Projet COMPLEX

Victor RACINE et Loic URIEN

Novembre 2017

Table des matières

1 Introduction

Au cours de ce projet, nous avons étudié un problème classique d'ordonnancement de tâches sur des machines indépendantes et les algorithmes permettant de le résoudre, approchés et exacts. La première partie se concentre sur un algorithme approché pour 3 machines basé sur l'algorithme exact de Johnson, qui résout le problème pour 2 machines. La deuxième partie consiste à mettre au point un algorithme d'arborescence de résolution exacte du problème, en se concentrant dans un premier temps sur l'études des différentes bornes inférieures et la manière d'étendre la résolution à k machines. Elle se conclura par la mise en oeuvre de la méthode en langage python, puis de son optimisation par les méthodes approchées "Limited Discrepancy Search" (LDS) et "Branch and Greed". La troisième partie, quant à elle, sera dédiée à l'étude expérimentale des performances des différents algorithmes.

2 Algorithme approché avec garantie de performance

Dans cette partie, nous allons nous intéresser aux algorithmes polynomiaux approchés permettant de résoudre le problème étudié, puis donner une garantie de performance pour ces deux algorithmes.

2.1 Algorithme 3-approché par permutation arbitraire

On considère le pire cas, dans lequel chaque tâche doit attendre que la précédente se termine sur chaque machine avant de se lancer. Ainsi, pour toute permutation P avec x la date de fin d'une permutation quelconque et OPT la date de fin de la permutation optimale, on a :

$$x \leq \sum_{i \in P} d_A^i + d_B^i + d_C^i$$

Ainsi, si on enlève la condition de séquentialité d'une tâche sur les machines, i.e qu'il est possible d'exécuter une tâches en parallèle sur chacune des machines, on obtient :

$$OPT \geq \sum_{i \in P} max(d_A^i, d_B^i, d_C^i)$$

De plus, on a bien que :

$$max(d_A^i, d_B^i, d_C^i) \geq \frac{d_A^i + d_B^i + d_C^i}{3}$$

Il s'ensuit que :

$$OPT \geq \frac{d_A^i + d_B^i + d_C^i}{3} \iff x \leq 3OPT$$

Ainsi, on a bien montré que la méthode présentée est 3-approchée.

2.2 Algorithme 2-approché par Johnson

Montrons qu'utiliser l'algorithme de Johnson sur un problème d'ordonnancement à trois machines produit une méthode 2-approchée. Soient $OPT_{johnson}$ la valeur retournée par l'algorithme de Johnson et X l'ensemble des tâches considéré. Soient OPT la date de fin de la permutation optimale pour X et x la valeur retournée par notre méthode. Ainsi, comme Johnson retourne une valeur optimale pour un ordonnancement sur deux machines, on a bien que :

$$x \leq OPT_{johnson} + \sum\nolimits_{i \in X} d_C^i$$

De plus, $OPT \geq OPT_{johnson}$ et $OPT \geq \sum_{i \in X} d_C^i$

Soit $m = max(OPT_{johnson}, \sum_{i \in X} d_C^i)$ et $\alpha = 2m$

Ainsi, on a bien :

$$x \leq 2m = \alpha$$

Finalement

$$OPT \geq \frac{\alpha}{2} \geq \frac{x}{2} \Longrightarrow x \leq 2OPT$$

La méthode est donc bien 2-approchée.

3 Méthodes arborescentes exactes et approchées

Dans cette section, nous allons exclusivement nous intéresser aux méthodes arborescentes, exactes et approchées. Nous allons tout d'abord nous concentrer sur la recherche de bornes inférieures avant de nous lancer dans l'écriture des algorithmes.

3.1 Recherche de bornes

Dans l'énoncé, trois bornes nous sont données
$b_A^\pi = t^p i_A + \sum_{i\in\pi'} d_A^i + min_{i\in\pi'}(d_B^i + d_C^i)$
$b_B^\pi = t_B^\pi + \sum_{i\in\pi'} d_B^i + min_{i\in\pi'}(d_C^i)$
$b_C^\pi = t_C^\pi + \sum_{i\in\pi'} d_C^i$

Il est facile de montrer que l'ont peut remplacer t_B^π par $max(t_B^\pi,\ t_A^\pi + min_{i\in\pi'}(d_A^i))$ dans la borne b_B^π en considérant les deux cas suivants :

- Si $t_B^\pi > t_A^\pi + min_{i\in\pi'}(d_A^i)$, alors même en exécutant la tâche k de π' avec le plus court temps sur A, k devra attendre son tour pour passer sur la machine B. Dans ce cas on ne peut pas faire mieux qu'attendre la fin de la tâche en cours sur la machine B. La tâche k ne pourra donc s'exécuter sur B qu'à partir de la date t_B^π.

- Si $t_B^\pi \leq t_A^\pi + min_{i\in\pi'}(d_A^i)$, alors lorsque la dernière tâche de π aura fini son exécution sur la machine B, la prochaine tâche ne pourra commencer son exécution sur cette machine qu'à la fin de son exécution sur la machine A. Parmis toutes les tâches de π', aucune ne se finit sur A avant que la dernière tâche de π ne se finisse sur B. Ainsi, toute tâche de π' ne pourra s'exécuter sur B qu'à partir de la date $t_A^\pi + min_{i\in\pi'}(d_A^i)$

Montrons de même que l'on peut remplacer t_C^π par $max(t_C^\pi, t_B^\pi + min_{i\in\pi'}(d_B^i), t_A^\pi + min_{i\in\pi'}(d_A^i + d_B^i))$. Notons t'^π_C cette valeur, on alors que :

- Si $t'^\pi_C = t_C^\pi$, alors aucune tâche de π' ne pourra s'exécuter sur C avant la fin de la dernière tâche de π sur C. Ainsi, la première date où une tâche de π' pourra s'exécuter sur la machine C sera t_C^π.

- Si $t'^\pi_C = t_B^\pi + min_{i\in\pi'}(d_B^i)$, alors soit $k \in \pi'$ la tâche la plus courte sur B, aucune tâche ne pourra s'exécuter sur C avant la fin de k sur B.

- Si $t'^\pi_C = t_A^\pi + min_{i\in\pi'}(d_A^i + d_B^i))$, alors soit $k \in \pi'$ la tâche ayant la somme du temps sur A et B la plus courte ($k = argmin_{i\in\pi'}(d_A^i + d_B^i)$), alors aucune tâche ne pourra s'exécuter sur C avant que k ait fini son exécution sur A et sur B.

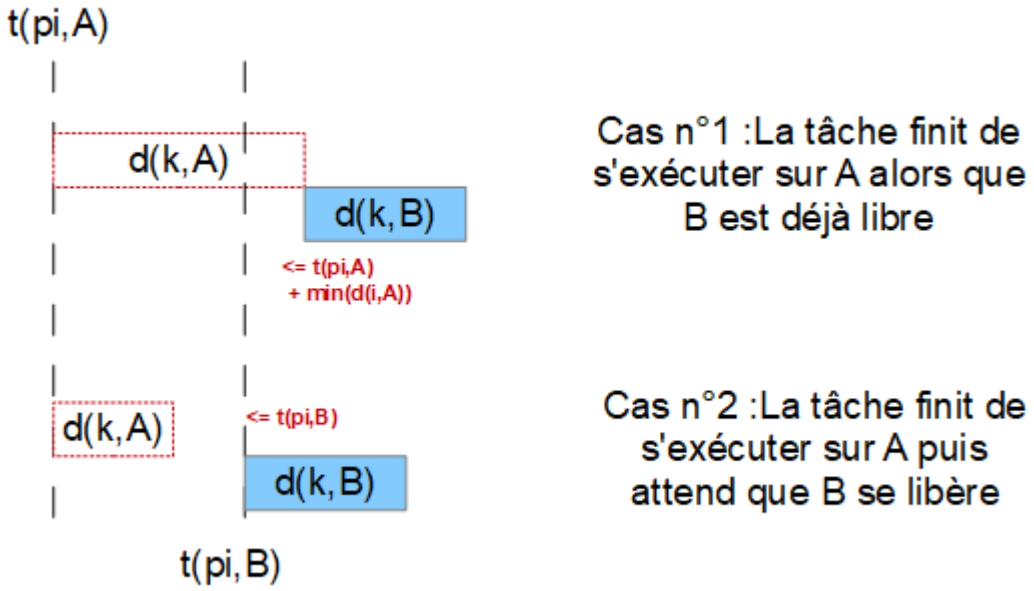

FIGURE 1 – Visualisation graphique de la preuve pour b1

Il est également possible de dériver deux autres bornes. En effet, soient P une permutation et π, π' deux ensembles tels que $P = \pi \cup \pi'$. Ainsi, π est le début de la permutation et π' la fin. Soient $k \in \pi'$, π_1 les tâches avant k dans π' et π_2 les tâches après k dans π'. On a alors

$$t_A^\pi + \sum_{i \in \pi_1, d_A^i \leq d_C^i} d_A^i + \sum_{i \in \pi_1, d_A^i > d_C^i} d_C^i = t_A^\pi + \sum_{i \in \pi_1} min(d_A^i, d_C^i) \leq t_A^\pi + \sum_{i \in \pi_1} d_A^i = t_\alpha$$

De plus,

$$t_\alpha + d_A^k + d_B^k + d_C^k \leq t_C^k$$

Ainsi, comme

$$t_C^k + \sum_{i \in \pi_2, d_A^i \leq d_C^i} d_A^i + \sum_{i \in \pi_2, d_A^i > d_C^i} d_C^i = t_C^k + \sum_{i \in \pi_2} min(d_A^i, d_C^i) \leq t_C^k + \sum_{i \in \pi_2} d_C^i \leq t_C^P$$

On a bien que

$$t_\alpha + d_A^k + d_B^k + d_C^k + \sum_{i\in\pi_2, d_A^i \leq d_C^i} d_A^i + \sum_{i\in\pi_2, d_A^i > d_C^i} d_C^i \quad (1)$$

$$\leq t_C^k + \sum_{i\in\pi_2, d_A^i \leq d_C^i} d_A^i + \sum_{i\in\pi_2, d_A^i > d_C^i} d_C^i \quad (2)$$

$$\leq t_C^P \quad (3)$$

En développant t_α, on en déduit que

$$c^k = t_A^\pi + (d_A^k + d_B^k + d_C^k) + \sum_{i\in\pi', i\neq k, d_A^i \leq d_C^i} d_A^i + \sum_{i\in\pi', i\neq k, d_A^i > d_C^i} d_C^i \leq t_C^P$$

On peut en déduire une borne b_2 telle que $b_2 = max_{k\in\pi'}(c^k)$

De même, en se plaçant sur la machine B, on obtient la formule :

$$c'^k = t_B^\pi + (d_B^k + d_C^k) + \sum_{i\in\pi', i\neq k, d_B^i \leq d_C^i} d_B^i + \sum_{i\in\pi', i\neq k, d_B^i > d_C^i} d_C^i$$

Similairement à la preuve précédente, on a que

$$t_B^\pi + \sum_{i\in\pi_1, d_B^i \leq d_C^i} d_B^i + \sum_{i\in\pi_1, d_B^i > d_C^i} d_C^i = t_A^\pi + \sum_{i\in\pi_1} min(d_B^i, d_C^i) \leq t_A^\pi + \sum_{i\in\pi_1} d_A^i = t_\beta$$

Et

$$t_\beta + d_B^k + d_C^k \leq t_C^k$$

Comme

$$t_C^k + \sum_{i\in\pi_2, d_B^i \leq d_C^i} d_B^i + \sum_{i\in\pi_2, d_B^i > d_C^i} d_C^i = t_C^k + \sum_{i\in\pi_2} min(d_B^i, d_C^i) \leq t_C^P$$

Alors en développant le résultat comme précédemment on prouve bien que $c'^k \leq t_C^P$. On en déduit une nouvelle borne $b_3 = max_{k\in\pi'}(c'^k)$

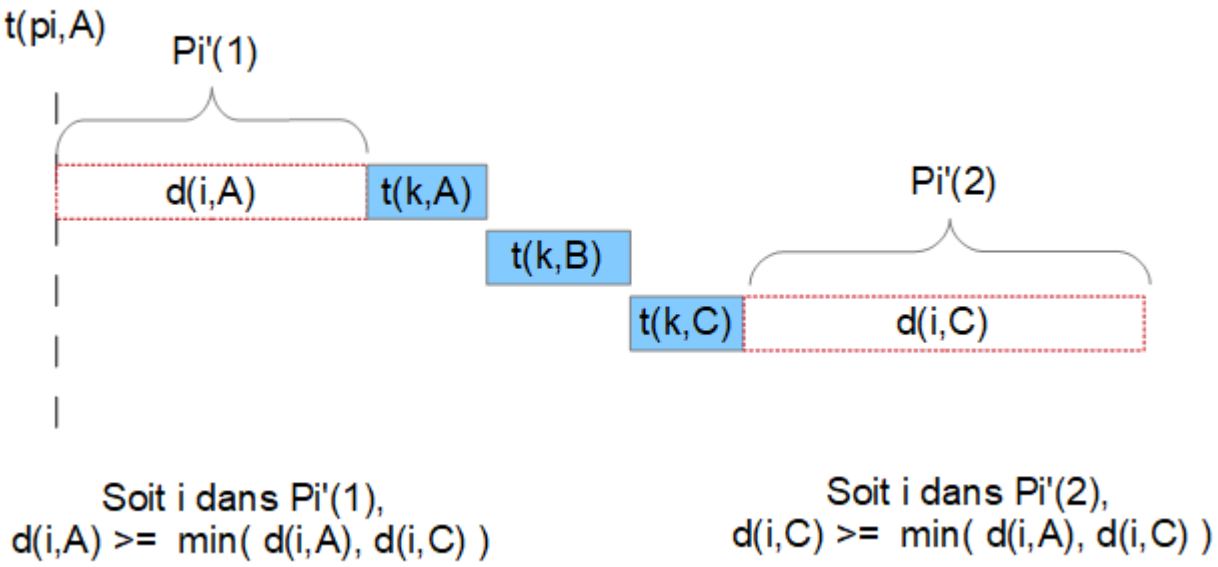

FIGURE 2 – Visualisation graphique de la preuve pour b2 et b3

3.2 Optimisation de la méthode exacte

Bien que la méthode exacte soit bien plus rapide qu'une méthode gloutonne, elle s'avère toujours relativement lente sur de grandes instances. Il est cependant possible de l'optimiser en utilisant l'heuristique de "la meilleure borne d'abord". Cette heuristique consiste à trouver un chemin de longueur k qui n'emprunte que des arêtes ayant la meilleure heuristique et à l'emprunter avant les autres, espérant ainsi trouver une meilleure valeur plus rapidement. L'algorithme effectue donc un pré-traitement afin de déterminer le chemin de longueur k qui emprunte les meilleurs bornes. Ainsi, plus la valeur k est grande, plus le pré-traitement est long, mais également meilleur est le chemin initial dans le cas général et inversement. Cet algorithme est implémenté sous le nom d'*eval*2.

Algorithm 1: Algorithme de pré-traitement (TrierMeilleureHeuristique)

Data: dureesTaches, listeTaches, dates : dates actuelles de fin des tâches sur les machines, h : heuristique de la borne inférieure, k : le nombre d'étapes de pré-traitement à effectuer

Result: ord : un ordonancement optimal

```
1  meilleureTache ← 0
2  meilleureBorne ← ∞
3  meileureDates ← ∅
4  foreach tache de listeTaches do
5  |  listeTaches ← listeTaches \ tache
6  |  nouvellesDates ← mettreAJourDates(dates, dureesTaches[tache])
7  |  inf ← h(listeTaches, dureeTaches, nouvellesDates)
8  |  if inf ≤ meilleureBorne then
9  |  |  meilleureBorne ← inf
10 |  |  meilleureTache ← tache
11 |  |  meilleureDates ← nouvellesDates
12 |  end
13 |  listeTaches ← listeTaches + [tache]
14 end
15 listeTaches ← listTaches \ [tache]
16 if (k-1) == 0 then
17 |  retourner [meilleureTache] + listeTaches
18 else
19 |  retourner [meilleureTache] +
   |   TrierMeilleureHeuristique(dureesTaches, listeTaches,
   |   meilleureDates, h, k - 1)
20 end
```

Algorithm 2: Algorithme exact optimisé (eval2)

Data: dureesTaches, nombreTaches, h : heuristique de la borne inférieure, k : le nombre d'étapes de pré-traitement à effectuer
Result: ord : un ordonancement optimal

```
1 dates ← (0, 0, 0)
2 listeTachesInitiale ← GenererListeTache(nombreTaches)
3 listeTaches ← TrierMeilleureHeuristique(dureesTaches,
    listeTachesInitiale, dates, h, k)
4 retourner evaluerNoeud(dureesDates, listeTaches, dates, h)
```

3.3 Optimisation de la méthode par algorithmes approchés

Afin de trouver une méthode de recherche arborescente approchée, on peut se concentrer sur deux points :

- Le calcul des bornes
- La fonction d'évaluation

En ce qui concerne le calcul des bornes, l'idée la plus évidente et de diminuer le nombre de tâches à traiter à chaque évaluation de la borne. Par exemple, soit une fonction h(dureesTaches, listeTaches) qui calcule une borne inférieure connaissant la liste des tâches à traiter, on peut par exemple couper la liste en deux et multiplier le résultat de l'appel par deux :

Algorithm 3: Algorithme de borne inférieure approchée

Data: dureesTaches, listeTaches, h
Result: une borne inférieure approximative

```
1 listeTachesDemi ← []
2 for i allant de 1 à ⌊taille(liste_taches)/2⌋ do
3 |   listeTachesDemi ← listeTachesDemi + liste_taches[i] end
4 |   retourner 2 * h(dureesTaches, listeTachesDemi)
5 |
```

Une version plus générique de cet algorithme pourrait s'écrire en prenant en compte un paramètre α :

Algorithm 4: Algorithme de borne inférieure approchée généralisé

Data: dureesTaches, listeTaches, h
Result: une borne inférieure approximative

```
1 listeTachesAlpha ← []
2 for i allant de 1 à ⌊taille(liste_taches) / α⌋ do
3 |   listeTachesDemi ← listeTachesDemi + liste_taches[i] end
4 |   retourner α * h(dureesTaches, listeTachesAlpha)
5 |
```

Tout en faisant attention, bien sûr, à ce que la taille de la liste des tâches soit supérieure à α.

En pratique, modifier les bornes de cette manière est une solution peu scalable qui a tendance à donner d'assez mauvais résultats. Le fait que les bornes inférieures ne soient plus forcément de vraies bornes inférieures peut également dans certains cas pathologiques (α * h $<$ tâches non traitées) conduire à une exploration trop exhaustive de l'arbre, là où l'on aurait certainement trouvé une solution plus rapidement.

La deuxième solution consiste à utiliser des algorithmes approchés tels que l'algorithme LDS et "Branch and Greed".

3.3.1 Algorithme LDS (Limited Discrepancy Search)

Le principe de cet algorithme et de limiter le nombre de déviation du chemin ayant la meilleure heuristique lors d'une recherche. Si on dévie trop de ce chemin, on cesse l'exploration, partant du principe que la meilleure réponse ne doit pas s'y trouver. Bien que plus efficace dans le cadre d'un arbre binaire, il s'est révélé relativement performant sur de nombreuses instances de notre problème. Cette méthode se base sur deux autres algorithmes, "LDSsearch" et "LDSprobe" [2] [4].Soit k le nombre de déviations autorisées, "LDSsearch" va lancer l'algorithme "LDSprobe" pour chaque valeur de 0 à k et garder la meilleure valeur trouvée. L'algorithme "LDSprobe" va, quant à lui, effectuer une simple recherche d'arbre pendant k étapes. A la k-ième étape, il se bornera à prendre le chemin ayant la meilleure heuristique.

Ci dessous quelques exemples tirés d'une présentation [1]

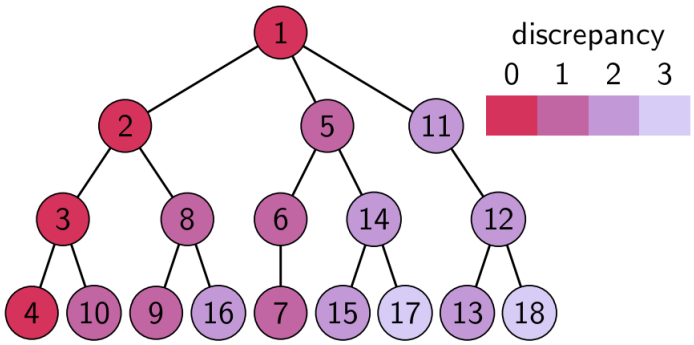

FIGURE 3 – Arbre de recherche LDS pour k = 4 (tout l'arbre)

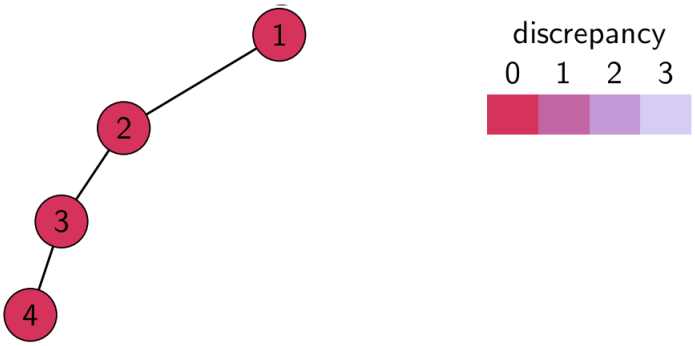

FIGURE 4 – Arbre de recherche LDS pour k = 0 (aucune déviation)

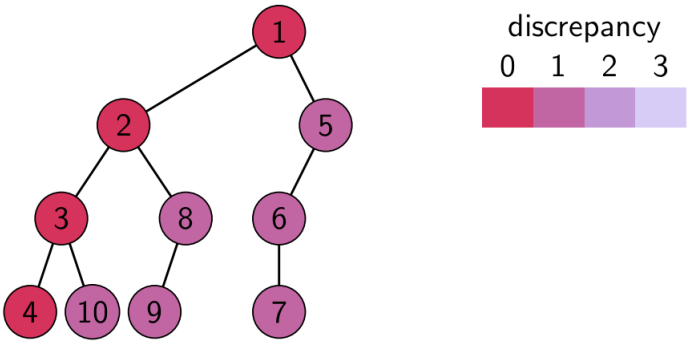

FIGURE 5 – Arbre de recherche LDS pour k = 1 (une seule déviation)

3.3.2 Algorithme Branch and Greed

Le quatrième et dernier algorithme est également un algorithme approché, connu sous le nom de "Branch and Greed"[3]. Le but de cet algorithme est que pour chaque arête à chaque niveau, on lance un algorithme glouton qui va arriver à une feuille. On prend la branche qui est arrivée à la feuille avec la meilleur valeur, et on recommence. On a au final un algorithme assez peu précis, mais extrêmement performant, capable de résoudre de grandes instances du problème. La qualité des solutions retournées se rapproche de la qualité des solutions de Johnson. Plus d'information dans la partie sur les résultats expérimentaux.

3.4 Adaptation des méthodes pour des instances à k machines

Nous pouvons remarquer que la taille et la topologie de l'arbre ne dépend en aucun cas du nombre de machines, en fait seuls deux éléments de notre méthode en dépendent :

- La fonction de mise à jour des dates de fin, appelée à chaque développement d'un noeud.

- Les bornes inférieures.

Nous pouvons adapter les bornes trouvées précédemment comme suit : Soient $M_1, M_2, ..., M_n$ les n machines de notre problème ordonnées selon leur index. Pour $k \in [\![1, n]\!]$ et une machine M_k, on a

$$b^{\pi}_{M_k} = t^{\pi}_{M_k} + \sum\nolimits_{i \in \pi'} d^i_{M_k} + min_{i \in \pi'}(\sum\nolimits_{k \in [\![1,n]\!]} d^k_{M_k}) \tag{4}$$

$$b_1 = max_{k \in [\![1,n]\!]}(b^{\pi}_{M_k}) \tag{5}$$

De même pour b2 :

$$c^k_{M_p} = t^{\pi}_{M_p} + \sum\nolimits_{i \in [\![1,n]\!]} d^k_{M_i} + \sum\nolimits_{i \in \pi', i \neq k, d^i_{M_p} \leq d^i_{M_n}} d^i_{M_p} + \sum\nolimits_{i \in \pi', i \neq k, d^i_{M_p} > d^i_{M_n}} d^i_{M_n} \tag{6}$$

$$b_2 = max_{p \in [\![1,n]\!]}(max_{k \in \pi'}(c^k_{M_p})) \tag{7}$$

4 Résultats expérimentaux

Nous allons maintenant nous concenter sur les résultats obtenus par notre implémentation. Tous les tests ont été réalisés sur les 3 classes d'instances aléatoires données dans le sujet, aucune corrélation, corrélation temps et corrélation machine. Nous nommerons ces classes respectivement dans la suite NC (non-correlated), TC (time correlated), MC (machine correlated). Tous les tests ont été effectués sur une machine disposant d'un processeur "Intel(R) Core(TM) i3-5010U CPU @ 2.10GHz", de 4Go de RAM, tournant sous ArchLinux.

4.1 Qualité des bornes

Afin de choisir la meilleure borne pour le reste des tests, nous avons choisi de commencer par l'évaluation des bornes. Ci-dessous les résultats obtenus en utilisant l'algorithme exact initial (non optimisé) en exécutant l'algorithme 100 fois pour les tailles de 3 à 8, puis 10 fois pour la taille 9 et 5 fois pour la taille 10 :

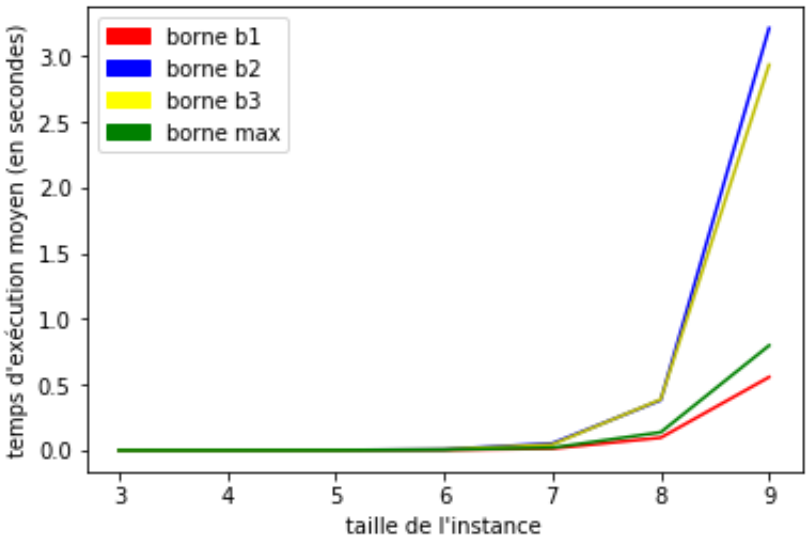

FIGURE 6 – Qualité des bornes pour NC

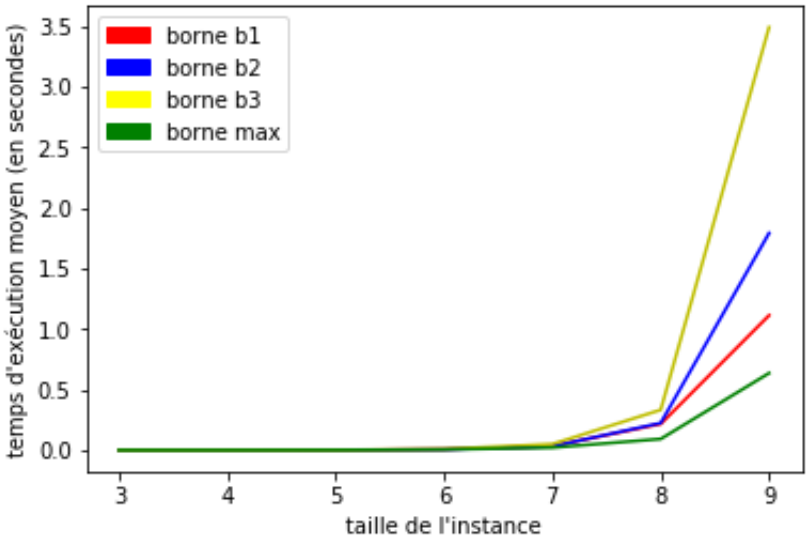

FIGURE 7 – Qualité des bornes pour TC

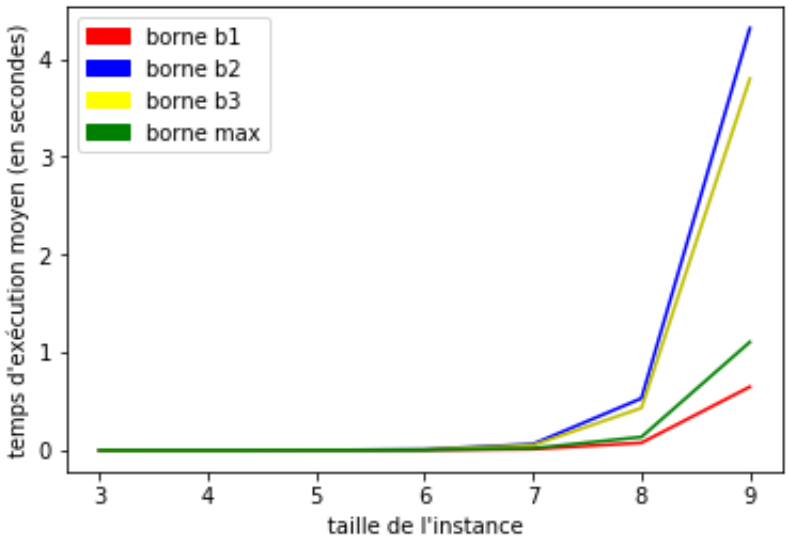

FIGURE 8 – Qualité des bornes pour MC

Nous pouvons remarquer que les bornes b_2 et b_3 donnent globalement d'assez mauvais résultats bien qu'elles soient de meilleures bornes en général, la raison la plus probable étant que leur coût de calcul dépasse le temps gagné en adoptant leurs meilleures valeurs. Les deux bornes b_1 et max produisent généralement des résultats assez similaires, avec b_1 légèrement meilleure que max, sauf pour TC. En effet, pour la classe TC, les durées sur chaque machines varient peu, il est facile d'éliminer beaucoup de chemins, avoir de très bonnes borne est donc un avantage. Dans la suite nous choisirons la borne max étant donné sa relative stabilité sur les différentes classes et taille d'iinstances.

4.2 Performances des algorithmes exacts

Le but de cette sections est d'évaluer les deux algorithmes, *eval* et *eval*2, afin de prouver que notre algorithme *eval*2 est bien plus rapide en général que l'algorithme *eval*. Similairement aux tests pour les bornes, dans un soucis de garder des temps d'exécution corrects, chaque algorithme a été exécuté 100 fois pour les tailles de 3 à 8, puis 10 fois pour la taille 9 et 5 fois pour la taille 10. Ci-dessous les résultats :

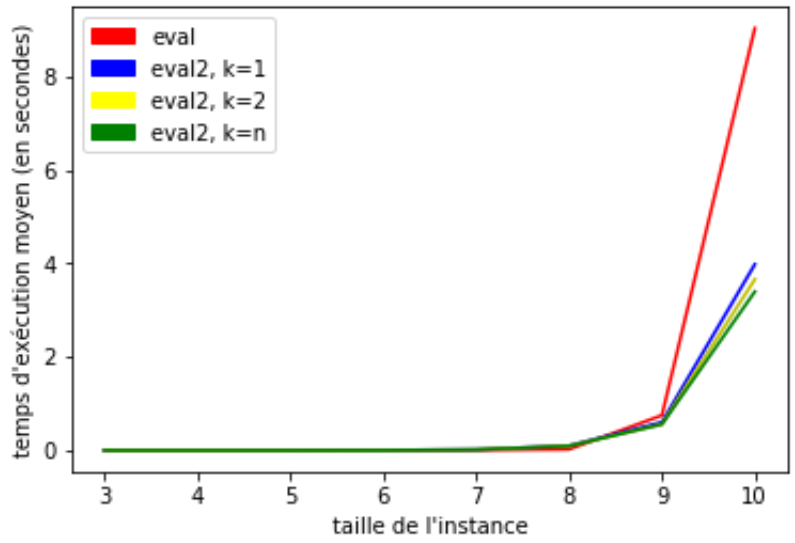

FIGURE 9 – Performances des algorithmes exacts pour NC

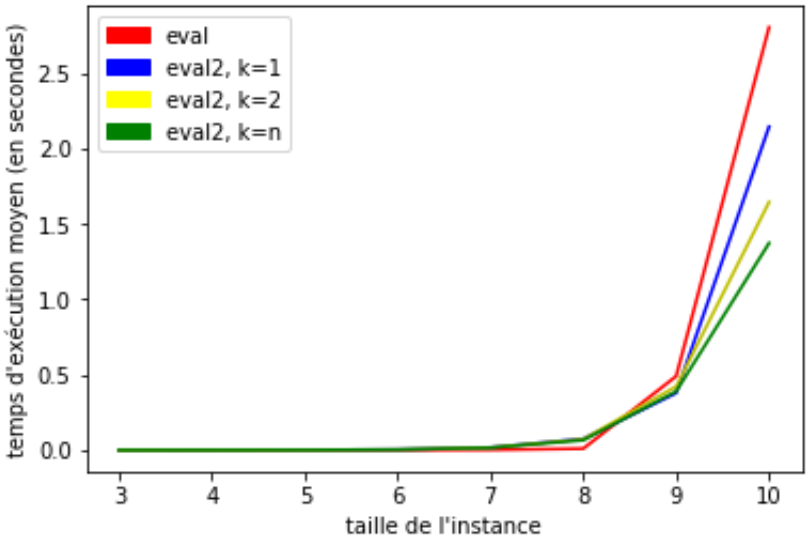

FIGURE 10 – Performances des algorithmes exacts pour TC

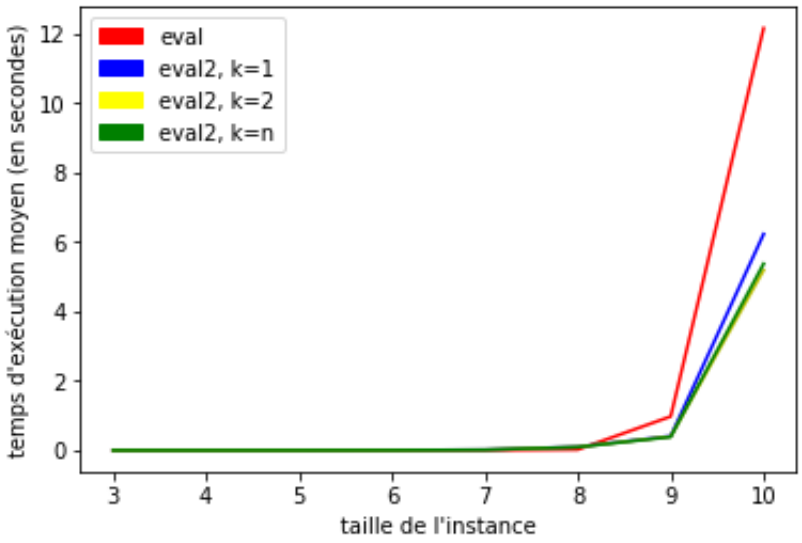

FIGURE 11 – Performances des algorithmes exacts pour MC

Il est intéressant de remarquer que les deux algorithmes "explosent" tous deux pour une taille d'instance supérieure à 9. Lors des tests, *eval* n'a pas pu résoudre d'instances de taille supérieure à 10 dans des temps raisonnables (en dessous de 10 minutes), et *eval*2 a été capable de résoudre relativement rapidement certaines instances de taille 12. Ces résultats montrent non seulement que notre algorithme optimisé est bien plus rapide en général que l'algorithme d'origine, mais il nous donne également quelques indices utiles sur la valeur de k à utiliser. Nous pouvons voir que dans le cas général, utiliser $k = nombreTaches$ est la meilleure solution. Nous avons pu vérifier cette assertion en testant avec $k \in [\![1, nombreTaches]\!]$ pour un nombre de tâches allant jusqu'à 10. Le coût du pré-traitement est donc compensé par l'avantage de prendre le chemin ayant les meilleurs heuristiques.

4.3 Évaluation des algorithmes approchés

Cette section présente les résultats obtenus pour les algorithmes approchés, aussi bien leurs performances que la qualité des solutions obtenues. Le calcul des performances utilise plus d'itération que pour les expérimentations précédentes étant donné que ces algorithmes sont en général plus rapides, allant jusqu'à 300, mais conserve le même schéma présenté ci-dessus pour le

calcul de la qualité de la solution, un algorithme exact devant être exécuté.

4.3.1 Performances

Nous allons tout d'abord étudier les performances des algorithmes, les graphes ci-dessous représentent les résultats :

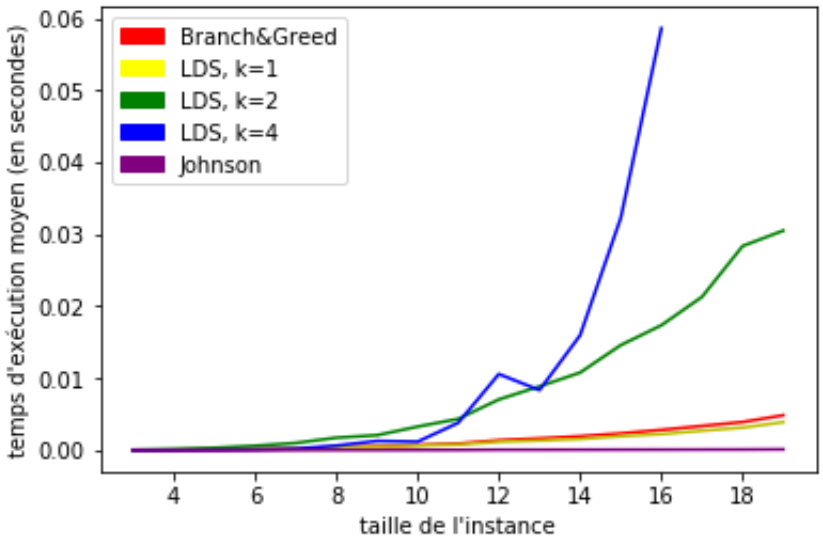

FIGURE 12 – Performances des algorithmes approchés pour NC (avec LDS, k = 4)

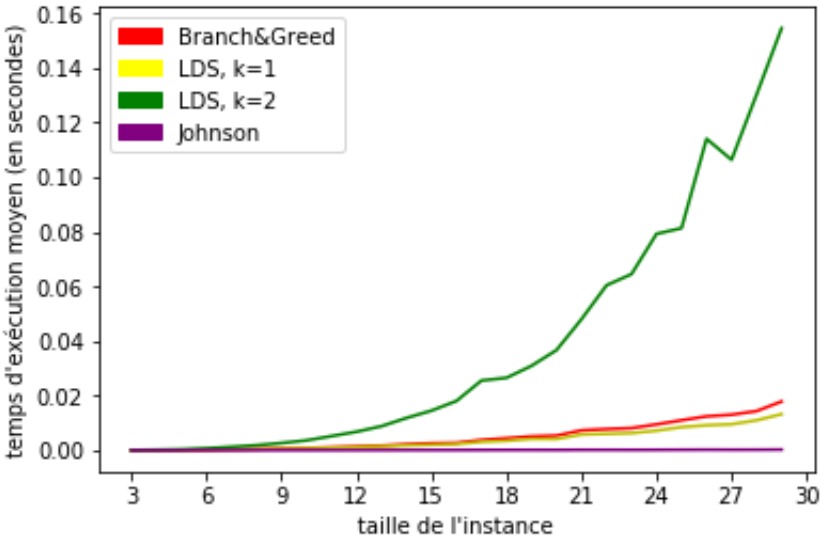

FIGURE 13 – Performances des algorithmes approchés pour NC (sans LDS, k = 4)

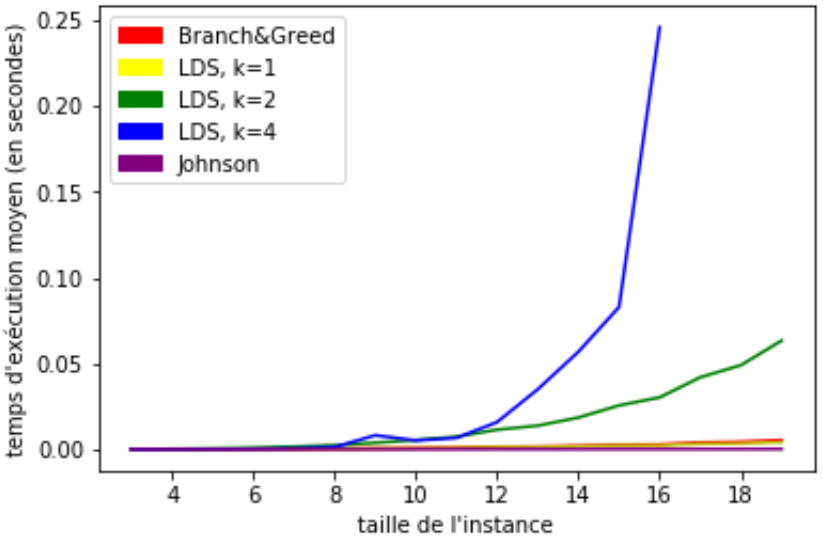

FIGURE 14 – Performances des algorithmes approchés pour TC (avec LDS, k = 4)

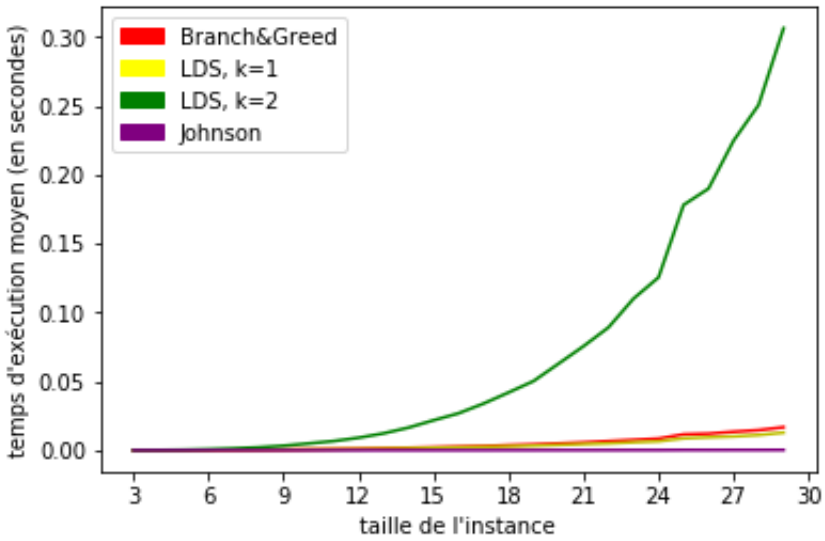

FIGURE 15 – Performances des algorithmes approchés pour TC (sans LDS, k = 4)

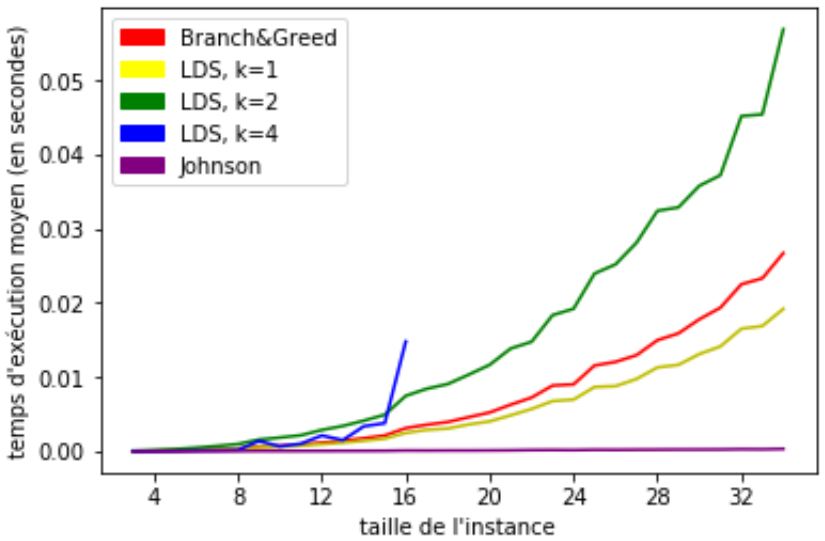

FIGURE 16 – Performances des algorithmes approchés pour MC (avec LDS, k = 4)

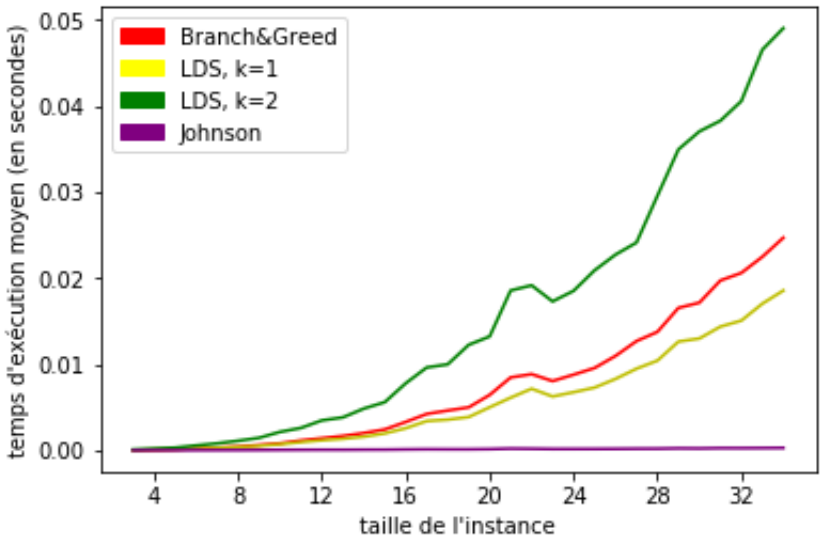

FIGURE 17 – Performances des algorithmes approchés pour MC (sans LDS, k = 4)

Nous pouvons tout d'abord remarquer de Johnson est incontestablement l'algorithme le plus performant, ce qui n'est pas étonnant étant donné la simplicité du code et le fait qu'il repose sur un simple algorithme polynomial. Les résultats s'avèrent ensuite assez cohérents avec les attentes initiales :

- Le *Branch&Greed* reste le plus performant après Johnson.
- Le *LDS* devient de moins en moins performant plus k devient grand.

Il est cependant étonnant de découvrir que le *LDS* est globalement plus rapide que le *Branch&Greed* pour $k = 1$. De plus, comme nous allons le voir dans la prochaine section, il donne également de bien meilleurs résultats.

4.3.2 Qualité de la solution

Dans cette section, nous allons nous concentrer sur la qualité des solutions produites par les différents algorithmes approchés implémentés. Les différents graphes ci-dessous représentent les erreurs moyennes des solutions produites par les algorithmes relativement à la solution optimal. Dans nos tests, la solution optimale est générée par $eval2$ avec $k = nombreTaches$, ce qui nous a permis de tester jusqu'à des instances de taille 11 :

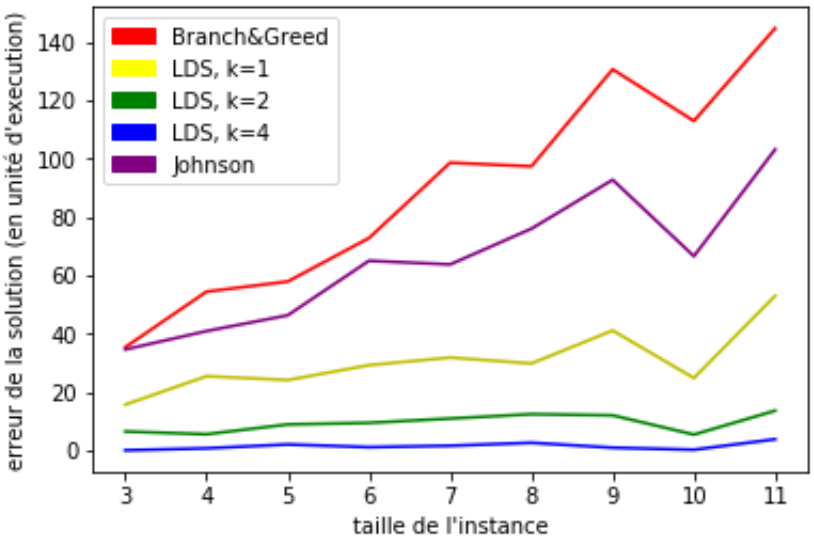

FIGURE 18 – Qualité des algorithmes approchés pour NC

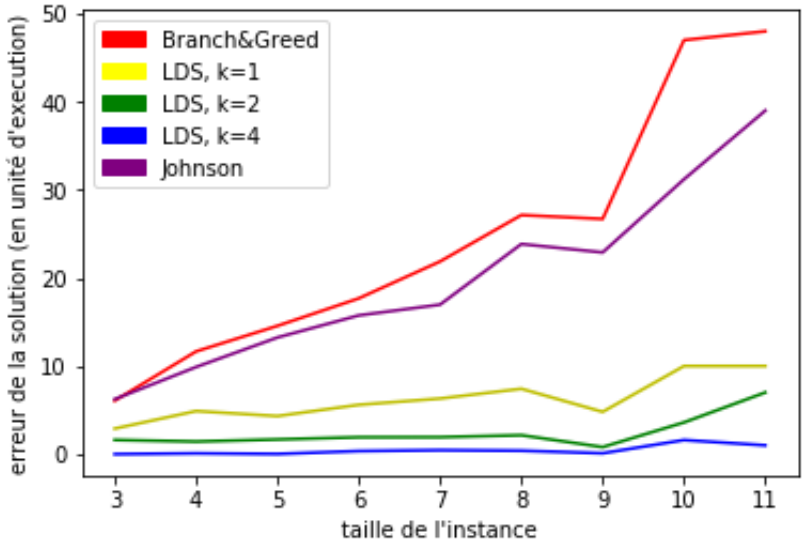

FIGURE 19 – Qualité des algorithmes approchés pour TC

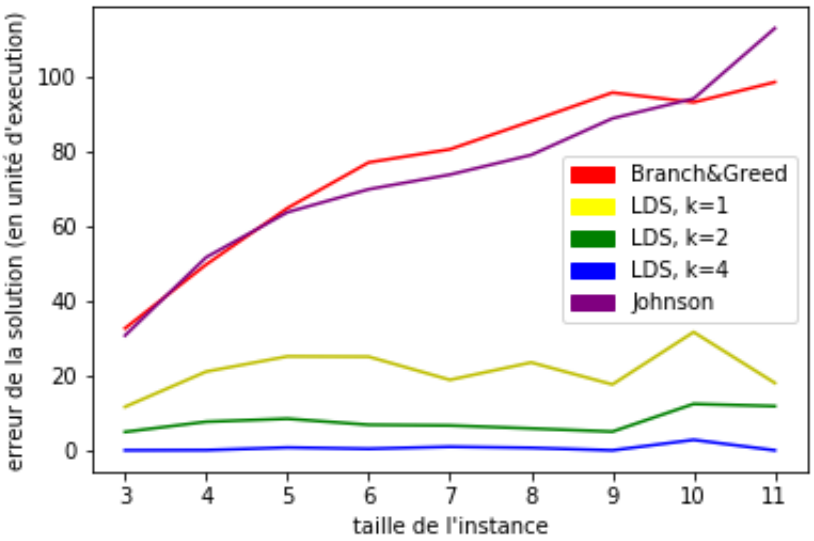

FIGURE 20 – Qualité des algorithmes approchés pour MC

Il est intéressant de remarquer que sur TC, les algorithmes approchés donnent en général de bonnes solutions par rapport aux instances de NC et MC. Cela est dû au fait que les tâches varient assez peu en temps d'exécution, il est donc beaucoup plus facile de trouver une solution non optimale qui se rapproche assez de l'optimale. Pour revenir sur la remarque faite à la fin de la section précédente, nous pouvons voir que l'algorithme LDS avec $k = 1$ donne de bien meilleurs résultats que $Branch\&Greed$. Plus surprenant ceci dit, le fait que Johnson donne globalement de meilleurs résultats que le $Branch\&Greed$, ce qui peut s'expliquer par le fait que Johnson exploite la structure du problème, alors que le $Branch\&Greed$ est un algorithme générique et à priori non polynomial. Finalement, utiliser LDS avec $k = 1$ est un bon compromis pour avoir une approximation relativement bonne sur des instances jusqu'à 30, au delà Johnson est incontestablement le meilleur, bien qu'il ne permette d'obtenir qu'une approximation assez grossière.

5 Conclusion

Au cours de ce projet nous avons pu explorer différentes méthodes arborescentes que nous avons testé afin d'en comprendre les qualités et les défauts. Nous avons montré expérimentalement que les méthodes exactes, bien qu'extrêmement rapides lorsqu'on les compare aux méthodes gloutonnes naïves, restent incapables de résoudre des problèmes pour de grandes instances. Les méthodes arborescentes approchées, telles que l'algorithme LDS, sont également un bon moyen de résoudre des problèmes pour des instances de taille moyenne. Dans le cas échéant, s'il existe un algorithme polynomial approché à garantie de performance pour le problème donné, il est presque sûr qu'il sera le seul à pouvoir donner une solution pour de très grandes instance du problème, à défaut de donner une bonne solution.

Références

[1] Christelle Gueret Agnès Le Roux, Odile Bellenguez-Morineau. Limited discrepancy search pour un problème d'ordonnancement de rendez-vous, 2014.

[2] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[3] Saffia Kedad-Sidhoum. Méthodes arborescentes exactes et approchées, 2014.

[4] Daniel Wilson. Important algorithms, 2010.