COMPLEX

ORDONNANCEMENT DE TÂCHES SUR DES MACHINES EN SÉRIE

31 octobre 2015

Emmanuel de Bézenac, Arthur Ramolet
Université Pierre et Marie Curie
Année Universitaire 2015-2016

Table des matières

1 Introduction **3**

2 Algorithme approché avec garantie de performance **4**
2.1 3-Approché 4
2.2 2-Approché 5
2.3 Implémentation 7

3 Méthode exacte **9**
3.1 Première borne 9
3.2 Deuxième borne 11
3.3 Troisième borne 12
3.4 Implémentation 12
3.5 K-machines 15

4 Méthode de recherche arborescente approchée **16**
4.1 Optimisations 16

5 Conclusion **17**

1 INTRODUCTION

L'objet de ce projet est de trouver une bonne solution au problème de passage de pièces sur 3 machines dans un atelier de fabrication, de façon à minimiser la date fin d'ordonnancement des taches. Nous savons que ce problème est NP-Difficile, et donc que nombre de possibilitées à envisager est exponentiel en le nombre de taches. Nous devons donc trouver des méthodes qui puissent optimiser le temps de calcul associé au problème.

Nous chercherons d'abord de trouver un bonne solution au problème, tout en étant s'assurant de la qualité de notre solution : on utilisera donc un algorithme approché avec garantie de performance. Nous nous aiderons de l'algorithme de Johnson qui nous fournit une solution optimale (et en temps polynomial) au problème d'ordonnancement sur 2 machines. Nous tanterons ensuite de réduire la complexité de cet algorithme.

Puis, nous chercherons trouver une solution exacte à ce problème, en se basant sur une méthode de 'Branch and Bound'. Nous chercherons ensuite des bornes inférieures et supérieures associés à cette méthode pour nous permettre d'explorer uniquement les branches intéressantes. Nous discuterons également de nos résultats suite à une implémentation de l'algorithme, de la complexité en temps temporelle ainsi qu'en espace mémoire en fonction de la taille des instances de problèmes données.

Nous finirons par discuter de méthodes envisageables pour optimiser nos algorithmes actuels, et tenterons de fournir une méthode au problème généralisé sur un nombre quelconque de machines.

2 ALGORITHME APPROCHÉ AVEC GARANTIE DE PERFORMANCE

2.1 3-Approché

Pour montrer que l'ordonnancement associé à P, une permutation arbitraire est 3-approché, il suffit de montrer que :

$$(1)\ Cout(P) \leq \sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C)$$

$$(2)\ OPT \geq \frac{1}{3} \sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C)$$

(1) et (2) nous donne bien :

$$3OPT \geq \sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C) \geq Cout(P)$$

L'ordonnancement associé à P serait donc 3-approché.

Montrons (1) : $Cout(P) \leq \sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C)$

$\sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C)$ correspond à la date de fin d'ordonnancement des taches {1,...,n} lorsqu'on est obligé d'attendre que la tache i soit terminée sur la machine C, avant de pouvoir commencer la tache i+1 sur la machine A. Cela reviendrait à exécuter toutes les taches sur une seule machine (Au niveau de la date de fin, et en supposant qu'une machine puisse exécuter les taches des machines (A,B,C)).

En pratique, lorsque la tache i sur la machine A (respectivement B) est terminée on peut commencer directement la tache i+1 sur la machine B (respectivement C).

Remarque : Nous avons l'égalité lorsqu'il n'y a qu'une seule tache à effectuer.

Montrons maintenant (2) :

$\frac{1}{3} \sum_{i=1}^{n} (d_i^A + d_i^B + d_i^C)$ correspond à la durée moyenne de travail des machines A,B,C sur les taches {1,...,n}. Il y a forcément une machine qui prend d'avantage de temps à effectuer ses

taches que la durée moyenne :

$$(3)\ \frac{1}{3}\sum_{i=1}^{n}(d_i^A + d_i^B + d_i^C) \leq \max(\sum_{i=1}^{n} d_i^A, \sum_{i=1}^{n} d_i^B, \sum_{i=1}^{n} d_i^C)$$

Comme toute les taches doivent passer sur chaque machine, nous avons :

$$(4)\ \max(\sum_{i=1}^{n} d_i^A, \sum_{i=1}^{n} d_i^B, \sum_{i=1}^{n} d_i^C) \leq OPT$$

avec (3) et (4) nous avons donc :

$$\frac{1}{3}\sum_{i=1}^{n}(d_i^A + d_i^B + d_i^C) \leq OPT$$

L'ordonnancement associé à P est donc 3-approché.

Exemples : Cas OPTIMAL et WSPT :
Soit $\epsilon > 0$,et 3 taches {1,2,3} de durées :

Tache/Machine	A	B	C
1	ϵ	ϵ	1
2	ϵ	1	ϵ
3	1	ϵ	ϵ

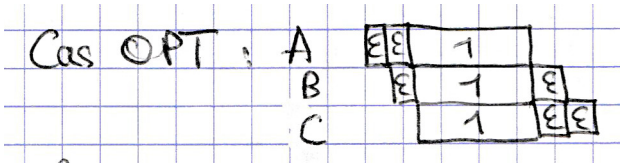

FIGURE 1: Cas OPT

Lorsque ϵ -> 0, durée de fin = 1. avec permutation (1,2,3).
Lorsque ϵ -> 0, durée de fin = 3. avec permutation (3,2,1).

2.2 2-Approché

Soit D_i^J la date de fin d'ordonnancement t retournée par l'algorithme de Johnson appliqué sur i machines. L'inégalité suivante est triviale : Pour les deux problèmes (sur 2 et sur 3 machines) l'algorithme de Johnson nous renvoie la même permutation. Les Durées de fin

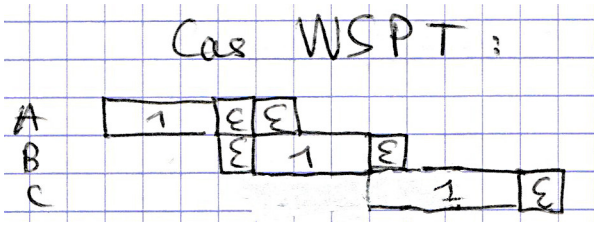

FIGURE 2: Cas WSPT

d'ordonnancement sur les machines A et B sont donc identiques. Il reste à rajouter les taches de la machine C.

$$D_3^J \leq D_2^J + \sum_{i=1}^{n} d_i^C$$

Nous avons bien l'énigalité, dans le second membre nous rajoutons les taches de C une fois que toutes les taches de B soient terminées.

Soient OPT_3 et OPT_2 étant la date de fin d'ordonnancement sur 3 machines (respectivement 2). Nous pouvons observer que (*) $D_2^J \leq OPT_3$: la durée $D_2^J \geq OPT$ est optimale sur 2 machines, et on ne prend pas en compte les taches de la machine C.

Nous avons aussi (**)$\sum_{i=1}^{n} d_i^C \leq OPT_3$: il y a au minimum la première tache de la machine A D_i^A qui devrait être prise en compte).

$$\text{Avec (*) et (**) nous avons : } D_3^J \leq D_2^j + \sum_{i=1}^{n} d_i^C \leq 2OPT$$

Exemples : Cas OPTIMAL et WSPT :
Soit $\epsilon > 0$,et 3 taches {1,2,3} de durées :

Tache/Machine	A	B	C
1	ϵ	1	ϵ
2	ϵ	ϵ	1
3	1	ϵ	ϵ

L'algorithme de Johnson nous renverrait la permutation (1,2,3), ce qui nous donne :

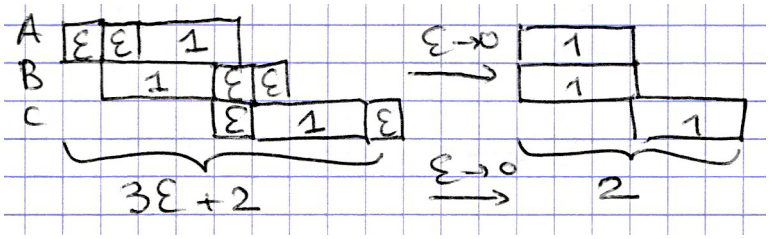

FIGURE 3: Cas WSPT : Permutation selon l'algorithme de Johnson.

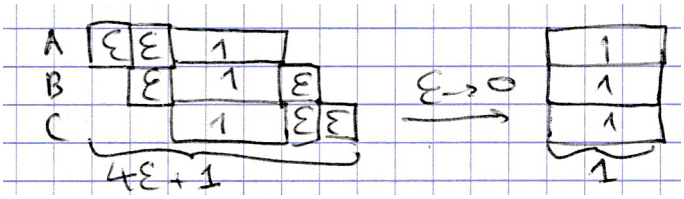

FIGURE 4: Cas OPTIMAL, avec la permutation (2,1,3)

2.3 Implémentation

Nous avons choisi d'implémenter nos algorithmes avec langage C. La gestion "A la main" de la mémoire en permet une utilisation optimale, de plus le langage étant proche de la machine et notre compilateur (GCC) permettant l'optimisation du code via des flags (On utilisera -02 pour notre projet) nous garantie des performances accrues par rapport à un langage plus haut niveau.

Le développement sous GNU/Linux nous permet également d'utiliser facilement le timestamp UNIX afin de chronométrer nos temps d'exécutions avec une précision raisonnable (En théorie, en microsecondes. En pratique, pour de petites instances le temps est difficile a évaluer à cause des nombreuses opérations des différents algorithmes, donc plutôt de l'ordre de la dixième de microseconde (À la louche)).

Nous avons implémenté deux versions de l'algorithme de Johnson afin de comparer leurs performances. La première version est celle fournie dans le sujet. La seconde trie au préalable les taches dans l'ordre croissant afin pouvoir directement sélectionner les taches souhaitées. La complexité de la première version est en $O(n^2)$ alors que la seconde nous offre une complexité de O(n Log n). On confronte alors ces deux algorithmes pour des instances avec un nombre de taches variant de 1 à 2500. Les durées respectives sont générées aléatoirement selon 3 classes : Données non-corrélées, Corrélation sur les durées d'exécution et Corrélation sur les machines.

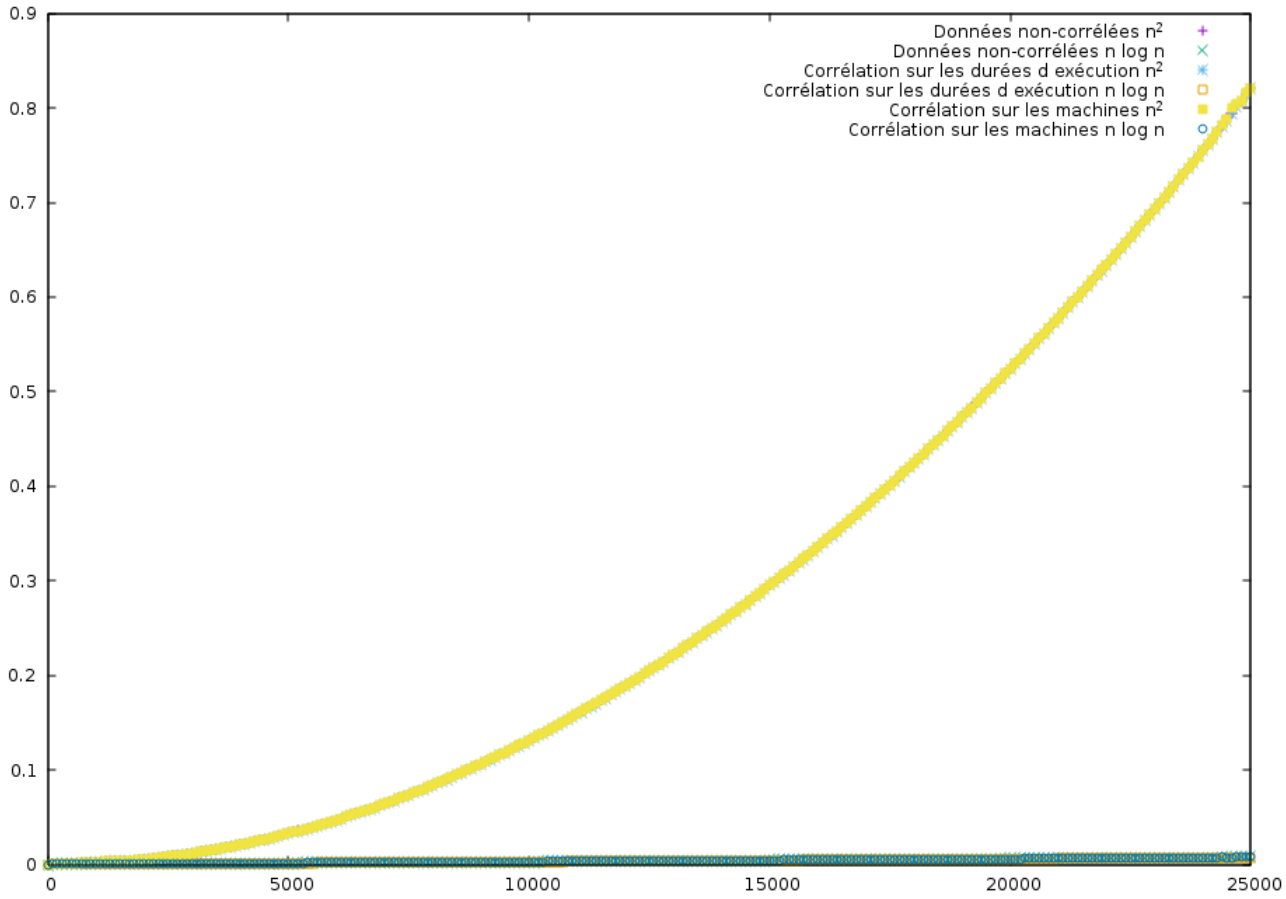

FIGURE 5: Temps de calcul des algorithmes de johnson et des différentes classes de génération d'instance.

On observe (Fig : 5) que l'algorithme avec tri propose des performances nettement supérieures à l'algorithme original. Celui-ci étant 2-approché, il pourra être utilisé afin de trouver une borne supérieure de notre instance. On observe aussi que les différentes classes n'influent pas sur le déroulement de l'algorithme. (Les courbes sont confondues pour chaque classe).

3 MÉTHODE EXACTE

3.1 Première borne

Montrons que dans b_B^π, on peut remplacer t_B^π par : (formule 12), c'est à dire $t_B^\pi \geq t'^\pi_B$ (On cherche la plus grande borne inférieure).

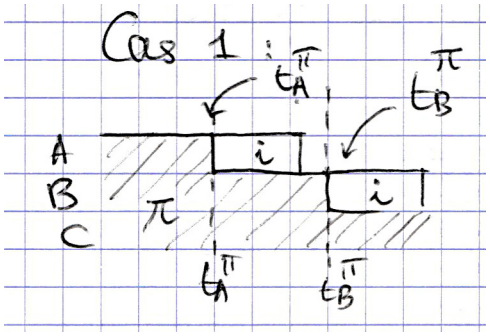

FIGURE 6: Cas 1

Ici (Fig : 6), une fois les taches π terminées sur A, on rajoute la tache i, $i \in \pi'$, $i = \arg\min_{i \in \pi'}\{d_A^i\}$, pour ne pas perdre de généralité.

Ici, $t_A + d_A^i < t_B$. Il faudra alors attendre que les taches de π se terminent sur B avant de pouvoir commencer la tache i sur B.

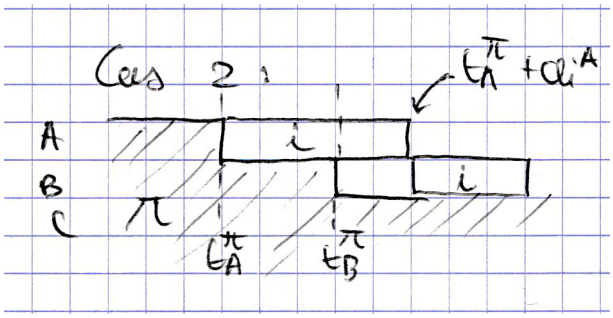

FIGURE 7: Cas 2

Ici (Fig : 7) , $t_A + d_i^A \geq t_B$. Il faudra donc attendre que la tache i se termine sur A avant de la commencer sur B.

Au lieu d'uniquement prendre en compte t_b^π pour la borne inférieur, ce serait plus efficace de prendre en compte au moins une durée en plus (ici $\min_{i \in \pi'}\{d_A^i\}$, car elle pourrait nous donner une borne inférieure plus grande (Cf cas 2).

On remplace donc t_B^π par $\max\{t_B^\pi, \min_{i \in \pi'}\{d_A^i\} + t_A^\pi\}$.

Montrons maintenant, qu'on peut toujours obtenir une borne inférieure en remplaçant t_C^π par t'^π_C :

On va faire un raisonnement analogue à (la première borne) qui consiste à chercher les taches contraignantes. On va également choisir les taches de durée minimale, pour ne pas perdre de généralité (Pour toute permutation de π', on aura toujours au moins le minimum).

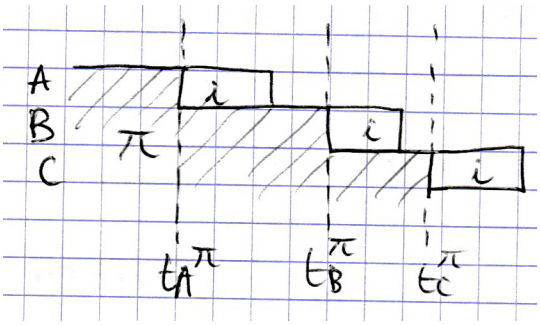

FIGURE 8: Taches de π contraignantes

Ici (Fig : 8) $t_C^\pi > t_B^\pi + \min_{i\in\pi'}\{d_B^i\}$ et $t_C^\pi > t_A^\pi + \min_{i\in\pi'}\{d_A^i + d_B^i\}$

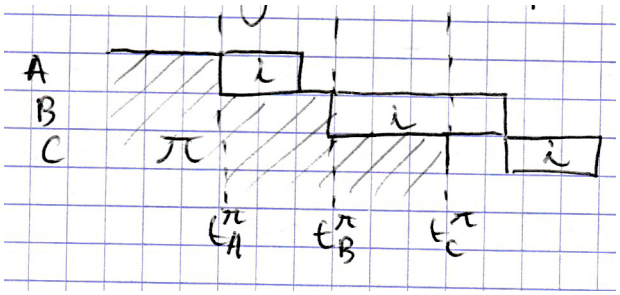

FIGURE 9: Tache i sur machine B contraignante

Ici (Fig : 9) $t_B^\pi + \min_{i\in\pi'}\{d_B^i\} > t_C^\pi$ et $t_B^\pi + \min_{i\in\pi'}\{d_B^i\} > t_A^\pi + \min_{i\in\pi'}\{d_A^i + d_B^i\}$

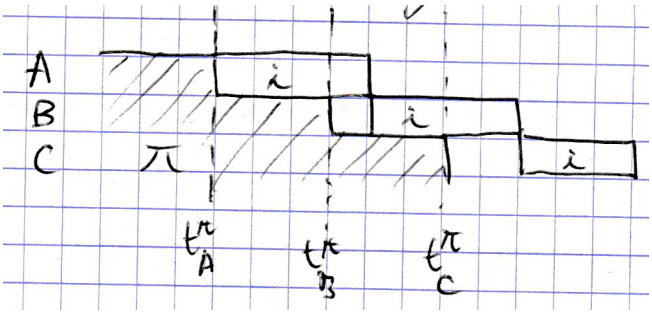

FIGURE 10: Tache i sur machine A et B contraignante

Ici (Fig : 10) $t_A^\pi + \min_{i\in\pi'}\{d_i^A + d_B^i\} > t_C$ et $t_A^\pi + \min_{i\in\pi'}\{d_i^A + d_B^i\} > t_B + m$...

Opter pour la borne inférieure de valeur maximale nous premet de construire une borne inférieure plus proche de la solution réalisable. Nous pourrons donc espérer un élage plus fréquent des branches de l'arbre. En choisissant la tache minimale comme la première tache on ne perds pas de généralité car on est à la recherche d'une borne inférieure. (si la tache $i \in \pi'$ était contraignante sur la machine $j \in \{A, B, C\}$, toutes les taches de π' le seraient également).

On peut donc remplacer t_C^π par ${t'}_C^\pi$, ce qui nous donne :

$$ {t'}_C^\pi = \max\{t_C^\pi, t_B^\pi + \min_{i\in\pi'} d_B^i, t_A^\pi + \min_{i\in\pi'}\{d_A^i + d_B^i\}\}$$

3.2 Deuxième borne

On schématise cette situation (Fig 11) ; pour simplifier, on suppose les taches {1,...,k-1,k,k+1,...,n} tel que : $d_i^A \leq d_i^C, \forall i \in \{1, ..., k-1\}$ et $d_i^A > d_i^C, \forall i \in \{k+1, ..., n\}$

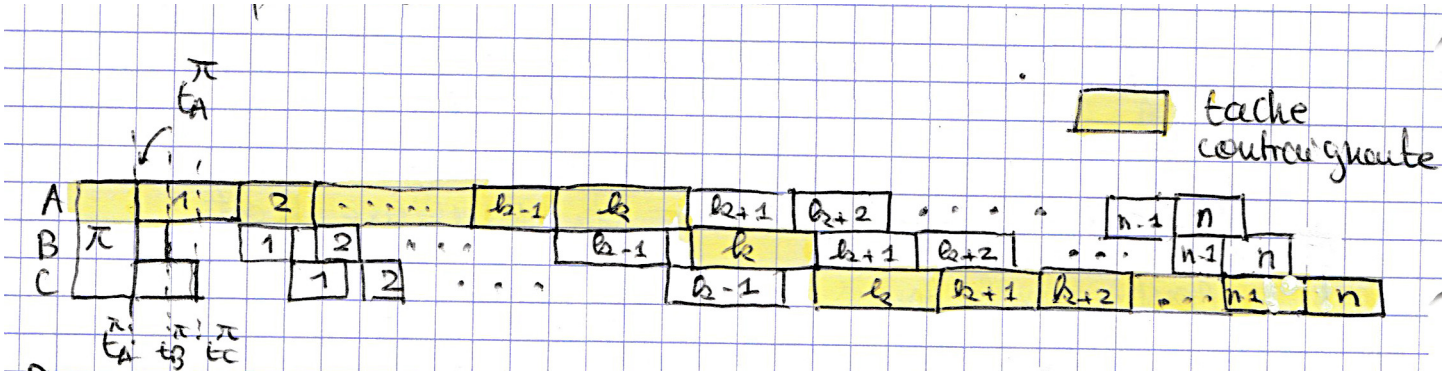

FIGURE 11

Pour simplifier, on suppose que les taches π des machines B et C ne sont pas contraignantes (si elles l'était, on aurait un temps total égal ou plus grand). Pour que l'ordonnancement soit tel que sa date de fin soit corresponde à :

$$t_A^\pi + \sum_{d_A^i \leq d_C^i} d_A^i + d_A^k + d_B^k + d_C^k + \sum_{d_A^i > d_C^i} d_C^i$$

, il faudrait faire les suppositions fortes que les taches {1,...,k} soient contraignantes sur la machine A (Alors que par définition, on a $d_A^i \leq d_C^i$), c'est à dire que ces mêmes taches ne soient pas contraignantes pour les machines B et C. Il faudrait également que les taches {k+1,...,n} soient contraignantes sur la machine C (alors que $d_i^A > d_i^C, \forall i \in \{k+1, ..., n\}$.

On a que $\forall k \in \pi'$ la date de fin d'ordonnancement associé à P est supérieure ou égale à cette borne inférieure. Pour trouver une borne inférieure, nous avons la liberté de choisir notre k, la meilleur option serait donc de choisir un k qui maximise notre borne, c'est à dire :

$$b2 = t_A^\pi + \max_{k\in\pi'}\{d_A^k + d_B^k + d_C^k + \sum_{i\in\pi', d_A^i \leq d_C^i} d_A^i + \sum_{i\in\pi', d_A^i > d_C^i} d_C^i\}$$

3.3 Troisième borne

En faisant le même raisonnement (Fig : 12), on peut déduire la borne :

$$b3 = t_B^\pi + \max_{k\in\pi'}\{d_B^k + d_C^k + \sum_{i\in\pi', d_B^i \leq d_C^i} d_B^i + \sum_{i\in\pi', d_B^i > d_C^i} d_C^i\}$$

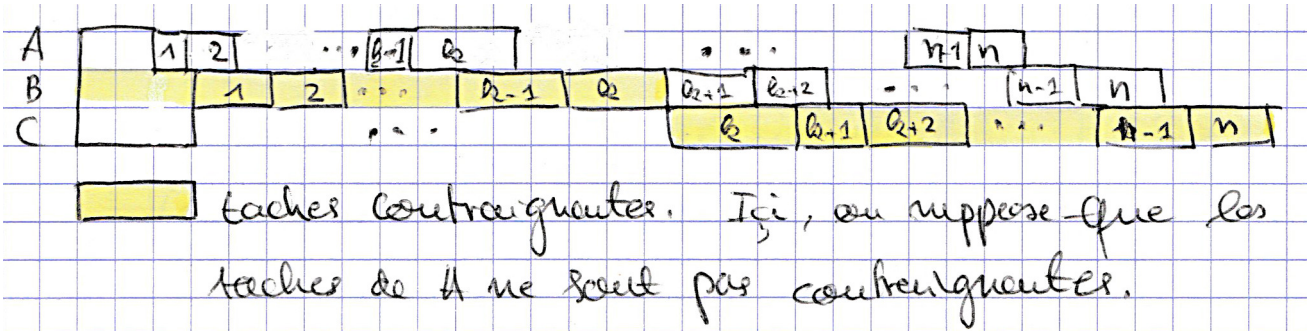

FIGURE 12

3.4 Implémentation

Pour traiter cette partie nous avons choisi le même langage pour les mêmes raisons que précédemment. Ainsi que pour une réutilisation aisée de l'algorithme de Johnson.

Ayant opté pour une implémentation récursive du branch and bound, celle-ci diffère un peu de celle vue en cours. Le cœur récursif de l'implémentation se charge d'effectuer toute les permutations possibles, et à chaque permutation, on calcule une borne supérieure et une borne inférieure. Celles-ci déterminent si oui ou non, l'exploration se poursuit dans la branche courante (comme traité en TD, pour un problème de minimisation). Nous avons choisi l'implémentation récursive car elle nous paraissait plus intuitive pour un parcours d'arbre et utilisant l'optimisation de GCC, la récursion est traitée comme une boucle avec une

pile. En soit, après optimisation, le code doit fonctionner de la même façon que l'algorithme étudié en cours.

L'algorithme pouvant difficilement traiter des instances à plus de 10 taches, ce sera notre taille maximale durant les tests. Les moyennes ont été calculées sur 200 tests.

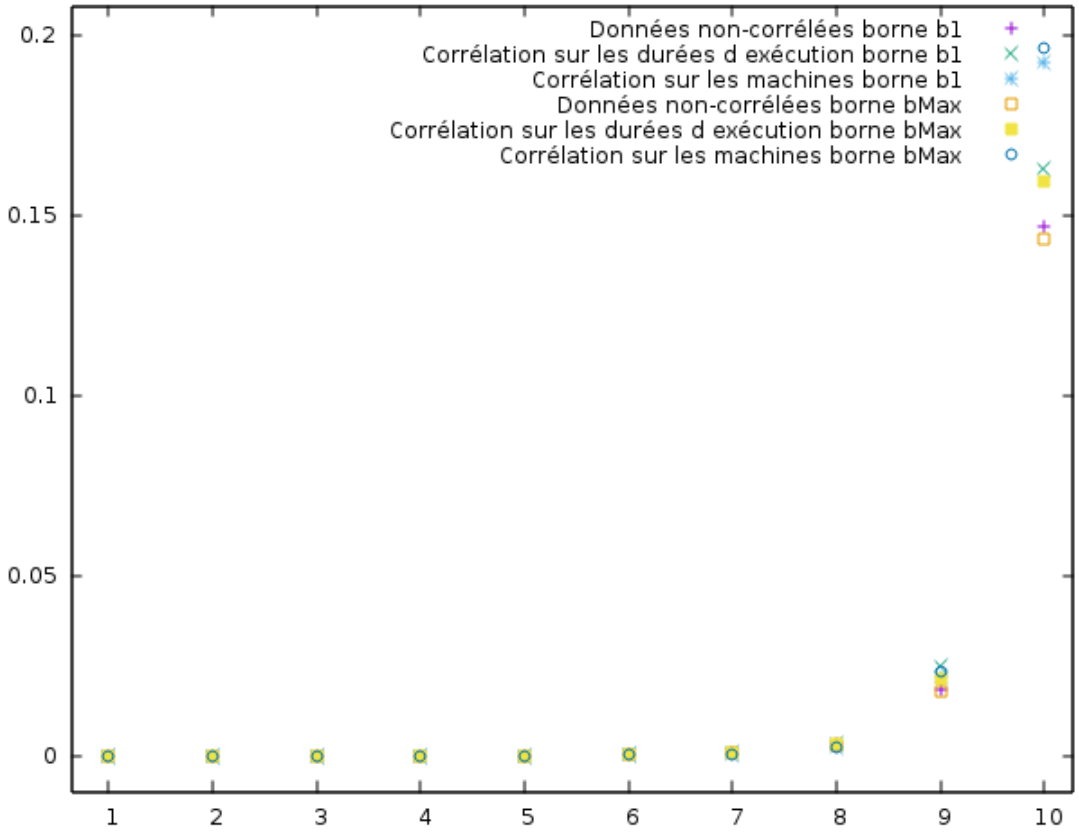

FIGURE 13: Temps de calcul moyen pour pour les bornes b1 et bMax ainsi que des différentes classes de génération d'instance.

L'algorithme ne permettant pas le traitement de grandes instances, il est difficile de faire une analyse pertinente de nos résultats.

On constate que l'utilisation du branch and bound ne réduit pas la complexité dans le pire des cas (la complexité reste exponentielle), mais en pratique le temps de calcul est fortement diminué (Sur les figures 13 et 14 mais explose avecle nombre de nœuds explorés à partir de 10 taches, et à 11 taches le temps de calcul est déjà trop élevé pour effectuer 200 tests avec chaque classe de génération).

On peut observer sur les temps d'exécutions (Fig : 13) que : La borne bMax dans les 3 classes montre des performances supérieures à la borne b1. Les données non corrélées sont les plus rapides à être traitées, suivies des données corrélées sur les durées d'exécution

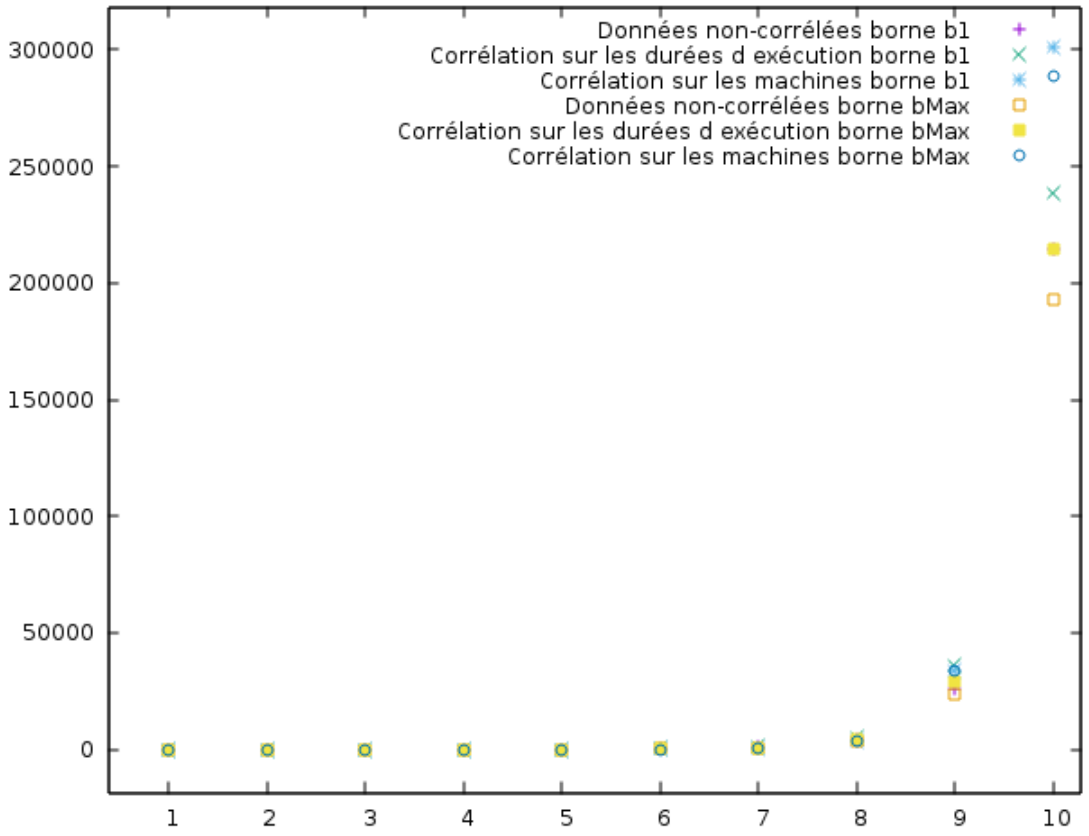

FIGURE 14: Nombre moyen de nœuds explorés pour les bornes b1 et bMax ainsi que des différentes classes de génération d'instance.

et pour finir, les données corrélées. Il est possible d'observer un résultat similaire avec le nombre de nœuds explorés (Fig : 14).

On peut observer l'utilisation mémoire de l'algorithme pour une instance à 10 taches à l'aide de massif ; un outil de valgrind.

On peut constater que l'allocation mémoire totale du programme est très lourde, c'est lié à l'implémentation de Johnson qui retourne une nouvelle instance à chaque appel. On pourrait réduire la consommation mémoire totale en modifiant cette implémentation. En contrepartie, l'allocation mémoire maximale durant le déroulement de l'algo est très faible (environ 11Ko). Cela permettrait traiter de grandes instances sans se soucier du problème de la mémoire.

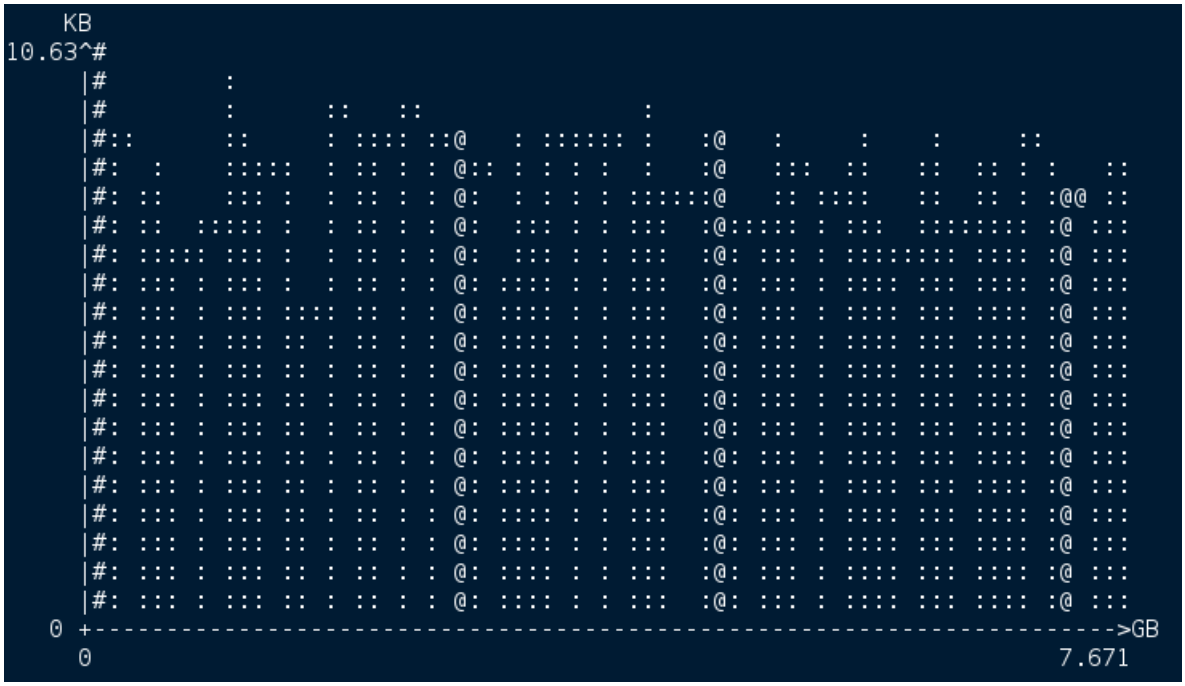

FIGURE 15: Occupation mémoire du branch and bound.

3.5 K-machines

Il devrait être possible d'utiliser cette méthode pour le problème sur k machines. Mais il faudrait estimer de nouvelles bornes inférieures et supérieures afin d'en garantir les performances.

4 MÉTHODE DE RECHERCHE ARBORESCENTE APPROCHÉE

4.1 Optimisations

Il existe plusieurs méthodes qui pourraient encore réduire le temps de calcul moyen de manière significative.

Nous pouvons dans un premier temps modifier notre algorithme de Branch and Bound, pour effectuer un parcours en largeur modifié : l'algorithme explore toutes les branches en calculant pour chaque noeud une solution réalisable à l'aide de l'algorithme de Johnson. On emprunte en premier les branches les plus prometteuses, c'est-à-dire celles qui nous renvoie les meilleures solutions réalisables. En utilisant le même principe d'élagage des noeuds, nous pouvons aussi élaguer les branches dès que possible. Cette algorithme révisé nous permettrai potentiellement de trouver la meilleure solution dans un temps plus court (pour encore gagner du temps, nous pouvons nous très bien nous arrêter après un nombre quelconque k feuilles rencontrées, mais cette méthode ne serait plus exacte).

Nous pouvons ensuite aussi optimiser le temps de calcul en appliquant l'algorithme de Branch and Greed, c'est-à-dire choisir uniquement la branche qui nous rende la borne supérieure la plus faible, puis réitérer l'opération sur les branches du sous arbre. Cette méthode nous assure uniquement une solution convenable, mais nous assure une complexité de $O(log(n))$ (nous descendons qu'une fois jusqu'à atteidre un noeud terminal). Cette méthode pourrait s'avérer utile pour de très grande instances. De plus, comme nous utilisons l'algorithme de Johnson, nous pouvons contrôler notre distance à l'ordonnancement optimal : notre solution serait au moins 2-approchée.

Faute de temps, nous étions dans l'incapacité d'implémenter ces méthodes.

5 CONCLUSION

Nous avons pu constater qu'apporter des solutions à un problème NP-difficile est fastidieuse : l'algorithme approché avec garantie de performance nous renvoie des résultats convenables, mais il n'existe pas nécéssairement d'algorithmes approchés pour un problème NP-difficile (exemple : problème du voyageur de commerce). Il faut dans le cas échéant utiliser la méthode de branch and bound, qui nous assure une solution exacte, mais sans que la complexité dans le pire des cas soit réduite. Nous avons néenmoins des temps de calcul corrects pour des instances de taches qui restes petites, mais nous ne pourrons pas appliquer cette méthode pour des grandes instances. En optimisant cette méthode, il serait également possible d'avoir un temps de calcul en moyenne plus petit. Dans notre méthode exacte il est possible de baisser la complexité, sans pour autant trouver de solution en temps polynomial (sauf si P=NP).

Nous nous sommes également aperçus que langage C n'est pas forcément pertinent dans le cadre de ce projet étudiant. En effet, même si les temps de calcul et l'occupation mémoire maximales sont intéressants, ces aspects sont peu intéressants dans cette étude. Les temps de calculs explosent trop rapidement pour rendre l'investissement consacré à la gestion de la mémoire et de l'optimisation intéressant par rapport à un langage plus haut niveau, qui nous aurais permis de traiter les parties manquantes.