en ensembles disjoints S_k, où $k \in \mathcal{M}$. Étant donné une tâche $i \in S_a$ et une tâche $j \in S_b$, la contrainte de précédence $i \prec j$ impose que $a \leq b$. Le temps de traitement total du poste k, noté $t(S_k)$, correspond à la somme des temps de traitement des tâches qui lui sont affectées. Si un temps de cycle fixe c est défini, une solution est considérée comme faisable si $t(S_k) \leq c, \forall k \in \mathcal{M}$.

Selon la fonction objectif choisie, différentes variantes du problème SALBP peuvent être définies :

- Dans le problème SALBP-1, le temps de cycle c est donné, et l'objectif est de minimiser le nombre de postes de travail m.
- Dans le problème SALBP-2, le nombre de machines m est donné, et l'objectif est de minimiser le temps de cycle c.
- Dans le problème SALBP-F, le nombre de machines m et le temps de cycle c sont tous les deux donnés, et l'objectif est de déterminer s'il existe une solution faisable pour ces valeurs.
- Dans le problème SALBP-E, l'objectif est de maximiser l'efficacité, ce qui revient à minimiser le produit du nombre de machines m et du temps de cycle c [16].

Dans cet article, on s'intéresse au problème SALBP avec minimisation du pic de consommation d'énergie (SALB3PM). Cette nouvelle variante du problème SALBP a pour objectif de répondre davantage aux problématiques énergétiques de notre époque. Dans le problème SALB3PM, le nombre de machines m et le temps de cycle c sont donnés. De plus, chaque tâche j possède maintenant une consommation d'énergie constante par unité de temps w_j, qui est indépendante de sa station d'affectation. Il est donc désormais possible de calculer, pour chaque unité de temps t, la consommation énergétique totale sur cette unité de temps, qui est la somme des consommations énergétiques des tâches en cours d'exécution à cet instant. L'objectif du problème SAL3BPM est d'ordonnancer chacune des tâches afin de minimiser le pic de consommation énergétique.

Exemple 2 *On considère une instance du problème SALB3PM avec $m = 3$ postes de travail et un temps de cycle de $c = 6$ unités de temps. Le tableau 1 indique, pour chaque tâche, sa durée d'exécution et sa consommation énergétique par unité de temps. Le graphe de précédence est celui précédemment décrit dans la figure 1.*

Tâche	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Durée d'exécution	2	3	2	2	3	2	2
Consommation d'énergie	10	20	10	20	20	20	10

Table 1: Durée d'exécution et consommation énergétique de chaque tâche

Un exemple de solution qui respecte toutes les contraintes du problème SAL3BPM est représenté dans la figure 2, où chaque rectangle représente une tâche dont la hauteur est sa consommation énergétique et sa longueur est sa durée d'exécution. Comme indiqué dans la figure 2 :

- *Au temps 0, les tâches 1, 3 et 4 commencent, ce qui implique une consommation d'énergie de 40 par unité de temps.*
- *Au temps 2, les tâches 1, 3 et 4 se terminent, alors que les tâches 2, 5 et 6 commencent. Cela augmente la consommation d'énergie à 60 par unité de temps.*
- *Au temps 4, la tâche 6 se termine alors que la tâche 7 commence. Cela diminue la consommation d'énergie à 50 par unité de temps.*
- *Au temps 5, les tâches 2 et 5 terminent. Cela diminue fortement la consommation d'énergie à 10 par unité de temps.*

Un exemple de solution optimale pour cette instance est proposé dans la figure 4, où l'on peut constater que les trois tâches avec la plus grosse consommation d'énergie (les tâches 2, 5 et 7) ne sont cette fois plus exécutées en même temps. On peut aussi noter que la consommation énergétique y est plus lissée, c'est à dire que la différence entre le pic de consommation énergétique

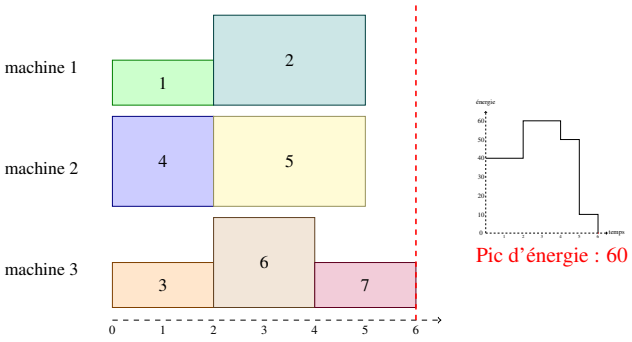


Figure 2: Solution réalisable pour le problème SALB3PM

(le moment où la consommation est la plus grande) et le creux de consommation énergétique (le moment où la consommation est la plus faible) est plus petite que dans la solution précédemment proposée. En outre, introduire des temps d'inactivité pour les machines semble être parfois nécessaire pour minimiser le pic de consommation, ce qui implique l'obtention d'ordonnancements non semi-actifs.

3 Résolution de SALB3PM grâce à une formulation MaxSAT

Dans cette section, on présente la formulation MaxSAT du problème SALB3PM initialement proposée dans [18]. Parmi les deux modèles proposés dans l'article original, on utilisera ici uniquement le deuxième modèle, basé sur une représentation binaire et dont les résultats expérimentaux donnent les meilleurs performances. Pour plus de détails, le lecteur peut se référer à l'article [18]. On rappelle que le problème de satisfiabilité maximum (MaxSAT) est l'extension naturelle de la satisfiabilité (SAT) en problème d'optimisation [12]. Dans la suite, on utilisera plus particulièrement la variante partielle pondérée de MaxSAT qui prend en entrée une formule bipartite pondérée $\phi = D \cup S$ contenant des variables booléennes, où D est l'ensemble des clauses dures qui doivent être satisfaites, comme dans SAT et S est l'ensemble des clauses souples pondérées (c, w_c) avec w_c un poids entier positif associé à la clause c. L'objectif de MaxSAT partiel pondéré est de trouver une affectation des variables de la formule qui maximise la somme des poids des clauses souples satisfaites tout en satisfaisant toutes les clauses dures. Avant d'introduire le modèle MaxSAT, on mentionne également l'existence de contraintes pseudo-booléennes ayant la forme $\sum_j a_j \cdot l_j \circ b$ où $a_j \in \mathbb{N}$, $b \in \mathbb{N}$, l_j est un littéral et $\circ \in \{=, \geq, \leq\}$ et pouvant être efficacement encodées sous forme clausale [15].

Pour introduire le modèle MaxSAT, le temps est discrétisé en l'ensemble des unités de temps $T = \{0, 1, \ldots, c-1\}$. On note $T^i = \{0, ..., c - t_i\}$ l'ensemble des unités de temps où il est possible de faire débuter la tâche $i \in O$. La valeur du pic de consommation énergétique appartient forcément à l'intervalle $[LB, UB]$, où LB est une borne inférieure calculée à partir de la plus haute

consommation énergétique parmi toutes les tâches et UB est une borne supérieure calculée à partir des m plus hautes consommations énergétiques. Plus formellement, on $LB = \max_{i \in \mathcal{O}} w_i$ et $UB = \sum_{w_i \in W_m} w_i$.

La formulation MaxSAT utilise quatre ensemble de variables de décision booléennes :

- **Les variables d'affectations** $X_{i,s}$ qui, pour une tâche $i \in \mathcal{O}$ et une machine $s \in \mathcal{M}$, sont affectées à $Vrai$ si la tâche i est assignée à la machine s.
- **Les variables d'ordonnancement** $B_{i,t}$ qui, pour une tâche $i \in \mathcal{O}$ et pour une unité de temps $t \in T$, sont affectées à $Vrai$ si la tâche i commence à l'unité de temps t.
- **Les variables d'activité** $A_{i,t}$ qui, pour une tâche $i \in \mathcal{O}$ et une unité de temps $t \in T$, sont affectées à $Vrai$ si la tâche i est en cours d'exécution à l'unité de temps t.
- **Les variables d'énergie** $binU_j$ qui encodent la valeur du pic de consommation énergétique. Chaque variable $binU_j$, pour une valeur $j \in \{0, \ldots, \lceil \log_2 UB \rceil\}$, représente la valeur du i$^{\text{ème}}$ bit du pic d'énergie de la solution. Le pic de consommation énergique W_{peak} est calculé de la manière suivante :

$$W_{peak} = \sum_{j \in [0,\ \lceil \log_2 UB \rceil]} 2^j \cdot binU_j \qquad (1)$$

Le modèle MaxSAT est représenté en figure 3 avec les clauses souples permettant de minimiser le pic de consommation énergétique les clauses dures garantissant le respect des contraintes intrinsèques du problème :

- Les clauses souples pondérées (ligne 2) comptabilisent lors du pic d'énergie la quantité d'énergie économisée par rapport à la borne supérieure UB (et l'objectif est de maximiser cette quantité, autrement dit de minimiser le pic de consommation énergétique).
- Les clauses dures permettent de :
  - garantir que chaque tâche soit affectée à exactement une machine (contrainte 3).
  - garantir le respect des contraintes de précédence (contraintes 4 et 5).
  - garantir que chaque tâche débute une et une seule fois et seulement dans l'intervalle de temps qui lui permet de terminer avant la fin du temps de cycle (contraintes 6 et 7).
  - lier les variables d'ordonnancement et les variables d'activité (contrainte 8).
  - garantir le non-chevauchement des tâches (contrainte 9).
  - gérer le calcul du pic de consommation énergétique (contraintes 10 et 11), avec $UB' = \sum_{j=0}^{\lceil \log_2 UB \rceil} 2^j$.

Clauses souples:

$$(\ \overline{binU_j}\ ,\ 2^j\ ) \qquad \forall j \in [0,\ \lceil \log_2 UB \rceil] \qquad (2)$$

Clauses dures:

$$\sum_{s \in \mathcal{M}} X_{i,s} = 1 \qquad \forall i \in \mathcal{O} \qquad (3)$$

$$\overline{X_{i,s1}} \vee \overline{X_{j,s2}} \qquad \forall (i,j,s_1,s_2) \in \mathcal{O}^2 \times \mathcal{M}^2 \text{ s.t } i \prec j \text{ and } s1 > s2 \qquad (4)$$

$$\overline{X_{i,s}} \vee \overline{X_{j,s}} \vee \overline{B_{i,t_1}} \vee \overline{B_{j,t_2}} \qquad \forall (i,j,s,t_1,t_2) \in \mathcal{O}^2 \times \mathcal{M} \times T^2 \text{ s.t } i \prec j \text{ and } t_1 > t_2 \qquad (5)$$

$$\sum_{t \in T} B_{i,t} = 1 \qquad \forall i \in \mathcal{O} \qquad (6)$$

$$\overline{B_{i,t}} \qquad \forall (i,t) \in \mathcal{O} \times T \setminus T^i \qquad (7)$$

$$\overline{B_{i,t}} \vee A_{i,t+\epsilon} \qquad \forall (i,t,\epsilon) \in \mathcal{O} \times T^i \times \{0, \ldots, t_i - 1\} \qquad (8)$$

$$\overline{X_{i,s}} \vee \overline{X_{j,s}} \vee \overline{A_{i,t}} \vee \overline{A_{j,t}} \qquad \forall (i,j,s,t) \in \mathcal{O}^2 \times \mathcal{M} \times T \text{ s.t } i < j \qquad (9)$$

$$\sum_{j \in [0,\ \lceil \log_2 UB \rceil]} 2^j binU_j \geq LB \qquad \forall t \in T \qquad (10)$$

$$\sum_{i \in \mathcal{O}} W_i * A_{i,t} + \sum_{j \in [0,\ \lceil \log_2 UB \rceil]} 2^j * \overline{binU_j} \leq UB' \qquad \forall t \in T \qquad (11)$$

Figure 3: Modélisation MaxSAT du problème SALB3PM

4 Résolution métaheuristique de SALBP3PM

Trouver des solutions optimales pour le problème SALB3PM est difficile [8, 14] ; c'est pourquoi nous avons étudié une approche métaheuristique. La méthode proposée repose sur une méthode à redémarrages multiples qui combine une génération semi-aléatoire de solutions et une recherche locale utilisant trois structures de voisinage différentes. L'approche proposée est présentée comme suit. La section 4.1 introduit l'approche proposée à démarrages multiples. Les sections 4.2 et 4.3 décrivent respectivement comment une solution est représentée et évaluée. La section 4.4 présente l'heuristique constructive semi-aléatoire utilisée pour générer les solutions initiales. Dans la section 4.5, la structure de voisinage est définie. Enfin, la section 4.6 décrit comment la recherche locale est effectuée sur les solutions générées.

4.1 Présentation de la métaheuristique

À l'origine, les procédures à redémarrages multiples étaient utilisées pour exploiter une procédure de recherche locale. Plusieurs solutions initiales sont générées aléatoirement, puis la recherche locale est appliquée [13]. L'algorithme 1 présente les différentes étapes d'une telle procédure pour un problème de minimisation. La procédure prend en entrée une fonction d'évaluation $f(\cdot)$ ainsi qu'un nombre maximal d'itérations. Tout d'abord, l'algorithme considère qu'il n'a aucune meilleure solution connue avec un pic de consommation énergétique arbitrairement infini (ligne 2). Itérativement, une valeur aléatoire est tirée à partir d'une distribution uniforme. Cette valeur est utilisée dans la fonction d'évaluation (ligne 4) et son rôle sera décrit dans la section 4.3. Une solution aléatoire est générée,

Algorithm 1 Approche à redémarrages multiples

```
1:  function MÉTAHEURISTIQUE(f(·), nb_runs)
2:      S ← ∅, f(S) ← ∞
3:      while on a fait moins de nb_runs itérations do
4:          W ← U(0, max_{j∈O}(w_j))
5:          S_courante ← Generer_solution()
6:          S_courante ← Rech._Locale(S_courante)
7:          if f(S_courante) < f(S) then
8:              S ← S_courante
9:          end if
10:     end while
11:     return S
12: end function
```

puis une procédure de recherche locale est appliquée à cette solution (lignes 5 et 6). Si la solution courante trouvée par la recherche locale est meilleure que la meilleure solution trouvée, celle-ci est mise à jour (lignes 7 et 9). La boucle allant de la ligne 3 à la ligne 10 est exécutée autant de fois qu'indiqué par le paramètre 'nb_runs'. Enfin, la meilleure solution est retournée à la fin de l'algorithme.

4.2 Représentation d'une solution

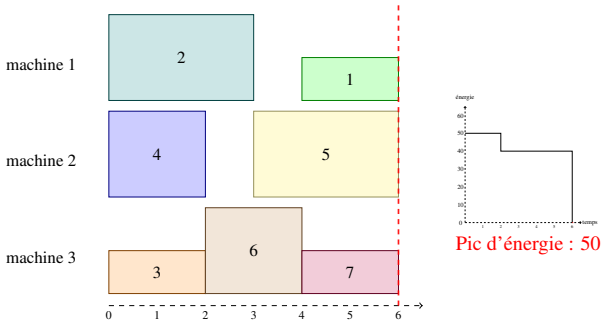
Représenter une solution est un facteur clé pour appliquer avec succès une métaheuristique à un problème. Dans l'approche proposée, nous considérons une représentation indirecte d'une solution. Ce type de représentation a été utilisé dans [11], où un vecteur d'affectation et un vecteur de séquencement des opérations étaient employés. Nous étendons cette représentation en introduisant un troisième vecteur qui constitue notre contribution. Plus précisément, nous utilisons trois vecteurs pour représenter une solution (comme illustré dans la figure 4) :

- Le premier vecteur, V, contient l'ordre d'exécution des tâches.
- Le deuxième vecteur, A, est un vecteur d'affectation qui précise le poste de travail attribué à chaque tâche.
- Le troisième vecteur, D, contient le délai de chaque tâche. Le délai d'une tâche j est défini comme le temps d'inactivité pendant lequel la tâche j est différée après la fin de la tâche précédente sur le même poste de travail.

Le processus de décodage des trois vecteurs d'une solution utilise le vecteur A pour affecter toutes les tâches aux postes de travail. L'ordre des tâches au sein des postes de travail suit l'ordre défini dans V, et le délai de chaque tâche est donné par D. À l'issue de cette procédure de décodage, il est possible de déterminer les instants de début et de fin de chaque tâche.

4.3 Évaluation d'une solution

La fonction objectif du problème est de minimiser le pic de consommation énergétique. Étant donné que des solutions violant la contrainte de temps de cycle peuvent être générées par notre algorithme, on utilise une fonction



Ordre (V)	2	1	5	3	4	6	7
Affectation (A)	1	1	3	2	2	3	3
Délai (D)	1	0	0	0	1	0	0

Figure 4: Exemple de solution optimale et de sa représentation

d'évaluation pénalisant les solutions non réalisables. La pénalité associée à une solution s est calculée comme le temps total par lequel le temps de cycle est dépassé. L'équation 12 résume le calcul de la pénalité, où d_j représente le délai de la tâche j.

$$penalty(s) = \sum_{k \in \mathcal{M}} \max(0, \sum_{j \in S_k} (t_j + d_j) - c) \qquad (12)$$

L'équation 13 définit la fonction d'évaluation. Soit $power\_peak(s)$ le pic de consommation énergétique de la solution s, $penalty(s)$ la pénalité calculée selon l'équation 12, et W un facteur de pénalité tiré d'une distribution uniforme $U(0, \max_{j \in \mathcal{O}}(w_j))$. La valeur de W est mise à jour à chaque itération, comme indiqué à la ligne 4.

$$f(s) = power\_peak(s) + penalty(s) \cdot W \qquad (13)$$

4.4 Génération d'une solution initiale

Les heuristiques constructives basées sur des règles de priorité pour SALBP ont été largement étudiées dans la littérature [17, 6]. Les méthodes basées sur des priorités utilisent un poids associé à chaque tâche pour l'ordonner. Comme l'algorithme à redémarrages multiples dépend de solutions initiales à la fois variées et de bonne qualité, nous utilisons une version semi-aléatoire de l'heuristique constructive basée sur le temps de traitement le plus long pour générer les solutions initiales [6].

Pour chaque tâche, la règle de priorité est calculée selon l'équation 14, où $x \in [0, 1]$ est une valeur aléatoire générée pour chaque tâche, et où le paramètre $\alpha \in [0, 1]$ contrôle le degré d'aléatoire de la fonction de priorité. Les durées de traitement minimale et maximale sont respectivement notées t_{min} et t_{max}. Pour $\alpha = 0$, la priorité est entièrement aléatoire, tandis que pour $\alpha = 1$, la priorité est purement gloutonne.

$$priority(j) = (1 - \alpha)x + \alpha \frac{t_j - t_{min}}{t_{max} - t_{min}} \qquad (14)$$

La solution générée respecte les contraintes de précédence, le délai est nul pour toutes les tâches, et tous les postes de travail, à l'exception du dernier, respectent la contrainte de temps de cycle. Pour comprendre la procédure d'affectation, certaines définitions sont nécessaires. Une tâche est dite disponible si toutes ses tâches prédécesseures directes ont été affectées. Une tâche j peut être affectée au poste de travail k si :

- Tous ses prédécesseurs ont été affectés aux postes $1, \ldots, k$, et
- Le temps total de traitement du poste k, additionné à la durée de la tâche j, ne dépasse pas le temps de cycle.

Toutefois, pour le dernier poste de travail, la contrainte de temps de cycle est relâchée. La procédure d'affectation est représentée dans l'algorithme 2 [6, 9].

À la ligne 2, la liste des tâches à affecter ($\mathcal{T}$), le compteur de postes de travail (k) et le temps disponible (CT) sont initialisés. La boucle des lignes 3 à 16 se poursuit tant que $\mathcal{T}$ n'est pas vide. À la ligne 4, la liste des tâches disponibles, LCT, est mise à jour avec toutes les tâches pouvant être affectées au poste k. Si LCT n'est pas vide, aux lignes 6 à 9, la tâche ayant la priorité la plus élevée, notée $next$, est sélectionnée dans LCT. Cette tâche est ensuite retirée de $\mathcal{T}$ et affectée au poste k. Le temps disponible CT est décrémenté de w_{next}. Si LCT est vide, le temps disponible CT est réinitialisé à la ligne 11. Si le nombre de postes ouverts est inférieur à m (c'est-à-dire $k < m$), le compteur de postes k est incrémenté à la ligne 13. Pour le dernier poste de travail ($k = m$), les tâches peuvent y être affectées sans respecter la contrainte de temps de cycle.

Algorithm 2 Procédure d'affectation

1: **function** GÉNÉRER_SOLUTION($f(\cdot), nb\_runs$)
2: $\mathcal{T} \leftarrow \{1, \ldots, n\},\ k \leftarrow 1,\ CT \leftarrow c$
3: **while** $\mathcal{T} \neq \emptyset$ **do**
4: $LCT = \{j \in \mathcal{T}, t_j \leq CT,$ j est disponible$\}$
5: **if** $LCT \neq \emptyset$ **then**
6: $next \leftarrow$ tâche prioritaire dans LCT
7: $LCT \leftarrow LCT \setminus \{next\}$
8: Affecter $next$ au poste de travail k
9: $CT \leftarrow CT - w_{next}$
10: **else**
11: $CT \leftarrow c$
12: **if** k < m **then**
13: $k \leftarrow k + 1$ ▷ Ouverture de poste
14: **end if**
15: **end if**
16: **end while**
17: **end function**

4.5 Structure de voisinage

Trois structures de voisinage sont considérées dans l'approche actuelle. La première est basée sur l'insertion de tâches ($\mathcal{N}_{insertion}$), la deuxième sur l'échange de tâches ($\mathcal{N}swap$), et la troisième sur l'augmentation du délai d'une tâche ($\mathcal{N}_{delay}$). Les deux premiers mouvements peuvent violer les contraintes de temps de cycle.

4.5.1 Insertion de tâche

Étant donnée une solution S et une tâche j :

- la position du prédécesseur le plus proche de la tâche j dans $\mathcal{O}$ est notée $pred_{near}(j)$, et son poste de travail est $ws_{pred}(j)$. Si la tâche j n'a pas de prédécesseurs, alors $pred_{near}(j) = 0$ et $ws_{pred}(j) = 1$;
- la position du successeur le plus proche de la tâche j dans $\mathcal{O}$ est notée $suc_{near}(j)$, et son poste de travail est $ws_{suc}(j)$. Si la tâche j n'a pas de successeurs, alors $suc_{near}(j) = n$ et $ws_{suc}(j) = m$.

Une solution voisine est obtenue en retirant la tâche j de sa position actuelle, puis en l'insérant dans l'intervalle $[pred_{near}(j) + 1, , suc_{near}(j) - 1]$, tout en l'affectant à un poste de travail dans l'intervalle $[ws_{pred}(j), , ws_{suc}(j)]$. Le retard de la tâche j est alors réinitialisé à zéro. Ce voisinage est de taille $O(mn^2)$.

4.5.2 Échange de deux tâches

Étant donnée une solution S et une tâche j_1, on calcule $pred_{near}(j_1)$ et $suc_{near}(j_1)$ comme décrit dans la Section 4.5.1. On sélectionne une tâche $j_2 \in [pred_{near}(j_1) + 1, , suc_{near}(j_1) - 1]$, telle que $j_1 \in [pred_{near}(j_2) + 1, , suc_{near}(j_2) - 1]$. Une solution voisine est obtenue en échangeant les positions de j_1 et j_2 dans le vecteur V ainsi que leurs affectations aux postes de travail. Les retards des tâches j_1 et j_2 sont alors réinitialisés à zéro. Ce voisinage est de taille $O(n^2)$.

4.5.3 Augmenter le délai d'une tâche

Étant donnée une solution faisable S et un poste de travail k dont le temps total de traitement est inférieur au temps de cycle c, on sélectionne une tâche j dans le poste de travail k. Une solution voisine est obtenue en incrémentant le délai de la tâche j. L'incrémentation ne doit pas faire terminer la dernière tâche du poste de travail k après le temps de cycle. Ce voisinage est de taille $O(nc)$.

4.6 Recherche Locale

Une procédure de recherche locale est utilisée pour améliorer les solutions générées. Les trois voisinages décrits ci-dessus sont employés dans la recherche locale. Une stratégie de première amélioration est adoptée (nommée $first\_improvement$). Pour chaque voisinage, étant donnée une solution, ses voisins sont évalués dans un ordre aléatoire jusqu'à ce qu'une solution améliorante soit trouvée. Cela signifie que si la solution initiale est un optimum local pour un voisinage donné, tous ses voisins seront évalués. La recherche locale s'arrête lorsqu'un optimum local est trouvé pour les trois voisinages.

Cette procédure est représentée dans l'algorithme 3. L'algorithme fonctionne comme suit. Une solution S et

Algorithm 3 Algorithme de recherche locale

```
1: function RECHERCHE_LOCALE(S, f(·))
2:     eval_begin ← f(S), is_local_optimum ← false
3:     while !is_local_optimum do
4:         if x > 0.5 | x ∼ U(0,1) then
5:             S ← first_improvement(S, N_swap)
6:             S ← first_improvement(S, N_insertion)
7:         else
8:             S ← first_improvement(S, N_insertion)
9:             S ← first_improvement(S, N_swap)
10:        end if
11:        if S est une solution faisable then
12:            S ← first_improvement(S, N_delay)
13:        end if
14:        if f(S) = eval_begin then
15:            is_local_optimum ← 1
16:        else
17:            eval_begin ← f(S)
18:        end if
19:    end while
20:    return S
21: end function
```

une fonction d'évaluation $f(\cdot)$ sont fournies en entrée. La boucle des lignes 3 à 19 continue jusqu'à ce qu'aucune solution voisine améliorante ne soit trouvée. À l'intérieur de cette boucle, la ligne 4 sélectionne aléatoirement l'ordre dans lequel $\mathcal{N}_{insertion}$ ou $\mathcal{N}_{swap}$ est appliqué. Quelque soit cet ordre, la fonction $first\_improvement$ retourne le premier voisin de S dont la valeur de la solution est meilleur que S. Ensuite, si S est une solution réalisable, le voisinage $\mathcal{N}_{delay}$ est utilisé à la ligne 12 pour tenter d'en améliorer la valeur. Les lignes 14 à 18 vérifient si aucun des trois voisinages n'a permis de trouver une meilleure solution, auquel cas, étant coincé dans un optimum local, la recherche locale est arrêtée et la meilleure solution est renvoyée.

5 Expérimentations

Un ensemble d'instances de référence[1], étendu à partir de travaux précédents, a été utilisé dans nos expérimentations. Il contient 130 instances, réparties en 13 familles de 10 instances chacune. Pour chaque famille, 5 instances possèdent un nombre fixe de postes de travail m, et le temps de cycle c est calculé à l'aide d'un solveur SALBP-2 (ces instances sont appelées *normales*) ; les 5 autres instances (appelées *étendues*) ont été générées avec $c' = \lceil 1{,}3 \cdot c \rceil$. La consommation énergétique w_j de chaque tâche a été générée à partir d'une distribution uniforme $U(5, 50)$ [8, 14, 18]. Le tableau 2 résume les caractéristiques des instances. Les deux premières colonnes donnent le nom de la famille d'instances et le nombre de tâches. Le nombre minimum et maximum de machines, ainsi que le temps de cycle, sont également fournis, séparés par un tiret.
La métaheuristique proposée a été implémentée en C++ et compilée avec GNU C++ 13.2.0 en utilisant les options -O3 et -march=native. La méthode MaxSAT, présentée dans la section 3 et issue de [18], a été implémentée en Python et utilise le package PySAT [10]. La méthode est lancée avec un solveur MaxSAT exact (afin de voir les instances pour lesquelles la valeur de la solution optimale est trouvée), ainsi qu'un solveur heuristique :

- **MaxHS** [4] (solveur MaxSAT exact) : Ce solveur repose sur une architecture hybride combinant les techniques modernes de résolution SAT avec des méthodes d'optimisation en programmation entière.
- **NuWLS-c** [3] (solveur MaxSAT heuristique) : Ce solveur utilise une approche par portfolio combinant deux moteurs : une recherche locale MaxSAT avancée et le MaxSAT TT-Open-WBO-Inc. La version que nous avons utilisée a remporté la première place dans les quatre catégories incomplètes du l'évaluation MaxSAT 2023 [2].

L'ensemble des expériences a été réalisé sur la plateforme MatriCS[3], sur un serveur fonctionnant sous Rocky Linux 8.6, équipé de processeurs Intel Xeon E5-2680 v4 cadencés jusqu'à 2,40 GHz. Un seul thread a été utilisé pour toutes les expériences. Pour chaque instance, la métaheuristique proposée a été exécutée 10 fois. La valeur α du générateur de solutions initiales a été fixée à 0,3. Pour la méthode exacte, la limite de temps a été fixée à 7200 secondes, et pour les méthodes heuristiques (MaxSAT avec NuWLS-c et la métaheuristique introduite), elle a été fixée à 1000 secondes.

Le Tableau 3 résume les résultats sur deux catégories d'instances : les instances dont les solutions optimales sont connues (OPT) totalisant 83 instances, et l'ensemble complet des 130 instances (ALL). Pour l'ensemble complet, la métaheuristique est comparée avec la valeur de la solution optimale. Pour les instances dont les solutions optimales sont connues, les résultats de la méta-heuristique ($MetaH$) et de la résolution MaxSAT heuristique sont comparés, avec les métriques suivantes :

- Les écarts relatifs maximum (gap.max) et moyen (gap.avg) par rapport à la meilleure solution connue pour chaque instance;
- Le pourcentage d'instances pour lesquelles chaque méthode a obtenu :
  - la meilleure solution (#best)
  - la meilleure solution de façon consistante sur tous les lancements (#best10)
  - au moins une solution faisable (#feasible)
  - des solutions faisables à chaque lancement (#feasible10)

[1] https://github.com/ZZFreya/SALB3PM/tree/main/benchmarks/1993

[2] https://maxsat-evaluations.github.io/

[3] https://www.matrics.u-picardie.fr

Table 2: Résumé des résultats des heuristiques sur chaque famille d'instances

name	#tasks	#machines	normal					extended				
			cycle	$\overline{gap}$ (%)	$\bar{t}$ (s)	$\#iterations$	feas. (%)	cycle	$\overline{gap}$ (%)	$\bar{t}$ (s)	$\#iterations$	feas. (%)
MERTENS	7	2 - 6	6 - 15	0.00	0.0	1.8	100.0	8 - 20	0.00	0.3	1122.8	100.0
BOWMAN8	8	3 - 7	17 - 28	0.00	0.0	2.8	100.0	23 - 37	3.87	0.1	271.5	100.0
JAESCHKE	9	4 - 8	6 - 10	0.00	0.0	5.1	100.0	8 - 13	5.63	4.1	16119.5	100.0
JACKSON	11	2 - 6	9 - 23	0.00	89.0	382136.4	100.0	12 - 30	1.38	8.6	19664.6	100.0
MANSOOR	11	1 - 5	45 - 185	0.00	0.1	103.0	100.0	59 - 241	9.75	24.0	40786.4	100.0
MITCHELL	21	3 - 7	16 - 35	0.36	35.0	31970.8	100.0	21 - 46	3.90	56.2	33399.2	100.0
ROSZIEG	25	5 - 9	16 - 26	1.71	139.5	96666.8	100.0	21 - 34	-7.57	270.7	112665.2	100.0
HESKIA	28	2 - 6	171 - 512	-1.65	385.6	48887.4	100.0	223 - 666	-31.68	303.3	8962.0	100.0
BUXEY	29	8 - 12	28 - 41	2.03	315.7	255088.0	100.0	37 - 54	-19.28	395.2	41585.9	100.0
SAWYER30	30	8 - 12	28 - 41	3.74	465.6	348708.0	100.0	37 - 54	-17.86	365.5	30136.5	100.0
GUNTHER	35	8 - 12	44 - 63	2.31	392.7	152396.7	100.0	58 - 82	-31.86	399.4	34261.2	100.0
WARNECKE	58	17 - 21	76 - 92	-	-	-	0.0	99 - 120	-35.99	428.4	3458.1	100.0
LUTZ2	89	22 - 26	19 - 23	-14.13	471.3	61436.6	80.0	25 - 30	-37.60	440.6	6246.6	100.0
				-0.21	**184.8**	**117344.3**	**90.8**		**-12.10**	**207.4**	**26821.5**	**100.0**

Métrique (%)	**ALL**	**OPT**	
	MetaH	**MetaH**	**MaxSAT**
gap.max	24.07	**24.07**	107.32
gap.avg	-6.50	**2.31**	4.27
#best	63.08	51.81	**80.72**
#best10	62.31	50.60	-
#feasible	95.38	**100.00**	**100.00**
#feasible10	95.38	100.00	-

Table 3: Résumé des résultats des heuristiques sur les 130 instances

Le Tableau 2 détaille les résultats de la métaheuristique sur chaque famille d'instances. Pour les instances normales et étendues, nous fournissons l'écart moyen entre la solution trouvée par la métaheuristique et la meilleure solution connue, le temps moyen ($\bar{t}$) mis par la métaheuristique pour trouver sa meilleure solution et le nombre moyen d'itérations ($\#iterations$) nécessaire à la métaheuristique pour trouver sa meilleure solution. Nous indiquons également le pourcentage de solutions réalisables trouvées par la métaheuristique.

En ce qui concerne les instances dont les solution optimales sont connues, nous fournissons des comparaisons détaillées entre notre méthode proposée et la formulation MaxSAT résolue via une approche heuristique, avec des métriques calculées par rapport aux valeurs des solutions optimales connues. Pour les autres instances, les métriques sont calculées en comparant les performances de notre méthode aux meilleures solutions obtenues par la formulation MaxSAT, qu'elle soit résolue exactement ou heuristiquement.

Les résultats expérimentaux présentés dans le Tableau 3 montrent que la métaheuristique proposée a égalé ou amélioré les meilleures solutions connues lors des 10 exécutions indépendantes pour 81 instances (soit 62,31% des instances). Si l'on considère les cas où la meilleure solution connue a été atteinte ou dépassée dans au moins une exécution, ce nombre passe à 82 instances (soit 63,08% des instances). En ce qui concerne la faisabilité, l'algorithme a trouvé des solutions réalisables dans toutes les exécutions pour 124 instances (95,38% des instances), et n'a pas produit de solutions réalisables pour 6 instances difficiles (4,62% des instances). L'analyse comparative montre que, pour les instances avec des solutions optimales connues, notre méthode surpasse le solveur heuristique MaxSAT en termes de gaps moyens et maximums d'optimalité. Toutefois, le solveur MaxSAT heuristique a identifié des solutions optimales pour 28,92% d'instances supplémentaires. Une analyse plus approfondie du Tableau 2 révèle que les performances de la métaheuristique s'améliorent considérablement pour les instances de plus grande taille, avec des gains moyens d'au moins 17% par rapport à la meilleure solution connue pour les cas comportant plus de 25 tâches et des temps de cycle étendus.

6 Conclusion

Dans cet article, nous avons proposé une approche métaheuristique pour le problème d'équilibrage de ligne d'assemblage avec minimisation des pics de consommation énergétique (SALB3PM). Nous nous sommes intéressés à deux méthodes : une métaheuristique à redémarrages multiples contenant une recherche locale exploitant trois structures de voisinage ; et une formulation MaxSAT issue de travaux précédents [18]. Les résultats expérimentaux montrent que la nouvelle métaheuristique est efficace pour trouver des solutions de haute qualité, en égalant ou en améliorant la meilleure solution connue pour 62,31% des instances lors de toutes les exécutions, et en obtenant des solutions réalisables pour 95,38% des instances. De plus, pour les instances comportant plus de 25 tâches et des temps de cycle étendus, la métaheuristique améliore la meilleure solution connue d'au moins 17%. Les approches MaxSAT ont quant à elles des résultats qui se confirment pertinents par comparaison avec la métaheuristique.

Une partie des travaux futurs concerne l'amélioration de la métaheuristique via l'ajout d'autres structures de voisinage afin d'examiner de nouveaux espaces de solutions, la prise en compte de l'historique des solutions pour accélérer le processus de recherche, et l'étude de représentations

alternatives des solutions. Nous prévoyons également de réguler automatiquement le paramètre α au cours de la recherche, afin d'équilibrer dynamiquement l'exploration et l'exploitation. Par ailleurs, pour les instances difficiles où il est compliqué de trouver des solutions faisables, une piste consiste à relâcher la contrainte sur le nombre de postes de travail afin de faciliter la recherche. Enfin, compte tenu de la pertinence à la fois de la métaheuristique et de l'approche MaxSAT et du comportement différent de chacun en fonction de la famille d'instances étudiée, nous comptons explorer l'hybridation entre les métaheuristiques et les méthodes exactes. Une direction possible serait de résoudre le sous-problème de détermination des délais des tâches à l'aide d'une formulation MaxSAT, pour obtenir des solutions plus précises et plus efficaces.

Remerciements

Ce travail a été pour partie financé grâce à l'appui du Centre International de Recherche "Innovative Transportation and Production Systems". Ce travail a également bénéficié d'un accès aux ressources de calcul de haute performance (HPC) de la plateforme MatriCS de l'Université de Picardie Jules Verne, cofinancée par l'Union européenne via le Fonds Européen de Développement Régional (FEDER) et par le Conseil régional des Hauts-de-France.

References

[1] IEA (2024). World Energy Outlook 2024. Technical Report Annex A, International Energy Association, Paris, 2024.

[2] İlker Baybars. A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem. *Management Science*, 32(8):909–932, August 1986.

[3] Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls-c-2023: Solver description. *MaxSAT Evaluation 2023*, page 23, 2023.

[4] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by Solving a Sequence of Simpler SAT Instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *LNCS*, pages 225–239. Springer, 2011.

[5] Xavier Delorme, Paolo Gianessi, and Damien Lamy. A new Decoder for Permutation-based Heuristics to Minimize Power Peak in the Assembly Line Balancing. *IFAC-PapersOnLine*, 56(2):3704–3709, 2023.

[6] Masood Fathi, Dalila Benedita Machado Martins Fontes, Matias Urenda Moris, and Morteza Ghobakhloo. Assembly line balancing problem: A comparative evaluation of heuristics and a computational assessment of objectives. *Journal of Modelling in Management*, 13(2):455–474, May 2018.

[7] Thiago Giachetto de Araujo, Matthieu Py, Laurent Deroussi, and Nathalie Grangeon. A metaheuristic for the Simple Assembly Line Balancing Problem with Power Peak Minimization. In *26ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF 2025)*, 2024.

[8] Paolo Gianessi, Xavier Delorme, and Oussama Masmoudi. Simple Assembly Line Balancing Problem with Power Peak Minimization. In Farhad Ameri, Kathryn E. Stecke, Gregor von Cieminski, and Dimitris Kiritsis, editors, *Advances in Production Management Systems. Production Management for the Factory of the Future*, volume 566, pages 239–247, Austin, TX, United States, September 2019. Springer International Publishing.

[9] Michel Gourgand, Nathalie Grangeon, and Sylvie Norre. Metaheuristics based on Bin Packing for the line balancing problem. *RAIRO - Operations Research*, 41(2):193–211, April 2007.

[10] Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.

[11] Damien Lamy, Xavier Delorme, and Paolo Gianessi. Line Balancing and Sequencing for Peak Power Minimization. *IFAC-PapersOnLine*, 53(2):10411–10416, 2020.

[12] Chu Min Li and Felip Manyà. MaxSAT, Hard and Soft Constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2021.

[13] Rafael Martí, Ricardo Aceves, Maria Teresa León, Jose M. Moreno-Vega, and Abraham Duarte. Intelligent Multi-Start Methods. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, pages 221–243. Springer International Publishing, Cham, 2019.

[14] Matthieu Py, Arnauld Tuyaba, Laurent Deroussi, Nathalie Grangeon, and Sylvie Norre. Application of SAT to the Simple Assembly Line Balancing Problem with Power Peak Minimization. In *15th Pragmatics of SAT international workshop, a workshop of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, 2024.

[15] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1087–1129. IOS Press, 2021.

[16] Armin Scholl and Christian Becker. State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, 168(3):666–693, February 2006.

[17] Armin Scholl and Stefan Voss. Simple assembly line balancing - Heuristic approaches. *Journal of Heuristics*, 2(3):217–244, 1997.

[18] Zhifei Zheng, Sami Cherif, and Rui Sá Shibasaki. Optimizing Power Peaks in Simple Assembly Line Balancing Through Maximum Satisfiability. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 363–370, 2024.