Chapitre 4 : Approche exacte pour le problème de flow-shop à deux machines avec des temps de latence

4.1 Introduction

Nous nous intéressons dans ce chapitre au problème de *flow-shop* à deux machines avec des temps de latence. Dans un premier temps nous étudions le problème de *flow-shop* à deux machines avec des temps d'exécution unitaires et des temps de latence quelconques. Dans un deuxième temps, nous étudions le problème de *flow-shop* à deux machines avec des temps de latence et d'exécution quelconques.

4.2 Branch and bound avec des temps d'exécution unitaires

Rappelons que le problème de *flow-shop* à deux machines avec des temps d'exécution unitaires et des temps de latence quelconques est *NP-difficile* au sens fort. L'utilisation de la méthode énumérative de *branch and bound* est par conséquent justifiée. Comme mentionné au Chapitre 3, le développement de cette méthode est basé sur l'utilisation des bornes inférieures et supérieures et une stratégie de branchement. Notons que cette partie est basée sur la thèse de doctorat de Yu [1996] et sur l'article de Moukrim et Rebaïne [2005]. Nous débutons par la description des quatre bornes inférieures utilisées pour élaguer les branches non potentielles. Ensuite, nous présentons les bornes supérieures qui sont un ensemble d'heuristiques qui servent à limiter l'exploration de certaines branches. Après cela, nous décrivons les règles de dominances qui vont nous orienter vers une meilleure exploration des branches

4.2.1. Bornes inférieures

Les bornes inférieures sont toujours utilisées dans les algorithmes de *branch and bound*. En effet, à chaque nœud de l'arbre de recherche, on calcule une borne inférieure, correspondant au plus petit *makespan* qui peut être trouvé à partir de ce nœud. Si la valeur de cette borne inférieure est plus grande que la valeur du *makespan* courant, alors

on élague toute la branche issue de ce nœud. Dans ce qui suit, nous calculons quatre bornes inférieures ; la borne qui représente la plus grande valeur sera gardée.

- **Première borne inférieure (LB1)**

La première borne inférieure *LB1* est calculée à la racine de l'arbre de *recherche.* Cette borne génère la valeur du *makespan* minimale [Lenstra, 1994]. Notons que si le nombre de jobs n est inférieur ou égal à 5, alors la borne LB1 donne toujours le *makespan* optimal [Yu, 1996]. Mais, si n est supérieur ou égal à 6, alors il existe des instances pour lesquelles la borne inférieure n'est pas atteinte. Par exemple, la séquence des jobs 4-4-4-0 0-0 produit une borne inférieure égale à 9, tandis que l'optimum génère un *makespan* de valeur 10.

Lemme 1 [Yu, 1996]

Si Opt représente la valeur optimale, alors la relation suivante est vérifiée :

$$Opt \geq LB1 = \underset{1 \leq k \leq n}{Max} \left\{ \left\lceil \sum_{i=1}^{k} \frac{\tau_i}{k} \right\rceil + k + 1 \right\}, \qquad (1)$$

où $\tau_1 \geq \tau_2 \geq \ldots \geq \tau_n$.

Exemple 4-1

Soit le nombre de job suivant : $n = 6$. Les temps de latence des différents jobs sont :

Jobs *j*	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Tableau 4-1 : Temps d'exécution du problème pour $n = 6$ avec des temps de latences de l'Exemple 4-1

Le calcul de la première borne inférieure *LB*1 est comme suit :

$$LB1=\max\{6+1+1, \lceil (6+5)/2 \rceil + 2+1, \lceil (6+5+4)/3 \rceil + 3+1, \lceil (6+5+4+3)/4 \rceil + 4+1,$$
$$\lceil (6+5+4+3+2)/5 \rceil + 5+1, \lceil (6+5+4+3+2+1)/6 \rceil + 6+1\} = \max\{8, 9, 9, 10, 10, 11\} = 11.$$

- **Deuxième borne inférieure (LB2) [Yu, 1996]**

La deuxième borne inférieure *LB*2 peut être calculée à chaque nœud interne de l'arbre de recherche. En fait, elle est utile pour les solutions partielles qu'on construit au fur et à mesure qu'on avance dans cet arbre.

Corollaire 1 [Yu, 2004]

Soit $\delta = (\alpha, \beta)$ *une permutation de N jobs où* α *est une sous séquence fixée. Alors la relation suivante est vérifiée.*

$$Opt \geq Max\{LB1(N), |\alpha| + LB1(N-\alpha)\},$$

où LB1(K) est la borne inférieure LB1 appliquée aux jobs de l'ensemble K.

Preuve :

Il est clair que les jobs restants appartenant à la séquence N-α vont commencer leur exécution à partir d'un temps $t = \alpha$. De plus, la borne inférieure, correspondant aux jobs appartenant à la séquence N-α, est clairement *LB1* (N -α). Par conséquent, le *makespan* de l'ensemble de jobs, étant donné la séquence α, est clairement, $|\alpha| + LB1(N-\alpha)$ (Figure 4-1).

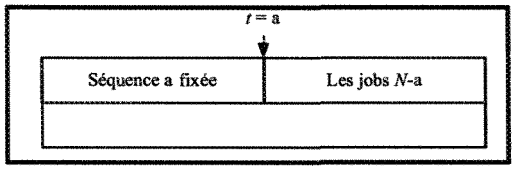


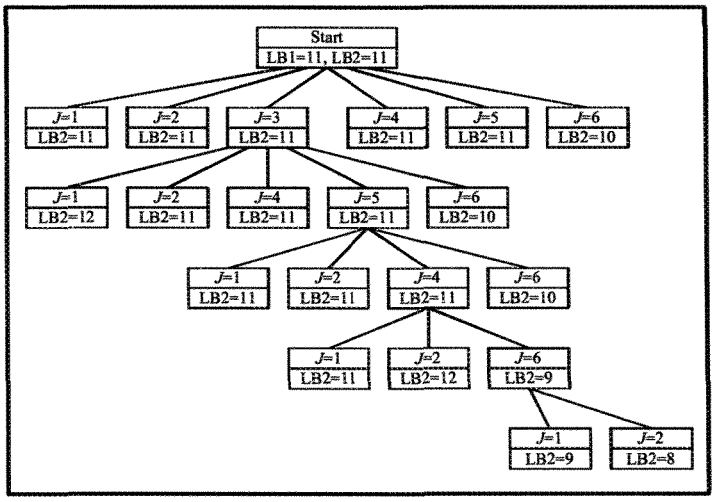
Figure 4-1: Le makespan de l'ensemble des jobs dans le cas unitaire

Exemple 4-2

Soit l'instance de l'Exemple 4-1 et $\alpha = (3,5,4)$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6. On les ordonne suivant l'ordre décroissant. On place tout d'abord la séquence α, ensuite les N -α jobs restants. Le temps de latence des jobs seront placés dans cet ordre : 3, 5, 4, 6, 2 et 1. On obtient :

LB1 (6, 2, 1)= max $\{\lceil(6)/4\rceil+4+1, \lceil(6+2)/5\rceil+5+1, \lceil(6+2+1)/6\rceil+6+1\}$= max {7, 8, 9}=9 alors ***LB2***=$|\alpha|$ + *LB1* (6, 2, 1)=3+9=12.

En appliquant la borne inférieure *LB2* sur quelques branches de l'arbre de recherche, on obtient l'Arbre 4-1 :



Arbre 4-1 : LB2 sur quelques branches de l'arbre dans le cas unitaire

- **Troisième borne inférieure (LB3) [Moukrim et Rebaïne, 2005]**

Soit, *r(j)*, la date au plus tôt d'un job *j* sur la machine *M2*. Étant donné α jobs déjà ordonnancés, la date au plus tôt d'un job *j* est $\Pi(j) + \tau_j$ où $\Pi(j)$ est la position d'exécution du job $j \in \alpha$ sur *M1*. Pour les jobs restants qui appartiennent à la séquence $n-\alpha$, leur date au plus tôt ne peut pas être inférieure à $|\alpha| + 1 + \tau_j$. Il est clair que l'ordonnancement des jobs suivant leur date au plus tôt va générer une borne inférieure du *makespan*.

Pour tout *p* de 1 à $|\alpha|$, le job $\alpha(p)$ est ordonnancé sur *M1* à la case *p*. Pour tout job *j*, nous allons définir sa date de début au plus tôt sur *M2*.

Pour p = 1 to $|\alpha|$ faire
début
$j = \alpha(p)$
$r(j) = p + \tau_j$
Fin pour

Algorithme 4-1 : Première partie de l'algorithme de la troisième borne inférieure dans le cas unitaire

Ensuite, pour tout job j n'appartenant pas à la sous séquence α, sa date d'exécution sur *M1* est au plus tôt $|\alpha|$. Donc, sa date de début d'exécution sur *M2* est au plus tôt : $|\alpha| + 1 + \tau_j$. En exécutant les jobs selon ces dates de début d'exécution au plus tôt sur *M2*, on obtient une borne inférieure à la fin de traitement de tous les jobs sur *M2*. La date au plus tôt de l'ensemble des jobs est donnée par l'Algorithme 4-2 [Moukrim et Rebaïne, 2005]:

```
▪Trier les jobs dans un tableau s selon l'ordre croissant les r(j). On aura:
▪r(s(1)) <= r(s(2)) <= ... <=r(s(n)).
▪Poser t(s(1)) = r(s(1))
▪Répéter
        i = i + 1;
        t(s(i)) = t(s(i-1)) + 1;
        Si t(s(i)) < r(s(i )) alors t(s(i)) = r(s(i));
Jusqu'à ( i > n)
```

Algorithme 4-2 : L'algorithme de la troisième borne inférieure dans le cas unitaire

La valeur de la borne inférieure *LB3* sera alors égale à $t(s(n)) + 1$.

Exemple 4-3

L'exemple ci-dessous illustre le calcul de la borne inférieure *LB*3. Notons que nous on traite ici le même exemple donné pour illustrer *LB*1 et *LB*2. Soit $\alpha = (3,5,4)$. Les jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6.

	$j \in \alpha$			$j \notin \alpha$		
Jobs *j*	3	5	4	1	2	6
Temps de latence τ_j	3	5	4	1	2	6
r(j)	4	7	7	5	6	10

Tableau 4-2 : Temps de latence pour $n = 6$ de l'Exemple 4-3

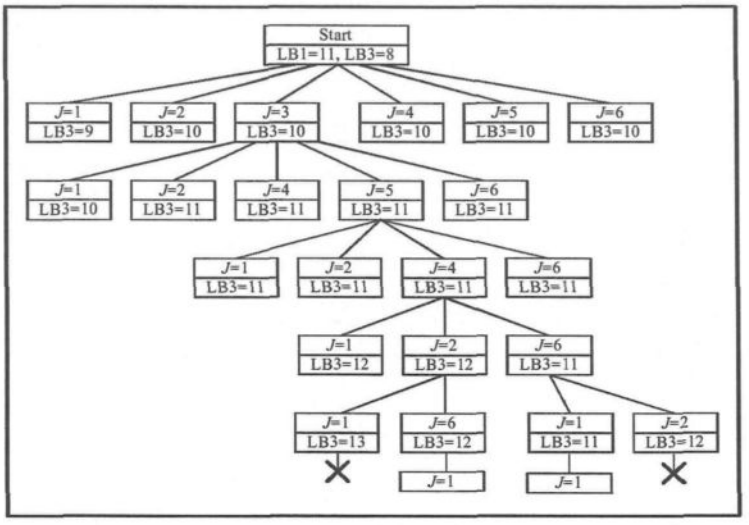
On trie les jobs dans un tableau s selon l'ordre croissant des $r(j)$. Ensuite, on applique l'algorithme pour obtenir une $LB3 = t(s(6)) + 1$:

Jobs *j*	3	1	2	4	5	6
Temps de latence τ_j	3	1	2	4	5	6
r(j)	4	5	6	7	7	10
s(i)	3	1	2	4	5	6
t(s(i))	4	5	6	7	8	10

Tableau 4-3 : Application de LB3 sur l'Exemple 4-3

On aura alors $LB3 = 10+1 = 11$.

En appliquant la borne inférieure *LB3* sur quelques branches de l'arbre de recherche, on obtient l'Arbre 4-2.



Arbre 4-2: Application de la borne *LB3* sur quelques branches de l'arbre de recherche dans le cas unitaire

- **Quatrième borne inférieure (LB4) [Moukrim et Rebaïne, 2005]**

La borne inférieure LB4 est similaire à la borne LB1, présentée dans le

Lemme 1. La seule différence réside dans le fait que *LB*4 tient compte des valeurs réelles des temps de latence. En effet, étant donnés α jobs déjà ordonnancés, leurs dates au plus tôt est $\Pi(j) + \tau_j$ pour tout job $j \in \alpha$, $\Pi(j)$ étant la position d'exécution du job j sur M1 dans cet ordonnancement. Si on place un job j à la position $\Pi(j) + \tau_j$ sur *M2*, on peut trouver un job $i \in \alpha$ et $i \neq j$ placé à la même position. Donc, on est obligé de modifier la position de ce dernier. Il s'ensuit que son temps de latence va augmenter. Cette modification consiste à augmenter le temps de latence correspondant au prochain emplacement libre sur M2. Les nouveaux temps de latence sont calculés à l'aide de l'algorithme [Moukrim et Rebaïne, 2005]:

Pour $p = 1$ to $|\alpha|$ faire
début
  j=$\alpha(p)$
  Soit q la période de temps où est exécuté le job j sur M2
    $\tau_j = q - p - 1$
  fin

Algorithme 4-3 : L'algorithme de la quatrième borne inférieure dans le cas unitaire

Notons que les temps de latence des jobs qui n'appartient pas à α ne sont pas modifiés. Pour calculer la borne *LB*4, on applique la même formule utilisée auparavant pour le calcul de la borne *LB*1. Cette formule sera appliquée sur l'ensemble des jobs après modification des temps de latence.

Exemple 4-4

Soit l'exemple ci-dessous :

Jobs *j*	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Soit $\alpha = (3,5,4)$, donc $|\alpha| = 3$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 1, 2 et 6. En appliquant la borne *LB*4 sur notre exemple, on obtient l'ordonnancement de la Figure 4-2:

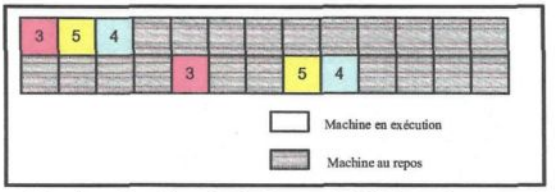


Figure 4-2: Application de la borne *LB*4 dans le cas unitaire

On traite seulement les temps de latence des jobs appartenant à α :

- Pour $p = 1$: $j = 3$, $\tau_3 = 3$. Donc, le temps modifié de $\tau_3 = 5-1-1 = 3$ (dans ce cas, on n'a pas de changement)
- Pour $p = 2$: $j = 5$, $\tau_5 = 5$. Donc, le temps modifié de $\tau_5 = 8-2-1 = 5$. (Dans ce cas, on n'a pas de changement)
- Pour $p = 3$: $j = 4$, $\tau_4 = 4$. Donc, le temps modifié de $\tau_4 = 9-3-1 = 5$. (Dans ce cas, on a fait un changement)

Après avoir modifié le temps de latence, notre tableau sera le suivant :

Jobs *j*	3	5	4	6	2	1
Temps de latence τ_j	3	5	5	6	2	1

Tableau 4-4 : Les nouveaux temps de latence de l'Exemple 4-4

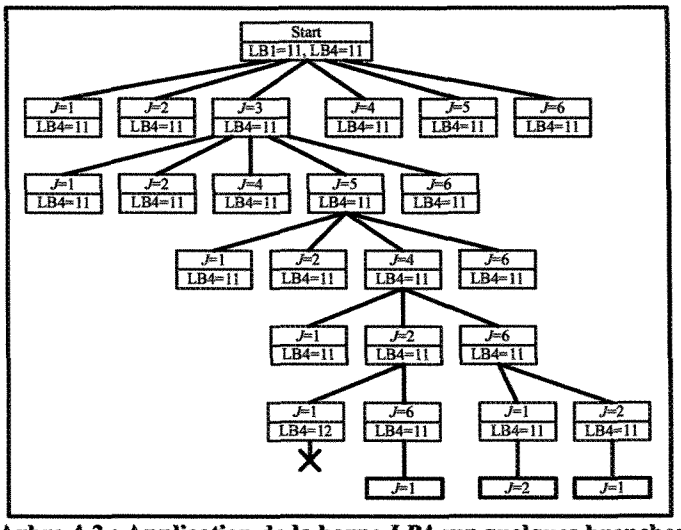
La borne *LB*4 = *LB*1 (avec temps de latence modifiés)

$$LB1=\max\{6+1+1,\lceil(6+5)/2\rceil+2+1,\lceil(6+5+5)/3\rceil+3+1,\lceil(6+5+5+3)/4\rceil+4+1,$$

$$\lceil(6+5+5+3+2)/5\rceil+5+1,\lceil(6+5+5+3+2+1)/6\rceil+6+1\}$$

LB1 = max {8, 9, 10, 10, 11, 11} = 11.

En appliquant la quatrième borne inférieure *LB*4 sur notre problème de *flow-shop,* on obtient une partie de notre arborescence. Ces branches sont décrites dans l'Arbre 4-3.



Arbre 4-3 : Application de la borne *LB4* sur quelques branches de l'arbre de recherche dans le cas unitaire

La borne inférieure retenue pour cet exemple est la borne qui nous donne la plus grande valeur entre ces quatre bornes inférieures calculées.

4.2.2. Les bornes supérieures [Moukrim et Rebaïne, 2005]

Une borne supérieure permet de rejeter certaines branches explorées, par simple propagation de contraintes. Ces bornes supérieures sont un ensemble d'heuristiques appliquées sur les nœuds de l'arborescence.

Dans cette section, on présente quatre familles d'heuristiques. Chacune d'elles ordonnance les jobs d'une séquence donnée suivant trois ordres différents. Les trois règles utilisées par chacune de ces heuristiques sont :

- Première règle :

Elle consiste à ordonnancer la séquence de jobs sur la première machine *M1* suivant l'ordre décroissant du temps de latence, et au plus tôt sur la deuxième machine *M2*.

Exemple 4-5

Soit une séquence de jobs de taille $n = 6$. S présente les temps de latence de cette séquence. On a alors $S = \{6, 5, 4, 2, 1, 3\}$. Après avoir appliqué la première règle, on aura cet ordre: 6, 5, 4, 3, 2 et1.

- Deuxième règle :

Cette règle consiste à placer les jobs d'une manière alternative sur la première machine *M1*. Elle est donnée par la stratégie qui permet d'extraire en premier une sous séquence de jobs ayant des temps de latence successifs de différence au moins de deux. Ensuite, on réapplique la même règle sur l'ensemble des jobs restants, et ainsi de suite jusqu'à épuisement de la séquence initiale. Nous obtenons ainsi un nouvel ordre qui sera exécuté au plus tôt sur la deuxième machine *M2*.

Exemple 4-6

Soient les temps de latence suivants : 6, 5, 4, 2, 1 et 3. Pour appliquer la deuxième règle sur la séquence *S*, on doit suivre les étapes suivantes :

1. Extraction d'une sous séquence de jobs ayant des temps de latence successifs de différence au moins de 2. Ces temps de latence seront stockés dans une nouvelle séquence *F*. On aura alors : $F = \{6, 4, 2\}$. Les temps de latence restant dans la séquence *S* sont : 5, 1 et 3
2. On répète le même processus, décrit par le point (1), sur les jobs restant de la séquence *S* jusqu'à son épuisement. On aura : 5 et 3. On fait une mise à jour de la séquence *F* qui sera $F = \{6, 4, 2, 5, 3\}$. Il reste seulement un job qui sera ajouté à la fin de la séquence *F*.
3. La séquence finale des temps de latence sera la suivante : $F = \{6, 4, 2, 5, 3, 1\}$.

- Troisième règle :

Elle présente le même principe que la règle précédente. Seulement dans ce cas, l'alternance ne s'applique pas sur toute la séquence. Elle consiste à prendre en premier une sous séquence qui contient seulement deux jobs ayant des temps de latence successifs de différence au moins de deux. Et on refait le même principe jusqu'au traitement de toute la séquence initiale.

Exemple 4-7

Soit les temps de latence suivants : 6, 5, 4, 3, 2 et 1. Dans ce cas, l'alternance se fait deux à deux, on aura alors : 6, 4, 5, 3, 2 et 1.

Il est clair que la complexité de ces différentes heuristiques décrites ci-dessus est $O(n\log n)$ puisqu'elle est dominée par la procédure de tri. Dans ce qui suit, on présente les différentes heuristiques utilisées pour les bornes supérieures. On distingue quatre bornes supérieures qui sont : *UB1, UB2, UB3, UB4*. Chacune de ces bornes utilise les trois règles citées ci-dessus. Au total, nous présentons 12 heuristiques. Notons que ces heuristiques sont conçues pour être utilisées pour les nœuds internes de l'arbre de recherche, c'est-à-dire qu'il est supposé qu'une sous-séquence jobs, α, est déjà fixée.

- **Première borne supérieure (UB1)**

La première heuristique utilise un ordre topologique (placer les jobs sur les n premières périodes de temps tels qu'ils se présentent). On considère une sous séquence de jobs α. Pour calculer la première borne supérieure, il faut tout d'abord compléter la sous séquence de jobs par le reste des jobs qui n'appartiennent pas à α. Ces derniers doivent être placés à la suite de la sous-séquence α. Donc, ces jobs vont être affectés au plus tôt sur la période de temps $|\alpha|$ de la première machine *M1*, suivant un certain ordre. L'ordre choisi est l'une des règles d'ordonnancement citées plus haut. La séquence finale obtenue sera exécutée sur *M1* et puis, sur *M2*. Par exemple, si on a déjà fixé l'ordre d'une sous séquence α = (1, 2 et 3) alors il reste les jobs 4, 5 et 6 ayant les temps de latence : 5, 2 et 7. Ensuite, on ordonnance ces derniers suivant l'ordre décroissant. Donc, la borne *UB1* est donnée par la valeur du *makespan* pour la séquences 1, 2, 3, 6, 4 et 5. On applique alors l'Algorithme 4-4.

p : position d'un job dans α

j : un job appartenant à la séquence α

n : nombre total des jobs

Pour $p = 1$ jusqu'à n faire

début

$$j=\alpha(p)$$

Placer j sur la période de temps p sur la machine M1

Placer j sur la première période de temps libre sur M2 à partir de la période de temps $p+1+\tau_j$

Fin pour

Algorithme 4-4 : l'algorithme de la première borne supérieure dans le cas unitaire

Mais, dans le cas où $\alpha = (1,2)$ et les temps de latence des jobs restants sont : 6, 5, 4 et 3. On applique la deuxième règle sur les jobs restants. On aura l'ordre suivant : 6, 4, 5, et 3. L'ordonnancement généré est illustré par la Figure 4-3. La première borne supérieure ***UB*1** est égale à **12.**

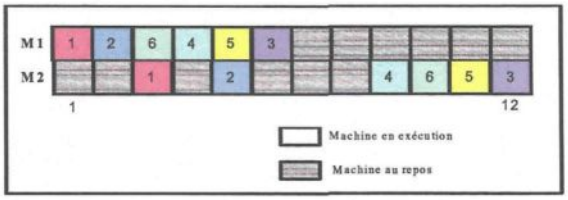


Figure 4-3: Application d'*UB*1 sur une séquence de jobs en utilisant la deuxième règle d'ordonnancement dans le cas unitaire

Mais dans le cas où on applique *UB*1 sur la séquence de jobs ordonnancée suivant la troisième règle, on obtient les mêmes résultats. Si on prend $\alpha = (6)$ alors les temps de latence des jobs restants seront : 5, 4, 3, 2 et 1. Le nouvel ordre sera : 5, 3, 4, 2 et 1. L'ordonnancement sur les deux machines est représenté par la Figure 4-4. Donc, La première borne supérieure ***UB*1** est égale à **12.**

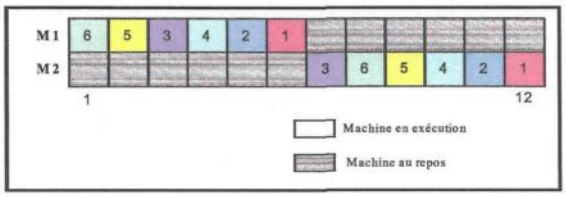


Figure 4-4: Application d'*UB*1 sur une séquence de jobs en utilisant la troisième règle d'ordonnancement dans le cas unitaire

- **Deuxième borne supérieure (UB2)**

La deuxième heuristique consiste tout d'abord, à placer les jobs appartenant à α Ensuite, on place les n-α jobs au plus tôt sur la deuxième machine M2 et au plus tard sur la première machine *M1*, par rapport à leur position sur *M2*. Donc, un job $j \in n-\alpha$ sera affecté au plus tôt à la position $|\alpha|+\tau_j+1$. On obtient l'Algorithme 4-5. $M(1,k)$ et $M(2,l)$ désignent respectivement l'emplacement d'un job i à la $k$ème position sur la première machine et la $l$ème position sur la deuxième machine [Moukrim et Rebaïne, 2005].

N : nombre total des jobs.
α : est une sous séquence de job ordonnancée.
On ordonnance les jobs n-α suivant l'une des règles d'ordonnancement
Pour i = 1 jusqu'à n faire
Début
$K = 0$
Répéter
$K = k+1$
Jusqu'à $M(1,k)$ = libre et $M(2,k+1+\tau_i)$=libre
Placer le job j à la position k sur M1 et à la position $k+1+\tau_i$ sur M2
Mettre Jusqu'à $M(1,k)$ = occupée et $M(2,k+1+\tau_i)$ = occupée
Fin pour

Algorithme 4-5 : L'algorithme de la deuxième borne supérieure dans le cas unitaire

Exemple 4-8

Pour calculer la borne *UB*2, on traite le même exemple pour illustrer le calcul d'*UB*1. Soit $\alpha = (1,2)$. Les temps de latence des jobs restants sont : 6, 5, 3 et 4. On applique les trois règles d'ordonnancement sur cet exemple:

Jobs j	1	2	3	4	5	6
Temps de latence τ_j	1	2	3	4	5	6

Tableau 4-5 : Temps de latence n = 6 de l'Exemple 4-8

- La séquence du job obtenue en appliquant la première règle est la suivante : 1, 2, 6, 5, 4 et 3 générant un *makespan* de valeur 12 comme illustré à la Figure 4-5.

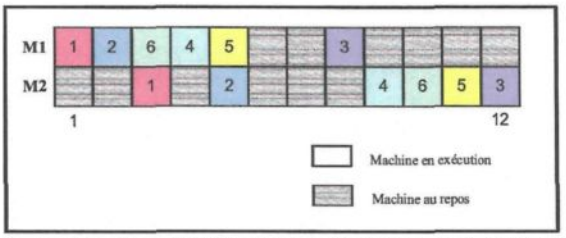


Figure 4-5: Application d'*UB2* sur une séquence de jobs en utilisant la première règle d'ordonnancement dans le cas unitaire

– En appliquant la deuxième règle d'ordonnancement, on obtient: 1, 2, 6, 3, 5 et 4. En exécutant cette séquence sur les deux machines, on obtient un *makespan* de valeur 12, (Figure 4-6). Par conséquent, *UB2 = 12*.

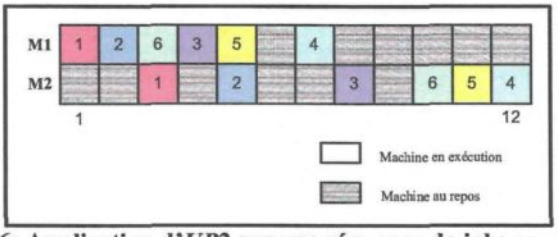


Figure 4-6: Application d'*UB2* sur une séquence de jobs en utilisant la deuxième règle d'ordonnancement dans le cas unitaire

– La troisième règle produira le même résultat que la règle précédente, puisqu'on obtient la même séquence de jobs 1, 2, 6, 3, 5 et 4.

- **Troisième borne supérieure (UB3)**

La stratégie de cette troisième borne *UB3* consiste à placer un job $j \in n-\alpha$, le plus tôt possible dans un emplacement libre sur *M1,* disons au temps *p*. Ensuite, on déduit sa place sur *M2*, car on peut se trouver dans le cas où deux jobs se confondent sur le même emplacement. À ce moment, on avance jusqu'à la prochaine position libre sur *M2*. En procédant de la sorte, le temps de latence de ce job va changer. Après avoir placé un job *j* au temps *q* sur *M2*, on doit replacer le job *j* sur *M1* (le plus tard possible) de telle manière vérifie que (*q-p*) soit le plus proche du temps de latence τ_j. L'Algorithme 4-6 illustre borne supérieure *UB3* [Moukrim et Rebaïne, 2005].

Ordonnancer les jobs n'appartenant pas à α suivant l'une des règles d'ordonnancement

Pour $i = 1$ à n faire

/* Trouver la première place libre sur M1 où le job $i \in n - \alpha$ pourrait être exécuté */

$k = 0$

Répéter

$k = k+1$

Jusqu'à $M(1, k)$ = libre

/* Déterminer la première case libre sur M2 où le job i pourrait être exécuté si on suppose que i est exécuté sur la case k sur M1 */

$q = k + \tau_i$

Répéter

$q = q+1$

Jusqu'à $M(2, q)$ = libre

Placer le job i à la période de temps q sur M2

/* Déterminer la case où le job i sera exécuté le plus tard possible sur M1 sachant qu'il est exécuté à la case q sur M2. */

$R = q - \tau_i$

Répéter

$r = r - 1$

Jusqu'à $M(1, r)$ = libre

Placer le job i sur la période de temps r sur M1

Fin pour

Algorithme 4-6 : l'algorithme de la troisième borne supérieure dans le cas unitaire

Exemple 4-9

Considérons l'instance de l'exemple précédent. On obtient l'ordonnancement illustré par la Figure 4-7. Notons que les trois règles produisent le même résultat. Donc, la valeur $UB3 = 13$.

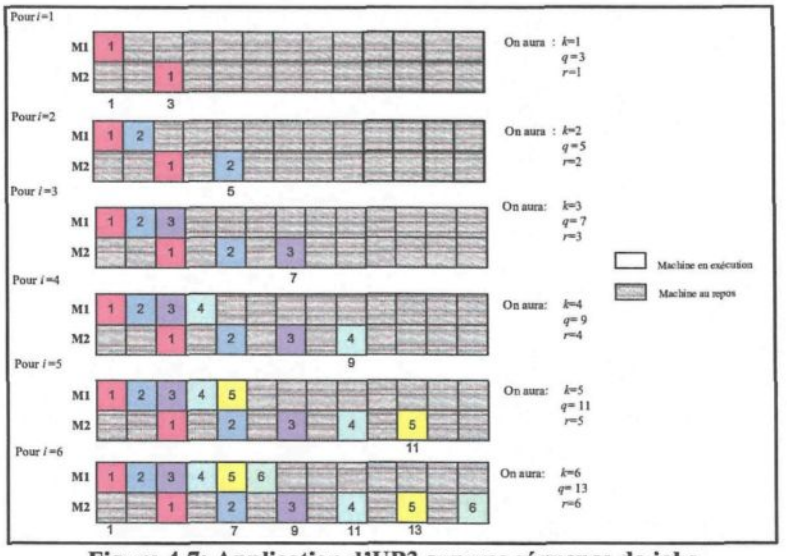


Figure 4-7: Application d'UB3 sur une séquence de jobs

- **Quatrième borne supérieure (UB4)**

La quatrième borne supérieure *UB*4 présente une heuristique qui traite séparément les jobs à temps de latence pairs et impairs. Tout d'abord, on commence par ordonner les jobs n'appartenant pas à α suivant l'une des règles d'ordonnancement. On place les jobs appartenant à α sur les deux machines. Après, on vérifie si le premier job entre les n-α est pair ou impair. Dans le cas où ce dernier est impair, on affecte le plus tôt possible sur *M1* et *M2* les jobs impairs suivis des jobs pairs. Mais dans le cas contraire, on affecte tout d'abord les jobs pairs. On suppose maintenant que le premier job de la séquence est impair. Soit un job impair $j \in n-\alpha$, placé au plus tôt à la position $|\alpha|$ sur *M1* et $|\alpha|+\tau_j$ sur M2. Ensuite, on place les jobs pairs le plus tard possible sur *M1* par rapport à leur position sur *M2*.

Soit un job *i* pair appartenant à $n-\alpha-(jobs\ impairs)$. Ce job est placé au plus tôt à la position $|\alpha|+|jobs\ impairs|+\tau_i$ sur *M2*. Si on se trouve dans un cas où deux jobs se confondent sur le même emplacement alors on passe à l'emplacement suivant. La valeur de la borne *UB4* est le *makespan* total obtenu après avoir placé les jobs sur *M1* [Moukrim et Rebaïne, 2005].

Exemple 4-10

Soit $\alpha = (3,5)$. Les temps de latence des jobs restants qui n'appartiennent pas à α sont : 6, 1, 2 et 4.

- On ordonnance les temps de latence des jobs suivants selon la première règle, notre séquence sera la suivante : 3, 5, 6, 4, 2 et 1. la valeur de la borne *UB*4 est égale à **12** (Figure 4-8).

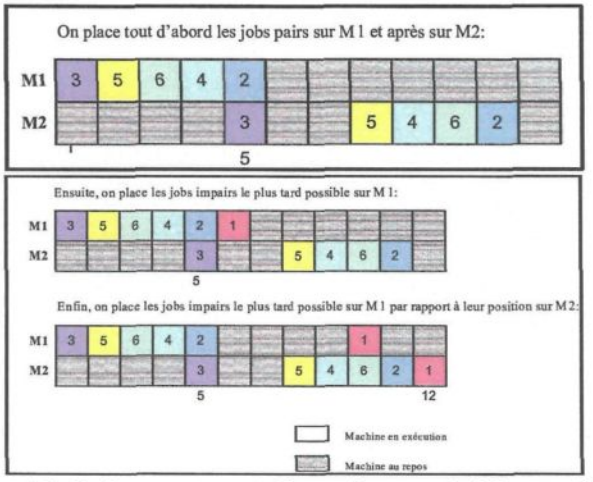


Figure 4-8: Ordonnancement d'une séquence de jobs en utilisant la première règle et la borne *UB*4

- En appliquant la deuxième et troisième règle sur les temps de latence de la séquence de jobs, on aura: 3, 5, 6, 1, 2 et 4 et la borne *UB*4 = 11. L'ordonnancement obtenu est illustré par la Figure 4-9.

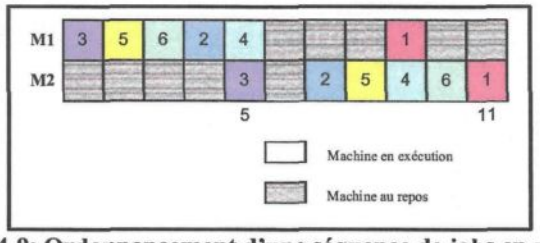


Figure 4-9: Ordonnancement d'une séquence de jobs en utilisant la deuxième règle et la borne *UB*4

4.2.3. Règles de dominance

Les règles de dominance présentent un ensemble de conditions et de critères. Elles permettent d'exclure des solutions partielles. Ces règles sont souvent utilisées pour raffiner la méthode de *branch and bound* et palier aux faiblesses de certaines bornes.

- **Première règle de dominance**

Une règle de dominance a pour but de réduire le temps de traitement et accélérer le calcul de l'algorithme de *branch and bound.*

Lemme 2 [Mokrim et Rebaïne, 2005]

Notons que M2(j) la période de temps où est exécuté le job j sur M2 sachant que j est exécuté sur M1 à la période de temps $|\alpha| + 1$. *La règle de dominance est la suivante : S'il existe un job j' dans I -* α *tel que* $\tau_{j'} > \tau_j$ *et* $M2(j) - (|\alpha| + 1) - 1 \geq \tau_{j'}$ *alors il est inutile de considérer le nœud où le job j est placé à l'emplacement* $|\alpha| + 1$ *sur M1.*

Exemple 4-11

Soit les jobs suivant $I = \{1, 2, 3, 4, 5\}$. On a $\alpha = (1, 2)$. Les jobs restants sont : 3, 4 et 5.

Jobs *j*	1	2	3	4	5
Temps de latence τ_j	2	2	2	1	0

Tableau 4-6 : Temps de latence *n* = 5 de l'Exemple 4-11

Si on place $j = 5$ alors $M2(j)=6$. On doit vérifier les deux conditions, on aura :

- $\tau'_j > \tau_j$ car $\tau'_j = 2$.
- $M2(j)-(|\alpha|+1)-1 = 6-(2+1)-1 = 2 \geq \tau'_j$

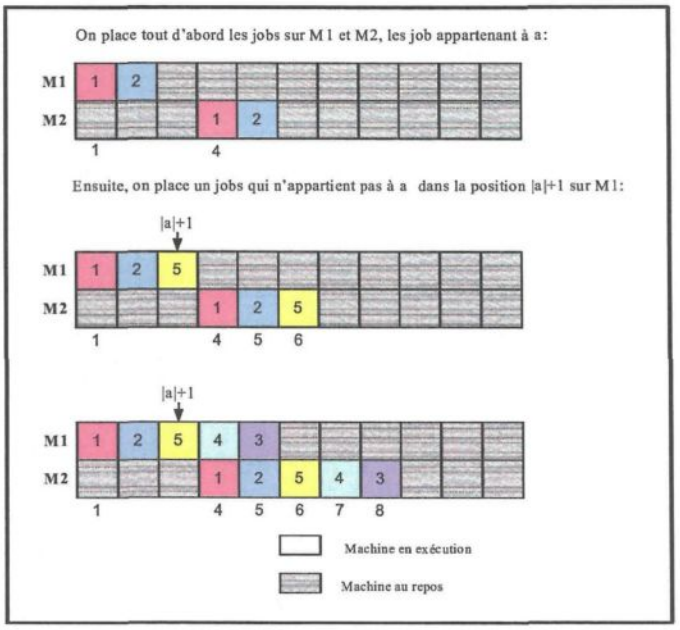


Figure 4-10 : Application de la première règle de dominance

Les deux conditions sont vraies. Donc, on n'a pas besoin d'explorer le reste de l'arbre à partir de ce nœud. On obtient alors l'ordonnancement suivant en échangeant les places des jobs 3 et 5. L'ordonnancement présenté à la Figure 4-11 a le même *makespan* que celui présenté à la Figure 4-10.

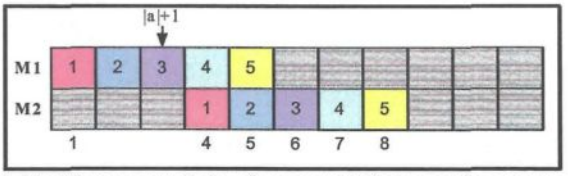
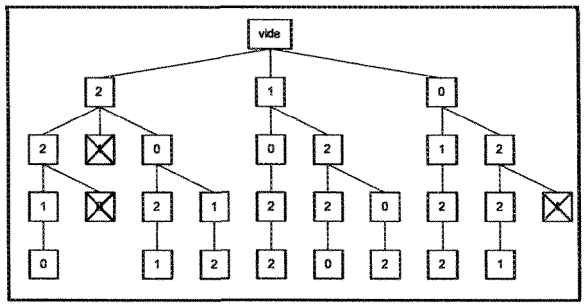


Figure 4-11: Placement de la séquence de jobs après avoir appliqué la première règle de dominance

Considérons l'exemple où les jobs ont les temps de latence suivants : 2, 2, 1, 0. On obtient l'Arbre 4-4. Les nœuds non explorés grâce à la première règle de dominance seront barrés.



Arbre 4-4 : Illustration de la première règle de dominance

- **Deuxième règle de dominance [Moukrim et Rebaïne, 2005]**

Dans cette règle, on regroupe les jobs et on les représente sous forme de blocs. Un bloc peut contenir un ou plusieurs jobs. Notons que l'ordre dans le même bloc n'est pas forcément le même sur M1 et M2. Par exemple, un bloc peut s'exécuter sur M1 dans cet ordre (3, 3, 0) alors que sur M2, il s'exécute dans l'ordre (0, 3, 3).

Par ailleurs, il a été remarqué par Moukrim et Rebaïne [2005] que les différents blocs peuvent être utilisés dans n'importe quel ordre sans que cela n'affecte la valeur du *makespan*.

Pour utiliser cette observation, on définit un ordre lexicographique entre ces différents ordonnancements trouvés et on ne générera que le plus grand d'entre eux. En appliquant cette règle de dominance, on obtient un arbre de *branch and bound* réduit.

Définition 10

Soient deux ensembles $A=(a_1,...,a_n)$ *et* $B=(b_1,...,b_m)$ *tels que* $a_1 \geq ... \geq a_n$ *et* $b_1 \geq ... \geq b_m$. *On dit que A est lexicographiquement plus grand que B si :*

1. *il existe j,* $1 \leq j < n$, $a_1 = b_1,...,a_j = b_j$ *et* $a_{j+1} > b_{j+1}$, *ou*

2. $n > m$ *et* $a_1 = b_1,...,a_m = b_m$.

Exemple 4-12

Soit les temps de latence suivants :

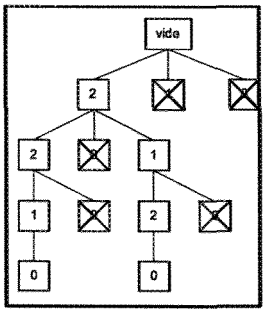
Jobs *j*	1	2	3	4	5	6	7	8	9
Temps de latence τ_j	3	3	0	5	4	1	1	4	2

Tableau 4-7 : Temps de latence *n* = 9 de l'Exemple 4-12

On crée trois blocs : (3, 3, 0), (5, 4, 1, 1) et (4, 2). En exécutant les blocs dans n'importe quel ordre on obtient le même *makespan*. Ainsi tous les ordonnancements présentés dans Figure 4-12 sont équivalents.

On va profiter de cette remarque pour ne générer qu'un seul représentant parmi ces 6 solutions. L'ordre lexicographique décroissant permet de classer ces différents blocs ainsi : (5, 4, 1, 1) > (4, 2) > (3, 3, 0). Ainsi, dans l'algorithme de *branch and bound*, on ne génère que l'ordonnancement « O4 » parmi ces 6 ordonnancements.

Considérons l'exemple suivant où les jobs ont les temps de latence suivants : 2, 2, 1, 0. On obtient l'Arbre 4-5. Les nœuds non explorés grâce à la deuxième règle de dominance seront barrés.



Arbre 4-5 : Illustration de la deuxième règle de dominance

Au début de l'ordonnancement, 0 constitue un bloc. De plus, (2,2,1)>(0). Donc, on élague la branche 0. On applique le même principe sur les autres branches jusqu'à l'obtention de l'Arbre 4-5.

4.2.4. Résultats

Dans cette section, sont résumés les résultats préliminaires de la simulation effectuée sur l'algorithme de *branch and bound* que nous venons de décrire. L'algorithme de *branch and bound* ainsi que toutes les heuristiques ont été développés avec le langage C++ sous l'environnement Visual Studio .NET 2003. L'application a été déployée sur une machine Pentium IV à 3.06 GHZ d'horloge et avec 512 Mo de mémoire vive. Nous avons limité les expérimentations à 900 secondes. Le **'X'** désigne que le temps limite de 900 secondes est dépassé. Il est utile de noter que ce problème est résolu par cet algorithme d'une manière efficace et ce, pour des instances n'excédant pas 30

jobs. Au-delà, il existe beaucoup d'instances pour lesquelles l'algorithme ne donne pas solution dans la limite fixée de 900 secondes. Ces instances ont été élaborées par Moukrim et Rebaïne [2005].

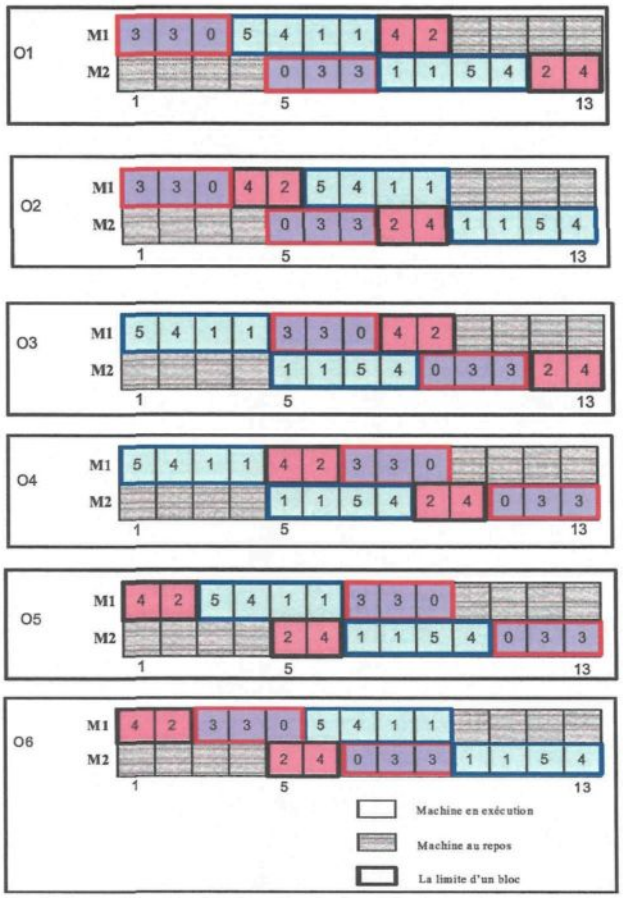


Figure 4-12: Les différents ordonnancements générés suite à la permutation des blocs

La première colonne représente la borne inférieure LB calculée à la racine de notre *Branch and Bound*. La deuxième colonne indique la meilleure borne supérieure UB des 4 bornes générées par notre algorithme. La troisième colonne représente le *Makespan* optimal. La quatrième colonne représente le nombre de nœuds visités lors de l'exploration de l'arbre de *Branch and Bound*. La dernière colonne représente le temps d'exécution de chaque instance.

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
17	17	17	0	0
18	19	18	23	0
16	17	16	21	0
17	17	17	0	0
15	16	15	23	0
17	17	17	0	0
16	16	16	0	0
15	16	15	0	0
15	16	15	21	0
17	18	17	0	0

Tableau 4-8 : Simulations pour n = 10, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
31	32	31	527	0,01
31	32	31	1815	0,02
32	35	32	133916	19,72
32	34	32	149	0
32	34	32	0	0
32	32	32	0	0
29	36	29	1368	0,02
31	31	31	0	0
31	32	31	354	0,01
32	33	32	57	19,85

Tableau 4-9 : Simulations pour n = 20, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
49	50	49	2597	0,02
47	48	47	5717	0,11
44	46	44	2295441	43,6
45	47	45	233383	4,4
48	49	48	346	0,01
45	48	45	118455	2,27
46	47	46	12079	0,23
45	47	45	176145	3,28
46	48	46	150676	2,82
44	45	44	57323	1,08

Tableau 4-10 : Simulations pour n = 30, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
63	64	63	3334873	82.98
62	64	62	6666241	165.12
62	63	62	1886276	45.15
61	64	61	29454523	712.43
64	65	64	34299	0.85
62	63	63	18845798	X
58	60	60	36507975	X
61	63	63	39355086	X
61	63	63	36565474	X
58	61	61	35060949	X

Tableau 4-11 : Simulations pour $n = 40$, cas unitaire

LB à la racine	UB initial	C_{max}	Nbre de nœuds visités	CPU
69	70	70	32817293	X
71	73	71	2642367	69,3
70	72	72	34840502	X
68	70	70	327551841	X
68	70	70	327551841	X
66	68	68	34502465	X
70	71	71	36424816	X
70	73	73	34083049	X
67	71	71	31712970	X
65	67	67	57	X

Tableau 4-12 : Simulations pour $n = 45$, cas unitaire

Nous pouvons remarquer que le nombre de nœuds croit exponentiellement à partir de 30 jobs. Les temps de traitement deviennent similaires puisque le temps d'exécution de l'instance ne doit pas céder 900 secondes. On a trouvé jusqu'à ici des résultats optimaux.

4.3 Branch and bound avec des temps d'exécution quelconques

Nous étudions dans cette partie le problème de *flow-shop* à deux machines avec des temps de latence et d'exécution quelconques. Comme dans la partie concernant les temps d'exécution unitaires, nous décrivons les différentes bornes inférieures et supérieures utilisées pour la réalisation de l'algorithme de *branch and bound*. Notons que cette partie est partiellement basée sur le travail de Yu (1996).

4.3.1. Bornes inférieures

Rappelons que le temps d'exécution d'un job j (j = *1, 2, ..., n*) sur la machine i (i = *1, 2*) est désigné par p_{ij} et son temps de latence par τ_j.

- **Première borne inférieure (LB1)**

La première borne inférieure *LB1*, que nous présentons, est donnée par le Lemme 3.

Lemme 3 [Yu, 1996]

Si Opt désigne la valeur optimale du makespan, alors

$$Opt \geq LB1 = \max\left(\sum_{j=1}^{n} p_{1j} + \min_{1\leq j\leq n}\left(\tau_j + p_{2j}\right), \min_{1\leq j\leq n}\left(p_{1j} + \tau_j\right) + \sum_{j=1}^{n} p_{2j}\right).$$

Preuve :

Soit j le dernier job exécuté sur *M1*. Il est clair que son temps de fin est $\sum_{j=1}^{n} p_{1j}$ comme illustré à la Figure 4-13. Ce job doit premièrement subir un temps de latence τ_j et ensuite être exécuté par *M2*. Dans le meilleur des cas, ces deux temps sont inférieurs à $\min_{1\leq j\leq n}\left(p_{1j} + \tau_j\right)$. D'où le résultat.

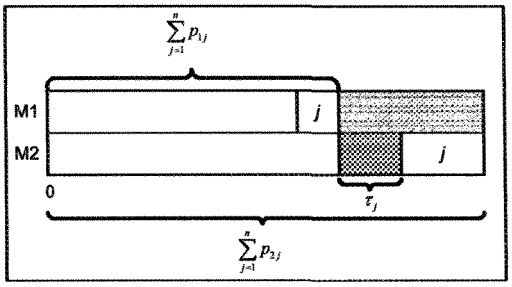


Figure 4-13 : Illustration du Lemme 3

Exemple 4-13

Soit l'instance ci-dessous avec n = 5 jobs.

Jobs *j*	**1**	**2**	**3**	**4**	**5**
Temps d'exécution p_{1j}	**13**	**12**	**2**	**1**	**1**
Temps d'exécution p_{2j}	**8**	**8**	**7**	**6**	**1**
Temps de latence τ_j	**9**	**7**	**2**	**1**	**0**
$p_{1j}+\tau_j$	**22**	**19**	**4**	**2**	**1**
$p_{2j}+\tau_j$	**17**	**15**	**9**	**7**	**1**

Tableau 4-13 : Application de LB1 sur l'Exemple 4-13 dans le cas quelconque

Les cases grises représentent respectivement le minimum de $p_{1j}+\tau_j$ et celui de $p_{2j}+\tau_j$. Le calcul de la première borne inférieure *LB1* est le suivant :

$LB1 = \max\{(13+12+2+1+1)+1, (8+8+7+6+1)+1\}=31.$

Étant donnée une sous séquence α des jobs déjà ordonnancés, nous présentons, dans ce qui suit, trois bornes inférieures *LB2*, *LB3*, *LB4*.

- **Deuxième borne inférieure (LB2)**

La deuxième borne inférieure LB2 est calculée à chaque nœud de l'arbre de recherche. Elle est donnée par le Lemme 4.

Lemme 4 [Yu, 1996]

$$Opt \geq LB2 = \max\left(P_{1j}+\tau_j+P_{2j}\right).$$

Preuve :

Quelque soit le job, il aura à s'exécuter sur *M1*, ensuite un temps de latence s'en suit avant de s'exécuter sur *M2*. Il est clair que le *makespan* ne peut être inférieur à la somme de ces trois valeurs.

Étant donné α jobs déjà ordonnancés, la date au plus tôt d'un job A sur la machine *M2* est $\sum_{j=1}^{A} p_{1j}+\tau_A$ pour tout job $A \in \alpha$. Il est clair que les jobs restants appartenant à la

séquence n - α vont commencer leur exécution après que les jobs de la séquence α soient exécutés. De plus, la borne inférieure correspondant aux jobs appartenant à la séquence n -α, est clairement *LB2* (n-α), c'est-à-dire LB2 appliquée aux jobs de l'ensemble n-α. Par conséquent, le *makespan* de l'ensemble de jobs, étant donnée la séquence de jobs α, est $\sum_{j\in\alpha} p_{1j} + LB2(n-\alpha)$. Cette borne peut être obtenue en appliquant l'Algorithme 4-7.

$|\alpha|$: **représente le cardinal de la sous séquence** α.

n - α : **étant le nombre de jobs non ordonnancés et qui n'appartiennent pas à** α.

On calcule la borne LB2 sur le reste de jobs n'appartenant pas à α **on aura alors :**

$$LB2 = \max_{j\in n-\alpha}\left(p_{1j} + \tau_j + p_{2j}\right).$$

La valeur de la borne inférieure finale est la suivante : LB2 = $\sum_{j\in\alpha} p_{1j} + LB2\left(n-\alpha\right)$

Algorithme 4-7 : Algorithme de la deuxième borne inférieure LB2 dans le cas quelconque

Exemple 4-14

Soit l'instance suivante avec n = 5 jobs.

	$j\in\alpha$		$j\in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p_{1j}+\tau_j$ p_{2j}			11	8	2

Tableau 4-14 : Application de LB2 sur l'Exemple 4-14 dans le cas quelconque

Pour calculer *LB2*, on doit procède comme suit : On détermine : $\sum_{j=1}^{2} p_{1j}$ = 13+12 = 25 avec $j\in\alpha$. Ensuite, on applique la formule de *LB2* sur les n - α jobs restants. La case grise représente le maximum de ($p_{1j}+\tau_j+p_{2j}$). La valeur finale de *LB2 est* comme suit :

$$\text{LB2} = \sum_{j=1}^{2} p_{1j} + \text{LB2}\ (n - \alpha) = 25 + \max\ \{11, 8, 2\} = 36.$$

- **Troisième borne inférieure (LB3)**

Notons que les deux bornes inférieures *LB1* et *LB2* contiennent seulement un seul terme représentant le temps de latence. Il est clair qu'il est préférable d'impliquer l'ensemble de temps de latence pour espérer avoir une borne inférieure efficace.

Théorème 7 [Yu, 1996]

Soient $q_j = \min(p_{1j}, p_{2j})$, $r_j = \max(p_{1j}, p_{2j})$ *avec (j = 1, 2, ..., n). Si Opt désigne le makespan de la solution optimale, alors*

$$Opt \geq LB3 = \left\lceil \left(\sum_{j=1}^{n} q_j (\tau_j + r_j - 1) \right) \Big/ \sum_{j=1}^{n} q_j \right\rceil + 1 + \sum_{j=1}^{n} q_j .$$

Preuve :

Sans perte de généralités, nous pouvons considérer que chaque job j contient q_j jobs artificiels avec un temps d'exécution unitaire chacun. Considérons les deux cas suivants :

Cas 1 : $p_{1j} \leq p_{2j}$

Dans ce cas, nous avons : $q_j = p_{1j}$ et $r_j = p_{2j}$. Soit k = 1, 2, ..., q_j. Définissons l'exécution de chaque job artificiel J_{jk} sur *M1*, ensuite sur *M2*. Le temps d'exécution d'un job artificiel J_{jk} sera considéré comme un temps unitaire : il sera exécuté sur le $k^{ème}$ emplacement du job j sur *M1*. Puis, il poursuit son exécution sur *M2*, et il exécutera sur le $(r_j - q_j + k)^{ème}$ emplacement du job j sur *M2*. Ces jobs artificiels ont le même temps de latence τ_j'.

$$\tau_j' = (q_j - k) + \tau_j + (r_j - q_j + k - 1) = \tau_j + r_j - 1$$

Cas 2 : $p_{1j} > p_{2j}$

Dans ce cas, nous avons : $q_j = p_{2j}$ et $r_j = p_{1j}$. Soit k = 1, 2, ..., n. Similairement, définissons l'exécution de chaque job artificiel J_{jk} sur *M1*, ensuite sur *M2*. Le temps d'exécution d'un job artificiel J_{jk} sur *M1* sera le même que celui décrit par le premier

cas, alors que J_{jk} sera exécuté sur le $k^{ème}$ emplacement du job j sur $M2$. De même que le cas précédent, les jobs artificiels ont le même temps de latence τ_j' comme ci-dessus, c'est-à-dire

$$\tau_j' = (r_j - k) + \tau_j + (k-1) = \tau_j + r_j - 1.$$

En appliquant le Lemme 1 sur toute la séquence de $(q_1 + q_2 + ... + q_n)$ jobs artificiels de temps unitaires et des temps de latence, décrits précédemment, on obtient la troisième borne inférieure *LB3*.

Cette borne peut être obtenue en appliquant l'Algorithme 4-8 sur notre arborescence.

$|\alpha|$: représente le cardinal de la sous séquence α.

$n - \alpha$: représente le nombre de jobs qui n'appartiennent pas à α.

$$q_j = \min_{j \in n-\alpha}(p_{1j}, p_{2j})$$

$$r_j = \max_{j \in n-\alpha}(p_{1j}, p_{2j})$$

On calcule la borne *LB3* sur le reste de jobs n'appartenant pas à α on aura alors :

$$LB3 = \left\lceil \left(\sum_{j \in n-\alpha} q_j (\tau_j + r_j - 1) \right) \Big/ \sum_{j \in n-\alpha} q_j \right\rceil + 1 + \sum_{j \in n-\alpha} q_j.$$

La valeur de la borne inférieure finale est la suivante : LB3 = $\sum_{j \in \alpha} p_{1j} + LB3(n-\alpha)$

Algorithme 4-8 : Algorithme de la troisième borne inférieure LB3 dans le cas quelconque

Exemple 4-15

L'exemple ci-dessous illustre le calcul de la borne inférieure *LB3*. On suppose qu'on a une sous séquence α de taille 2 comme dans l'exemple précédent.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$r_j = \max_{j \in n-\alpha}(p_{1j}, p_{2j})$			7	6	1
$q_j = \min_{j \in n-\alpha}(p_{1j}, p_{2j})$			2	1	1
$\tau_j + r_j - 1$			8	6	0
$q_j(\tau_j + r_j - 1)$			16	6	0

Tableau 4-15 : Application de LB3 sur l'Exemple 4-15 dans le cas quelconque

On peut procéder comme suit pour le calcul de *LB3* :

1. On calcule *LB3* pour les jobs n'appartenant pas à α. On aura :

$$LB3(n-\alpha) = \lceil (16+6+0)/4 \rceil + 1 + 4 = \lceil (22)/4 \rceil + 5 = 11.$$

2. Enfin, la valeur finale de la borne *LB3* est : $LB3 = \sum_{j \in \alpha} p_{1j} + LB3(n-\alpha) =$ 25+11= 36.

- **Quatrième borne inférieure (LB4)**

Théorème 8 (Yu, 1996)

Soit p'_{ij} *la somme de j plus petites valeurs de* $\{p_{i1}, p_{i2}, ..., p_{in}\}$ *avec i = 1, 2 et j = 1, 2, ..., n, alors on aura :*

$$Opt \geq LB4 = \left\lceil \left(\sum_{j=1}^{n} \tau_j + \sum_{j=1}^{n} p'_{1j} + \sum_{j=1}^{n} p'_{2j} \right) \Big/ n \right\rceil.$$

Où p'_{ij} *désigne la somme des j plus petites valeurs de* $\{p_{i1}, p_{i2}, ..., p_{in}\}$.

Preuve :

Soit *S* un ordonnancement d'une séquence du problème de *F2D*. On suppose que σ et ε représentent, respectivement, les séquences de jobs sur *M1* et *M2*, d'une solution donnée. Il est évident que le *makespan* *C*[*S*] vérifie la condition suivante :

$$C[S] \geq \sum_{j=1}^{\sigma^{-1}(i)} p_{1j} + \tau_j + \sum_{j=\varepsilon^{-1}(i)}^{n} p_{2j} \; ; i = 1, 2, \ldots, n. \qquad (1)$$

On a $\sigma^{-1}(i)$ et $\varepsilon^{-1}(i)$ désignent respectivement la position d'un job i sur la séquence σ et ε. Soit σ et $\varepsilon \in PER_n$[7], sachant que $\alpha(i) = \sigma^{-1}(i)$ et $\beta(i) = n+1-\varepsilon^{-1}(i)$; $i \in [1, n]$. On remarque alors que la signification de p'_{ij} est la suivante :

$$\sum_{j=1}^{\sigma^{-1}(i)} p_{1j} \geq p'_{1\alpha(i)} \text{ et } \sum_{j=\varepsilon^{-1}(i)}^{n} p_{2j} \geq p'_{2\beta(i)} \text{ avec } (i = 1, 2, \ldots, n). \qquad (2)$$

Donc, si on somme (1) et (2), on obtient :

$$n * C[S] \geq \sum_{i=1}^{n} \tau_j + \sum_{i=1}^{n} p'_{1\alpha(i)} + \sum_{i=1}^{n} p'_{2\beta(i)}. \qquad (3)$$

On aura alors :

$$\sum_{i=1}^{n} p'_{1\alpha(i)} = \sum_{i=1}^{n} p_{1i}, \sum_{i=1}^{n} p'_{2\beta(i)} = \sum_{i=1}^{n} p'_{2i}. \qquad (4)$$

La borne inférieure *LB4* découle ainsi des inégalités (3) et (4).

Pour appliquer la quatrième borne inférieure *LB4*, on doit d'abord vérifier les temps de latence de chaque job appartenant à α car ils peuvent augmenter en plaçant, par exemple, deux jobs sur le même emplacement. Pour cela, on applique l'Algorithme 4-9.

q : fin de temps d'exécution d'un job sur *M2*.
Pour p = 1 to $|\alpha|$ faire
début
Soit q l'emplacement de la fin d'exécution du job j sur *M2*

$$\tau_j = q - \sum_{j=1}^{p} p_{1j} - p_{2j} \}$$

fin pour

Algorithme 4-9 : Algorithme de vérification des temps de latence

Après avoir obtenu les nouveaux temps de latence, on applique le Théorème 8 sur l'ensemble des jobs.

[7] PER_n est la permutation d'un ensemble $N = \{1, 2, \ldots, n\}$ de jobs.

Exemple 4-16

Soit l'instance suivante avec $n = 5$ jobs. On suppose qu'on a une sous séquence $\alpha = 3$ de jobs.

	$j \in \alpha$			$j \in n-\alpha$	
Jobs j	4	5	3	1	2
Temps d'exécution p_{1j}	1	1	2	13	12
Temps d'exécution p_{2j}	6	1	7	8	8
Temps de latence τ_j	1	0	2	9	7

Tableau 4-16 : Les temps d'exécution et de latence de l'Exemple 4-16

Pour calculer la borne *LB4*, on doit tout d'abord vérifier s'il a un changement dans les temps de latence des jobs appartenant à α . Donc, on doit tout d'abord ordonnancer les jobs appartenant à α comme illustré dans la Figure 4-14 :

Figure 4-14 : Application de *LB4* avec des temps d'exécution quelconques

De la Figure 4-14, on peut remarquer qu'il y'a eu des changements dans les temps de latence des jobs appartenant à α . On a donc :

- Pour $p = 1$: $j = 4$, $\tau_4 = 1$. Donc, le temps modifié de $\tau_4 = 8-1-6 = 1$ (dans ce cas, on n'a pas de changement).
- Pour $p = 2$: $j = 5$, $\tau_5 = 0$. Donc, le temps modifié de $\tau_5 = 9-2-1 = 6$. Notons que, dans ce cas, on a fait un changement.
- Pour $p = 3$: $j = 3$, $\tau_3 = 2$. Donc, le temps modifié de $\tau_3 = 16-4-7 = 5$. Notons que, dans ce cas, on a fait également un changement.

Après avoir modifié les temps de latence, notre tableau sera comme suit :

	$j \in \alpha$			$j \in n-\alpha$	
Jobs j	4	5	3	1	2
Temps d'exécution p_{1j}	1	1	2	13	12
Temps d'exécution p_{2j}	6	1	7	8	8
Temps de latence τ_j	1	6	5	9	7
P'_{1j}	1	2	4	17	29
P'_{2j}	6	7	14	22	30

Tableau 4-17 : Application de LB4 sur l'Exemple 4-16 dans le cas quelconque

On a: $LB4 = \left\lceil \left(\sum_{j=1}^{n} \tau_j + \sum_{j=1}^{n} p'_{1j} + \sum_{j=1}^{n} p'_{2j} \right) \Big/ n \right\rceil = \lceil ((1+6+5+9+7) + (1+2+4+17+29)$

$+ (6+7+14+22+30)) / 5 \rceil = 32.$

4.3.2. Bornes supérieures

Comme dans le cas unitaire, nous utilisons dans notre algorithme de *branch and bound* quatre bornes supérieures basées sur des heuristiques ci-dessous. Notons que ces heuristiques sont conçues pour être utilisées pour les nœuds internes de l'arbre de recherche, c'est-à-dire qu'il est supposé qu'une sous-séquence de jobs α est déjà fixée.

- **Première borne supérieure**

Pour calculer la première borne supérieure *UB1*, nous utilisons l'algorithme de Johnson, présenté à la section 3.2.4. Puisqu'il est à l'origine de plusieurs travaux sur deux machines, nous avons jugé utile de l'utiliser pour calculer *UB1*.

Pour calculer la première borne supérieure, on place tout d'abord les jobs appartenant à α sur machine *M1*. Ensuite, on passe à traiter les $n-\alpha$ jobs restants. Nous devrons appliquer le Théorème 6 sur les $n-\alpha$ jobs. Ce théorème consiste à modifier les temps d'exécution de chaque job j, en appliquant : $p'_{ij} = p_{ij} + \tau_j$ ($i = 1, 2, j = 1, 2, \ldots, n$). Après avoir obtenu les nouveaux temps d'exécution, on applique l'algorithme de Johnson sur les $n-\alpha$ jobs. Et enfin, on place les $n-\alpha$ jobs à la suite des α jobs déjà placés sur *M1*. L'heuristique *UB1* est obtenue en appliquant l'Algorithme 4-10.

- $|\alpha|$: **représente le cardinal de la sous séquence α .**
- $n - \alpha$ **: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**
- **Pour $i = 1$ jusqu'à 2 faire**

  Pour $j \in n - \alpha$ n faire

  $$p'_{ij} = p_{ij} + \tau_j$$

  Fin pour

  Fin pour

- **Appliquer l'algorithme de Johnson sur les nouveaux temps d'exécution p'_{ij} ($i = 1, 2$ et $j \in n - \alpha$)**
- **Après la fin d'exécution de chaque job j sur M1, on essaie de le placer le plus tôt possible sur M2**

Algorithme 4-10 : Algorithme de la première borne supérieure UB1 pour le cas quelconque

Exemple 4-17

L'exemple ci-dessous illustre le calcul de la borne supérieure $UB1$: soit le nombre de jobs suivant $n = 5$.

Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0

Tableau 4-18 : Les temps d'exécution et de latence de l'Exemple 4-17

- Soit la sous séquence $\alpha = (1,2)$. Les jobs restants sont : 3, 4 et 5. On doit en premier lieu calculer les nouveaux temps d'exécution en appliquant: $p'_{ij} = p_{ij} + \tau_j$ ($i = 1, 2, j = 1, 2, \ldots, n$). On obtient alors le Tableau 4-19.

On applique l'algorithme de Johnson sur les temps d'exécution modifiés. On aura alors deux sous ensembles U et V qui contiendront les jobs suivant : $U = \{4, 3\}$ et $V = \{5\}$.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs *j*	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p'_{1j} = \tau_j + p_{1j}$			4	2	1
$p'_{2j} = \tau_j + p_{2j}$			9	7	1

Tableau 4-19 : L'application de UB1 sur l'Exemple 4-17 dans le cas quelconque

La séquence finale des jobs sera ordonnancée comme suit : Après avoir placé les α jobs, on met les n -α jobs restants. On obtient la permutation: 1, 2, 4, 3 et 5. L'ordonnancement obtenu est présenté par la Figure 4-15. La valeur du *makespan* est donc $UB1 = 53$.

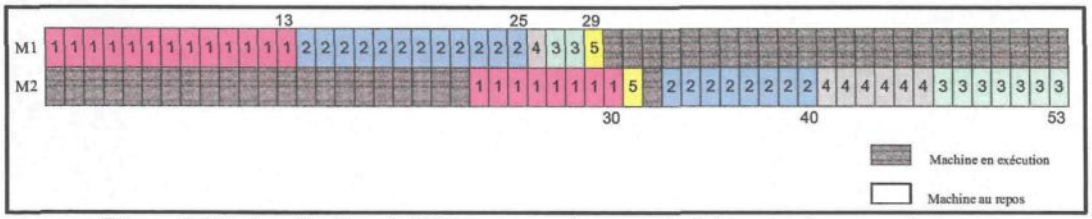


Figure 4-15 : Application de UB1 sur une séquence de jobs pour le cas quelconque

- **Deuxième borne supérieure**

La deuxième borne supérieure *UB2* est obtenue comme suit : on commence par placer les α jobs *M1*. Puis, on applique le Théorème 6 sur les n-α jobs restants. Autrement dit, les nouveaux temps d'exécution sont: $p'_{ij} = p_{ij} + \tau_j (i = 1, 2, j = 1, 2, ..., n)$. Ensuite, on ordonnance les jobs suivant ces nouveaux temps d'exécution obtenus dans l'ordre décroissant et on place la séquence obtenue sur la première machine *M1*, ensuite on place les jobs le plus tôt possible sur *M2*. Quoique très simple, cette heuristique s'est avérée efficace. Il s'agit d'appliquer l'algorithme *LPT*[8] (*Longest Processing time*) sur la première machine en incluant les temps de latence ensuite de placer au plus tôt les jobs

[8] LPT est un algorithme d'ordonnancement qui priorise les jobs avec les plus grands temps d'exécution.

sur la deuxième machine. On calcule enfin le *makespan* pour obtenir alors la deuxième borne supérieure *UB2*. Le calcul de la deuxième borne est illustré à l'Algorithme 4-11.

- $|\alpha|$: représente le cardinal de la sous séquence α.
- $n - \alpha$: étant le nombre de jobs non ordonnancés et qui n'appartiennent pas à α
- $p'_{ij} = p_{ij} + \tau_j$ avec $i = 1,2$ et $j = 1,2,...n$.
- Ordonnancer les jobs suivants l'ordre décroissant de p'_{1j} et on les place sur M1
- Après la fin d'exécution de chaque job j sur M1, on essaie de le placer le plus tôt possible sur M2

Algorithme 4-11 : Algorithme UB2 pour le cas quelconque

Exemple 4-18

Soit l'instance suivante avec $n = 5$ jobs. On doit tout d'abord appliquer la formule suivante: $p'_{ij} = p_{ij} + \tau_j$ sur l'instance. On aura alors des nouveaux temps d'exécution présentés dans le tableau ci-dessous:

	$j \in \alpha$		$j \in n-\alpha$		
Jobs j	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0
$p'_{1j} = \tau_j + p_{1j}$			4	2	1
$p'_{2j} = \tau_j + p_{2j}$			9	7	1

Tableau 4-20 : Application de UB2 sur l'Exemple 4-18 dans le cas quelconque

Ensuite, on ordonnance les jobs suivant l'ordre décroissant de p'_{1j}, on obtient alors la séquence de jobs : 1, 2, 3, 4 et 5, illustrée par la Figure 4-16 . La valeur de la borne *UB2* est donnée par le *makespan* qui est égal à 53.

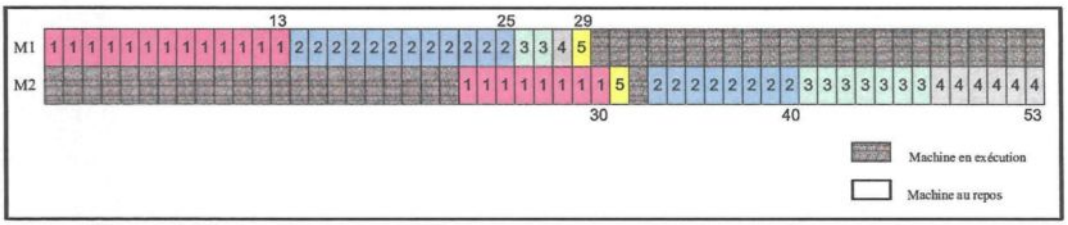


Figure 4-16: Application de UB2 sur une séquence de jobs pour le cas quelconque

- **Troisième borne supérieure**

L'heuristique pour obtenir la troisième borne supérieure *UB3* consiste à affecter des priorités aux différents jobs. On applique cette priorité au $n - \alpha$ jobs. Le job qui a la priorité la plus élevée sera exécuté en premier et ainsi de suite. Pour affecter une priorité S_j à un job J_j on calcule cette formule : $S_j = \left(\sum_{i=1}^{2} (m - 2i + 1) p_{ij} \right) + \tau_j$ avec $j = 1,2,\ldots, n$ et $i = 1,2$. Ensuite, on ordonnance les S_i suivant l'ordre décroissant. Puis on place les $n - \alpha$ jobs obtenus sur la machine *M1* à la suite des α jobs déjà ordonnancés et au plus tôt possibles sur *M2*. On calcule enfin le *makespan*. Cette formule pour le calcul des priorités est une modification de celle utilisée par Palmer [1965]. En effet, ce dernier avait utilisé la formule suivante pour calculer les priorités des jobs pour un *flow-shop* à condition que le nombre de machine m ≥ 1 : $S_j = \sum_{i=1}^{m} (m - 2i + 1) p_{ij}$ avec $j = 1,2,\ldots, n$ et $i = 1, 2$. En développant cette formule on obtient :

$$S_j = -(m-1)p_{1j} - (m-3)p_{2j} - (m-5)p_{3j} - \ldots + (m-3)p_{m-1j} + (m-1)p_{mj}, \quad \text{ou} \quad (m\text{-}1)$$

par exemple peut s'écrire sous cette forme : m-$2i$+1 pour $i = 1$.

Nous avons introduit les temps de latence τ_j pour calculer les priorités associés aux jobs. Ainsi, un job pour lequel les temps d'exécution tendent à augmenter d'une machine à une autre aura une priorité plus élevé que celle d'un job qui aura des temps d'exécution tendant à diminuer d'une machine à une autre tout en prenant compte des temps de latence. Le calcul de la troisième borne est illustré à l'Algorithme 4-12.

- **$n - \alpha$: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**
- **Pour $i = 1$ jusqu'à 2 faire**

  Pour $j \in n - \alpha$ n faire

  $$S_j = \left(\sum_{i=1}^{m} (m - 2i + 1) p_{ij} \right) + \tau_j$$

  Fin pour

  Fin pour
- **Ordonnancer les S_j suivant l'ordre décroissant**

Algorithme 4-12 : Algorithme de UB3 pour le cas quelconque

Exemple 4-19

Pour mieux comprendre cette borne, considérons l'exemple suivant. Soit $\alpha = (3,4)$. Les jobs restants sont : 1, 2 et 5. Le tableau ci-dessous présente la priorité de chaque job n'appartenant pas à α.

	$j \in \alpha$		$j \in n-\alpha$		
Jobs *j*	3	4	1	2	5
Temps d'exécution p_{1j}	2	1	13	12	1
Temps d'exécution p_{2j}	7	6	8	8	1
Temps de latence τ_j	2	1	9	7	0
$S_j = \left(\sum_{i=1}^{m}(m-2i+1)p_{ij}\right)+\tau_j$			14	11	0

Tableau 4-21 : L'application de UB3 sur l'Exemple 4-19 dans le cas quelconque

Le nouvel ordre de la séquence des jobs sera le suivant : 3, 4, 1, 2 et 5. La Figure 4-17 représente l'ordonnancement de la séquence sur les deux machines. La borne *UB3* est égale à 43.

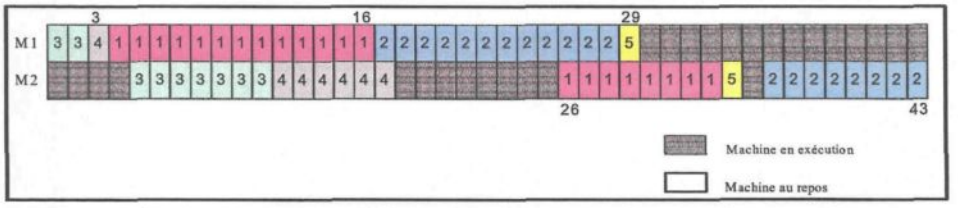


Figure 4-17: Application de UB3 sur une séquence de job pour le cas quelconque

- **Quatrième borne supérieure**

Pour le calcul de la quatrième borne supérieure *UB4*, nous appliquons l'heuristique *NEH* [Nawaz et *al.*, 1983]. Cette heuristique est basée sur l'hypothèse qu'un lot de jobs ayant un temps total d'exécution élevé est prioritaire (le lot est positionné en priorité dans un ordonnancement partiel) par rapport à un job de faible priorité. On classe les jobs dans l'ordre décroissant de leur durée de traitement totale : temps d'exécution sur les deux machines et celui de latence. Puis, on construit progressivement la séquence correspondant à la solution en insérant, à chaque étape, le job suivant dans la liste à la meilleure place dans la séquence, de manière à minimiser le *makespan*. L'adaptation de

l'algorithme *NEH* pour le calcul de la borne *UB4* est illustrée à l'Algorithme 4-13. L'utilisation de cette heuristique est justifiée par la bonne qualité des résultats donnés au sein de différents travaux, notamment ceux de Taillard [1990], Sarin et Lefoka [1993], Framinan et *al.* [2003].

- **$|\alpha|$: représente le cardinal de la sous séquence α.**
- **$n-\alpha$: représente le nombre de jobs non ordonnancés et qui n'appartiennent pas à α**

  Pour $j \in n-\alpha$ n faire

  $p_{ij} = p_{1j} + \tau_j + p_{2j}$ **avec $i = 1,2$.**

  Fin pour
- **Adaptation de NEH :**
  - **Ordonner les tâches selon l'ordre croissant de temps d'exécution des jobs p_{ij} sur les machines.**
  - **Créer une séquence vide *S*.**
  - **On ordonnance les deux premiers jobs à la suite de la séquence α.**
  - **Soit une séquence de jobs vide *T***
  - **Tant que (*T* non vide) faire**

    ***T* = *T* – premier élément *j* de *T* ;**

    Tester l'élément *j* à tous les emplacements dans *S*;

    Insérer *j* dans *S* à l'emplacement qui minimise le *makespan*.
  - **Fin Tant que**

Algorithme 4-13 : Algorithme d'adaptation NEH pour le calcul de UB4 pour le cas quelconque

Exemple 4-20

Cet exemple illustre le calcul de la borne supérieure *UB*4. Considérons l'instance suivante pour $n = 5$.

Jobs *j*	1	2	3	4	5
Temps d'exécution p_{1j}	13	12	2	1	1
Temps d'exécution p_{2j}	8	8	7	6	1
Temps de latence τ_j	9	7	2	1	0

Tableau 4-22 : Les temps d'exécution et de latence de l'Exemple 4-20

Soit une sous séquence $\alpha = (3,4)$. Les jobs restants n'appartenant pas à α sont : 1, 2 et 5. On doit tout d'abord calculer le temps d'exécution de chaque job sur les deux machines: $p_{ij} = p_{1j} + \tau_j + p_{2j}$ $(i = 1, 2, j = 1, 2, \ldots, n)$. On aura alors le tableau suivant :

	$j \in \alpha$		$j \in n - \alpha$		
Jobs *j*	3	4	1	2	5
Temps d'exécution p_{1j}	2	1	13	12	1
Temps d'exécution p_{2j}	7	6	8	8	1
Temps de latence τ_j	2	1	9	7	0
$p_{ij} = p_{1j} + \tau_j + p_{2j}$			30	27	2

Tableau 4-23 : Application de UB4 sur l'Exemple 4-20 dans le cas quelconque

On ordonnance les jobs n'appartenant pas à α suivant l'ordre croissant du temps p_{ij}. La séquence sera la suivante : 5, 2 et 1. Après avoir ordonnancé les jobs, on doit placer les jobs appartenant à α ainsi que les deux premiers jobs des $n - \alpha$ jobs restants. C'est-à-dire, on place les jobs 3 et 4, ensuite les jobs 5 et 2, tout en respectant l'ordre. La Figure 4-18 illustre cet ordonnancement.

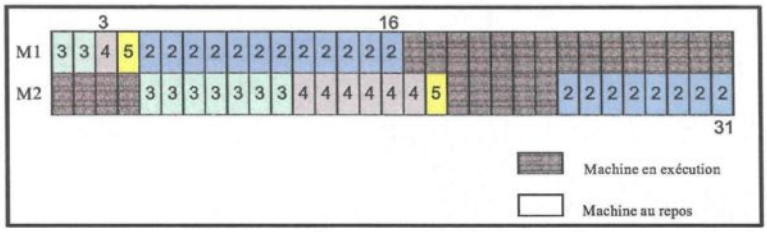


Figure 4-18 : Ordonnancement d'une partie de la séquence en appliquant UB4

On applique notre algorithme sur le reste des jobs non encore ordonnancées. Le job restant est le job 1. Ce dernier va être placé dans trois différentes positions, comme illustré dans la Figure 4-19. A partir de là, on doit choisir l'ordonnancement qui génère le plus petit *makespan*.

On a trois différents ordonnancements, (a), (b) et (c). L'ordonnancement qui a le *makespan* minimal est l'ordonnancement (b) ou (c), puisqu'ils ont le même *makespan*. Pour conclure, nous allons choisir par défaut le premier ordonnancement, (b). La valeur de la borne *UB4* est égale à 44.

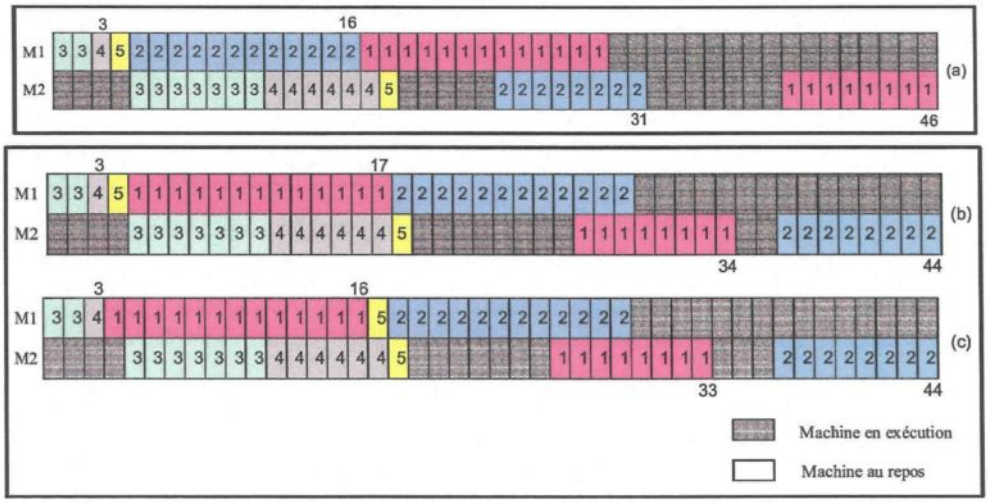


Figure 4-19 : Les différents ordonnancements induits par l'application de UB4

4.3.3. Illustration de l'algorithme de branch and bound

Appliquons l'algorithme de *branch and bound* sur l'exemple ci-dessous pour $n = 5$, où les temps de latence et d'exécution sur *M1* et *M2* de jobs sont résumés comme suit :

Jobs *j*	**1**	**2**	**3**	**4**	**5**
Temps d'exécution p_{1j}	**13**	**12**	**2**	**1**	**1**
Temps d'exécution p_{2j}	**8**	**8**	**7**	**6**	**1**
Temps de latence τ_j	**9**	**7**	**2**	**1**	**0**

Tableau 4-24 : Les temps d'exécution et de latence d'un problème de branch and bound

* On calcule d'abord la valeur de $LB1 = 31$.
* Les différentes bornes supérieures initiales sont :
  - $UB1 = 46$; $UB2 = 46$; $UB3 = 49$; $UB4 = 44$;
* On fait appel au *branch and bound*. On obtient alors une séquence minimale : 3, 1, 4, 2 et 5 et un *makespan* = 43, (Figure 4-20).

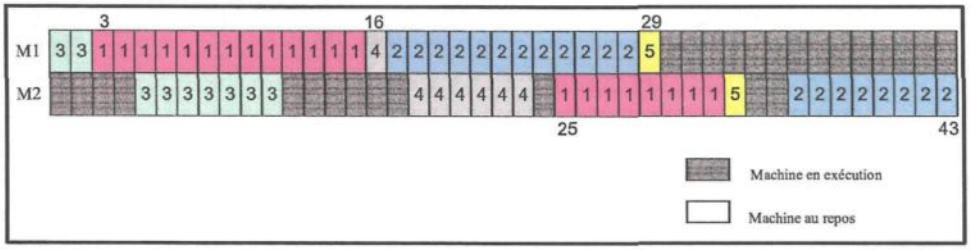
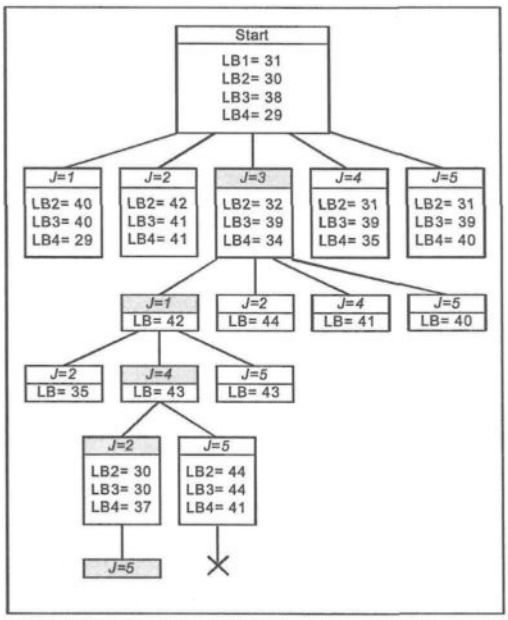


Figure 4-20 : Exécution de l'algorithme branch and bound sur une séquence de jobs dans le cas quelconque

En appliquant l'algorithme général de *branch and bound* sur notre problème, on obtient l'Arbre 4-6. Dans cet arbre, *LB* représente la valeur de la borne choisie entre *LB2*, *LB3* et *LB4*. Les cases grisées de cet arbre représentent la séquence optimale.



Arbre 4-6 : L'arbre de recherche dans le cas quelconque

4.3.4. Résultats empiriques

Nous présentons dans cette partie les différents résultats obtenus sur le problème de flow de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Dans un premier temps, nous rapportons les résultats relatifs aux heuristiques que nous avons utilisées. Ensuite, nous présentons les résultats de l'algorithme de *branch and bound* dont l'exécution pour une instance donnée été limitée à 3600 secondes, soit 1 heure. L'algorithme du *branch and bound* a été développé avec le

langage C++ sous l'environnement Visual Studio .NET 2003. Il a été exécuté sur une machine Pentium IV de 3.06 GHZ d'horloge et 512 Mo de mémoire vive.

- **Heuristiques**

Nous présentons dans cette partie les différents résultats relatifs aux bornes supérieures du problème de *flow-shop* à deux machines avec des temps d'exécution et de latence quelconques. Nous rappelons que les bornes supérieures, que nous avons utilisées, sont les heuristiques de Johnson modifié, ordre décroissant, *NEH* et de priorité. Le Tableau 4-25 résume les différents résultats trouvés pour les quatre heuristiques sur 10 instances de 10, 15, 20, 30, 40, 50 et 60 jobs. La deuxième colonne représente le temps CPU moyen d'exécution. Il est clair que l'heuristique de priorité (*UB4*) est la plus gourmande en temps. En moyenne *UB3* consomme moins de 6 secondes. Quant à *UB1* et *UB2*, elles consomment en moyenne moins d'un dixième de seconde. La troisième colonne du tableau représente la déviation de chaque heuristique par rapport à *LB1*. Les heuristiques *UB3* et *UB4* ont une déviation moyenne inférieure à 7 %, qui est relativement assez bonne. Au vue de ces résultats, il est clair que *UB4* est la meilleure borne supérieure, même si le rapport *déviation / temps* penche nettement en faveur de *UB3*.

UB	moyenne Temps CPU	moyenne |(Ubi-LB1)/LB1|
UB1 : Johnson	0,101557	0,109765
UB2 :Ordre décroissant	0,024957	0,121885
UB3 : NEH	5,991757	0,066290
UB4 : Priority	193,002157	0,062279

Tableau 4-25 : Résultats des heuristiques dans le cas quelconque

- **Algorithme de branch and bound**

Les tableaux ci-dessous présentent les résultats obtenus pour les classes de n = 5, 7, 9, 10, 12 et 15 jobs. Pour chaque classe, l'algorithme a été exécuté sur 10 instances générées d'une manière aléatoire. Les temps d'exécution et de latence sont générés aléatoirement dans un intervalle de temps entre 0 et 100. Nous y trouvons la borne *LB* initiale, la meilleure borne supérieure *UB* initiale, le meilleur *makespan* atteint par l'application, le nombre de nœuds visités et le temps *CPU* pour les différentes classes

d'instances sur lesquelles l'algorithme de *branch and bound* a été exécuté. Le symbole '**x**' signifie que le temps limite de 3600 secondes a été dépassé.

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
202	314	304	101	0,5
212	287	250	35	0,235
157	256	252	282	0,672
230	279	278	72	0,547
271	360	351	188	0,844
172	289	287	48	0,469
267	306	286	30	0,531
260	280	280	213	0,766
248	311	304	60	0,531
180	284	277	123	0,328

Tableau 4-26 : Simulation pour $n = 5$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
396	459	458	1842	5,923
329	416	410	1452	4,172
359	431	431	8683	16,627
356	415	412	8305	15,142
267	324	324	5801	9,422
328	389	386	2386	5,845
285	368	362	1535	3,203
218	263	254	2619	4,297
197	331	318	1503	2,375
316	337	337	516	2,079

Tableau 4-27 : Simulation pour $n = 7$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
440	485	483	16299	34,679
540	581	579	12147	32,022
580	598	598	88907	252,183
527	595	594	81137	188,64
391	490	483	216786	448,603
461	544	544	67399	160,353
383	440	438	23947	44,835
364	458	458	128987	262,364
420	560	543	168720	384,458
332	375	374	245562	441,542

Tableau 4-28 : Simulation pour $n = 9$ dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
559	597	596	79560	734,808
478	563	563	383515	X
496	523	523	416468	3482,21
396	517	517	398584	X
488	534	534	376227	X
511	556	551	216960	X
635	724	724	165542	X
560	585	584	41506	637,995
358	427	423	298785	X
325	464	446	280694	X

Tableau 4-29 : Simulation pour n = 10 dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
501	580	569	1139286	X
669	725	725	922775	X
618	650	650	877778	X
446	568	564	1147817	X
741	767	767	784949	X
657	673	673	1138119	X
459	508	508	1242814	X
622	643	643	1104795	X
632	670	664	1114856	X
477	649	608	1170930	X

Tableau 4-30 : Simulation pour n = 12 dans le cas quelconque

LB à la racine	*UB à la racine*	C_{max}	*Nbre de nœuds visités*	*CPU*
701	767	767	154795	X
664	711	706	155179	X
659	768	768	152572	X
828	876	862	165754	X
736	771	771	151057	X
648	709	709	159106	X
803	858	856	183950	X
655	705	705	168341	X
796	888	888	136598	X
729	828	828	145226	X

Tableau 4-31 : Simulation pour n = 15 dans le cas quelconque

Comme nous pouvons le constater à travers ces résultats, l'algorithme de *branch and bound*, présenté ci-dessus, semble donner d'assez bons résultats, mais juste sur de petites instances, comme il fallait s'y attendre. En effet, à partir de $n = 10$, l'algorithme

ne produit pas de résultats, pour la majorité des instances, après une heure d'exécution allouée à chaque instance. Toutefois, nous restons convaincus que la conception de quelques règles de dominance améliorerait grandement l'efficacité de cet algorithme.