Université de Franche-Comté
Master 2 I2A année 2020 / 2021
Florian Colas / 21512327

Module PIR

Évaluation d'algorithmes d'ordonnancement

	Université de Franche-Comté	Encadrant	Jury
Laurent Philippe	Professeur des Universités	X	X
Louis-Claude Canon	Maître de conférence	X	X
Julien Bernard	Maître de conférence	X	X
Veronika Sonigo	Maître de conférence		X

Dédicaces

Ce mémoire marque la fin de 5 ans (avec 1 année de pause) d'études en Master 2 Informatique Avancée et Applications (I2A) effectuées au CTU de Besançon. Je voudrais dédier ce document à Léo et Lou qui ont vécu le quinquennat "Master papa" avec patience et compréhension et à Anne-Sophie dont le soutien inconditionnel, la tolérance et l'indulgence à mon égard ont rendu possible ce projet.

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé dans la réalisation de ce projet de recherche ainsi que la rédaction de ce document : Laurent Philippe, professeur à l'université de Franche-Comté, Louis-Claude Canon, maître de conférence et Julien Bernard, maître de conférence. Ils ont tous trois su me guider dans les directions et choix à prendre. Ils m'ont transmis des informations précieuses concernant le sujet et les questions techniques. Je les remercie également pour leur disponibilité, leur écoute, et la qualité des échanges qu'on à pu avoir tout au long de la réalisation de ce projet d'initiation.

Et merci à Anne-Sophie pour ses relectures, l'aide à la correction de ce document.

Sommaire

1 Introduction générale

La question d'ordonnancement est omniprésente dans la vie pratique, l'industrie, le transport, les institutions, la santé et toutes autres disciplines qui nécessitent une organisation. L'ordonnancement consiste à agencer au mieux des tâches, des jobs reliés ou non entre eux pour atteindre un but tout en optimisant un critère particulier. Dans la construction d'une maison, depuis la pose de la première dalle jusqu'à la pose du dernier carrelage, tout a été organisé pour réduire au maximum le retard total. Certaines tâches ont pu être effectuées en parallèle (pose de la plomberie, pose de l'électricité) quant à d'autres, elles n'ont pu commencer qu'après l'achèvement d'ouvrages (pose des tuiles après la finition de la charpente).

L'ordonnancement intervient aussi dans l'attribution et l'ordre d'exécution des jobs à réaliser par processeur. Même pour une machine monoprocesseur l'ordre a son importance si des jobs sont liés par des relations de précédences. La question est d'autant plus essentielle depuis l'apparition des ordinateurs à structure parallèle. En effet, la composition d'un ensemble de jobs sur plusieurs processeurs, même sans lien entre eux, a des répercutions sur le temps total nécessaire pour exécuter tous ces jobs.

Le parallélisme est un type d'architecture informatique dans lequel plusieurs processeurs exécutent ou traitent une application, un calcul simultanément. Cette structure aide à effectuer de grands calculs en divisant la charge de travail entre plusieurs processeurs capables de communiquer et de coopérer.

Le calcul de l'attribution de chaque tâche d'un ensemble de jobs aux ressources disponibles dans le but d'optimiser un critère est un algorithme appelé algorithme d'ordonnancement (Scheduling algorithm). Ce calcul doit être effectué le plus rapidement possible et donner la réponse d'ordonnancement en adéquation avec ce qui est attendu.

Un cas particulier et fréquent d'ordonnancement est le problème $P||C_{\max}$ tel que définit dans la classification à trois champs de Graham *et al.* [12]. Le but de $P||C_{\max}$ consiste à planifier un ensemble $J = \{j_1, j_2, \ldots, j_n\}$ de n jobs (ou tâches) indépendants, sur m machines identiques, dans le but d'optimiser le temps total de traitement appelé makespan (noté C_{max}). Le temps nécessaire de réalisation de chaque job $p_i \in P = \{p_1, p_2, \ldots, p_n\}$ (avec $1 \leq i \leq n$) est connu à l'avance. Un job commencé est complété sans interruption (non préemptif) et est exécuté par une seule machine. Une machine ne traite qu'un seul job à la fois.

Mais, comme l'ont démontré Garey et Johnson, $P_2||C_{\max}$ (i.e avec $m = 2$) est un problème NP-Difficile [9], et $P||C_{\max}$ avec $m \geq 3$ est un problème NP-Difficile au sens fort [10]. En plus, $P||C_{\max}$ devient un problème NP-

Difficile, du moment que le nombre de machines est fixé [3], comme l'a montré Rothkopf [23] qui a présenté un algorithme de programmation dynamique.

Généralement, donner la solution optimale à un problème d'ordonnancement $P||C_{\max}$ n'est pas réaliste. La résolution de celui-ci demanderait un temps excessif et donc rédhibitoire. Comme les machines sont identiques, et travaillent à la même vitesse, la difficulté repose uniquement sur le regroupement des jobs. La résolution de ce problème d'ordonnancement va reposer sur des méthodes d'approche qui consistent à calculer en temps polynomial une solution "assez" proche de la valeur optimale.

Les jobs sont exécutés sans interruption ni coupure. Donc le makespan ne peut pas être inférieur à la taille du jobs le plus long (i.e. $\max_i\{p_i\}$). Il ne peut pas être inférieur non plus à la moyenne des tailles des jobs par machine i.e. $\frac{1}{m}\sum_{i=1}^{n} p_i$. Donc toutes les solutions ont une borne minimale [18] :

$$\text{borne}_{min} = \max\{\max_i\{p_i\}, \tfrac{1}{m}\sum_{i=1}^{n} p_i\}$$

L'existence d'une solution qui résout efficacement le problème de manière optimale n'est pas pensable, à moins que $P = NP$. Dans la littérature, l'étude d'ordonnancement est très riche et abondante. Le but étant d'améliorer le temps de calcul, et d'approcher le résultat optimal. Les solutions proposées sont des heuristiques ou des approximations.

Le but d'une heuristique (du grec *heuriskein* : trouver) est de produire une solution respectant les contraintes du problème et de bonne qualité selon le critère d'optimisation considéré. La solution ne sera pas forcément optimale mais une heuristique efficace tente de trouver une solution de bonne qualité suivant le temps de résolution imparti. Ces algorithmes calculent des solutions dont la borne maximale au pire des cas n'est pas maîtrisée : une étude du comportement est nécessaire pour définir cette borne maximale. Pour chaque étude de comportement, les notions suivantes sont utilisées :

- $C_m^A(J)$: Le résultat (makespan) de l'ordonnancement d'un ensemble J de jobs sur m machines parallèles identiques obtenu par l'algorithme A ;
- $C_m^\star(J)$: Le makespan optimal idéal de l'ordonnancement d'un ensemble J de jobs sur m machines parallèles identiques ;
- $\Gamma(A) = \frac{C_m^A(J)}{C_m^\star(J)}$: Le ratio d'approximation atteint par l'algorithme A au pire cas.

Parfois, cette borne est arbitrairement large et n'a pas été analysée. Une simulation permet d'obtenir des informations générales sur les tendances. Parmi les heuristiques les plus étudiées, nous pouvons citer LPT (Longest Time Processing) [11], LDM (Largest Differencing Method) [15], COMBINE [17].

Ce mémoire prend le texte de Della Croce et Scatamacchia [5] comme point de départ. Ces derniers revisitent l'heuristique LPT en appliquant aux données d'entrée du problème une stratégie gloutonne. L'algorithme développé appelé SLACK devient très compétitif par rapport à d'autres heuristiques, tant en complexité en temps qu'en ratio d'approximation. Cette conclusion est le résultat d'un protocole expérimental défini dans [14]. Les algorithmes sont testés sur des instances basées sur 2 types de générations de nombres pseudo-aléatoires, contenant 10, 50, 100, 500 et 1000 jobs, à planifier sur 5, 10 et 25 machines. Pour $n > m$ cela représente 13 points de comparaisons qui permettent aux auteurs de déclarer SLACK comme étant une alternative sérieuse à LPT, LDM et COMBINE. Ce résultat est intéressant. En effet la borne d'approximation se rapproche un peu plus de l'optimal. Mais ce résultat empirique repose sur un nombre assez restreint d'échantillons. Sont-ils assez représentatifs pour en dégager une conclusion absolue ? Est-ce le comportement général de SLACK (ou tout autre algorithme) d'être le plus efficace s'il l'est pour ces 13 points ?

Le but de ce document est de répondre à cette question à l'aide des démarches suivantes :

- Reproduire le protocole expérimental de Della Croce et Scatamacchia sur les mêmes algorithmes et les mêmes types d'instances afin de valider sa reproductibilité ;
- Élargir ce protocole expérimental aux principaux algorithmes mais à d'autres types de générations aléatoires, voire à de vraies archives de listes de tâches et à un spectre plus large de nombres de jobs et de machines.

Le chapitre 2 présente un état de l'art au problème $P||C_{\max}$. La création des listes de temps et les instances sont traitées dans le chapitre 3. Le chapitre 4 abordée l'environnement de tests, avant de présenter en chapitre 5 le résultat de la reproduction du protocole expérimental de Della Croce et Scatamacchia et du protocole élargi. Ce document se termine par une discussion sur les protocoles de tests des heuristiques.

Les annexes abordent certains points importants de l'implémentation des algorithmes et de la création des listes de temps.

2 État de l'art

Le problème d'ordonnancement se pose depuis l'apparition des premières machines parallèles et est d'autant plus d'actualité que les postes personnels sont équipés depuis quelques années de processeurs (CPU) et de cartes gra-

phiques (GPU) multi-cœurs. Les centres de calculs sont dotés de parcs assez uniformes et maintenant les clouds, offrent des instances VM qui permettent des environnements d'exécution homogènes. L'ordonnancement fait partie de la catégorie des problèmes d'optimisation combinatoire. C'est un champ de la recherche opérationnelle, actif depuis plus d'un siècle et abondant dans la littérature.

Pour le problème $P||C_{\max}$ de nombreuses pistes sont explorées. Il est pratiquement impossible de les énumérer toutes. Aussi sont présentées ici les plus étudiées et/ou utilisées pour être comparées ou servir de référent.

Nous abordons dans l'ordre, certaines heuristiques, un type d'algorithme d'approximation nommé schéma d'approximation en temps polynomial (PTAS) et une résolution du problème basée sur la programmation linéaire.

2.1 Heuristiques

Les heuristiques représentent la plus grande partie des recherches concernant le problème $P||C_{\max}$. Celles présentées sont basées sur le principe des LS (List Scheduling), sur une stratégie gloutonne, sur le problème bin-packing et sur le problème de partitionnement de nombres.

LS (List Scheduling) l'idée d'un algorithme d'ordonnancement de liste est de stocker l'ensemble des jobs dans celle-ci, éventuellement les trier dans un ordre particulier et/ou les regrouper selon une règle définie dans le but de leur assigner une priorité. Ces jobs sont ensuite affectés un à un à une machine suivant un principe déterminé. Les machines sont toujours occupées tant qu'il existe au moins un job en attente.

LPT Rule (Longest Time Processing) proposé par Graham [11], améliore l'algorithme LS. Contrairement à LS, LPT rule ordonne les jobs dans le sens décroissant de leur temps de traitement et affecte le job à la machine la moins chargée à ce moment là. LPT a un ratio d'approximation $\Gamma(LPT) \leq \frac{4}{3} - \frac{1}{3 \cdot m}$ et une complexité en temps de $O(n \log(n) + n \log(m))$. De fait, sa simplicité d'implémentation et ses caractéristiques d'approximation font de cet algorithme un référent de tests et l'un des plus repris dans la littérature.

Algorithme 1 : LPT Rule

Data :
instance de $P||C_{\max}$, avec
m machines,
n jobs et leur temps d'exécution

1 Trier les jobs de l'ensemble J dans l'ordre décroissant de leur temps d'exécution et ré-indexer l'ensemble de telle manière à obtenir : $p_1 \geq p_2 \geq \ldots \geq p_n$

2 Parcourir la liste et affecter chaque job à la machine la moins chargée à ce moment là.

Stratégie gloutonne Della Croce et Scatamacchia [5] revisitent LPT rule dans le but de l'optimiser. L'étude est articulée autour du lien qui existe entre le nombre de machines m, le nombre de jobs n, la relation qu'il peut y avoir entre les deux et la probabilité que LPT donne un résultat au pire cas. S'ensuit SLACK, un algorithme basé sur la stratégie gloutonne suivante, avant d'affecter les jobs un à un à la machine la moins chargée à ce moment là :

— Trier les jobs par ordre décroissant de leur taille ;
— Découper l'ensemble trié en tuples de m jobs ;
— Soit "slack" la différence entre la taille du premier job et la taille du dernier job de chaque tuple ;
— Trier l'ensemble des tuples dans l'ordre décroissant de leur "slack".

SLACK a une complexité en temps de $O(n\log(n) + n\log(m))$.

Algorithme 2 : SLACK

Data :
instance de $P||C_{\max}$, avec
m machines,
n jobs et leur temps d'exécution

1 trier la liste des jobs dans l'ordre décroissant des temps nécessaires de traitements

2 réindexer les jobs, de manière à obtenir $p_1 \geq p_2 \geq ... \geq p_n$

3 découper l'ensemble obtenu en $\lceil \frac{n}{m} \rceil$ tuples de m jobs (ajout de jobs "dummy" de taille nulle pour le dernier tuple, si n n'est pas un multiple de m)

4 considérer chaque tuple avec la différence de temps ("Slack") entre le premier job du tuple et le dernier.

5
$$\left\{ \begin{array}{lll} \{1,...,m\} & \{m+1,...,2\cdot m\} & ... \\ p_1 - p_m & p_{m+1} - p_{2\cdot m} & ... \end{array} \right\}$$
trier les tuples par ordre décroissant de "Slack" et ainsi former un nouvel ensemble `// e.g:` $\{\{m+1,...,2\cdot m\}\{1,...,m\}\}$ `si` $p_{m+1} - p_{2\cdot m} > p_1 - p_m$.

6 appliquer l'ordonnancement (affectation à la machine la moins chargée à ce moment là) à l'ensemble ainsi obtenu.

Bin-Packing Un des problèmes semblable à $P||C_{\max}$ est celui de Bin-Packing.

Soient
- un ensemble d'objets à ranger $t_i \in T = \{t_1, t_2, \ldots, t_n\}$,
- les tailles de ces objets $L(t_i)$ (avec $1 \leq i \leq n$).
- une taille de bac C.

Un packing est une partition $B < B_1, B_2, ...B_k >$ telle que $L(B_j) \leq C$ (avec $1 \leq j \leq k$ et $L(B_j)$ la somme des tailles des objets pacqués dans B_j). Autrement dit, cela consiste à placer des objets t_i dans des bacs B_j de taille maximum C. Le problème Bin-Packing qui est NP-Complet, tente de minimiser le nombre de bacs k.

Bin-Packing peut être considéré comme le double de $P||C_{\max}$ car l'ensemble des objets à ranger associés à leur taille et la partition B peuvent être vus respectivement comme un ensemble J de n jobs, leurs tailles propres, et l'ordonnancement recherché sur m machines. La taille C des bacs correspond au makespan recherché, et le nombre de bacs obtenus au nombre m de machines.

Coffman, Garey et Johnson [4] utilisent Bin-Packing pour tenter de résoudre $P||C_{\max}$ et proposent l'heuristique MULTIFIT (algorithme 4). Cet algorithme est basé sur FFD (First-Fit-Decreasing, algorithme 3, [22]), un outil (heuristique) de résolution de Bin-Packing. Il accepte en entrée un ensemble de tailles d'objets et une taille maximale C de bacs et propose en retour un packing, donc un nombre de bacs k. L'idée de MULTIFIT est de proposer des valeurs de C à FFD, jusqu'à avoir un nombre de bacs $k = m$. Ceci est réalisée à l'aide d'une recherche dichotomique dont les bornes de départ qui vont confiner C sont :

borne inférieure
$$C_l = \max\{\max_i\{L(t_i)\}, \frac{1}{m} \cdot L(T)\}$$
ou rammené au problème $P||C_{\max}$
$$C_l = \text{borne}_{min} = \max\{\max_i\{p_i\}, \frac{1}{m}\sum_{i=1}^{n} p_i\}$$

borne inférieure
$$C_u = 2 \cdot C_l$$

MULTIFIT accepte en entrée un ensemble de temps de jobs P, un nombre de machines m et un nombre d'itérations k pour la recherche dichotomique. Selon la taille de l'instance du problème k doit avoir une valeur suffisante pour que MULTIFIT donne une réponse convenable. Il est estimé que cette valeur est atteinte pour toute taille d'instance quand $k = 7$ [4]. MULTIFIT a un ratio d'approximation $\Gamma(MULTIFIT) \leq 1,220 + 2^{-k}$ et une complexité en temps de $O(nlog(n)+knlog(m))$. FFD, quant a lui a un ratio d'approximation de $C_m^{FFD}(J) \leq \frac{11}{9} \cdot C_m^{\star}(J) + \frac{6}{9}$ [6] et une complexité en temps de $O(n \log(m))$.

Algorithme 3 : FFD

Data :
instance de Bin-Packing avec :
C taille maximale des bacs,
un ensemble $T = \{t_1, t_2, \ldots, t_n\}$ de n objets à ranger (pacquer) dont $L(t_i)$ est la taille de chaque objet t_i (avec $1 \leq i \leq n$)

```
1  Trier l'ensemble T par ordre décroissant des L(t_i) et ré-indexer
   l'ensemble de telle manière à obtenir : t_1 ≥ t_2 ≥ ... ≥ t_n
2  NB_b = 0 // nombre de bacs
3  Pour tout objet t_i avec i = 1, 2, ..., n faire
4      Pour tout bac existant B_j (avec j ≤ NB_b) faire
5          Si L(t_i) + L(B_j) ≤ C alors
6              pacquer t_i dans le bac B_j
7              sortir de la boucle
8      Si t_i n'a pas été pacqué alors
9          incrémenter NB_b
10         Créer un nouveau bac B_{NB_b}
11         pacquer t_i dans B_{NB_b}
```

Algorithme 4 : MULTIFIT

Data :
T un ensemble de jobs
m un nombre de processeurs
k un nombre d'itérations

borne supérieure : $C_u = \max\left\{\frac{2}{m} \cdot L(T), \max_i\{L(T_i)\}\right\}$
borne inférieure : $C_l = \max\left\{\frac{1}{m} \cdot L(T), \max_i\{L(T_i)\}\right\}$
$i = 1$
Tant que $i \leq k$ **faire**
    $C = \frac{C_u + C_l}{2}$
    $NB_b = FFD(T, C)$ `// FFD renvoie le nombre de bacs créés`
    Si $NB_b \leq m$ **alors**
        $C_u = C$
    sinon
        $C_l = C$
    incrémenter i

`// Après` k `itérations, MULTIFIT renvoie` C_u `qui correspond à la plus petite valeur de` C `pour laquelle` $FFD[T, C] \leq m$
Renvoyer C_u

Lee et Massey [17] améliorent MULTIFIT en utilisant LPT pour calculer la borne supérieure de départ pour réduire l'amplitude de la recherche dichotomique et proposent l'heuristique COMBINE. Contrairement à la recherche dichotomique de MULTIFIT qui effectue k itérations, celle de COMBINE s'arrête lorsque la différence des deux bornes est inférieure à $\alpha \cdot A$.

$C_u - C_l \leq \alpha \cdot A$
avec

$$A = \sum_{i=1}^{n} \left(\frac{p_i}{m}\right) \qquad \text{(moyenne des poids des jobs par processeur)}$$

$$\alpha = 0.005 \qquad \text{(constante arbitraire)}$$

Le nombre d'itérations k de recherche dichotomique est variable mais n'excède pas 6. Or, LPT a déjà tourné une fois, donc $k \leq 7$.

COMBINE a un ratio d'approximation $\Gamma(COMBINE) \leq \frac{13}{12} + 2^{-k}$ et une complexité en temps de $O(n\log(n) + kn\log(m))$.

Algorithme 5 : COMBINE

Data :
instance de $P||C_{\max}$, avec
m machines,
n jobs,
α un coefficient arbitraire ($\alpha = 0,005$ par défaut)

A $= \sum_{i=1}^{n}(\frac{p_i}{m})$
M $\leftarrow C_m^{lpt}(J)$
Si $M \geq 1,5 \cdot A$ **alors**
    $M^\star = M$
sinon
    $C_u \leftarrow M$
    $C_l \leftarrow \max\left\{\frac{M}{\frac{4}{3}-\frac{1}{3\cdot m}}, p_1, A\right\}$
    Tant que $C_u - C_l > \alpha \cdot A$ **faire**
        appliquer MULTIFIT
    `// on arrête lorsque` $C_u - C_l \leq \alpha \cdot A$

Partitionnement de nombres Un autre problème similaire à l'ordonnancement est le problème de partitionnement de nombres.

Soit un ensemble $E = \{e_1, e_2, \ldots, e_n\}$ de n entiers.

Le problème de partitionnement de nombres consiste à diviser l'ensemble de départ en NB_e sous-ensembles mutuellement exclusifs et collectivement exhaustifs de sorte que les sommes des nombres dans chaque sous-ensemble soient aussi égales que possible [16]. Ce problème est NP-complet et peut être assimilé au problème d'ordonnancement $P||C_{\max}$, avec un ensemble $E = P = \{p_1, p_2, \ldots, p_n\}$ de taille de tâches à partitionner en $NB_e = m$ sous-ensembles i.e. m machines (ou processeurs).

Karmarkar et Karp [15] proposent LDM (Largest Differencing Method) pour deux partitions puis pour m partitions. Cet algorithme consiste à remplacer les 2 plus grands (Largest) nombres de l'ensemble de n éléments de départ par leur différence (Differencing) pour obtenir un nouvel ensemble de $n-1$ éléments. LDM a un ratio d'approximation [19] $\Gamma(LDM) \leq \frac{7}{6}$ pour $m = 2$, $\frac{4}{3} - \frac{1}{3\cdot(m-1)} \leq \Gamma(LDM) \leq \frac{4}{3} - \frac{1}{3\cdot m}$ pour $m \geq 3$ et une complexité en temps [19] de $O(n \cdot \log n)$.

Algorithme 6 : LDM

Data :
Un ensemble $P = \{p_1, p_2, \ldots, p_n\}$ de n nombres positifs réels (temps d'exécution des n jobs indépendants).
m un nombre de partitions à obtenir (nombre de machines cible)

1 Convertir les n p_i en n m-tuples $A_i = [\ m-1$ valeurs vides $\{\}$ et $\{p_i\}]$
`// e.g. pour` $m = 3$ p_i `devient` $[\{\}\{\}\{p_i\}]$
2 **Pour** $n-1$ *itérations* **faire**
3 Calculer pour chaque m-tuple A_i la différence $d(A_i)$ entre la plus grande et la plus petite valeur des éléments de A_i et considérer A_a et A_b les 2 m-tuples dont les différences $d(A_a)$ et $d(A_b)$ sont les plus grandes
4 Combiner A_a et A_b en un seul m-tuple A_{ab} en joignant le premier plus petit élément de A_a (peut être un ensemble vide, un élément, ou un ensemble d'éléments provenant de précédentes combinaisons et dont la valeur est la somme des éléments qui le composent) avec l'élément de A_b le plus grand, puis, le deuxième élément le plus petit de A_a avec le deuxième élément le plus grand de A_b, et ainsi de suite

2.2 Approximation

Contrairement à une heuristique, qui doit être étudiée pour connaître sa borne d'approximation, celle-ci est maîtrisée par un algorithme d'approximation qui fournit une garantie d'approche.

PTAS (Polynomial-Time Approximation Scheme) Une famille d'algorithmes d'approximation, les PTAS, regroupe des algorithmes qui calculent, pour tout $\epsilon > 0$ donné une solution proche de l'optimal d'un facteur $(1+\epsilon)$ pour un problème de minimisation ou $(1-\epsilon)$ pour un problème de maximisation, en temps polynomial et dépendant de ϵ.

Hochbaum et Shmoys sont les premiers à proposer un PTAS [13] adapté au problème $P||C_{\max}$, PTAS DUAL. Cet algorithme prend comme idée de départ MULTIFIT (algorithme 4) en reformulant le problème Bin-Packing. Soient :

- Un ensemble d'objets $t_i \in T = \{t_1, t_2, \ldots, t_n\}$;
- Leur taille $L(t_i)$ avec $1 \leq i \leq n$ et $0 \leq L(t_i) \leq 1$;
- Une taille maximale C des bacs ;
- $T^\star(T)$ le Bin-Packing optimal, i.e. le nombre minimum de bacs nécessaires pour l'organisation des pièces de l'ensemble T.

Le but de l'algorithme proposé est de construire un arrangement Bin-Packing qui utilise au plus $T_m^\star$ bacs remplis avec des pièces totalisant une taille au plus de $1+\epsilon$. $P||C_{\max}$ et Bin-Packing sont reliés de la façon suivante :

$T^\star(\frac{J}{C} \leq m$ si et seulement si $C_m^\star \leq C$.

Donc, la taille minimale des bacs $C^\star$ qui correspond au makespan optimal est telle que $T^\star\left(\frac{J}{C^\star}\right) \leq m$. Il y a 2 paramètres critiques m (le nombre de machines identiques) et C (la taille des bacs qui est aussi le makespan recherché). Ceci est un problème de décision à 2 paramètres $R(p_1, p_2)$ qui se décline en 2 problèmes d'optimisation dont l'un (problème primal) consiste à fixer le premier paramètre pour optimiser l'autre et inversement pour le deuxième (problème dual).

le problème primal $(J, \bar{p_1})$ a pour valeur optimale $OPT_P(J, \bar{p_1})$. L'algorithme d'approximation pour ce problème $\epsilon-$primal prend en paramètre p_1 et optimise p_2 de la manière suivante :

— p_1 est une valeur inférieure ou égale à $\bar{p_1}$;

— p_2 est une valeur optimisée $PRIMAL_{P,\epsilon} \leq (1+\epsilon) \cdot OPT_P(J, \bar{p_1})$.

Le problème dual $(J, \bar{p_2})$ a pour valeur optimale $OPT_D(J, \bar{p_2})$. L'algorithme d'approximation pour ce problème $\epsilon-$primal prend en paramètre p_2 et optimise p_1 de la manière suivante :

— p_1 est une valeur optimisée $DUAL_{D,\epsilon}(J, \bar{p_2}) \leq OPT_D(J, \bar{p_2})$;

— p_2 est une valeur $\leq (1+\epsilon) \cdot \bar{p_2}$.

Donc, trouver un algorithme $\epsilon-$primal pour le problème primal peut être réduit à trouver un algorithme $\epsilon-$dual pour le problème dual.

Trouver un algorithme d'approximation $\epsilon-$dual peut être réduit à trouver un algorithme d'approximation $\epsilon-$dual pour une instance du problème où la taille des pièces sont strictement supérieures à ϵ.

De fait, l'intervalle de la taille des pièces (devenu $]\epsilon, 1]$) est découpé en S $= \lceil \frac{1}{\epsilon^2} \rceil$ sous intervalles de tailles identiques $]\epsilon = l_1, l_2],]l_2, l_3], \ldots,]l_S, L_{S+1} = 1]$. Chaque bac P_i, sera rempli au maximum avec $\lfloor \frac{1}{\epsilon} \rfloor$ pièces.

Soit x_k avec $1 \leq k \leq S$ le nombre de pièces d'un bac dont la taille est comprises dans $]l_k, l_{k+1}]$. Chaque x_k peut prendre une valeur dans l'intervalle $[0, \frac{1}{\epsilon}[$

La configuration d'un bac P_i peut être donnée par un s-tuple $(x_1, x_2, \ldots, x_S)$. Soit $cf(x_1, \ldots, x_S)$ une configuration dite faisable si $\sum_{k=1}^{S} x_k \cdot l_k \leq 1$ avec l_k de $]l_k, l_{k+1}]$.

Nous avons donc

$L(P_i) = 1 + \epsilon$

Soit b_k avec $1 \leq k \leq S$ le nombre de pièces utilisées dans tous les bacs dont la taille appartient à $]l_k, l_{k+1}]$.

Soit $Bins(b_1, b_2, \ldots, b_S)$ le nombre minimum de bacs nécessaires, lorsqu'il

y a b_k pièces de taille l_k, et considérons que le premier bac soit rempli. Nous obtenons :

$$Bins(b_1, b_2, \ldots, b_S) = 1 + \min_{cf(x_1,\ldots,x_S)} Bins(b_1 - x_1, b_2 - x_2, \ldots, b_S - x_S)$$

Ce qui est résolu par programmation dynamique.

Il existe 2 versions d'algorithmes optimisée pour $\epsilon = \frac{1}{5}$ et $\epsilon = \frac{1}{6}$ qui sont présentés en annexe.

2.3 Programmation linéaire

Le problème $P||C_{\max}$ s'inscrit parfaitement dans l'énoncé d'un problème de programmation linéaire. en effet, la fonction objectif qui consiste à minimiser le makespan et les contraintes sont des fonctions linéaires. Les variables et le résultat attendu sont discrets, ce qui rend la résolution du problème difficile. Ces algorithmes donnent une solution faisable exacte.

PA Mokotoff [20] présente un algorithme basé sur la formulation de la programmation linéaire en utilisant des variables booléennes d'affectation des jobs à une machine, i.e. x_{ij} est égal à 1 si le job j_j est affecté à la machine m_i, ou 0 dans le cas contraire.

La minimisation du makespan peut être posée ainsi :
Minimiser y tel que :

— $\sum_{i=1}^{m} x_{ij} = 1 \quad$ pour $1 \leq j \leq n$
Sur toutes les machines, au moins un et un seul x_j est égal à 1. Un job est affecté à une et une seule machine.

— $y - \sum_{j=1}^{n} p_j \cdot x_{ij} \geq 0 \quad$ pour $1 \leq i \leq m$
Pour une machine donnée, la somme des temps est inférieure ou égale à y.

Où la valeur optimale de y est $C_{\max}$ et

$$x_{ij} = \begin{cases} 1 \text{ si le job } j_j \text{ est affecté à la machine } m_i \\ 0 \text{ sinon.} \end{cases}$$

Le programme linéaire est donc composé de

— $n \cdot m + 1$ variables (les variables x_{ij} et la variable y)

— $n + m$ contraintes

La zone F peut être définie ainsi :

$$F = \{(x, y) : x \in B^{n \cdot m}, y \in \mathbb{R}_+ : \sum_{i=1}^{m} x_{ij} = 1 \forall j; y - \sum_{j=1}^{n} p_j \cdot x_{ij} \geq 0 \forall i\}$$

avec

$$B = \begin{bmatrix} x_{11} & & x_{n1} \\ & \ddots & \\ x_{1m} & & x_{nm1} \end{bmatrix}.$$

Le polytope P relatif à F est défini ainsi :

$$P = \{(x, y) : x \in {\mathbb{R}_+}^{n \cdot m}, y \in \mathbb{R}_+ : \sum_{i=1}^{m} x_{ij} = 1 \forall j; y - \sum_{j=1}^{n} p_j \cdot x_{ij} \geq 0 \forall i\}$$

Il est possible de construire un ensemble fini d'**inégalités**

$$Ax + Dy \leq \bar{b}$$

telles que

$$\min\{y : (x, y) \in F\} = \min\{y : x \in {\mathbb{R}_+}^{n \cdot m}, y \in \mathbb{R}_+ Ax + Dy \leq \underline{b}$$

NB : une solution $(x^\circ, y^\circ) \in P$ doit être exclue (car n'est pas un vecteur entier) si $(x^\circ, y^\circ) \notin P$.

Des **inégalités transitoires** peuvent être générées (nombre maximum de jobs par machine)

$$\sum_{j \in S_i} x_{ij} \leq L_i \quad (L_i = h - 1 \iff S_{j_h} > LB \text{ et } S_{j_{(h-1)}} \leq LB)$$

LB : borne inférieure.

Pour un problème $P||C_{\max}$, même de taille modeste, le nombre de variables et contraintes est très important, et certaines sont inutiles. L'algorithme va donc utiliser la méthode des plans sécants (Cutting Plane Method). À chaque itération , des inégalités valides sont générées, puis une relaxation est exécutée, jusqu'à l'obtention d'une solution faisable.

L'algorithme est présenté en annexe.

Nous venons de parcourir les principaux algorithmes et méthodes de tentative de résolution du problème $P||C_{\max}$, notamment LPT, COMBINE et LDM utilisées dans le document de Della Croce et Scatamacchia [5] pour comparer les performances de SLACK. Le tableau 1 récapitule les caractéristiques de ces heuristiques qui nous intéressent.

algorithme	Domaine	[Ref]	Complexité en temps	Ratio d'approximation
LPT Rule	List Scheduling	[11]	$n\ \log(n)$	$\frac{4}{3} - \frac{1}{3m}$
SLACK	List Scheduling stratégie gloutonne	[5]	$n\ \log(n)$	non précisé
LDM	partitionnement	[15]	$n\ \log(n)$	entre $\frac{4}{3} - \frac{1}{3(m-1)}$et $\frac{4}{3} - \frac{1}{3m}$pour $m \geq 3$
COMBINE	bin-packing	[17]	$n\ \log(n) + kn\ \log(m)$	$\frac{13}{12} + 2^{-k}$
MULTIFIT	bin-packing composante de COMBINE	[17]	$n\ \log(n) + kn\ \log(m)$	$\frac{13}{12} + 2^{-k}$
FFD	bin-packing composante de MULTIFIT	[17]	$n\ \log(n) + kn\ \log(m)$	$\frac{13}{12} + 2^{-k}$

TABLE 1 – Heuristiques étudiées

3 Listes de temps et instances

Les instances, paramètres de chaque algorithme, reposent sur des listes de temps de jobs. Le résultat de la comparaison de ces heuristiques dépend de cette liste de temps, de la distribution, de l'hétérogénéité ou de l'homogénéité de ses valeurs. Il est important de pouvoir caractériser ces listes de temps pour faire un lien avec le comportement des heuristiques. Mais pour des caractéristiques identiques les comportements ne sont pas forcément identiques. En effet, une même valeur d'une caractéristique peut représenter plusieurs ensembles de valeurs de temps différentes et ainsi impliquer des comportements algorithmiques différents. Il est donc nécessaire de multiplier les tests pour voir se dessiner des tendances, des liens entre distributions de valeurs, caractéristiques, et résultats obtenus.

C'est précisément ce que font Della Croce et Scatamacchia en générant 10 instances de 2 différents types de génération de listes de temps pour chaque couple $\{m, n\}$. Cette méthode paraît correcte pour le nombre d'instances mais le cas échéant insuffisante pour les types de listes de temps. Le nombre restreint des lois statistiques utilisées dans leur modèle expérimental ne permet pas d'affirmer ou d'infirmer que SLACK n'est pas sensible à la répartition des nombres constituant les listes de temps. Pour cela, il est nécessaire de tester SLACK, ou tout autre algorithme, sur des instances aux caractéristiques variées.

Il existe deux façons d'obtenir des listes de temps de jobs : Soit les créer entièrement à partir d'un nombre n désiré ; Soit les récupérer.

Pour la création d'une liste synthétique des générateurs de nombres aléatoires suivant des lois statistiques différentes peuvent être utilisés. Della Croce et Scatamacchia pour leur protocole ont adopté deux types de générations de listes de temps : une répartition uniforme et non-uniforme. Il est intéressant d'étendre ce protocole à d'autres types de listes.

Pour la récupération, des sites internet mettent à disposition au téléchargement des listes de temps soit synthétiques, soit réelles.

Après une présentation des choix effectués sur les méthodes d'acquisition des listes de temps, nous aborderons les différentes lois statistiques utilisées pour générer des listes de nombres synthétiques et le moyen pour en récupérer des réelles. Ensuite, nous examinerons une méthode de transformation d'une liste de temps en une instance dont l'optimal est connu, pour finir sur la caractérisation des listes de nombres.

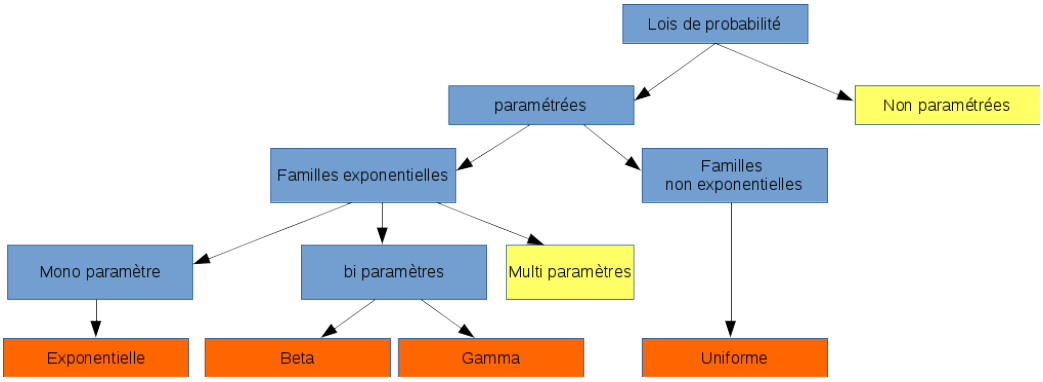

FIGURE 1 – Taxonomie des lois statistiques utilisées [21].

3.1 Choix de production des listes de temps

Nous souhaitons d'abord valider la reproductibilité du protocole expérimental de Della Croce et Scatamacchia. Nous devons donc reproduire les expériences avec les mêmes types de génération de nombres qu'ils utilisent :

— Uniforme ;
— Non-uniforme.

Selon la taxonomie des distributions statistiques [21] il existe plusieurs familles. Pour la génération de listes de temps, nous utilisons des variables statistiques indépendantes et identiquement distribuées (i.i.d.) qui suivent pour chaque production la même loi de probabilité.

Dans le but d'élargir ce protocole et de comparer SLACK avec d'autres types de listes de temps 3 autres distributions qui font partie de la famille des distributions exponentielles (figure 1) sont implémentées :

— Exponentiel ;
— Gamma ;
— Beta.

Pour soumettre les algorithmes à des données de situations réelles, notre choix se porte sur la récupération (téléchargement) d'archives existantes de listes de temps provenant des activités de vrais systèmes informatique :

— PWA (Parallel Workload Archive).

3.2 Génération des listes de temps synthétiques

Les listes synthétiques sont créées de toutes pièces en suivant des lois statistiques et de distributions particulières. Sont présentées ici les lois exponentielle, gamma, beta et les 2 méthodes utilisées par Della Croce et Scatamacchia, uniforme et non-uniforme.

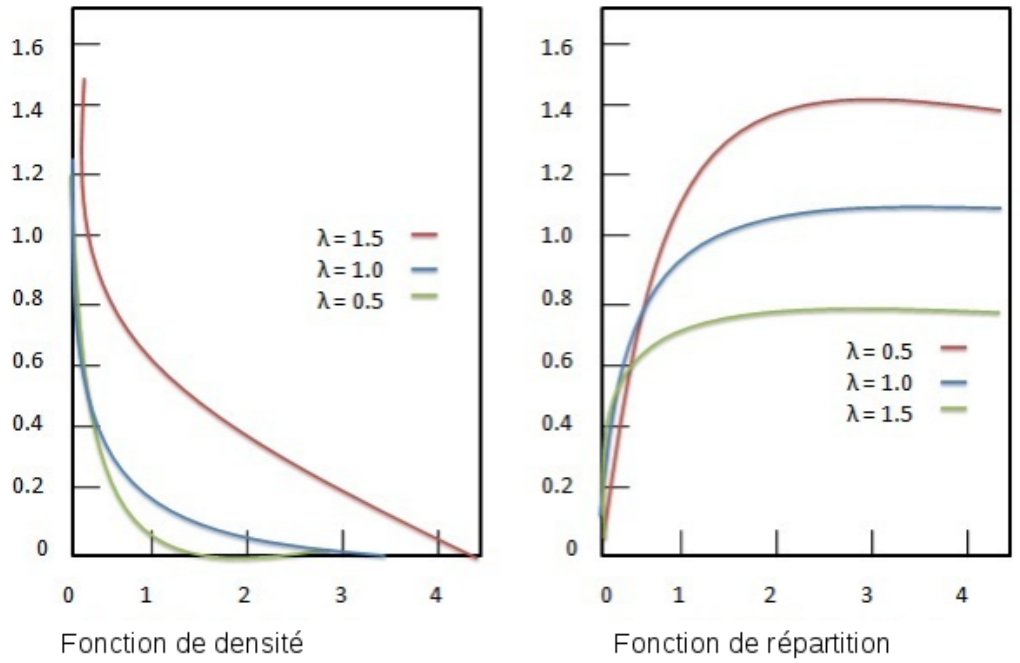

FIGURE 2 – Loi statistique exponentielle.

les informations concernant les lois exponentielles, beta et gamma proviennent du site de Marie Etienne :
https://marieetienne.github.io/CoursHalieut/rappel_loi.html

Loi exponentielle ou loi de durée de vie sans vieillissement (figure 2). Soit X une variable aléatoire définie dans $[0, \infty[$. X suit une loi exponentielle de paramètre λ si elle a pour densité :

$$[X = t] = \begin{cases} \lambda e^{-\lambda t} & \text{si } t \geq 0 \\ 0 & \text{sinon} \end{cases}$$

L'espérance et la variance sont fonctions de λ et sont égales a :

$$\mathbb{E}(X) = \frac{1}{\lambda}$$
$$\mathbb{V}(X) = \frac{1}{\lambda^2}$$

Loi gamma ou modélisation de phénomènes qui se déroulent au cours du temps (figure 3). La loi de statistique Gamma est une loi de la famille des lois exponentielles à 2 paramètres et se définit sur $[0, \infty]$. Si $Y_1, Y_2, \ldots, Y_n$ sont des variables i.i.d. de loi exponentielle de paramètre λ alors $X = \sum_{i=1}^{\alpha} X_i$ suit une loi gamma $\Gamma(\alpha, \beta)$ avec α le paramètre de forme et β $(= \lambda)$ le paramètre d'échelle.

Si X suit une loi $\Gamma(\alpha, \beta)$ alors la densité est de la forme :

$$[X = t] = \begin{cases} \frac{\beta^\alpha t^{\alpha-1} e^{-\beta t}}{\Gamma(\alpha)} & \text{si } t \geq 0 \\ 0 & \text{sinon} \end{cases}$$

Les paramètres α et β peuvent être définis aussi ainsi :

$$k \text{ tel que } \alpha = k \text{ et}$$
$$\theta \text{ tel que } \beta = \frac{1}{\theta}$$

L'espérance et la variance sont fonctions de k et θ et sont égales a :

$$\mathbb{E}(X) = k\theta$$
$$\mathbb{V}(X) = k\theta^2$$

Loi beta (figure 4) La loi Beta a été développée par Thomas Bayes (1702 - 1761) pour répondre à son propre problème de succès/échecs dont la proportion de succès possible est inconnue [1]. Cette proportion inconnue, qui est la variable aléatoire représentée par une loi beta, est déduite par tirages successifs avec remise. La loi de statistique Beta est une loi de la famille des lois Exponentielles à 2 paramètres et se définit sur $[0, \infty]$.

Si la variable aléatoire X suit une loi Beta de paramètres α et β alors la loi de densité est de la forme :

$$[X = t] = \begin{cases} \frac{\Gamma(\alpha+\beta) t^{\alpha-1} (1-t)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)} & \text{si } t \in [0, 1] \\ 0 & \text{sinon} \end{cases}$$

Beta est intéressante dans l'analyse de proportions d'événements de type succès/échec ou (moyennant un changement d'échelle) dans la modélisation du temps avant la fin d'une tâche.

L'espérance et la variance sont fonctions de α et β et sont égales a :

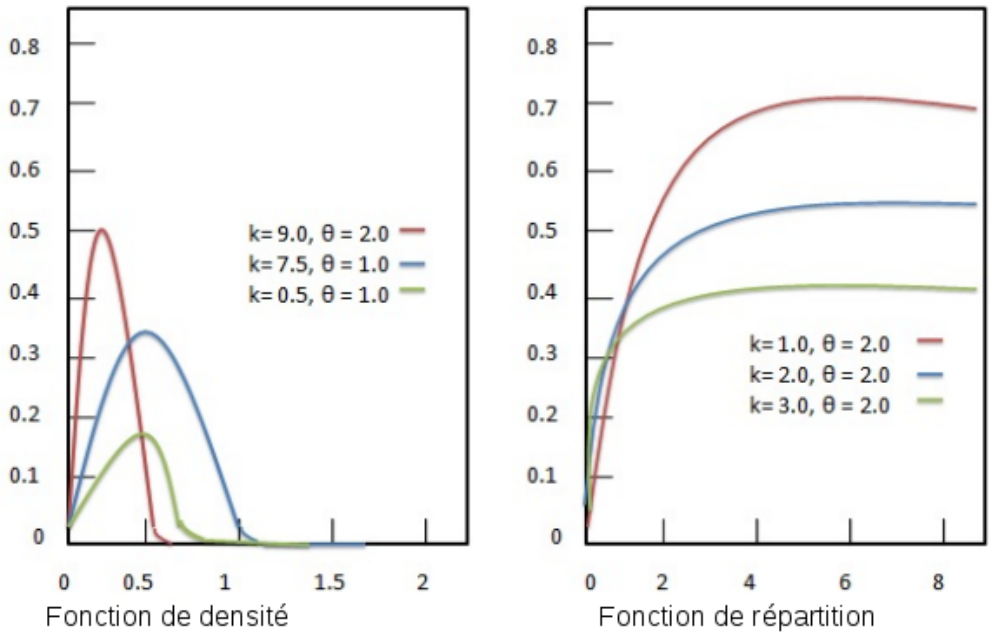

FIGURE 3 – Loi statistique Gamma.

$$\mathbb{E}(X) = \frac{\alpha}{\alpha + \beta}$$
$$\mathbb{V}(X) = \frac{\alpha\beta}{(\alpha + \beta)^2(\alpha + \beta + 1)}$$

Loi uniforme Cette loi statistique utilisée pas Della Croce et Scatamacchia est non exponentielle. Une loi uniforme sur $[a, b]$ est une loi de probabilité P dont la fonction de densité est constante sur $[a, b]$. i.e. les nombres générés sont uniformément répartis dans l'intervalle $[a, b]$

$$[X] = \frac{1}{b-a}$$

Liste non-uniforme Ce type de génération de liste de temps est utilisé dans le protocole expérimental de [8] et est repris par Della Croce et Scatamacchia [5] pour éprouver SLACK. Une liste de nombres aléatoires générée via une règle NON-UNIFORME de paramètres a et b est l'union de 2 listes générées via une loi uniforme comme suit :

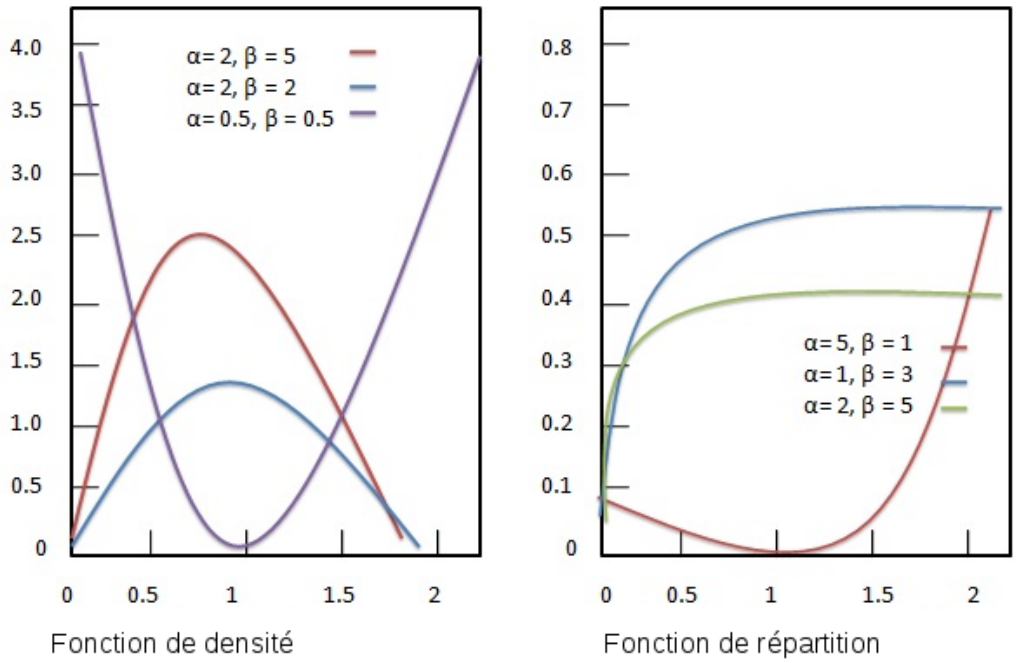

FIGURE 4 – Loi statistique Beta.

$$98\% = \text{loi uniforme } [0.9(b-a), b]$$
$$2\% = \text{loi uniforme } [a, 0.2(b-a)]$$

Exemple 1. *Pour la génération d'une liste suivant la règle non-uniforme [1, 100],*

$$98\% \text{ est uniformément répartis sur } [89.1\ ,\ 100]$$
$$2\% \text{ est uniformément répartis sur } [1\ ,\ 19.8]$$

ce qui peut donner pour $n = 10$ $\{95, 95, 90, 93, 97, 89, 90, 90, 99, 2\}$

Ce qui donne des instances très hétérogènes.

3.3 Récupération de listes de temps réelles

PWA Parallel Workload Archive. L'idée est de collecter des données de vrais systèmes et de supposer que les futurs workload seront similaires [7].

PWA est un dépôt de fichiers journaux de temps disponible à l'URL :

https://www.cs.huji.ac.il/labs/parallel/workload/

Ces archives proviennent de l'enregistrement direct des événements qui se produisent sur les systèmes informatiques et ont l'avantage de représenter un panel de besoins réels. Mais elles peuvent comporter certains inconvénients :

- Le nombre de jobs n n'est pas maîtrisé et ne représente qu'une vingtaine de fichiers, donc un nombre limité de tests et comparaisons empiriques à effectuer ;
- Ces archives ne sont pas forcément complètes et des informations importantes peuvent manquer ;
- Incohérence de données. Certaines informations ne passent pas le contrôle de contraintes d'intégrités ;
- Données erronées. Des données peuvent être fausses. e.g dues au dépassement d'adressage de nombres ;
- Modification d'environnement. Le problème inhérent aux machines est leur hétérogénéité qui change avec le temps ;
- Comportement non représentatif. L'aspect humain entre aussi en ligne de compte ;
- Effet "END". Les jobs sont consignés une fois terminés. Entre une tâche extrêmement courte et une tâche longue le temps d'enregistrement est le même. Cela peut provoquer un décalage entre la fin du journal et le temps calculé ;
- Les temps d'arrêts ne sont pas consignés.

3.4 Instance et maîtrise de la solution optimale

Pour comparer les résultats obtenus il est nécessaire d'avoir un référent par rapport au Makespan. Il n'est pas possible d'obtenir le ratio d'approximation de l'algorithme A ($\Gamma(A) = \frac{C_m^A(J)}{C_m^\star(J)}$) car l'optimal n'est pas connu et c'est précisément ce que l'on cherche. La seule donnée connue est la borne minimale de la liste de tailles de jobs. Or cette borne ($\text{borne}_{min} = \max\{\max_i\{p_i\}, \frac{1}{m}\sum_{i=1}^n p_i\}$) n'est pas forcément l'optimal et le rapport $\frac{Optimal}{\text{borne}_{min}}$ varie.

Exemple 2. *Soit $P = \{8, 5, 7, 8, 5, 5\}$, l'ensemble des p_i à appliquer sur 4 machines parallèles identiques. La borne minimale est :*

$$borne_{min} = \max\left\{\max_i\{p_i\}, \frac{1}{m}\sum_{i=1}^{n} p_i\right\} = 9.5$$

L'optimal $C_4^\star(J) = 12$ (figure 5).

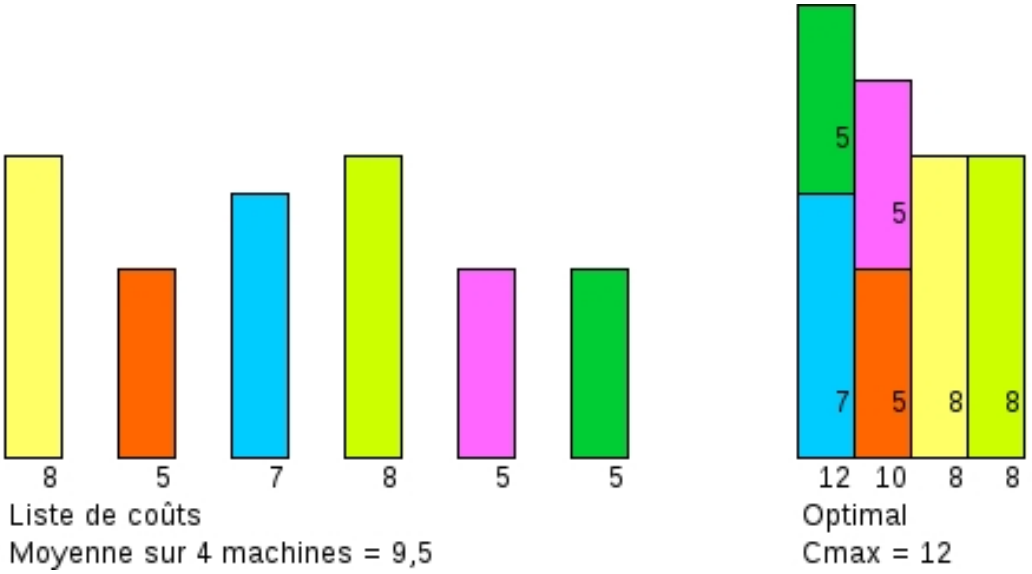

FIGURE 5 – Liste de coût de départ, moyenne des tailles des jobs pour 4 machines, et makespan optimal.

Cette borne minimale est égale à l'optimal uniquement si toutes les charges des machines sont identiques. À partir d'une liste de coûts (synthétique ou réelle) il est donc possible d'obtenir une instance dont l'optimal est connu [2]. Cette opération nécessite la transformation de la liste de coût de départ pour un nombre de machines défini (figure 6).

- Soit une liste de n jobs et l'ensemble des p_i à appliquer sur m machines parallèles identiques ;
- Ordonnancement : chaque job est affecté à la machine la moins chargée à ce moment là ;
- Tri : tri des machines par ordre décroissant des charges ;
- Complétion : la machine la plus chargée (la première) représente le makespan. La charge des $m-1$ machines restantes est complétée avec des jobs fictifs pour obtenir le même temps que la machine la plus chargée.
- Toutes les machines ont le même temps d'exécution. L'optimal est dans ce cas égal à la moyenne des temps par machine. La nouvelle liste de coûts contient désormais $n+m-1$ éléments.

La liste de départ est modifiée suite à l'ajout de $m-1$ jobs fictifs. Plus le nombre de machines m augmente par rapport à n et plus la distribution utilisée pour la création de la liste de départ est perturbée. En effet, les jobs fictifs ajoutés ne répondent pas à la distribution utilisée. Il est donc possible qu'un biais existe entre la distribution de départ et l'instance complété avec

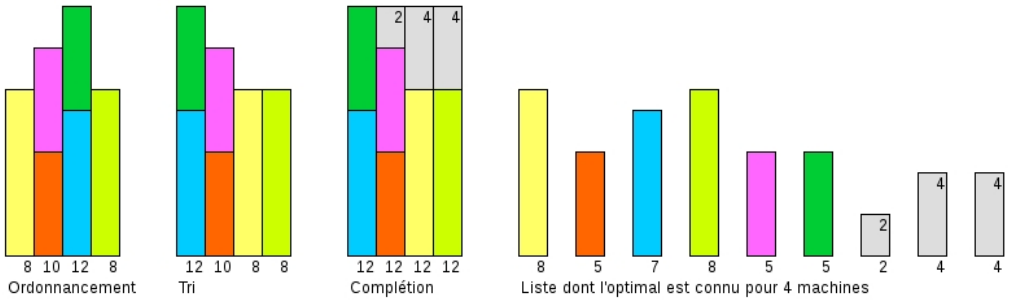

FIGURE 6 – transformation d'une liste pour obtenir l'optimal.

$m-1$ jobs.

Il convient de distinguer la liste de coût native obtenue soit par génération, soit par récupération, dont seule la borne minimale est connue et liste complétée avec $m-1$ temps de jobs nommée M1 qui est une instance dont l'optimal est connu.

Tous les protocoles expérimentaux sont effectués sur les instances M1 et sur le nombre réel de tâches $(n+(m-1))$.

Nous venons de voir comment les listes de temps sont générées et utilisées. Elles peuvent être créées via la production de nombres pseudo-aléatoires selon un nombre défini de jobs et une distribution statistique choisie ou récupérées d'une archive de fichiers journaux de tâches relatifs à de réelles activités de production. De ces listes peuvent être calculées des instances (M1) dont l'optimal est connu, ce qui va permettre une estimation de l'efficacité des algorithmes. Et enfin, plusieurs indicateurs statistiques caractérisant chaque instance les accompagnent.

Nous abordons maintenant la notion de "campagnes" de tests où les instances sont soumises à des heuristiques.

4 Campagnes

Chaque expérience se déroule en "campagne". Une campagne est un ensemble de paramètres expérimentaux (liste de nombres de machines, liste de nombres de jobs, type d'instances à créer, algorithmes à comparer) qui aboutit à un fichier résultat global pour y être analysé.

Ce chapitre aborde l'environnement de tests, le choix des heuristiques implémentés et le déroulement d'une campagne. Est expliqué pour finir, comment les Makespan sont comparés entre eux.

4.1 Environnement de tests

L'environnement de tests est développé en Python pour la partie applicative (génération des listes de temps, calcul de l'instance M1, algorithmes) et en R pour la partie analyse et graphes. Des informations sur la plate-forme sont disponibles en annexe.

Toutes les campagnes ont été effectuées sur un portable doté de 4 Go de ram et équipé d'un processeur Intel core I3 4330u cadencé à 1900 MHz.

4.2 Choix des heuristiques

Pour reproduire l'expérience de Della Croce et Scatamacchia et comparer les résultats, les mêmes heuristiques sont implémentées à l'environnement de test. Soit :

— LPT rule ;
— SLACK ;
— LDM ;
— COMBINE.

Ceci a aussi l'avantage d'expérimenter 4 pistes différentes pour résoudre le problème $P||C_{\max}$:

— LS (list-Scheduling) avec LPT rule ;
— Bin-Packing avec COMBINE ;
— Stratégie gloutonne avec SLACK ;
— Partitionnement de nombres avec LDM.

COMBINE repose sur deux autres heuristiques qui sont aussi implémentées : FFD et MULTIFIT.

4.3 Déroulement d'une campagne

L'objectif est de reproduire le protocole expérimental de Della Croce et Scatamacchia et aussi de l'élargir à d'autres types d'instances et de possibilités de comparaisons. En plus des possibilités de tests sur des instances de 10, 50, 100, 500 et 1000 jobs à planifier sur 5, 10 et 25 machines [5], chaque campagne peut faire varier n de 1 en 1 avec un nombre de départ et un nombre d'arrivée ou suivant une liste. Il en est de même pour le nombre de machines m. Les deux valeurs peuvent aussi évoluer conjointement.

Une campagne boucle sur le nombre de machines choisies, le nombre de jobs à créer pour les listes de temps, par nombre et type de loi statistiques. Pour chaque triplet $\{m, n,$ liste de temps$\}$ sont créées des instances native et M1 qui sont soumises aux algorithmes LPT, SLACK LDM et COMBINE (figure 7).

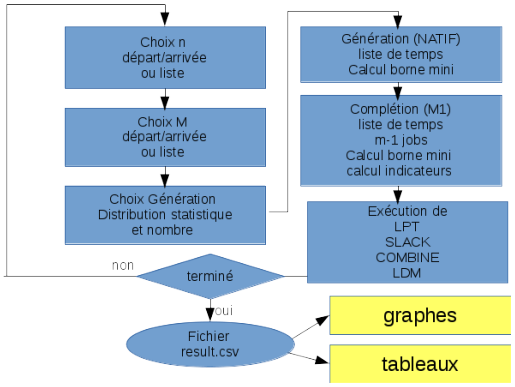

FIGURE 7 – Déroulement d'une campagne.

generateMethode	m	id	seed	n	[a-b]	LowBound	m1_n	m1LowBound	m1Optimal	resultConcerns	algoName	makespan	time
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	Results	LPT	201.0	0.0006611347198486328
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	Results	SLACK	204.0	0.0012745857238769531
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	Results	LDM	202.0	0.1558094024658203
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	Results	COMBINE	200.48	0.008884668350219727
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	m1Results	LPT	264.0	0.0013446807861328125
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	m1Results	SLACK	265.0	0.0023484230041503906
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	m1Results	LDM	264.0	0.2230684757232666
UNIFORM	25	1	9572	100	1.0-100.0	198.92	124	260.0	260.0	m1Results	COMBINE	261.0	0.009851455688476562
UNIFORM	25	2	8625	100	1.0-100.0	209.6	124	271.0	271.0	Results	LPT	215.0	0.0006451606750488281
UNIFORM	25	2	8625	100	1.0-100.0	209.6	124	271.0	271.0	Results	SLACK	217.0	0.0010120868682861328
UNIFORM	25	2	8625	100	1.0-100.0	209.6	124	271.0	271.0	Results	LDM	215.0	0.1554090976715088
UNIFORM	25	2	8625	100	1.0-100.0	209.6	124	271.0	271.0	Results	COMBINE	210.27499999	0.009083271026611328

FIGURE 8 – Contenu du fichier result.csv.

Les 4 algorithmes sont donc exécutés à chaque itération aux mêmes instances i.e aux mêmes valeurs.

8 lignes résultat sont générées et sont intégrées dans un fichier global "result.csv" une fois les boucles de la campagne terminées pour comparaison (figure 8). La structure se décompose ainsi :

- *generateMethode* : Est la méthode qui est utilisée pour générer la liste de temps ;
- m : Est le nombre de machines identiques pur les instances Native et M1 ;
- *id* : Est l'id du triplet {m, n, numéro de liste de temps}. Sert aux regroupements et comparaisons des algorithmes pour une même instance ;
- *seed* : Est la graine produit du moteur de génération de nombres pseudo-aléatoires utilisée pour la création de la liste de temps. Peut servir pour recréer à l'identique une instance ;
- n : Nombre de jobs servant à créer une liste de temps native ;
- $[a - b]$: Paramètres a et b pour les listes de temps uniforme et non-uniforme. Cette information est utilisée pour recréer le protocole expérimental de Della Croce et Scatamacchia ;

— *LowBound* : Borne inférieure de l'instance native ;
— *m1_n* : Nouveau nombre de jobs de l'instance M1 après complétion et calcul de l'optimal ;
— *m1LowBound* : Borne inférieure de l'instance M1 ;
— *m1Optimal* : Optimal calculé sur l'instance M1 ;
— *resultConcerns* : Indique si le Makespan calculé concerne l'instance native ou M1 ;
— *algoName* : Nom de l'algorithme soumis ;
— *makespan* : Makespan trouvé par l'algorithme soumis ;
— *time* : Temps qu'il a été nécessaire pour calculer le makespan par l'algorithme.

4.4 Comparaison Makespan et normalisation

Comparer les makespans sous leur forme brute n'est pas significatif. Les valeurs obtenues sont fonction du contenu des instances.

Il est donc nécessaire de normaliser le résultat pour permettre des comparaisons. 3 pistes sont explorées :

— $\frac{Makespan}{borne\ inférieure}$: Seule piste possible de normalisation des instances natives car seule leur borne inférieure sont connues. C'est la normalisation qui serait utilisée pour comparer les algorithmes sur des instances natives. Mais celles-ci ne sont utilisées que pour calculer des instances M1.
— Makespan - Optimal : Ou makespan absolu [2]. Permet de retirer du résultat l'optimal connu de l'instance M1. Cette valeur tend vers 0 lorsque le résultat converge vers l'optimal mais ne permet pas une comparaison avec le ratio d'approximation $\Gamma(A) = \frac{C_m^A(J)}{C_m^\star(J)}$;
— $\frac{Makespan}{Optimal}$: Ou makespan relatif [2]. Permet d'obtenir le rapport entre le makespan calculé et l'optimal de l'instance M1. Cette valeur tend vers 1 lorsque le résultat converge vers l'optimal. Cette normalisation permet aussi une comparaison avec le ratio d'approximation $\Gamma(A) = \frac{C_m^A(J)}{C_m^\star(J)}$. Cette normalisation est donc utilisée pour comparer les résultats obtenus sur les instances M1.

5 Résultats

LP: Dans la suite on utilise que des instances M1 ?

[a,b]	m	instances NON UNIFORMES	SLACK vs LPT			SLACK vs COMBINE			SLACK vs LDM		
		#	W	E	L	W	E	L	W	E	L
1-100	5	50	31	16	3	30	16	14	0	44	6
	10	40	32	8	0	29	8	3	0	40	0
	25	40	23	17	0	23	17	0	1	39	0
1-1000	5	50	39	10	1	39	10	1	1	44	5
	10	40	40	0	0	33	0	7	3	36	1
	25	40	27	12	1	27	12	1	2	36	2
1-1000	5	50	39	10	1	39	10	1	2	39	9
	10	40	40	0	0	32	0	8	7	29	4
	25	40	28	10	2	28	10	2	5	30	5

[a,b]	m	instances UNIFORMES	SLACK vs LPT			SLACK vs COMBINE			SLACK vs LDM		
		#	W	E	L	W	E	L	W	E	L
1-100	5	50	12	37	1	10	38	2	0	44	6
	10	40	14	20	6	9	21	10	2	27	11
	25	40	10	29	1	4	23	13	4	28	8
1-1000	5	50	32	15	3	31	15	4	1	30	19
	10	40	27	5	8	21	5	14	4	6	30
	25	40	24	12	4	17	6	17	3	10	27
1-1000	5	50	36	12	2	36	11	3	0	12	38
	10	40	37	0	3	30	0	10	2	1	37
	25	40	22	11	7	15	6	19	4	10	26
		780	513	224	43	453	208	129	41	505	234
		%	65,77	28,72	5,51	58,08	26,67	16,54	5,26	64,74	30,00

	Total temps	SLACK	LPT	COMBINE	LDM
		104	75	105	5453
	Rapport à LPT	1,39	1,00	1,40	72,71

FIGURE 9 – Résultat obtenu par Della Croce et Scatamacchia.

5.1 Protocole expérimental de Della Croce et Scatamacchia

Pour comparer SLACK aux autres heuristiques LPT LDM et COMBINE, Della Croce et Scatamacchia utilisent 10 instances uniformes et non-uniformes d'intervalle [a, b] de [1, 100], [1, 1000] et [1, 10000] pour un nombre de machines de 5 et un nombre de jobs de 10, 50, 100, 500 et 1000, et sur un nombre de machines de 10 et 25 pour un nombre de jobs de 50, 100, 500 et 1000. soit un total de :

pour chaque intervalle [a, b] = [1, 100], [1, 1000], [1, 10000] et chaque classe uniforme et non-uniforme.

— 10 instances $\cdot$ 5 couples $\{m = 5, n\}$:
$\{5, 10\}, \{5, 50\}, \{5, 100\}, \{5, 500\}, \{5, 1000\}$;
— 10 instances $\cdot$ 4 couples $\{m = 10, n\}$:
$\{10, 50\}, \{10, 100\}, \{10, 500\}, \{10, 1000\}$;
— 10 instances $\cdot$ 4 couples $\{m = 25, n\}$:
$\{25, 50\}, \{25, 100\}, \{25, 500\}, \{25, 1000\}$.

780 instances.

La figure 9 présente le résultat obtenu dans [5].

		machines		
Archive	algorithme	5	10	25
Early CTC SP2	COMBINE	1,0000000032	1,0000000126	1,0000000316
	LDM	1	1	1
	LPT	1,0000000063	1,0000000253	1,0000000316
	SLACK	1	1	1
LANL CM5	COMBINE	1	1	1
	LDM	1	1	1
	LPT	1	1	1
	SLACK	1	1	1
NASA IPSC	COMBINE	1	1	1
	LDM	1	1	1
	LPT	1	1	1
	SLACK	1	1	1
SDSC Par95	COMBINE	1,0000000631	1,0000000841	1,0000001044
	LDM	1,0000000631	1,0000000841	1,0000001044
	LPT	1,0000000631	1,0000000841	1,0000001044
	SLACK	1,0000000631	1,0000000841	1
SDSC Par96	COMBINE	1	1	1
	LDM	1	1	1
	LPT	1	1	1
	SLACK	1	1	1

FIGURE 10 – Makespan obtenu par heuristique sur 5 archives PWA pour 5, 10, 25 machines.

Le principe est de compter combien de fois SLACK calcule un makespan plus proche de l'optimal que LPT, LDM ou COMBINE. Lorsqu'il gagne contre un algorithme, un point W (win) est comptabilisé. Lorsqu'il perd, un point L (loose) est comptabilisé. Lorsqu'il est équivalent à l'algorithme avec lequel il se mesure, un point E (equivalent) est comptabilisé.

Les auteurs annoncent une amélioration significative de LPT avec SLACK car celui-ci est meilleur à 65.8% des cas contre LPT suivant les 780 instances de référence de la littérature [5]. De ce fait, SLACK peut être vu comme une alternative précieuse à LPT.

5.2 SLACK sur des instances réelles

Les 4 algorithmes sont exécutés sur des listes de temps réelles, provenant de 5 PWA NASA IPSC, Early CTC SP2, LANL CM5, SDSC Par95 et SDSC Par96, qui contiennent respectivement 18066, 75895, 122058, 32128 et 53744 jobs. Ces 5 archives sont transformées en instances de 5, 10 et 25 machines

[a,b]	m	instances NON UNIFORMES	SLACK vs LPT			SLACK vs COMBINE			SLACK vs LDM		
		#	W	E	L	W	E	L	W	E	L
1-100	5	50	44	5	1	40	7	3	0	46	4
	10	40	40	0	0	34	0	6	0	40	0
	25	40	40	0	0	20	0	20	1	38	1
1-1000	5	50	50	0	0	45	0	5	4	42	4
	10	40	40	0	0	36	2	2	0	37	3
	25	40	40	0	0	17	0	23	0	36	4
1-10000	5	50	49	0	1	44	0	6	1	35	14
	10	40	40	0	0	36	0	4	1	22	17
	25	40	40	0	0	16	0	24	1	37	2

[a,b]	m	instances UNIFORMES	SLACK vs LPT			SLACK vs COMBINE			SLACK vs LDM		
		#	W	E	L	W	E	L	W	E	L
1-100	5	50	17	31	2	12	24	14	2	32	16
	10	40	15	17	8	11	6	23	1	16	23
	25	40	22	14	4	7	2	31	7	16	17
1-1000	5	50	40	8	2	32	5	13	3	18	29
	10	40	30	4	6	16	6	18	3	3	34
	25	40	30	4	6	8	3	29	10	0	30
1-10000	5	50	43	4	3	36	2	12	3	4	43
	10	40	32	2	6	21	2	17	5	0	35
	25	40	30	2	8	7	0	33	15	0	25
		780	642	91	47	438	59	283	57	422	301
		%	82,31	11,67	6,03	56,15	7,56	36,28	7,31	54,10	38,59
		Total temps	SLACK		LPT		COMBINE		LDM		
			4,6828		3,0783		12,0274		238,7196		
		Rapport à LPT	1,52		1,00		3,91		77,55		

FIGURE 11 – Résultat obtenu par reproduction Della Croce et Scatamacchia.

et le résultat (makespan relatif) est présenté en figure 10.

SLACK est efficace mais de très peu. Les heuristiques dans cet échantillon sont plus souvent équivalentes.

5.3 Reproduction du protocole expérimental de Della Croce et Scatamacchia

Le même protocole expérimental est exécuté. Le résultat est indiqué figure 11. Les résultats obtenus ne sont pas tout à fait identiques ou avoisinants.

Côté temps, SLACK passe d'un facteur 1.3 à un facteur 1.5 par rapport à LPT. Mais c'est COMBINE qui a le plus de différence en passant d'un facteur 1.40 à un facteur 3.9 par rapport à LPT. Cela peut être du à l'implémentation des algorithmes et à la technologie utilisée.

Côté makespan, SLACK cumule plus de victoires contre LPT en passant de 65.8% à 82.3% de "WIN". Par contre, SLACK perd de son efficacité contre LDM et surtout contre COMBINE. COMBINE qui récupère 36.3% de "WIN" contre SLACK, contre 16.5% précédemment indiqué.

	campagnes	instances	SLACK vs LPT			SLACK vs COMBINE			SLACK vs LDM		
			W	E	L	W	E	L	W	E	L
Min			639	83	33	408	46	284	58	394	287
Max			650	97	58	440	56	317	67	431	326
Moyenne			645,9	92,98	41,12	430,16	53,2	296,64	61,4	409,42	309,18
Ecart type			4,32	5,10	5,17	12,87	4,17	12,22	2,70	13,69	13,95
Uniform/Non Uniform	50	39000	32295	4649	2056	21508	2660	14832	3070	20471	15459
%			82,81	11,92	5,27	55,15	6,82	38,03	7,87	52,49	39,64

FIGURE 12 – Résultat obtenu avec 50 campagnes de tests type “Della Croce et Scatamacchia”.

Peut-être que notre protocole est une singularité statistique. L'expérience est exécutée 50 fois pour s'approcher le plus de la tendance la plus représentative.

Le résultat est résumé en figure 12. La tendance des 50 campagnes corrèle le résultat obtenu précédemment et non pas celui de Della Croce et Scatamacchia.

5.4 Facteurs d'influence sur le comportement des heuristiques

Della Croce et Scatamacchia ne comparent les algorithmes que sur 2 familles de distributions : uniforme et non-uniforme. Distributions générées avec 3 paramètres différents, le résultat est synthétisé avec des instances de 5, 10, 25 machines, et 10, 50, 100, 500, 1000 jobs. Les différences trouvées entre notre expérience et celle de Della Croce et Scatamacchia peuvent être dues à une variété trop grande de paramètres résumés en un seul résultat.

Influence de la distribution Pour vérifier si la distribution statistique utilisée pour générer les listes de temps a une influence sur le résultat nous reprenons la même expérience avec 2 autres familles de distributions : Gamma et beta de paramètres $\alpha = 1$ et $\beta = 1$, sur des instances de 5, 10, 50, 100, 500, 1000 jobs, à ordonnancer sur 5, 10, 25 machines, sur 10 campagnes.

Le résultat est donné figure 13. SLACK descend à 68.7% contre LPT, n'est plus qu'à 43,4% contre COMBINE et à 13.5% contre LDM

Le type d'instance a donc une influence sur le comportement des algorithmes, notamment sur l'efficacité de SLACK.

Influence du nombre de jobs Les résultats précédents sont calculés et cumulés avec des instances de 100, 1000 et 10000 jobs. Peut-être que SLACK

			SLAC vs LPT			SLACK vs COMBINE			SLAC vs LDM		
	campagnes	instances	W	E	L	W	E	L	W	E	L
Min			529	96	134	332	9	420	96	69	599
Max			543	110	152	349	13	435	110	74	613
Moyenne			536,1	100,5	143,4	338,5	11,1	430,4	105,3	71	603,7
Ecart type			5,57	4,12	6,59	4,62	1,20	4,33	5,54	1,63	5,56
Gamma/Beta	10	7800	5361	1005	1434	3385	111	4304	1053	710	6037
%			68,73	12,88	18,38	43,40	1,42	55,18	13,50	9,10	77,40

FIGURE 13 – Résultat obtenu avec 10 campagnes de tests avec des distributions Gamma et Beta.

			SLAC vs LPT			SLACK vs COMBINE			SLAC vs LDM		
	campagnes	instances	W	E	L	W	E	L	W	E	L
Min			28	24	8	14	2	51	15	18	33
Max			46	31	21	23	6	61	28	24	46
Moyenne			38,4	27,5	14,1	18	4,9	57,1	20,9	20,6	38,5
Ecart type			7,69	2,72	5,84	4,03	1,37	4,36	4,91	2,32	6,17
Gamma/Beta	10	800	384	275	141	180	49	571	209	206	385
%			48,00	34,38	17,63	22,50	6,13	71,38	26,13	25,75	48,13

FIGURE 14 – Résultat obtenu avec 10 campagnes de tests avec des distributions Gamma et Beta sur des instances de 1000 jobs.

est meilleurs pour des instances de 100 jobs et cacherait, avec le cumul des résultats, des difficultés pour des instances de 1000 jobs. Pour vérifier si le nombre de jobs a une incidence sur le comportement des heuristiques, nous reprenons la même expérience avec les 2 familles de distributions gamma et beta de paramètres $\alpha = 1$ et $\beta = 1$, des instances uniquement de 1000 jobs à ordonnancer sur 5, 10 et 25 machines, sur 10 campagnes.

Le résultat en figure 14 indique que SLACK ne domine plus LPT avec 48% de “WIN” et laisse l’efficacité à COMBINE et LDM avec respectivement 22,5% et 26,1%.

Influence du cumul des résultats Les campagne précédentes cumulaient des données hétérogènes (nombre de jobs, distributions). Pour vérifier si un cumul de paramètres différents peut masquer dans le résultat un comportement par paramètre, nous retirons la loi de distribution beta dans la génération des listes de temps. Donc 1 familles de distributions gamma de paramètres $\alpha = 1$ et $\beta = 1$, des instances uniquement de 1000 jobs à ordonnancer sur 5, 10 et 25 machines, sur 10 campagnes.

SLACK n’est plus qu’à 25% contre LPT, 15% contre COMBINE et 17% contre LDM (figure 15). Une tentative d’explication de l’efficacité de SLACK par distribution est faite en 5.6.

			SLAC vs LPT			SLACK vs COMBINE			SLAC vs LDM			
	campagnes	instances	W	E	L	W	E	L	W	E	L	
Min			7	18	5	4	1	28	4	11	16	
Max			14	23	13	9	6	32	10	19	21	
Moyenne			10,1	20,9	9	6,3	3,6	30,1	6,8	14,4	18,8	
Ecart type			2,51	1,60	2,58	1,57	1,43	1,85	2,10	2,59	1,32	
Gamma	10	400	101	209	90	63	36	301	68	144	188	
	%		25,25	52,25	22,5	15,75	9	75,25	17	36	47	

FIGURE 15 – Résultat obtenu avec 10 campagnes de tests avec une distribution Gamma sur des instances de 1000 jobs.

Donc un cumul de paramètres différents masque le comportement des heuristiques par paramètres.

5.5 Cartographie des heuristiques

Della Croce et Scatamacchia utilisent des instances de 10, 50, 100, 500 et 1000 jobs à ordonnancer sur 5, 10 et 25 machines. Ce qui représente 13 points de tests (les instances $\{10\ jobs, 10\ machines\}$ et $\{10\ jobs, 25\ machines\}$ n'existent pas car $m \geq n$). En faisant évoluer n de 10 à 1000 et m de 5 à 50, et en consignant l'algorithme (entre LPT et SLACK) qui donne le makespan relatif le plus bas, nous obtenons le graphe en figure 16. Les points rouges représentent les 13 points de test en question.

Les zones jaunes (algorithmes sont équivalents) correspondent à la partie "optimale", où les heuristiques calculent un makespan égal à l'optimal. Cette partie se situe entre une demi-droite $n = m$ et une demi-droite $n = 2m - 1$.

En distribution uniforme SLACK domine largement avec un petit nombre de machines. Mais en montant en m LPT devient un peu plus présent. En distribution non-uniforme SLACK est vraiment plus efficace que Par contre, en distribution gamma LPT devient vraiment plus efficace.

Il est à remarquer que les points rouges du protocole expérimental de Della Croce et Scatamacchia sont placés principalement à des coordonnées où SLACK domine (dans les instances à distribution uniforme et non-uniforme).

La figure 17 donne la photographie d'une infime partie du comportement de chaque algorithme par famille de distribution (gamma, non-uniforme et uniforme) pour des instances, dont le nombre de machines est compris entre 5 et 50 et le nombre de jobs varie de 10 à 1000.

Il est à noter plusieurs phénomènes remarquables :

— En distribution gamma les algorithmes se stabilisent assez rapidement plus le nombre de jobs n est élevé.
Le makespan relatif dessine une bande oblique de "pires cas" qui suit la ligne $n = 2m$ pour rapidement converger vers 1 lorsque n augmente.

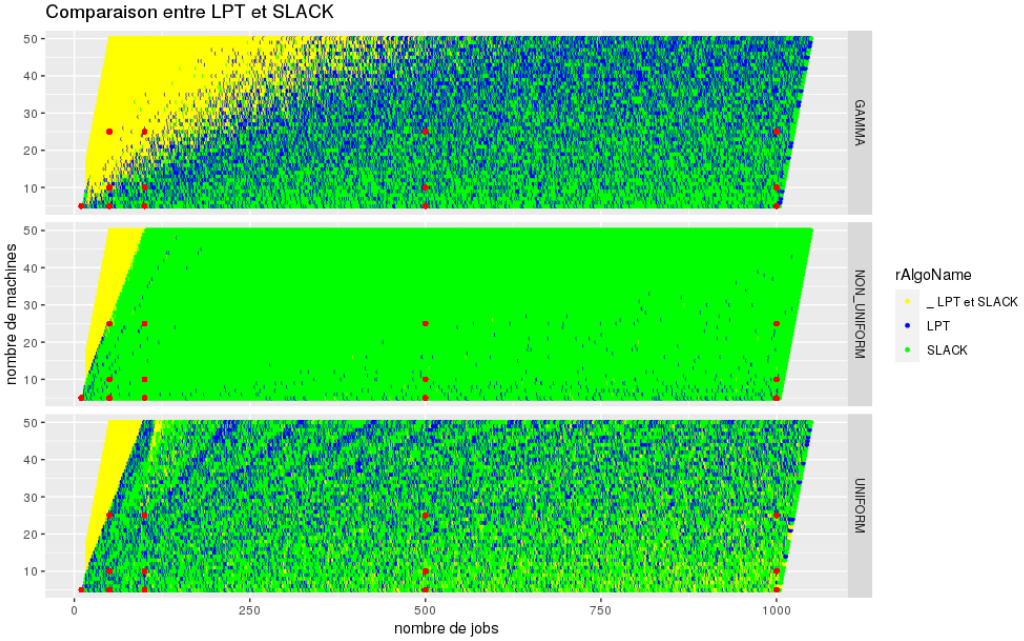

FIGURE 16 – comparaison LPT et SLACK sur 3 distributions avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

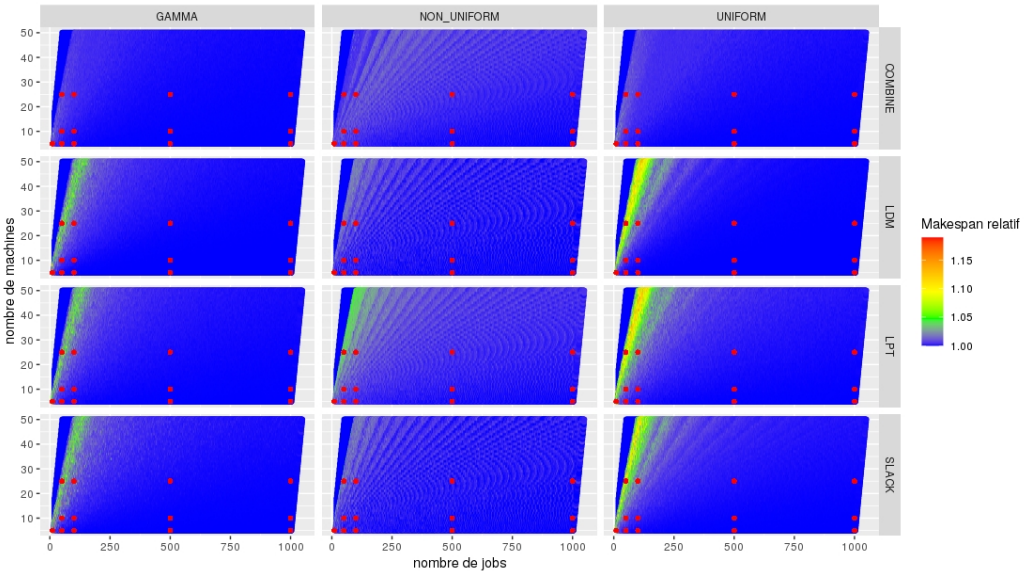

FIGURE 17 – comparaison LPT SLACK COMBINE et LDM par algorithme et par distribution avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

Cette bande est de plus en plus épaisse avec un nombre élevé m de machines ;

— En distribution uniforme SLACK converge moins rapidement que LPT et LDM. Mais de façon générale plus le nombre de machines m est élevé et plus la convergence vers 1 du makespan relatif nécessite un nombre élevé n de jobs.
Le makespan relatif dessine des vagues obliques de "pires cas" qui débutent suivant la ligne $n = 2m$ pour s'estomper petit à petit et converger vers 1 lorsque n augmente. Ces vagues sont de plus en plus larges avec un nombre élevé m de machines et selon l'algorithme ;

— En distribution non-uniforme tous les algorithmes ont du mal à converger vers un makespan relatif à 1. Les 4 heuristiques sont à peu près équivalentes en valeurs et SLACK est plus efficace par rapport à LPT de très peu.
Le makespan relatif dessine des formes de "pires cas" qui font penser à des interférences ;

— les 13 points rouges ne sont pas forcément placés à des coordonnées stratégiques qui nous permettraient une comparaison plus catégorique.

5.6 SLACK et la famille NON-UNIFORME

Pourquoi SLACK est très efficace dans une instance dont les temps sont générés à l'aide de la règle NON-UNIFORME ?

La stratégie gloutonne de SLACK consiste à trier les temps dans le sens décroissant, créer des tuples de m jobs de cet ensemble trié et trier ensuite chaque tuple par ordre décroissant de la différence entre le premier et le dernier élément de chaque tuple (slack).

En reformulant la stratégie, SLACK crée des paquets de m jobs qui vont s'empiler exactement un à un sur les m processeurs. Le premier paquet (tuple) présente une importante différence entre le premier et le dernier job, ce qui peut faire penser à un trapèze rectangle dont le sommet du haut est fortement penché. Le deuxième paquet dont la différence des deux jobs d'extrémités est un peu moins importante va un peu compenser celle du premier paquet, pour former un trapèze rectangle dont le sommet est moins penché. Et ainsi de suite jusqu'à avoir empilé tous les tuples, et rendre le sommet du haut du trapèze rectangle le plus horizontal possible. Et à chaque itération, chaque élément de chaque tuple a été également répartis sur chacun des processeurs.

Exemple 3. *Déroulement de SLACK dans un environnement idéal.*
Soit la liste de temps déjà ordonnée

$$P = \{89, 84, 55, 51, 51, 49, 47, 15, 10, 9, 7, 4\}$$

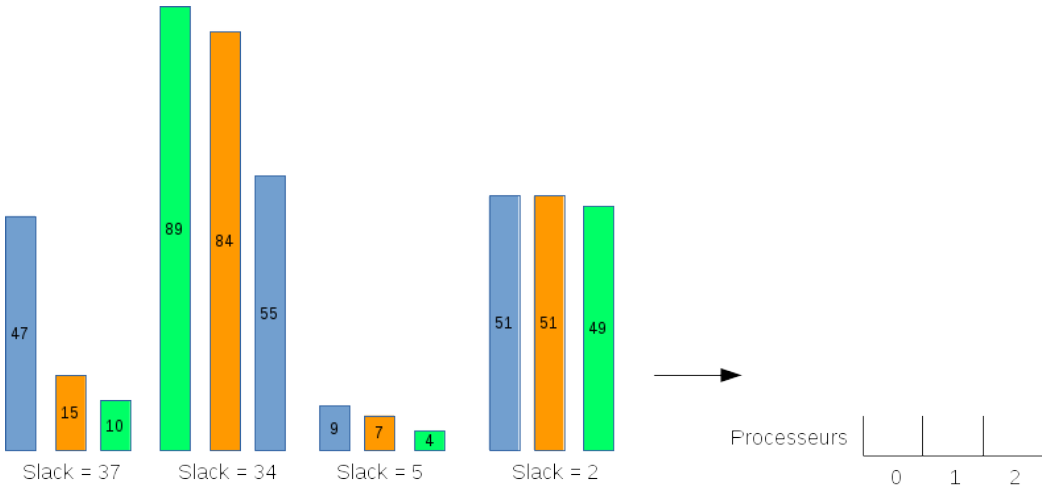

FIGURE 18 – Préparation des tuples pour SLACK avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

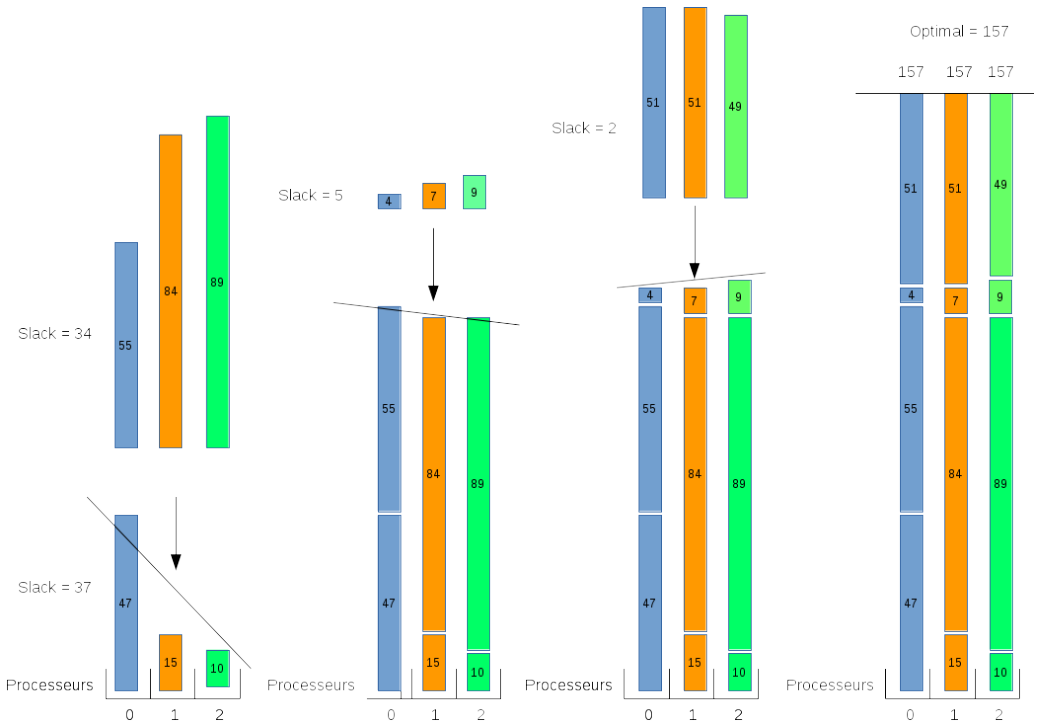

FIGURE 19 – Ordonnancement des tâches par tuple avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

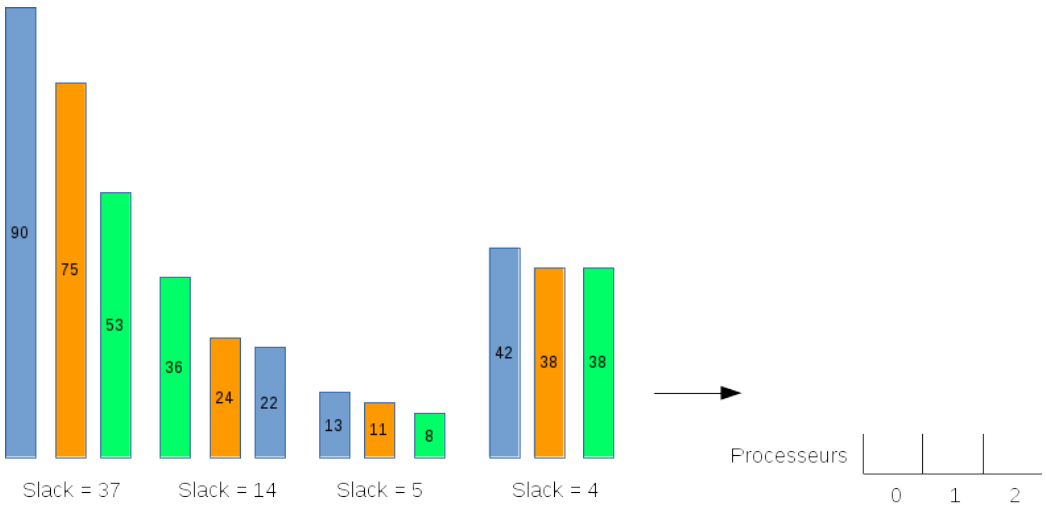

FIGURE 20 – préparation des tuples pour SLACK avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

à ordonnancer sur 3 machines. Ce qui donne les tuples ordonnés suivants (figure 18) :

- $T_1 = \{47, 15, 10\}$, *slack = 37 ;*
- $T_2 = \{89, 84, 55\}$, *slack = 34 ;*
- $T_3 = \{9, 7, 4\}$, *slack = 5 ;*
- $T_4 = \{51, 51, 49\}$, *slack = 2.*

Pour chaque tuple, chaque tâche est affectée à chaque processeur (figure 19), ce qui abouti vers une répartition équilibrée de temps par machine. Le makespan est égale à l'optimal

Par contre si un job a un temps qui déstabilise cette stratégie, comme par exemple le premier job du deuxième tuple qui a un temps inférieur au slack du premier tuple, celui-ci provoque un déséquilibre a chaque itération qui fini par un décalage entre makespan et optimal.

Plus formellement si dans

$$P = \{p_1, \ldots, p_m\}, \{p_{m+1}, \ldots, p_{2m}\}, \ldots, \{p_{n-m}, \ldots p_n\}$$

nous avons $p_{m+1} < p_m - p_1$, à l'opération du deuxième tuple, 2 jobs seront affectés au même processeur. Ce qui aura comme effet de déstabiliser l'harmonie des affectations des jobs de chaque tuple à chaque processeur et de provoquer un décalage entre l'optimal et le makespan.

Exemple 4. *Déroulement de SLACK dans un environnement non idéal.*

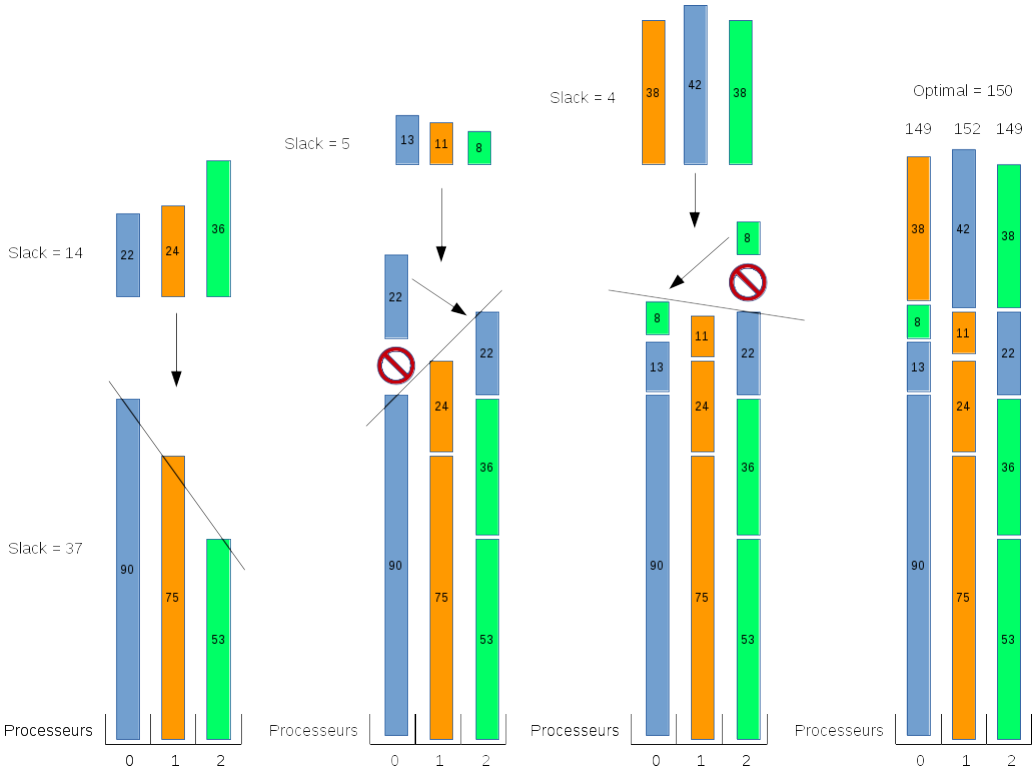

FIGURE 21 – Ordonnancement des tâches par tuple avec $10 \leq n \leq 1000$ et $5 \leq m \leq 50$

Soit la liste de temps déjà ordonnée

$$P = \{90, 75, 53, 42, 38, 38, 36, 24, 22, 13, 11, 8\}$$

à ordonnancer sur 3 machines. Ce qui donne les tuples oordonnés suivants

- $T_1 = \{90, 75, 53\}$, *slack* $= 37$;
- $T_2 = \{36, 24, 22\}$, *slack* $= 14$;
- $T_3 = \{13, 11, 8\}$, *slack* $= 5$;
- $T_4 = \{42, 38, 38\}$, *slack* $= 4$.

(figure 20 Le premier job de $p_{1 \in T2} = 36 < slack(T_1) = 37$ *Après l'affectation de* T_1 *nous avons la répartition des charges des processeurs suivante* $\{(90), (75), (53)\}$. *Puis vient l'affectation de* T_2, *avec en premier le job* $p_{T_21} = 36$ *nous avons la répartition des charges des processeurs suivante* $\{(90), (75), (89 : 53, 36)\}$. *Le troisième processeur vient d'être chargé mais son poids* (89) *est toujours inférieur au premier processeur* (90). *Ce qu'aucun job de ce tuple ne sera affecté au premier processeur cette fois-ci (figure 21).*

Cette perturbation va se propager jusqu'à la fin de l'algorithme. Le makespan trouvé sera différent de l'optimal.

Dans une liste de temps de type non-uniforme de paramètres [a, b] le slack maximum possible est égal à $slack_{max} = 0.9(b - a)$. Pour perturber le fonctionnement de SLACK il faudrait une valeur de temps inférieure à $slack_{max}$. Or, contrairement à gamma, beta, uniforme, voire reel avec PWA, une liste de temps non-uniforme ne contient aucune valeur comprise entre $0.9(b-a)$ et $0.2(b-a)$ et seulement 2% des éléments de la liste sont inférieurs à $0.2(b - a)$. Ce qui réduit presque à 0 les possibilités de générer une valeur qui peut potentiellement perturber SLACK.

Tester SLACK dans une instance de type non-uniforme oriente le résultat d'efficacité. Les listes réelles ne peuvent pas être non uniforme comme défini dans [8] car il y aura toujours une valeur entre les bornes $0.9(b-a)$ et $0.2(b-a)$ pour invalider l'appartenance d'une archive de tâches réelles à une règle non uniforme.

6 Discussion

SLACK est plus efficace que LPT suivant 13 points de tests dans un protocole expérimental. Mais il suffit de sortir de ce périmètre pour obtenir une conclusion différente. Della Croce et Scatamacchia ont utilisé et combiné des variables soit, trop différentes pour sortir un schéma de causalité pertinent, soit pas assez représentatives pour en déduire un comportement général. En

effet, certains points où SLACK est très efficace ont masqués d'autres résultats où SLACK est moins percutant. Aussi, certains points représentent des instances où tous les algorithmes convergent sur l'optimal ce qui réduit encore le nombre de points de tests. L'expérience est donc conditionnée par quelques points. Nous nous retrouvons devant un échantillon de tests biaisé et ne pouvons pas affirmer, même sur une distribution non-uniforme, que SLACK améliore LPT.

Au delà de ce problème se pose la question : comment tester une heuristique par rapport à d'autres heuristiques par l'expérimentation ? D'un point de vue théorique ces heuristiques ont un ratio d'approximation, une complexité en temps, voire un comportement asymptotique. Mais le ratio d'approximation n'est qu'une indication prouvée existante à un instant donné et prévaut tant qu'un nouveau ratio d'approximation plus proche de l'optimal n'est pas trouvé et/ou prouvé. Même si entre 2 heuristiques dont l'une à un ratio d'approximation supérieur à l'autre, cette heuristique supposée théoriquement moins efficace peut obtenir de meilleurs résultats par l'expérience. Le principe de randomisation est essentiel pour créer des échantillons servant de support expérimental. Or un algorithme comparé à d'autres, qui présente de meilleurs résultats sur une distribution supposée perturber le comportement de toute heuristique ne peut pas être gratifié d'algorithme améliorant un autre algorithme car il peut devenir mauvais sur une autre distribution. Pour tester une heuristique et en extraire une conclusion catégorique, il faudrait la tester sur toutes distributions et combinaisons de distributions en faisant évoluer les paramètres de chacune d'elles (a et b pour uniforme et non-uniforme, α et β pour les distributions gamma et beta ...), et par rapport à une large variation de n et m. Ce qui constitue un problème combinatoire.

Pour finir, les temps des jobs soumis aux processeurs sont des séquences dont la répartition ressemble à des lois statistiques déduites d'observations de phénomènes ou d'activités naturels. Or la règle non-uniforme n'est pas inspirée des hasards naturels. Les lois gamma, beta, exponentielles ont beaucoup plus de chance de représenter un échantillon de jobs contrairement à la règle non-uniforme.

7 Conclusion

Nous venons de reproduire à l'identique le protocole expérimental de Della Croce et Scatamacchia qui compare les heuristiques LPT, LDM, COMBINE et SLACK sur les 2 distributions uniformes et non-uniformes. Ce protocole n'est pas reproductible car nous ne trouvons pas tout à fait les mêmes résultats. Estimant ce protocole peut-être restreint, nous élargissons les pa-

ramètres expérimentaux à d'autres valeurs de n et m et à d'autres distributions statistiques telles que beta, gamma et exponentielle. Les résultats obtenus montrent que SLACK n'améliore pas de façon significative LPT contrairement à la conclusion de Della Croce et Scatamacchia. Le protocole expérimental des auteurs n'est pas assez large pour avoir une bonne idée du comportement de leur stratégie gloutonne par rapport aux autres algorithmes.

L'expérience n'est pas reproductible car l'échantillon comporte un biais. Les points utilisés pour tester les heuristiques ne sont pas assez nombreux. SLACK parait plus efficace dans une règle non-uniforme mais cela est dû à la construction de l'heuristique elle-même. Della Croce et Scatamacchia ont sélectionné des instances non-uniformes car cette règle de génération de nombres est supposée perturber LPT. Mais cette règle n'est pas universelle et ne représente pas les autres lois statistiques qui ont plus de chance d'être représentées dans une instance réelle et où les algorithme se comportent différemment.

Comparer des heuristiques par l'expérimentation est une tâche difficile tant les possibilités sont nombreuses. Les bornes d'approximations ne donnent qu'une indication de pire cas. L'étude des surfaces asymptotiques dépendantes de n et m de ces algorithmes serait beaucoup plus intéressante. car plus que le ratio d'approximation ces surfaces bornent la morphologie des heuristiques et peut-être indépendamment d'une loi de distribution.

Appendices

génération des listes de temps avec Python

Pour créer des listes de temps, nous utilisons la bibliothèque par défaut de génération de nombres pseudo-aléatoires de Python, selon différentes règles statistiques.

Les deux types de générations suivants implémentés génèrent des nombres entiers [1]. Ce sont les deux seuls types de génération utilisés dans le protocole expérimental de comparaisons des heuristiques dans [5].

— Uniforme : les nombres pseudo-aléatoires générés, sont uniformément répartis entre deux entiers a et b.

```
for i in range(n):
        rand = random.uniform(a,b)
```

— Non-uniforme. définie en tant que "NON-UNIF" dans [8]. Cette méthode produit une liste de temps de traitements dont 98% des éléments sont uniformément répartis dans l'intervalle $[0.9(b-a), b]$, et le reste (2%) est uniformément réparti dans l'intervalle $[a, 0.2(b-a)]$.

```
n98 = int((98*n) / 100)
    a1 = 0.9*(b-a)
b1 = b
a2 = a
b2 = 0.2*(b-a)
        ...
for i in range(n98):
        rand = random.uniform(a1,b1)

...

for i in range(n-n98):
        rand = random.uniform(a2,b2)
```

D'autres type de générations sont implémentées basés sur des lois statistiques (distribution). Celles-ci génèrent des nombres réels et acceptent des paramètres qui ont un effet sur "l'écrasement" des courbes représentant les distributions et entrent dans le calculs de la variance.

— Loi gamma : Accepte deux paramètres k ("alpha" dans l'implémentation) qui affecte la forme, et Θ ("beta" dans l'implémentation) qui affecte l'échelle.

1. La commande random.uniform(a,b) de Python renvoie un réel. La transformation en entier est effectuée à l'aide de la commande round(n).

```
for i in range(n):
    rand = random.gammavariate(alpha, beta)
```

— Loi exponentielle : Cas particulier de la loi gamma. Elle représente la durée de vie d'un événement. Elle accepte un paramètre λ (lambda dans l'implémentation) qui affecte l'échelle.

```
for i in range(n):
    rand = random.expovariate(lambd)
```

— Loi beta. Accepte deux paramètres de forme α qui affecte la forme, et β (toujours à 1 dans notre cas) qui affecte aussi la forme.

```
for i in range(n):
    rand = random.betavariate(alpha,1)
```

Algorithmes PTAS optimisés pour $\epsilon = \frac{1}{5}$ et $\epsilon = \frac{1}{6}$

Algorithme 7 : PTAS $\frac{1}{5}$-dual

1 Tant qu'il y a 1 pièce j avec $p_j \in [0, 6, 1]$, emballer j avec $L[1 - p_j]$, si cette pièce existe. Emballer j toute seule sinon.

2 Tant qu'il y a 2 pièces i, j avec $p_i, p_j \in [0, 5, 0.6[$, emballer i et j ensembles.

`// Chaque pièce restante a une taille inférieure à 0.5`

3 Tant qu'il existe 3 pièces, dont la plus grande fait au moins 0.4, trouver $L[0.3, 0.4, 0.5]$ et les pacquer ensembles.

4 Tant qu'il existe des pièces dont la taille est dans [0.4,0.5[, pacquer les 2 plus grandes ensembles.

`// Chaque pièce restante a une taille inférieure à 0.4`

5 Prendre la plus petite pièce des pièces restantes. Si $p_j > 0.25$, alors pacquer le reste des pièces en 3-bin. Sinon, $p_j = 0.25 - \delta$ pour $\delta \geq 0$. Si 3 autres pièces existent, pacquer j avec $L[0.25, \frac{+\delta}{3}, 0.25 + \delta, 0.25 + 3 \cdot \delta]$. Sinon, pacquer le reste dans 3-bin.

Algorithme 8 : PTAS $\frac{1}{6}$-dual

1 Tant qu'il existe une pièce j, telle que $p_j \in (\frac{2}{3}, 1)$, emballer p_j avec $L[1 - p_j]$.

2 Estimer le nombre total de 1-bin ou 2-bin d'une solution optimale du reste de l'instance. Pour chacun de ces bacs, l'emballer avec $L[\frac{1}{2}, \frac{2}{3}]$ si de telles pièces existent, sinon, l'emballer avec $L[\frac{2}{3}]$.

`// Pour le reste de la procédure, seuls les bacs qui ont // au moins 3 pièces sont considérés.`

`// Chaque pièce restante a une taille inférieure à` $\frac{2}{3}$

3 Pour chaque pièce j restante, avec une taille $\frac{1}{2} + \delta; \delta \geq 0$, emballer j avec $L[\frac{1}{4} - \frac{\delta}{2}, \frac{1}{3} - \delta]$.

`// Chaque pièce restante a une taille inférieure à` $\frac{1}{2}$

4 Estimer le nombre de 4-bin qui contient une pièce dont la taille est dans l'intervalle $[\frac{5}{12}, \frac{1}{2}[$ dans un packing optimal. Pacquer chaque bac avec $L[\frac{7}{36}, \frac{5}{24}, \frac{1}{4}, \frac{1}{2}]$.

5 Pour chaque pièce restante de taille $\frac{5}{12} + \delta, \delta \geq 0$, l'emballer dans un 3-bin avec $L[\frac{7}{24} - \frac{\delta}{2}, \frac{5}{12} - \delta]$.

`// Chaque pièce restante a une taille inférieure à` $\frac{5}{12}$

6 Estimer le nombre de 3-bin d'une solution optimale pour le reste de l'instance. Pour chacun d'eux, le pacquer avec $L[\frac{1}{3}, \frac{5}{12}, \frac{5}{12}]$.

`// Pour le reste de la procédure, seuls les bacs qui ont // au moins 4 pièces sont considérés.`

7 Pour chaque pièce j de taille $\frac{1}{3} + \delta, \delta \geq 0$, emballer i avec $L[\frac{2}{9} - \frac{\delta}{3}, \frac{1}{4} - \frac{\delta}{2}, \frac{1}{3} - \delta]$.

`// Chaque pièce restante a une taille inférieure à` $\frac{1}{3}$

8 Estimer le nombre de 5-bin qui contienne une pièce dont la taille est comprise dans l'intervalle $[\frac{7}{24}, \frac{1}{3}[$ d'une solution optimale. Emballer de tels bacs avec $L[\frac{17}{96}, \frac{13}{72}, \frac{9}{48}, \frac{5}{24}, \frac{1}{3}]$.

9 Tant qu'il existe des pièces dont la taille est supérieure à $\frac{7}{24}$ Prendre la plus grande pièce de taille $p_j = \frac{7}{24} + \delta, \delta \geq 0$. Pacquer i avec $L[\frac{17}{72} - \frac{\delta}{3}, \frac{13}{48} - \frac{\delta}{2}, \frac{7}{24} + \delta]$.

10 Considérer la plus petite pièce j. Si $p_j > \frac{1}{5}$, emballer les pièces restantes à raison de 4 pièces par bac. Si $p_j = \frac{1}{5} - \delta, \delta \geq 0$, pacquer j avec $L[\frac{1}{5} + \frac{\delta}{4}, \frac{1}{5} + \frac{2 \cdot \delta}{3}, \frac{1}{5} + \frac{3 \cdot \delta}{2}, \frac{7}{24}]$.

Algorithme PA

Algorithme 9 : PA

1 Déterminer la borne inférieure (LB) suivant l'algorithme de McNaughton [18].
2 Déterminer la borne supérieure UB suivant l'heuristique LPT (algorithme 1).
3 **Si** $LB = UB$ **alors**
4 **Renvoyer** la solution optimale est trouvée par l'heuristique LPT
5 **répéter**
6 Résoudre le programme de relaxation linéaire avec $C_{max} = LB$
7 **Si** *la relaxation est possible* **alors**
8 **Si** *la solution obtenue est entière (donc faisable)* **alors**
9 **Renvoyer** la solution obtenue qui est optimale
10 **sinon**
11 Générer des nouvelles inégalités (inégalités et/ou inégalités transitoires) et les ajouter à la nouvelle relaxation linéaire
12 **sinon**
13 incrémenter LB d'une unité
14 **jusqu'à** *Génération des d'inégalités impossible*
15 **Renvoyer** Résoudre le problème avec un algorithme Branch&Bound .

Bibliographie

[1] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. 1763. *MD computing : computers in medical practice*, 8(3) :157–171, 1991.

[2] Anne Benoit, Louis-Claude Canon, Redouane Elghazi, and Pierre-Cyrille Heam. *Update on the Asymptotic Optimality of LPT*. PhD thesis, Inria Grenoble-Rhône-Alpes, 2021.

[3] Bo Chen and N Chris. Potts, and gerhard j woeginger. a review of machine scheduling : Complexity, algorithms and approximability. *Handbook of combinatorial optimization*, pages 1493–1641, 1999.

[4] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1) :1–17, 1978.

[5] Federico Della Croce and Rosario Scatamacchia. The longest processing time rule for identical parallel machines revisited. *Journal of Scheduling*, 23(2) :163–176, 2020.

[6] Gyorgy Dosa. The tight bound of first fit decreasing bin-packing algorithm is ffd (i). In *International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer, 2007.

[7] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10) :2967–2982, 2014.

[8] Antonio Frangioni, Emiliano Necciari, and Maria Grazia Scutella. A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. *Journal of Combinatorial Optimization*, 8(2) :195–220, 2004.

[9] Michael R Garey and David S Johnson. "strong"np-completeness results : Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3) :499–508, 1978.

[10] MR Garey and DS Johnson. Computers and intractability : A guide to the theory of np-completeness. freeman, san francisco, 1979. 1982.

[11] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9) :1563–1581, 1966.

[12] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier, 1979.

[13] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1) :144–162, 1987.

[14] Manuel Iori and Silvano Martello. Scatter search algorithms for identical parallel machine scheduling problems. In *Metaheuristics for scheduling in industrial and manufacturing applications*, pages 41–59. Springer, 2008.

[15] Narendra Karmarkar and Richard M Karp. *The differencing method of set partitioning*. Computer Science Division (EECS), University of California Berkeley, 1982.

[16] Richard E Korf. Multi-way number partitioning. In *IJCAI*, volume 9, pages 538–543, 2009.

[17] Chung-Yee Lee and J David Massey. Multiprocessor scheduling : combining lpt and multifit. *Discrete applied mathematics*, 20(3) :233–242, 1988.

[18] Robert McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1) :1–12, 1959.

[19] Wil Michiels, Jan Korst, Emile Aarts, et al. Performance ratios for the karmarkar-karp differencing method. *Electronic Notes in Discrete Mathematics*, 13 :71–75, 2003.

[20] Ethel Mokoto. Scheduling to minimize the makespan on identical parallel machines : an lp-based algorithm. *Investigacion Operative*, 97107, 1999.

[21] Frank Nielsen and Vincent Garcia. Statistical exponential families : A digest with flash cards. *arXiv preprint arXiv :0911.4863*, 2009.

[22] Bastian Rieck. Basic analysis of bin-packing heuristics. *arXiv preprint arXiv :2104.12235*, 2021.

[23] Michael H Rothkopf. Scheduling independent tasks on parallel processors. *Management Science*, 12(5) :437–447, 1966.

Abstract

Consider the scheduling problem $P||C_{\max}$ which consists in scheduling n independent jobs on m identical machines in order to minimize the total processing time (makespan). By revisiting the LPT rule heuristic, SLACK, an algorithm based on a greedy approach, was developed. To test the efficiency of SLACK, the authors compare it to the LPT rule, COMBINE and LDM heuristics by subjecting them to uniform and non-uniform instances. The results show that SLACK significantly improves LPT. We try to reproduce the experiment identically to validate the reproducibility. But we do not find the same results. To validate this bias we extend the experimental protocol to compare other gamma, beta instances. With these new experimental parameters SLACK does not significantly improve LPT rule. The comparison protocol of Della Croce et Scatamacchia is not generalist, and the non-uniform instance is not sufficient to characterize the behavior of heuristics.

Key words: Scheduling . Identical parallel machines . Heuristics . LPT rule . Empirical .

Résumé

Considérons le problème d'ordonnancement $P||C_{\max}$ qui consiste à planifier n jobs indépendants sur m machines identiques dans le but de minimiser le temps total de traitement (makespan). Della Croce et Scatamacchia ont développé, en revisitant l'heuristique LPT rule, SLACK un algorithme basé sur une stratégie gloutonne. Pour éprouver l'efficacité de SLACK, les auteurs le comparent aux heuristiques LPT rule, COMBINE et LDM en les soumettant à des instances uniformes et non-uniformes. D'après le résultat SLACK améliore de façon significative LPT. Nous tentons de reproduire l'expérimentation à l'identique afin valider la reproductibilité. Mais nous ne trouvons pas les mêmes résultats. Pour valider ce biais nous élargissons le protocole expérimental de comparaisons d'autres instances gamma, beta. Avec ces nouveau paramètres expérimentaux SLACK n'améliore pas de façon significative LPT rule. Le protocole de comparaison de Della Croce et Scatamacchia n'est pas généraliste, et l'instance non-uniforme ne suffit pas pour caractériser le comportement d'heuristiques.

Mot clés : Ordonnancement . Machines parallèles identiques . Heuristiques . LPT rule . Empirique .