31/10/2015

COMPLEX : Projet ordonnancement

Guillaume Hivert && Darius Mercadier

Partie 1 – Algorithme approché avec garantie de performance

Question 1

Indépendamment de l'ordonnancement *P*, On a :

$$\sum d_A^i \leq OPT, \quad \sum d_B^i \leq OPT, \quad \sum d_C^i \leq OPT$$

Donc :

$$\sum (d_A^i + d_B^i + d_C^i) = \sum d_A^i + \sum d_B^i + \sum d_C^i \leq OPT + OPT + OPT = 3 \times OPT$$

D'où :

$$OPT \geq \sum \frac{(d_A^i + d_B^i + d_C^i)}{3}$$

L'algorithme *P* qui renvoi une solution aléatoire renvoie donc une solution qui dans le pire des cas est égale à 3 fois la solution optimale.
Par définition, l'algorithme *P* est donc 3-approché.

Question 2

Johnson retourne une solution optimale OPT2 pour 2 machines. Donc en rajoutant une machine, la solution optimale OPT est supérieure à OPT2.
Donc on a : $Johnson \leq OPT \; et \; \sum d_C^i \leq OPT$. La complexité de notre algorithme est donc inférieure ou égale à $Johnson + \sum d_C^i$, $et \; Johnson + \sum d_C^i \leq OPT + OPT$.
La solution retournée est donc inférieure à $2 \times OPT$, donc par définition, l'algorithme est 2-approché.

<u>Complexité de Johnson :</u>

– *n* itérations.

A chaque itération, on parcourt toutes les taches restantes, donc *n* pour la première itération, *n - 1* à la seconde, etc.

Donc au total, on fait :

$$n + (n-1) + \cdots + 2 + 1 = \frac{n \times (n-1)}{2} = O(n^2)$$

Notre algorithme se contente de renvoyer la solution retournée par Johnson, sa complexité est donc celle de Johnson : $O(n^2)$.

Une paragraphe est consacré aux performance de l'algorithme de Johnson après la question 10.

Partie 2 – méthode exacte

Question 4

L'idée de ce remplacement est que l'on veut tenir compte du temps pendant lequel la machine B n'est pas active car elle a finis de traiter les taches π, et est en attente de la tâche qui s'exécute actuellement sur A.

C'est le cas lorsque qu'on a $t_B^\pi \leq t_A^\pi + min_{i \in \pi'} d_A^i$: B a fini d'exécuter toutes les taches de π, et est donc inactive au moins jusqu'à ce que A ait fini toutes les taches de π (t_A^π), puis la plus courte de π' ($min_{i \in \pi'} d_A^i$).

On peut donc remplacer t_B^π par $t'^{\pi}_B = \max\{t_B^\pi, t_A^\pi + min_{i \in \pi'} d_A^i\}$.

En suivant le même raisonnement, on remarque que : $t_C^\pi < t_B^\pi + min_{i \in \pi'} d_B^i$ => C est libre, et doit attendre la fin des taches de π sur B puis au moins la tâche la plus courte de π' sur B.

$t_C^\pi < t_A^\pi + min_{i \in \pi'}\{d_A^i + d_B^i\}$ => C est libre, mais doit attendre que la fin des taches de π sur A, puis la fin de la plus courte de π' sur A et B.

On peut donc remplacer t_C^π par $t'^{\pi}_C = max\left\{t_C^\pi, t_B^\pi + min_{i \in \pi'} d_B^i, t_A^\pi + min_{i \in \pi'}\{d_A^i + d_B^i\}\right\}$.

Question 5

t_A^π est le temps à laquelle la machine A a fini d'exécuter les taches de π.

$\sum_{i \in \pi', i \neq k, d_A^i \leq d_C^i} d_A^i$ est la manière la plus rapide pour arriver au début de la tache k.

$d_A^k + d_B^k + d_C^k$ est le temps que met la tache A à s'exécuter sur chacune des machines A, B puis C.

$\sum_{i \in \pi', i \neq k, d_A^i > d_C^i} d_C^i$ est la manière la plus rapide pour finir l'ordonnancement après la tache k.

On peut donc prendre comme seconde borne inférieur :

$$b_2 = max_{k \in \pi'}\left\{t_A^\pi + \left(d_A^k + d_B^k + d_C^k\right) + \sum_{i \in \pi', i \neq k, d_A^i \leq d_C^i} d_A^i + \sum_{i \in \pi', i \neq k, d_A^i > d_C^i} d_C^i\right\}$$

Question 6

En suivant le même raisonnement qu'à la question précédente, on peut prendre comme borne b_3 :

$$b_3 = max_{k \in \pi'}\left\{t_B^\pi + \left(d_B^k + d_C^k\right) + \sum_{i \in \pi', i \neq k} min\{d_B^i, d_C^i\}\right\}$$

(En ayant réunis les 2 sommes en une avec un *min*)

Question 7

Parlons tout d'abord de la borne supérieur : nous avons pris comme borne supérieure la valeur retournée par l'algorithme de Johnson développé en partie 1. Celle-ci est égale au plus à 2 OPT.

Nous la calculons une seule fois avant de lancer le Branch and Bound puis l'utilisons comme majoration pour le branchement.

Les tests qui suivent permettent de comparer l'efficacité des différentes bornes en termes de nombre de nœuds visités, ainsi que les différents algorithmes :

Pour l'algorithme de Branch and Bound classique, nous avons testé comme borne inférieur :

– $b_{inf} = b_1$
– $b_{inf} = max\{b_1, b_2, b_3\}$

De plus, nous avons testé au passage un algorithme greedy, c'est-à-dire qu'il visite en premier les nœuds dont la borne inférieure (pour cet algorithme également nous avons testé b_1 et $min\{b_1, b_2, b_3\}$) est la plus faible. Il vient en réponse à la question 8, mais nous le présentons ici pour le comparer à l'algorithme classique en même temps que nous comparons les bornes inférieures de ce dernier.

Remarque

Le nombre de nœuds de l'arbre exhaustif des ordonnancements possible croit très vite lorsque le nombre de taches n augmente (ce nombre est supérieur à $n!$).

La méthode Branch & Bound n'est qu'une construction et un parcours astucieux de cet arbre, qui permet de ne pas explorer certaines branches grâce à un système de minoration (borne inférieure) des solutions que cette branche peut produire.

Sur certaines instances, les temps d'exécution des taches seront tels que soit les bornes inférieures que l'on calcule ne sont pas assez précise, ce qui entraine le parcours de plus de branches de l'arbre, ou bien tout simplement que l'ont n'ai pas de « chance » sur l'ordre de parcours.

Ces deux cas de figure ont pour conséquence que notre algorithme de B & B explorera un très grand nombre de nœuds avant de renvoyer la solution optimale.

Le principe des tests qui suivent est le suivant : pour chaque nombre de taches $n \in [5, 15]$, on a généré 40 instances de chaque classe, et testé chacun des algorithmes dessus.

Pour que ces tests se terminent en un temps raisonnable, nous avons limité le nombre de nœuds maximum que chaque algorithme peut explorer à 200 000.

Nous présenterons les résultats de ces tests dans les pages suivantes. Nous présentons 2 types de résultats :

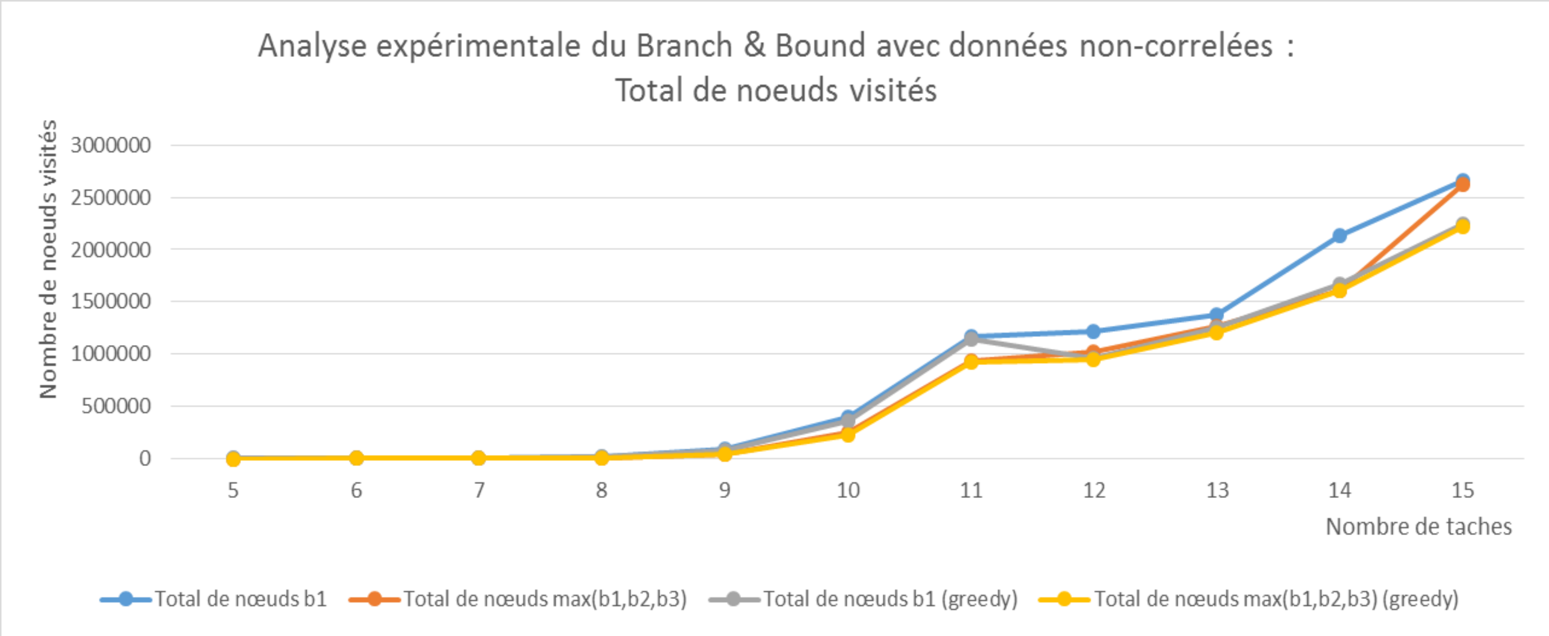
- Le nombre total de nœuds visités lors des 40 tests
- Le temps total d'exécution des 40 tests.

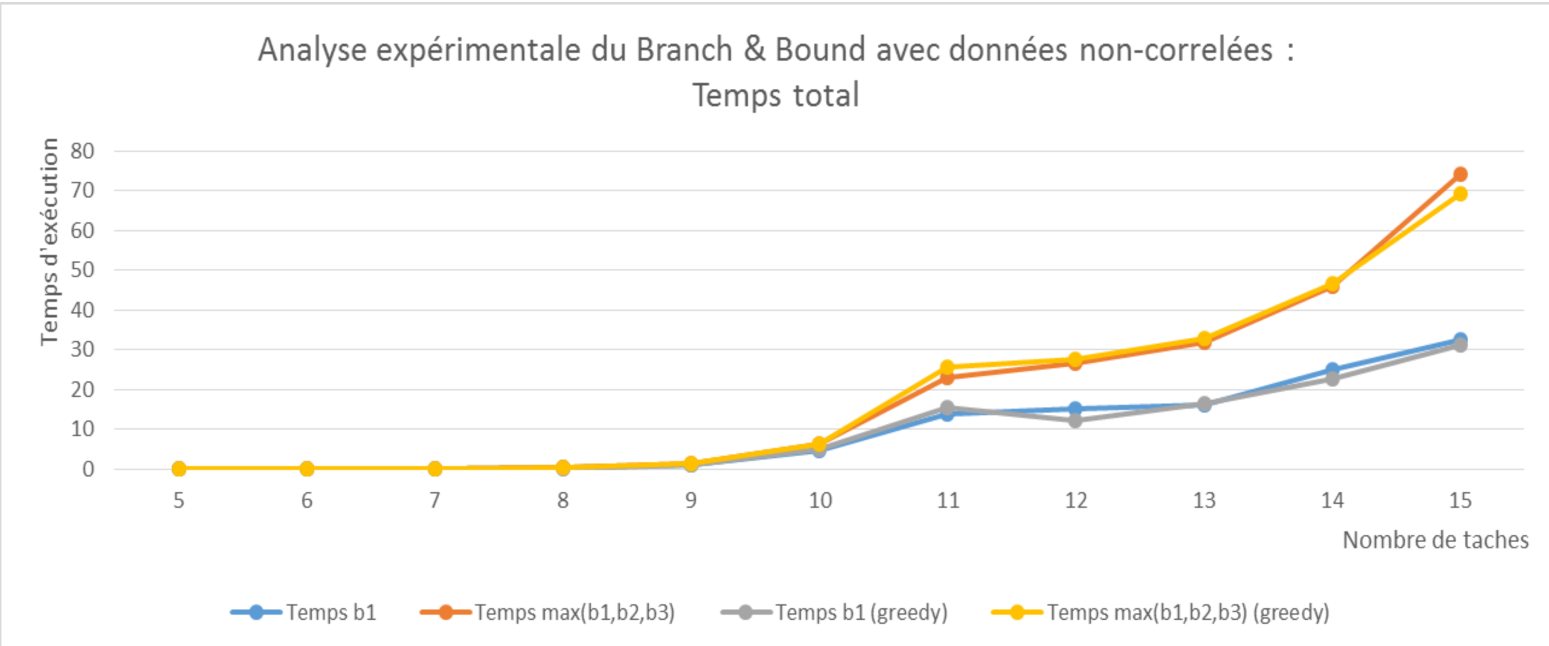
Il faut donc noter que ces valeurs peuvent être faussées lorsque des instances sur lesquelles un ou plusieurs algorithmes ont été interrompus à 200.000 nœuds visités. Mais nous donneront des précisions sur ce point lors de l'interprétation des courbes.

Les données ayant servis à tracer les graphiques sont disponibles à la fin du rapport.

Par ailleurs, nous reviendront à la question 10 sur les instances sur lesquelles les algorithmes présentés ci-dessus terminent en un temps très élevé.

Données non-corrélées





Interprétation des résultats expérimentaux

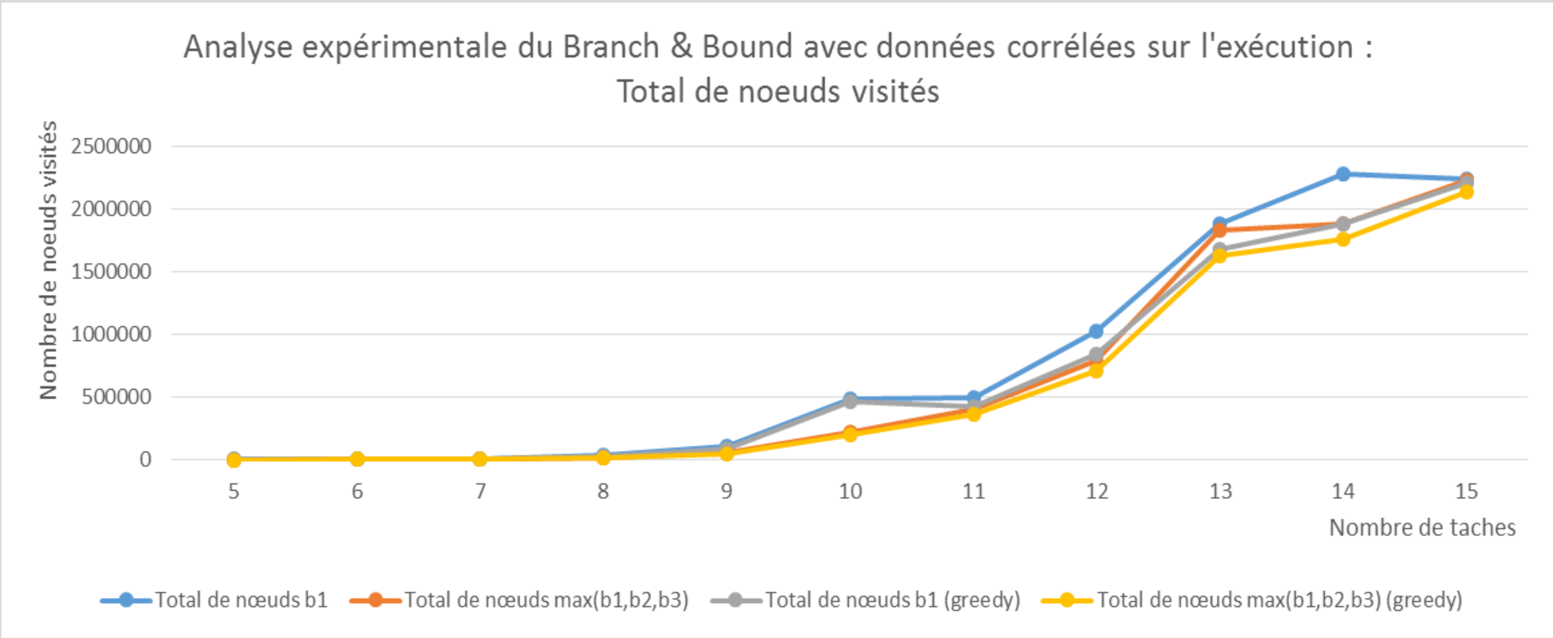
On constate que b_1 engendre une quantité de nœuds visités bien supérieur à $max\{b_1, b_2, b_3\}$. Cependant, le temps de calcul de $max\{b_1, b_2, b_3\}$ est plus bien important que celui de b_1. Donc en regardant le temps d'exécution et non le nombre de nœuds visités, on constate que la borne b_1 est la plus rentable des deux.

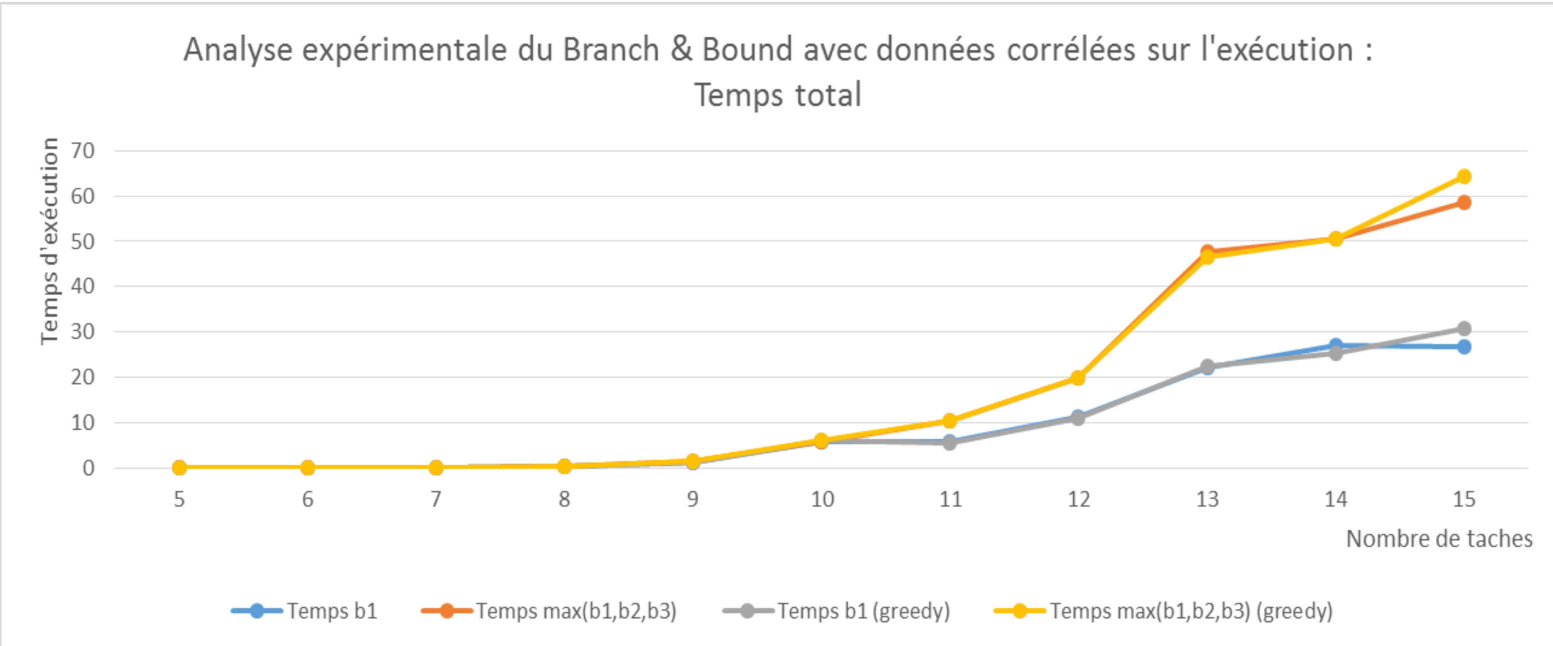
A noter qu'à partir de 12 taches, il commence à y avoir quelques d'instance nécessitant un temps de traitement long (plus de 200.000 nœuds), et de plus en plus lorsque le nombre de taches augmente.

Nous avons limité le nombre de taches à 15 au maximum, plus pas nécessité que par choix : en effet, à partir, et au-delà de 15 taches, il y a de plus en plus d'instances sur lesquelles notre algorithme demande énormément de temps pour se terminer (minutes, heures, voir plus).

On note au passage que l'algorithme greedy est très souvent plus intéressant que l'algorithme normal, mais nous reviendront sur ce point.

Données corrélées sur les durées d'exécution



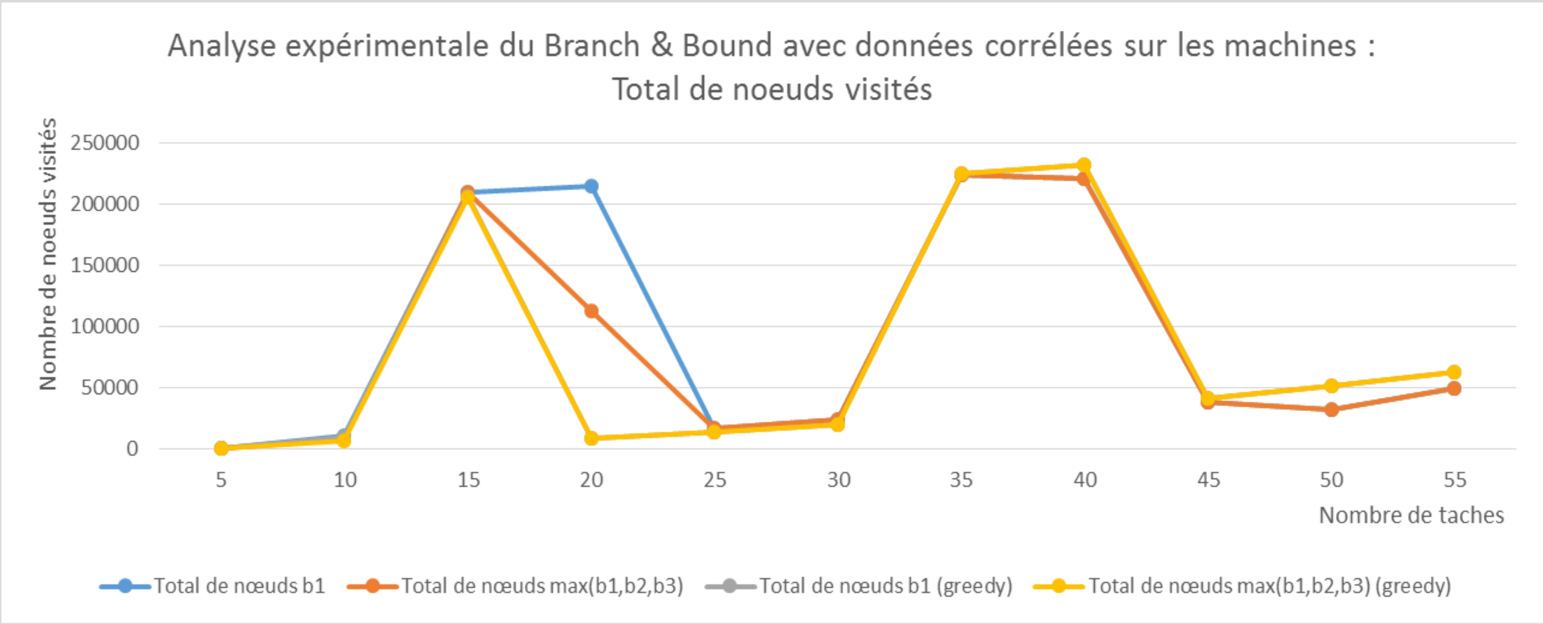


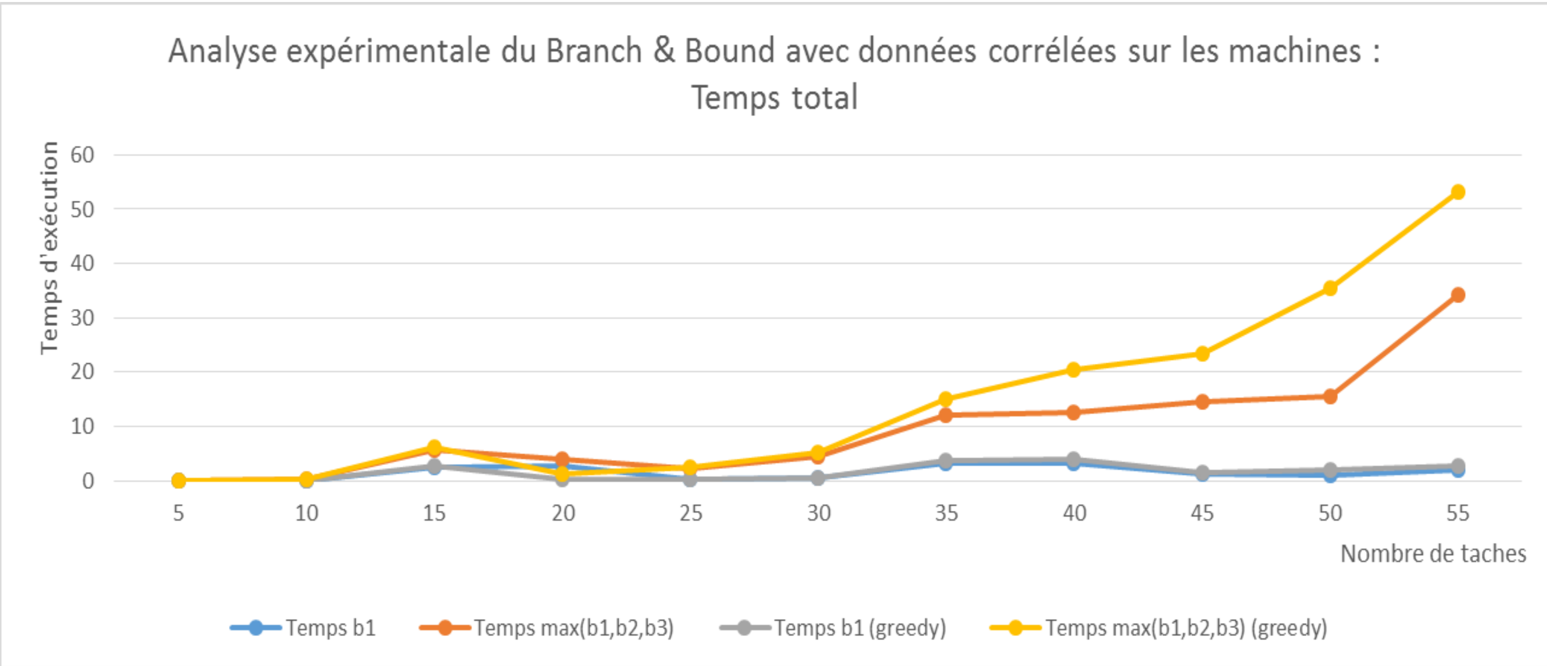
Interprétation des résultats expérimentaux

L'efficacité de la borne b_1 par rapport à $max\{b_1, b_2, b_3\}$ est du même ordre que sur les instances de données non-corrélées : plus de nœuds visités, mais un temps de calcul plus faible.

A noter également qu'avec ce type d'instance, il est beaucoup plus rare (que sur les données non-corrélées) que nos Branch & Bound aient besoin d'explorer un très grand nombre de nœuds avant de retourner une solution optimale. Nous avons cependant une fois de plus limité de taches à 15 car bien que plus rares, ces cas de figures ralentissant les performances se produisent régulièrement.

Données corrélées sur les machine





Interprétation des résultats expérimentaux

Notre algorithme de Branch and Bound se comporte très différemment sur cette classe d'instance par rapport aux classes testées précédemment :

-– Tout d'abord, on a presque toujours $b_1 = max\{b_1, b_2, b_3\}$. Or b_1 est bien plus rapide à calculer que b_2 et b_3 , surtout sur des grandes instances.

C'est d'ailleurs l'occasion de constater un peu plus la complexité de $max\{b_1, b_2, b_3\}$ (O(n²)). On voit bien que lorsque la taille de l'instance augmente, le temps de calcul pour $max\{b_1, b_2, b_3\}$ devient très largement supérieur au temps de calcul de b_1.
Cela explique que la courbe du nombre de nœuds visités est très différente de la courbe du temps : la première ne fait pas apparaitre n, tandis que les secondes sont proportionnelles à n ou à son carré.

Nombre de taches	$b_{inf} = b_1$	Greedy
5	16	16
10	60	61
20	135	211
30	621	466
40	645	821
50	810	1 276
60	970	1 831
70	1 050	2 486
80	1 322	3 241
90	1 310	4 096
100	1 342	5 051
200	2 000*	20 101*
300	4 520*	45 151*
400	3 902*	80 201*
500	22 779*	125 521*

– Le temps d'exécution est (expérimentalement) en O(n), et ce quel que soit la taille de l'instance, cela nous a donc permis de pouvoir tester des instances d'une taille allant jusqu'à 500 taches, comme le montre le tableau à gauche. Nous avons omis les temps d'exécution de ce tableau, mais sachez qu'ils sont d'environ 2min pour 500 taches.

– L'algorithme Greedy est souvent moins efficace. De plus, pour toutes les instances de taille n fixé, il explore autant de nœuds.

– Il n'est pas arrivé un seul fois que nos test soient interrompus en raison d'un nombre de nœuds visités trop élevés.

Nombre de nœuds visités en fonction du nombre de taches composant les instances (corrélation sur les machines).

** pour ces instances, nous avons effectué uniquement 1 test par nombre de taches en raison du temps de traitement relativement élevé.*

Conclusion (rapide) des résultats expérimentaux

On a donc pu constater que la borne b_1, bien que moins précise que $max\{b_1, b_2, b_3\}$, est bien plus rapide à calculer, ce qui la rend généralement plus intéressante à utiliser.

De plus, on constate une nette différence de performance de nos méthodes arborescentes selon les types d'instance testée : il est bien plus optimale est rapide sur des données corrélées que sur d'autres classes d'instance.

Question 8

Comme expliqué dans la question 7, nous avons essayé d'optimiser notre méthode arborescente avec une méthode de branchement Greedy : à un niveau donné, on branche sur chaque nœud dans l'ordre croissant de la valeur de leur borne inférieure.

Bien que ce ne soit pas très visible sur les graphiques montré lors de la question 7, cette solution s'est avérée presque toujours bien plus optimale que le Branch and Bound basique sur des instances constituées de données non-corrélées ou de données corrélées sur les durées d'exécution. Il arrive cependant que sur des instances de ces classes, le branchement Greedy fasse visiter plus de nœuds que le branchement naïf, mais cela est rare.

En revanche, sur les instances constituées de données corrélées sur les machines, cette solution n'est généralement pas rentable.

Curieusement, sur cette classe d'instance, pour n fixé, quel que soit l'instance de taille n, le Branch & bound Greedy visite autant de nœuds.

Question 9

Dans le cadre de cette méthode arborescente, nous n'avons que trois machines en série, toutefois, il est tout à fait possible d'élargir cette méthode à un nombre k de machines. En effet, pour chaque machine m, il est possible de calculer une borne inférieure similaire à celle aux bornes b_1 b_2 ou b_3 en se basant par exemple sur $t'^{\pi}_{m} = max\{t^{\pi}_{m}, t^{\pi}_{n} + min_{i \in \pi}, T^{i}_{o}\}$, ou *n* représente l'ensemble des autres machines précédents *m*, T^{i}_{o} l'ensemble des d^{i}_{o} avec *o* représentant l'ensemble des machines précédents *n*. Il est donc possible de trouver une borne dans tous les cas, et d'appliquer une méthode arborescente.

Question 10

Une méthode arborescente exacte peut être extrêmement performante, ou bien désastreuse selon les situations. L'algorithme Branch and Bound peut parfois donner sur des instances une réponse extrêmement rapidement (de l'ordre de quelques secondes) comme une réponse extrêmement lente (de l'ordre de plusieurs minutes ou heures selon la taille de l'instance). Or nous ne savons pas avant d'exécuter l'algorithme quel va être son comportement sur l'instance testée.

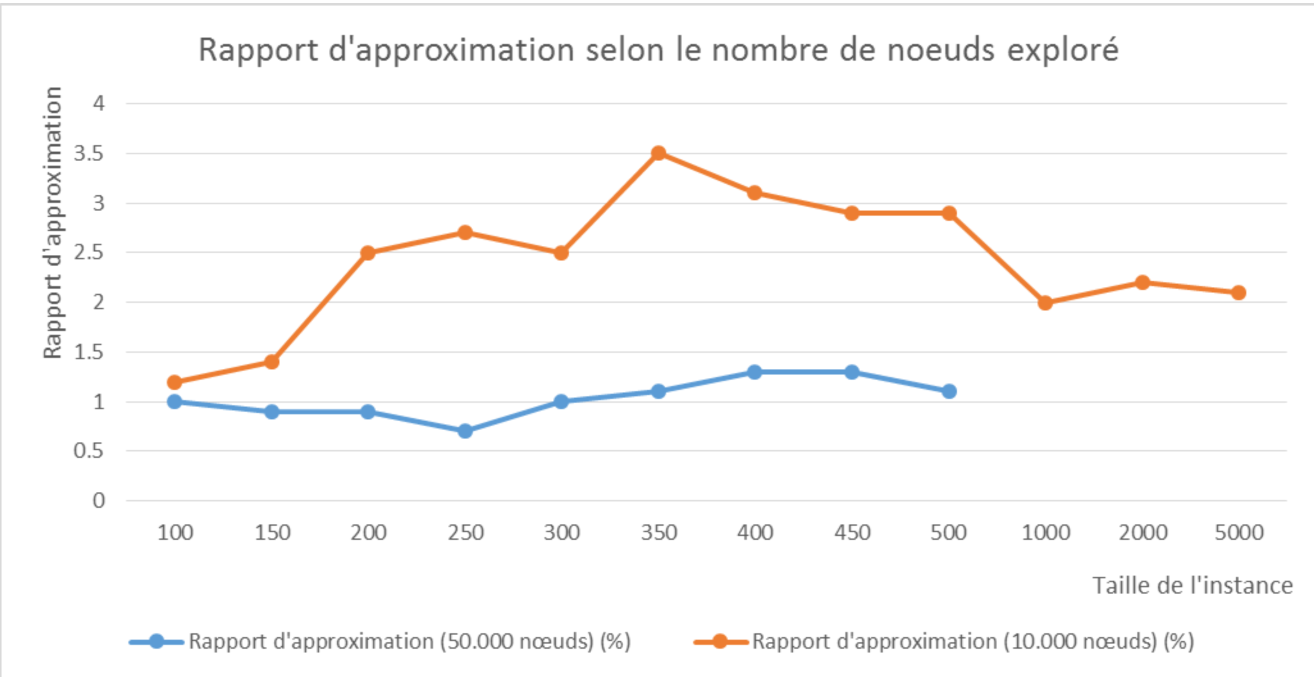
Or il se trouve qu'arrivé à un certain stade de l'algorithme, la réponse approché est suffisamment efficace et intéressante pour se passer des étapes de calculs supplémentaires (et donc diminuer le temps de calcul). Pour cela, nous avons décidé de mettre en place un calcul du rapport d'approximation de l'algorithme selon la formule $gap = \frac{z_{best} - z_{lb}}{z_{lb}}$, qui permet de déterminer l'écart entre la solution optimale théorique z_{lb} (la meilleure borne inférieure de la valeur optimale) et la meilleure solution trouvée par l'algorithme z_{best}.

En calculant ce rapport $\frac{z_{best} - z_{lb}}{z_{lb}}$ à chaque itération, on peut alors arrêter l'algorithme :

- soit une fois que le gap est atteint,
- soit une fois qu'un certain nombre de nœud a été visité, auquel cas il faut retourner la valeur du rapport en plus de la solution pour pouvoir évaluer sa validité.

En suivant cette méthode, on obtient alors un algorithme approché permettant de donner une réponse presque exacte (quasiment constamment de l'ordre inférieur à 5 % d'écart), tout en conservant un temps de calcul extrêmement rapide, généralement de l'ordre d'une à deux minutes pour les instances les moins favorables.

Afin d'illustrer l'efficacité de cette méthode approchée, nous avons lancé l'algorithme sur des instances de grande taille (de données corrélées sur les temps d'exécution), et mesuré les performances que l'on obtenait après 50.000, puis 10.000 nœuds explorés.



Précisons que les tests ont été effectués sur peu d'instances différentes en raison du temps qu'ils peuvent prendre, ce qui explique une certaine absence de pattern sur le graphique.

On constate qu'en moyenne ce rapport d'approximation est en moyenne compris entre 1 et 3.5%, ce qui est une bonne valeur.

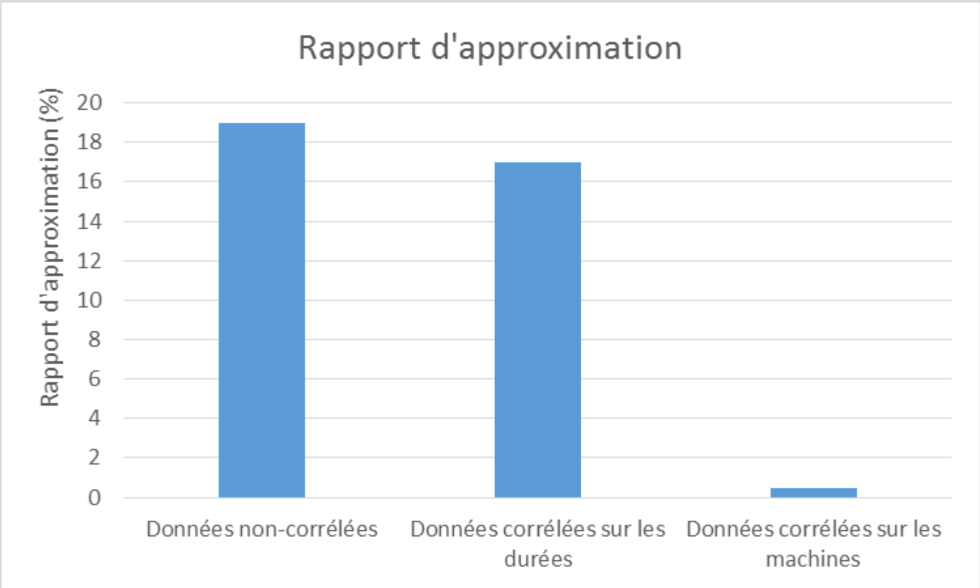
Retour sur l'algorithme de Johnson

Après avoir examiné la complexité et l'efficacité des algorithmes utilisant des méthodes arborescentes, exactes et approximé, faisons un petit retour sur l'algorithme de Johnson.

Pour commencer, sa complexité expérimentale est en $O(n^2)$, comme attendu. (L'algorithme étant relativement simple, il ne s'agit pas d'une surprise).

Pour avoir une référence par rapport aux méthodes arborescentes présentées précédemment : l'algorithme de Johnson dure environ 0.10 secondes sur une instance de taille 1.000, et 8 secondes sur une instance de taille 10.000.

Analysons plutôt la pertinence des solutions trouvées. Le graphique ci-dessous illustre la performance de l'algorithme.



Ces rapports d'approximations ont été mesurés à partir d'instances de taille 1.000, 2.000 et 5.000. Les valeurs qui apparaissent sont des moyennes, mais presque l'intégralité des solutions trouvées est dans un intervalle de +- 20% de ces valeurs (soit entre 24% et 16% sur les données non corrélées, entre 21% et 14% sur les données corrélées sur les durées, etc.).

L'algorithme de Johnson est donc généralement bien plus performant que ce que nous avions pu anticiper en constatant qu'il est 2-approché.

Il reste cependant bien moins précis (instances avec corrélation sur les machines à part) que la méthode approchée arborescente bien qu'il soit plus rapide à exécuter.

Conclusion

Nous avons effectué au cours de ce projet des tests de performances sur différents algorithmes, avec différentes méthodes de calcul d’optimale.

Les classes d'instances sur lesquelles s'exécutent nos algorithmes influent beaucoup sur leur performance : ils sont tous plus performants sur des données avec corrélations des temps sur les machines.

De plus, nous avons constaté que la qualité d'une borne ne se mesure pas seulement à sa précision, mais également à son temps de calcul : il a été préférable d'utiliser une borne inférieure qui n'était pas la plus élevée possible, mais qui se calculait rapidement.

Nous avons également constaté les limites de certaines méthodes arborescentes, qui peuvent, dans le pire des cas, nécessiter un temps de l'ordre de O(n!), ce qui n'est pas gérable dès que n dépasse 8-10.

En revanche, une approximation de ces méthodes arborescentes nous a permis d'obtenir des résultats très bons dans un temps assez court.

Annexes

Données non-correlées								
Nombre de taches	Total de nœuds b1	Total de nœuds max(b1,b2,b3)	Total de nœuds b1 (greedy)	Total de nœuds max(b1,b2,b3) (greedy)	Temps b1	Temps max(b1,b2,b3)	Temps b1 (greedy)	Temps max(b1,b2,b3) (greedy)
5	1255	936	1116	815	0.017	0.026	0.017	0.026
6	3713	2243	2987	1604	0.042	0.054	0.039	0.046
7	7494	4028	6582	3328	0.088	0.103	0.086	0.101
8	19687	10401	15595	7488	0.23	0.277	0.206	0.229
9	89799	49864	86022	47327	1.035	1.253	1.118	1.318
10	403079	253677	364088	224517	4.685	6.44	4.808	6.302
11	1173977	941926	1139127	917951	13.807	23.07	15.575	25.633
12	1212639	1017055	961688	943810	15.144	26.71	12.242	27.734
13	1377945	1268882	1258621	1211199	16.2	32.036	16.612	32.919
14	2131266	1608421	1673122	1606849	25.099	45.994	22.533	46.501
15	2661509	2627163	2248943	2225020	32.369	74.325	31.153	69.429

Corrélation sur les temps d'exécution								
Nombre de taches	Total de nœuds b1	Total de nœuds max(b1,b2,b3)	Total de nœuds b1 (greedy)	Total de nœuds max(b1,b2,b3) (greedy)	Temps b1	Temps max(b1,b2,b3)	Temps b1 (greedy)	Temps max(b1,b2,b3) (greedy)
5	1341	868	1249	720	0.015	0.019	0.016	0.019
6	2622	1636	2361	1335	0.029	0.039	0.031	0.038
7	4417	3435	3020	2418	0.052	0.087	0.041	0.073
8	33482	15358	30638	13100	0.38	0.394	0.393	0.378
9	110871	58735	92035	49865	1.282	1.449	1.195	1.338
10	484656	215843	463541	202967	5.735	5.847	6.175	6.044
11	496521	405545	422533	362943	5.776	10.418	5.548	10.27
12	1019731	794213	838498	707672	11.234	19.784	10.823	19.882
13	1880089	1828882	1671829	1621262	22.191	47.562	22.439	46.622
14	2281845	1885026	1881851	1755794	27.107	50.686	25.176	50.669
15	2235602	2223773	2208638	2138246	26.727	58.74	30.897	64.447

Correlation sur les machines								
Nombre de taches	Total de nœuds b1	Total de nœuds max(b1,b2,b3)	Total de nœuds b1 (greedy)	Total de nœuds max(b1,b2,b3) (greedy)	Temps b1	Temps max(b1,b2,b3)	Temps b1 (greedy)	Temps max(b1,b2,b3) (greedy)
5	842	759	760	687	0.009	0.017	0.009	0.018
10	10309	7420	9866	6334	0.13	0.247	0.139	0.236
15	209472	209216	205887	205644	2.505	5.658	2.835	6.271
20	214258	112952	8599	8560	2.68	3.866	0.195	1.152
25	17262	17188	13359	13340	0.355	2.228	0.343	2.595
30	23616	23536	20120	20108	0.546	4.358	0.563	5.161
35	223238	223238	225205	225205	3.232	12.026	3.649	14.986
40	220747	220747	232020	232020	3.164	12.626	3.908	20.531
45	38023	38023	41440	41440	1.173	14.451	1.565	23.301
50	32097	32097	51040	51040	1.069	15.439	2.128	35.452
55	49814	49814	62675	62675	1.917	34.146	2.802	53.242
60	59331	59331	73240	73240	2.308	40.633	3.51	73.365
65	75090	75090	85840	85840	3.18	64.243	4.38	100.519
70	43853	43853	99440	99440	1.963	42.705	5.524	134.736
75	76522	76522	114040	114040	3.569	83.215	6.579	177.905
80	38860	38860	129640	129640	1.945	50.799	8.003	231.465