Mémoire: The k-track assignment

Lola Moisset, encadrée par Mme Gabrel

2017

Table des matières

1 Présentation du problème **2**

2 Problème simple (P1) : machines utilisables à tout moment **2**
2.1 Premier Algorithme : A1P1 2
2.1.1 Pseudo-code 2
2.1.2 Exemple 3
2.2 Deuxième Algorithme (A2P1) 3
2.2.1 Pseudo-code 3
2.2.2 Exemple 4
2.3 Preuve des algorithmes 4
2.4 Une autre façon de faire : A3P1 6

3 L'aspect on-line du problème : modification des algorithmes **6**
3.1 Définition du contexte on-line 6
3.1.1 A1P1 modifié 7
3.1.2 A2P1 modifié 7

4 Problème P2 : machines avec des intervalles d'indisponibilité **7**
4.1 Explication du problème 7
4.2 Algorithme exact : A3P2 8
4.3 Exemple 8
4.4 Algorithmes en pseudo-code 9
4.4.1 A1P1 modifié 9
4.4.2 A2P1 modifié 9

5 Tests **10**
5.1 Protocole 10
5.2 Résultats 10

6 Conclusion **15**

7 Remerciements **15**

8 Références **16**

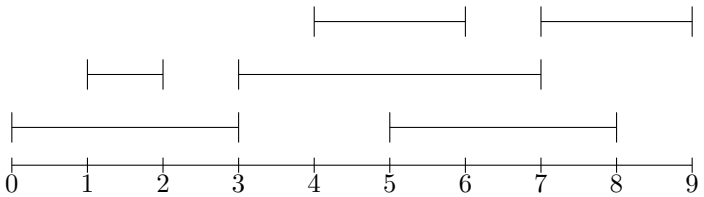
1 Présentation du problème

Le k-track assignment est un problème complexe qui consiste à essayer d'optimiser l'attribution d'un certain nombre de tâches à une ou plusieurs machines. Ce mémoire est basé sur le travail de Kovalyov, Ng, Cheng (2007) [1]. On part d'abord d'un problème simple (P1), où les machines sont disponibles à tout moment. On pourra alors utiliser deux algorithmes simples (A1P1 et A2P1) pour résoudre le problème de façon optimale en un temps polynomial. On peut ensuite s'attarder sur l'aspect on-line de ce problème, où on considère que les informations arrivent au fur et à mesure, et qu'il faut les trier sur le volet. Nous allons ensuite chercher à résoudre un problème plus complexe (P2), où on considère que les machines peuvent avoir des intervalles d'indisponibilité (par exemple des moments où on doit les réparer). On va montrer que ce problème est NP-complet, c'est à dire qu'il est impossible de le résoudre en un temps polynomial.

On considère que les tâches s'exécutent pendant un intervalle de temps fixé, chaque tâche doit être réalisée au plus une fois, chaque machine ne peut exécuter au plus qu'une tâche à chaque instant. Dans tout le reste de ce mémoire, on notera k le nombre de machines, n le nombre de tâches, et les intervalles $I_i = [s_i, d_i]$.

2 Problème simple (P1) : machines utilisables à tout moment

Il existe des algorithmes permettant de trouver la solution optimale au problème en un temps d'exécution polynomial. Prenons d'abord un exemple sur lequel on va exécuter les algorithmes du problème P1 : on prend $k = 2$, $n = 6$



Sont ici représentés sur cette timeline les différents intervalles de tâches des machines pour notre exemple.

2.1 Premier Algorithme : A1P1

2.1.1 Pseudo-code

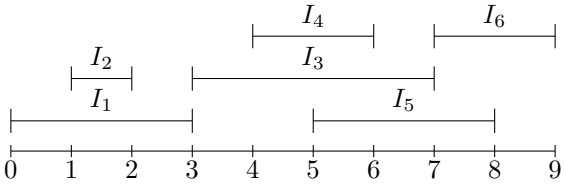
A1P1 consiste à placer chaque intervalle à la suite, dans l'ordre des dates de début, et à supprimer si besoin celle qui prend le plus de temps :

- Etape 1 - numéroter les tâches dans un ordre non décroissant des dates de début, tels que $s_1 \leqslant \cdots \leqslant s_n$, initialiser l'ensemble $S = \emptyset$

- Etape 2 - pour $j = 1, \ldots, n$, faire :
  ajouter le tâche j à S. S'il y a une machine qui peut débuter le travail j au moment s_j, lui affecter j. Sinon, retirer de S le travail avec la date de fin d_j la plus grande. Si ce n'est pas I_j, l'affecter à la machine qui a été libérée.

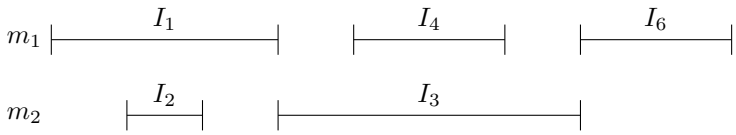
2.1.2 Exemple

Exécutons l'algorithme sur l'exemple pris en début de la partie 2. On commence d'abord par numéroter les intervalles :



- $S = \{I_1\}$, on assigne I_1 à m_1
- $S = \{I_2; I_1\}$, on assigne I_2 à m_2
- $S = \{I_3; I_2; I_1\}$, on assigne I_3 à m_2
- $S = \{I_4; I_3; I_2; I_1\}$, on assigne I_4 à m_1
- $S = \{I_5; I_4; I_3; I_2; I_1\}$, mais aucune machine n'est libre pour débuter I_5. On supprime donc de S I_5 car il est l'intervalle dans S dont la date de fin est la plus grande.
- $S = \{I_6; I_4; I_3; I_2; I_1\}$, on assigne I_6 à m_1

Ce qui nous donne, en décrivant l'utilisation des machines :



2.2 Deuxième Algorithme (A2P1)

2.2.1 Pseudo-code

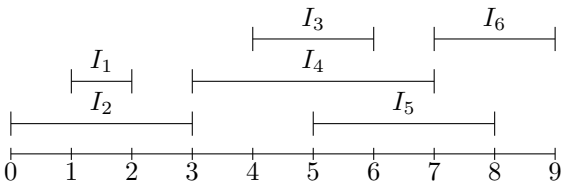
A2P1 consiste à regarder les tâches dans l'ordre des dates de fin croissantes, et attribuer à chaque machine son propre ensemble de tâches, tout en considérant une "machine poubelle" (P_0) où on place les tâches qui ne sont pas dans la solution optimale. Contrairement à A1P1, il ne va pas enlever une tâche déjà placée.

- Etape 1 - numéroter les tâches dans un ordre non décroissant des dates de fin, tels que $d_1 \leqslant \cdots \leqslant d_n$, initialiser les ensembles $P_l = \emptyset, l = 0, 1, \ldots, k$. L'ensemble P_0 sera constitué des tâches rejettés, et P_l sera constitué des intervalles des tâches affectés à la machine l.

- Etape 2 - pour $j = 1, \ldots, n$, faire :
  essayer de trouver un intervalle I_i tel que $i < j$ et :
  - $I_i \in P_1 \cup \cdots \cup P_n$
  - I_i est d'indice maximal dans l'ensemble P_l auquel il appartient.
  - $I_i \cap I_j = \emptyset$
- Etape 3 - si on a trouvé un I_i qui convient, affecter I_j à P_l si $I_i \in P_l$. Sinon, si il existe un ensemble $P_l = \emptyset, 1 \leqslant l \leqslant n$, alors $P_l = \{I_j\}$. Si on a aucun des deux, alors affecter I_j à P_0.

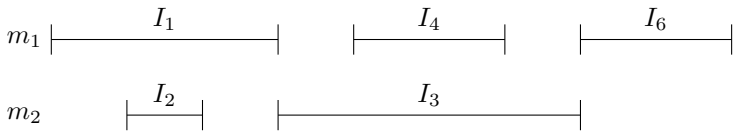
2.2.2 Exemple

On commence par numéroter les intervalles :



- $P_0, P_2 = \emptyset, P_1 = \{I_1\}$
- $P_0 = \emptyset, P_1 = \{I_1\}, P_2 = \{I_2\}$
- $P_0 = \emptyset, P_1 = \{I_1\}, P_2 = \{I_2, I_3\}$
- $P_0 = \emptyset, P_1 = \{I_1, I_4\}, P_2 = \{I_2, I_3\}$
- $P_0 = \{I_5\}, P_1 = \{I_1, I_4\}, P_2 = \{I_2, I_3\}$
- $P_0 = \{I_5\}, P_1 = \{I_1, I_4, I_6\}, P_2 = \{I_2, I_3\}$

Ce qui nous donne, en décrivant l'utilisation des machines :



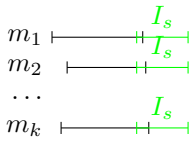
On a bien obtenu avec les deux algorithmes la même solution, qui est optimale.

2.3 Preuve des algorithmes

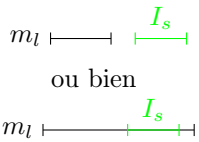
Proposition 1. *Ces deux algorithmes permettent d'obtenir un même résultat optimal à notre problème.*

Démonstration. Montrons d'abord que ces deux algorithmes donnent le même résultat. Supposons que tous les travaux ont des intervalles différents. Posons S_1 l'ensemble S obtenu avec A1P1, et $S_2 = P_1 \cup \cdots \cup P_m$ l'ensemble des travaux

associés à une machine par A2P1. Posons comme indices pour les intervalles ceux obtenus avec l'algorithme A2P1 (ordonnés par date de fin). Soit $I_s \in S_1$ l'intervalle avec le plus petit indice qui n'appartienne pas à S_2. S'il n'appartient pas à S_2, cela veut dire que lors de l'exécution de A2P1, au moment d_s, toutes les machines étaient occupées par des travaux dont les dates de fins sont plus petites que d_s :

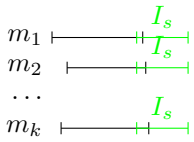


Or, si $I_s \in S_1$, alors cela veut dire que pendant l'algorithme A1P1, au temps d_s, au moins une machine était libre, ou bien occupée par un travail donc la date de fin était plus grande que celle de I_s. on a alors un l tel que :

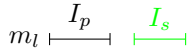


Ce qui contredit le fait que I_s est le travail dont le d_s est le plus petit et tel qu'il soit dans S_1 et non S_2.

Considérons maintenant $I_s \in S_2$ l'intervalle avec le plus petit indice qui n'appartient pas à S_1 (en ordonnant les indices par rapport à A1P1). S'il n'appartient pas à S_1, cela veut dire que lors de l'exécution de A1P1, on retrouve que au moment s_s, toutes les machines sont déjà occupées par des travaux dont les dates de fin sont inférieures à d_s :

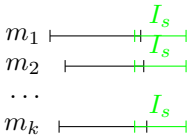


Or $I_s \in S_2$, donc au moment d_s lors de l'exécution de l'algorithme A2P1, on a au moins une machine telle que le dernier travail p qu'on lui a associé est tel que :

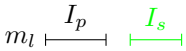


ou bien elle n'a pas encore de travaux associés, ce qui contredit le fait que I_s est le travail avec le plus petit indice (donc d_k) tel qu'il appartienne à S_2 et non à S_1. On a donc prouvé que les deux algorithmes donnent le même résultat.

Montrons maintenant que celui-ci est optimal en se basant sur A1P1. Supposons qu'il existe S' une configuration optimale du problème, telle que $|S'| > |S|$ (S étant le résultat obtenu par A1P1), en indiçant les intervalles par rapport à A1P1 (dates de début croissantes). Posons I_s le travail de plus petit indice tel que $I_s \in S', I_s \notin S$. Cela veut alors dire que dans la configuration obtenue avec A1P1, on a :



or dans la configuration de S', on a obtenu une machine telle que :



Ici, $p < s$, et clairement $I_p \notin S$, ce qui contredit le fait que I_s est l'intervalle au plus petit indice qui se trouve dans S' et non S. Ces deux algorithmes nous donnent donc une solution optimale à notre problème.

□

2.4 Une autre façon de faire : A3P1

On peut aussi écrire un algorithme qui est lui aussi exact, mais plus complexe, en utilisant les cliques (dans un graphe, une clique est un ensemble de points tous reliés deux à deux) :

- Etape 1 : On créé un ensemble P de sommets qui associe tous les intervalles avec chaque machines, tel que un sommet est composé d'un numéro d'intervalle et d'un numéro de machine.
- On créé la matrice d'adjacence associée à tous ces sommets. La matrice est carrée et $N[i][j] = 1$ si et seulement si les deux sommets i et j sont compatibles, c'est à dire que soit les deux intervalles associés sont disjoints, soient les deux machines et intervalles associées sont différents.
- Pour obtenir une solution optimale, il suffit simplement de trouver la plus grande clique maximale.

Cet algorithme permet d'obtenir une combinaison optimale d'intervalles.

3 L'aspect on-line du problème : modification des algorithmes

3.1 Définition du contexte on-line

Être dans un contexte on-line signifie que les informations arrivent en temps réel : on cherche toujours à obtenir une solution optimale, alors que les n tâches

ne sont pas toutes connues, on les découvre au fil du temps. Il faut donc à chaque nouvelle arrivée d'information faire tourner un algorithme et les placer au fur et à mesure (on considère qu'il est possible d'arrêter une tâche en cours).

3.1.1 A1P1 modifié

Pour adapter A1P1 au contexte on-line, il faut appliquer l'algorithme A1P1 sur chaque nouvelle tâche, tout en considérant l'ensemble S utilisé comme l'ensemble des tâches précédemment affectées aux machines - et donc ne pas le réinitialiser. Le pseudo-code de l'algorithme à effectuer à chaque nouvelle arrivée de tâche, en supposant que la tâche qui arrive est la $m$ième, est :

- On ajoute I_m à S
- S'il y a une machine disponible pour exécuter I_m, l'associer à cette machine
- Sinon, supprimer de S le travail avec la date de fin la plus grande (on considère qu'on peut arrêter une tâche en cours). Si ce n'est pas I_m, l'affecter à la machine qui a été libérée

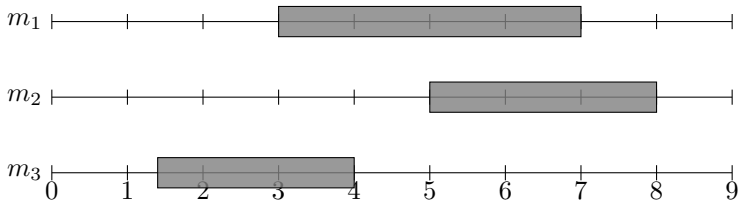
3.1.2 A2P1 modifié

Lorsqu'on essaye d'adapter A2P1 au contexte on-line, on se rend compte qu'on obtient une solution différente et de rendu moins bon que celle obtenue avec A1P1. En effet, il ne permet pas de retirer une tâche à une machine, il va forcément la laisser dans son ensemble P_l associé. Par exemple, si on a une tâche qui dure à peu près toute la durée de notre timeline (le temps sur lesquelles les tâches peuvent s'exécuter), elles sera placée dans les premières, mais ne pourra pas être déplacée, et empêchera donc plusieurs tâches de la remplacer.

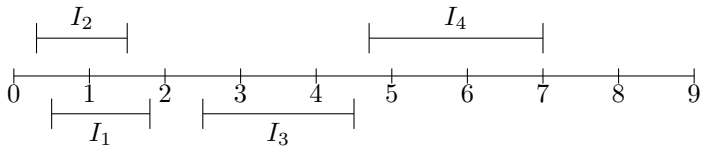
4 Problème P2 : machines avec des intervalles d'indisponibilité

4.1 Explication du problème

On va maintenant s'intéresser à ce qui se passe lorsque des machines ne sont pas disponibles à certains moments fixés, ces périodes d'indisponibilité peuvent par exemple être causées par des réparations des machines. Par exemple, prenons trois machines dont les timelines sont :



Comment associer les tâches ci-dessous aux machines ci-dessus ?



4.2 Algorithme exact : A3P2

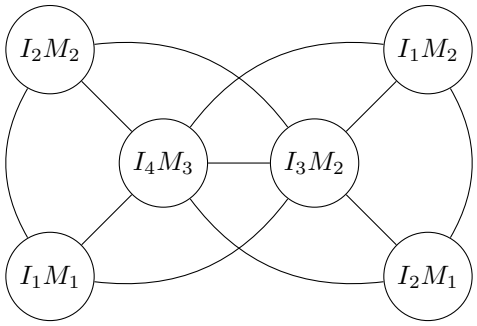
On peut d'abord adapter l'algorithme A1P3 qui nous donne des solutions exactes et optimales tout en utilisant les cliques :

- Etape 1 - créer le graphe associé aux tâches et machines, tel qu'il existe un sommet I_iM_j s'il est possible d'effectuer la tâche I_i sur la machine M_j, et une arête entre deux sommets s'ils sont compatibles (i.e. si deux taches i et j s'intersectent, on ne peut pas avoir d'arête entre I_iM_k et I_jM_k, de même entre I_iM_k et I_iM_l).
- Etape 2 - chercher une clique maximale dans le graphe : c'est une solution à notre problème (une clique est un ensemble de sommets deux à deux adjacents).

Mais le problème est NP-complet, c'est à dire qu'il n'existe pas d'algorithme exact s'exécutant en un temps polynomial de la taille du problème. Dans l'algorithme ci-dessus, déterminer la clique maximale est un problème qui a été prouvé comme NP-complet [2]. On ne peut donc pas l'utiliser pour de grandes tailles

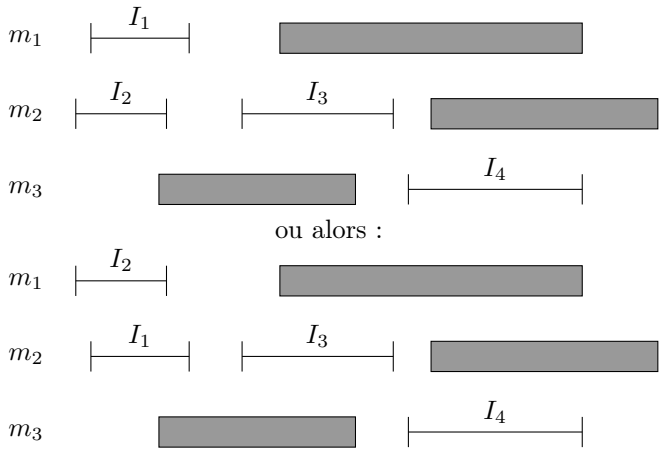
4.3 Exemple

En reprenant l'exemple de la partie 4.1, on a comme sommets possibles : $I_1M_1, I_2M_1, I_1M_2, I_2M_2, I_3M_2, I_4M_3$.



Les cliques maximales du graphe sont : $\{I_1M_1, I_2M_2, I_4M_3, I_3M_2\}$ et $\{I_1M_2, I_2M_1, I_3M_2, I_4M_3\}$.

On obtient donc :



4.4 Algorithmes en pseudo-code

On peut aussi modifier les autres algorithmes vus précédemment, mais le résultat peut ne pas être exact sur de grandes tailles

4.4.1 A1P1 modifié

On note A l'ensemble des numéros des intervalles représentant l'indisponibilité :

- Etape 1 - numéroter les tâches dans un ordre non décroissant des dates de début, tels que $s_1 \leqslant \cdots \leqslant s_n$, initialiser l'ensemble $S = \emptyset$ et ajouter à l'ensemble A les numéros des intervalles d'indisponibilité (considérés comme des tâches).
- Etape 2 - pour $j = 1, \ldots, n$, faire :
  Si j n'est pas dans A, ajouter le tâche j à S. S'il y a une machine qui peut débuter le travail j au moment s_j, lui affecter j. Sinon, retirer de S le travail avec la date de fin d_j la plus grande et tel que il n'appartient pas à A. Si ce n'est pas I_j, l'affecter à la machine qui a été libérée.
  Si j appartient à A, faire : regarder si sa machine associée n'a bien aucune tâche à ce moment là. Si ce n'est pas le cas, supprimer de S toutes les tâches associées à cette machines telles que $I_k \cap I_l = \emptyset$.

4.4.2 A2P1 modifié

Pour adapter A2, il faut considérer que les tâches déjà placées peuvent être déplacées à la machine corbeille. On note A l'ensemble des numéros des intervalles représentant l'indisponibilité :

- Etape 1 - numéroter les tâches dans un ordre non décroissant des dates de fin, tels que $d_1 \leqslant \cdots \leqslant d_n$, tout en considérant que les intervalles d'indisponibilité sont aussi des tâches. Ajouter à A les numérotations des

intervalles d'indisponibilité. initialiser les ensembles $P_l = \emptyset, l = 0, 1, \ldots, k$. L'ensemble P_0 sera constitué des tâches rejetés, et P_l sera constitué des intervalles des tâches affectés à la machine l.

- Etape 2 - pour $j = 1, \ldots, n$, faire :

  Si j n'appartient pas à A, essayer de trouver un intervalle I_i tel que $i < j$ et :

  - $I_i \in P_1 \cup \cdots \cup P_n$
  - I_i est d'indice maximal dans l'ensemble P_l auquel il appartient.
  - $I_i \cap I_j = \emptyset$
  - Etape 3 - si on a trouvé un I_i qui convient, affecter I_j à P_l si $I_i \in P_l$. Sinon, si il existe un ensemble $P_l = \emptyset, 1 \leqslant l \leqslant n$, alors $P_l = \{I_j\}$. Si on a aucun des deux, alors affecter I_j à P_0.

  Si j appartient à A, alors faire :

  - Ajouter I_j au P_l associé à sa machine
  - Vérifier que pour tout $I_k \in P_l$, $I_i \cap I_j = \emptyset$. Si ce n'es pas le cas, déplacer les I_k tels que c'est faux dans P_0.

5 Tests

5.1 Protocole

Pour prouver mon hypothèse, j'ai décidé d'établir un protocole faisant varier un facteur à la fois. Je vais d'abord faire tourner sur des machines sans intervalles d'indisponibilité les trois algorithmes (A1P1, A2P1, et A3P1, celui utilisant les cliques adapté au problème P1) en augmentant uniquement le nombre de tâches (Test 1). Je vais ensuite faire la même chose mais avec un intervalle d'indisponibilité sur chaque machine (Test 2).

On devrait remarquer dans les deux tests que l'algorithme A3 devrait prendre bien plus de temps que les autres, qui ne sont pas exacts. Je vais ensuite augmenter le nombre d'intervalles d'indisponibilité de chaque machine (Test 3). Pour ce test le temps d'exécution devrait fluctuer, car rajouter des indisponibilités ne va pas compliquer les calculs, mais plutôt même diminuer le nombre de sommets utilisés dans le calcul de cliques.

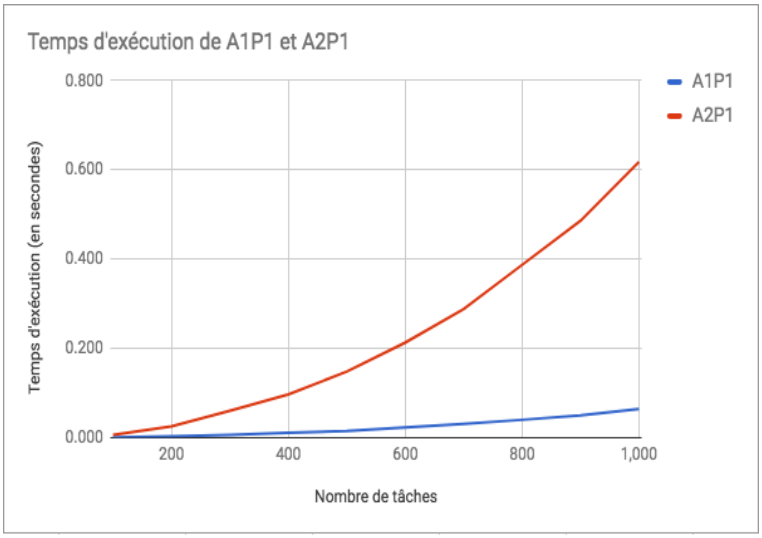
5.2 Résultats

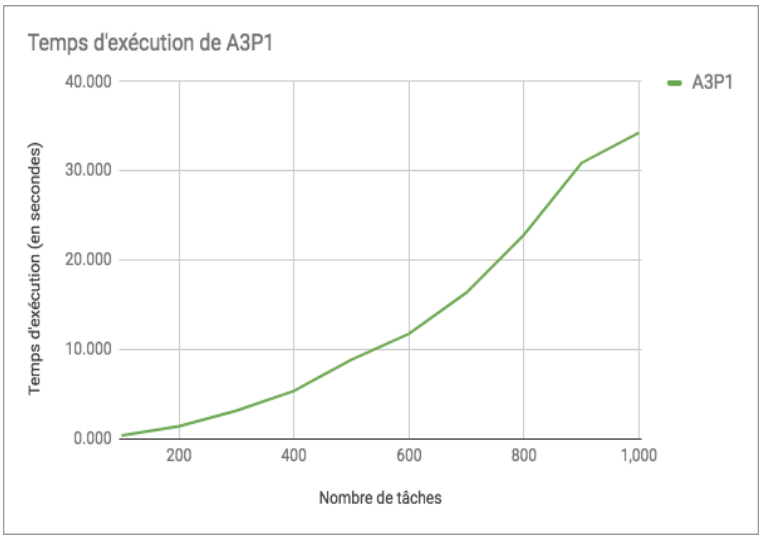
En faisant tourner les algorithmes du premier test, on obtient :

:

Nombre de tâches	Temps d'exécution (s)		
	A1P1	A2P1	A3P1
100	0.001	0.006	0.375
200	0.003	0.025	1.428
300	0.006	0.060	3.175
400	0.011	0.097	5.380
500	0.015	0.148	8.884
600	0.023	0.213	11.791
700	0.031	0.288	16.382
800	0.040	0.387	22.864
900	0.050	0.486	30.889
1000	0.064	0.618	34.278

FIGURE 1 – Test 1 : n varie, k=7





Ces résultats montrent bien qu'il y a une grande différence dans le temps d'exécution des trois algorithmes. A2P1 prend plus de temps que A1P1 car il doit faire plusieurs vérifications à chaque itération sur tous les intervalles déjà placés, et A3P1 est encore plus lent car trouver une clique prend un temps important pour une machine. De plus, dans l'algorithme A3P1, on aura toujours un nombre croissant de sommets parmi lesquels il faudra trouver une clique maximale (n*k), puisque chaque sommet peut-être associé à chaque machine, d'où la courbe obtenue.

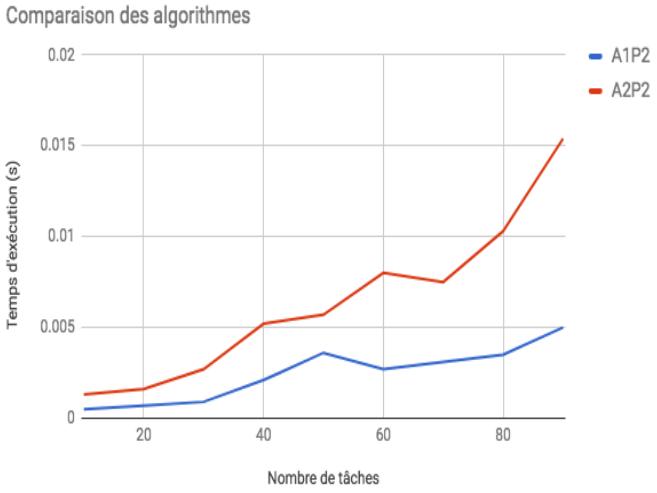
Passons maintenant au test 2, avec p le nombre de sommets utilisés pour trouver la clique maximale dans A3P2 :

Nombre de tâches	Temps d'exécution (s)			
	A1P1	A2P1	A3P1	p
10	0.0005	0.0013	0.0053	14
20	0.0007	0.0016	0.0057	24
30	0.0009	0.0027	0.0755	40
40	0.0021	0.0052	59.7142	93
50	0.0036	0.0057	1.4865	78
60	0.0027	0.0080	0.5772	60
70	0.0031	0.0075		
80	0.0035	0.0103		
90	0.0050	0.0154		

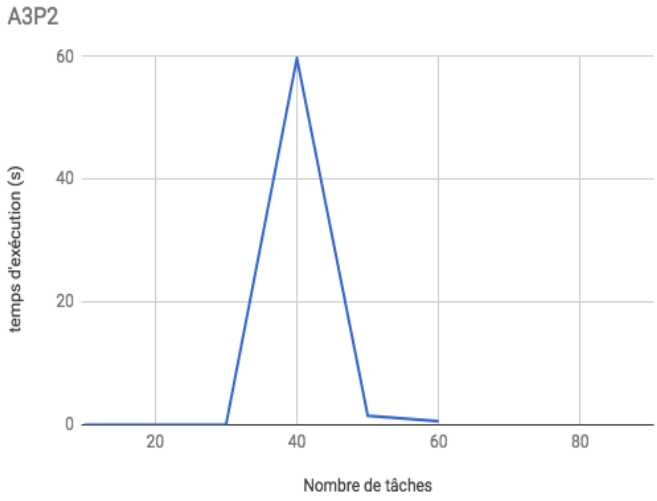
:

FIGURE 2 – Test 2 : n varie, k=4, chaque machine a un intervalle d'indisponibilité

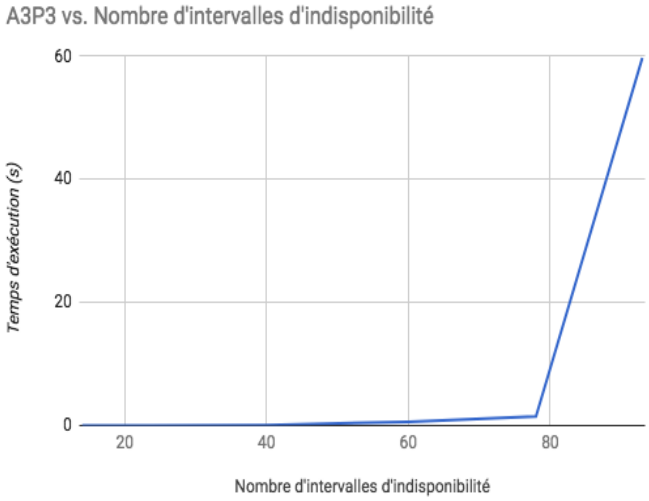
À partir d'un certain nombre de tâches, la recherche des cliques était trop poussée pour mon ordinateur. En traçant les courbes, on peut à nouveau voir que A2P2 est plus complexe que A1P2 :



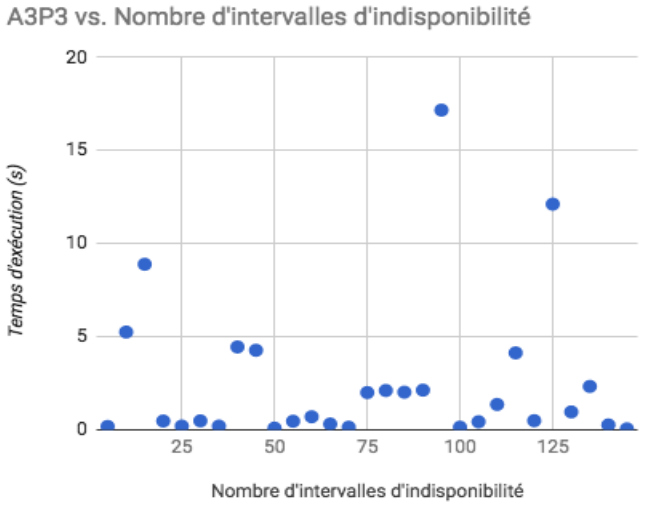
On peut aussi voir que le temps d'exécution de A3P2 ne dépend pas du nombre de tâches :



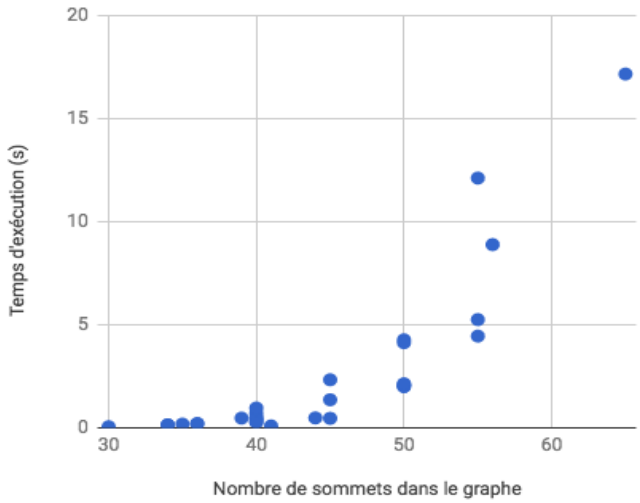
Mais, lorsqu'on trace la courbe du temps d'exécution de A3P2 par rapport à p, on obtient :



On a donc un semblant de corrélation entre la taille du graphe et le temps d'exécution. On peut en apprendre plus à l'aide du troisième test, où j'ai adapté l'algorithme A3P2 à des machines ayant plusieurs intervalles d'indisponibilité, et fait varier le nombre de ces derniers :



En traçant les temps d'exécution par rapport aux nombres d'intervalles, on voit bien que les temps d'exécution fluctuent beaucoup, et qu'il n'y a pas vraiment de corrélation entre eux. Par contre, si on trace les temps d'exécution en fonction de p, on obtient :



On voit bien que le temps d'exécution de l'algorithme croît de façon exponentielle avec la recherche de cliques maximales, ce qui est logique car la recherche d'une clique maximale est un problème NP-complet. Cela justifie donc pourquoi notre problème P2 est un problème NP-complet : on ne pourra pas trouver d'algorithme exact qui permette de résoudre le problème d'une façon polynomiale.

6 Conclusion

Nous avons donc pu voir, à travers les différents algorithmes que j'ai utilisés et adaptés afin de trouver une solution optimale (ou approchée dans les cas de A1P2 et A2P2), que dès que les machines commencent par exemple à tomber en panne, trouver une solution optimale devient tout de suite beaucoup plus long. De plus, si, à nouveau, on passait au contexte on-line avec le problème P2 (c'est à dire qu'on ne peut pas prédire le moment où les machines vont tomber en panne), ce serait à nouveau très compliqué d'obtenir la meilleure solution, car si on adaptait l'algorithme A3, il faudrait à chaque nouvelle tâche effectuer une recherche de clique maximale.

7 Remerciements

Je voudrais particulièrement remercier Virginie Gabrel, mon encadrante, qui m'a aidée à trouver un sujet qui m'a intéressée et m'a guidée dans la réalisation de ce mémoire.

8 Références

[1] Kovalyov MY, Ng CT, Cheng TCE. Fixed interval scheduling : models, applica-tions, computational complexity and algorithms. European Journal of Opera-tional Research 2007 ;178 :331–342.

[2] M. S. Garey and D. S. Johnson. Computers and Intractibility : A Guide to NPCompleteness. New York, NY, USA : Freeman & Co., 1979.