

# Rapport de Projet

## Algorithmes & Complexité

### Sommaire :

<b>Fonctionnalités développées</b>	<b>2</b>
<b>Génération de fichiers</b>	<b>2</b>
Ecriture	2
Lecture	3
<b>Description des méthodes de scheduling</b>	<b>5</b>
Préambule :	5
Version GAMMA	5
Description	5
Version ALPHA	6
Description	6
Exemple	7
Version BETA	9
Description à partir d'un exemple concret	9
Description en schéma	11
Interprétation de cette méthode	11
LOAD BALANCING	13
<b>Benchmarking des méthodes</b>	<b>14</b>
Résumé des résultats obtenus sur l'exemple :	14
Avant de comparer les temps d'exécution, veuillez à exécuter une première fois à blanc le programme.	14
Comparaison de la répartition des tâches	14
Comparaison asymptotique des Timeline obtenues	15
Comparaison asymptotique des temps de génération des méthodes	17
Limites de Beta :	18
Résumé	18
<b>ANNEXE : Classes développées</b>	<b>19</b>

## Fonctionnalités développées

- Génération aléatoire d'un fichier de configuration (machines et tâches)
- Lecture de ce fichier (input\_scheduler.txt)
- Traitement selon les méthodes de scheduling gamma, alpha et beta.
- Affichage des timeline d'attribution de tâches générées selon 2 modes :
  - Console
  - JavaFX

## Génération de fichiers

Pour effectuer les opérations de lecteur et d'écriture, nous avons créé une classe "FileGenerator" permettant de regrouper ces fonctionnalités. Cette classe se caractérise par 4 méthodes statiques :

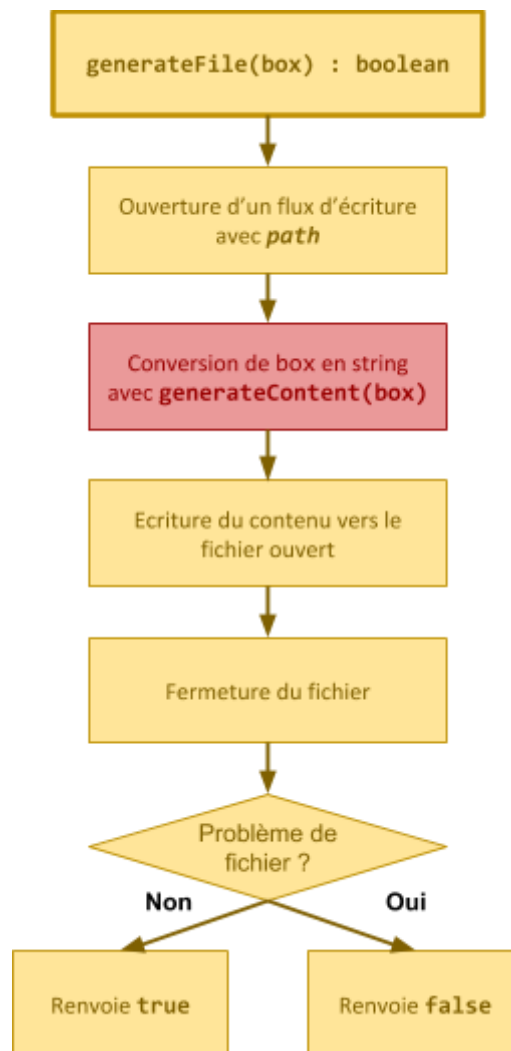
- `public static boolean generateFile(Box box)`
- `public static String generateContent(Box box)`
- `public static Box readBox()`
- `public static String readContent()`

Les trois premières méthodes utilisent la classe "Box", qui contient un cluster de machine (ArrayList de machine), ainsi qu'une liste de Job. L'objet "Box" est utilisé pour regrouper toutes les ressources du problème à traiter.

### Ecriture

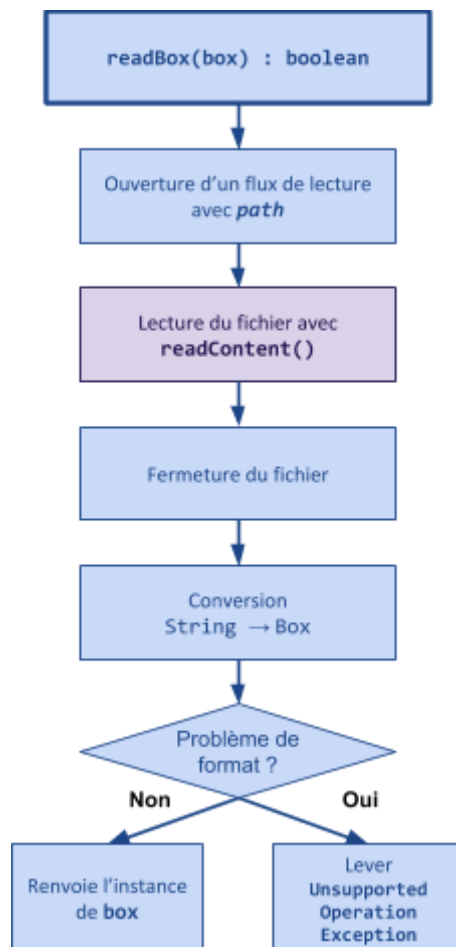
L'écriture d'un objet Box se déroule avec la fonction `generateFile(Box box)`, qui génère un fichier dont le contenu est l'ensemble des machines, suivi de l'ensemble des jobs et des tâches du problème, en précisant les dépendances entre ces dernières. Le format utilisé est le même que celui vu en cours.

La méthode statique `generateFile(Box box)` utilise l'attribut statique `path` de la classe `FileGenerator` et la méthode `String generateContent(Box box)` qui renvoie la représentation `String` de `box` passé en argument, selon le format utilisé. `generateContent` représente donc le cœur de l'écriture du problème car il formate les données. Pour simplifier, voici comment se déroule la génération d'un fichier :



## Lecture

La lecture d'un fichier contenant une instance de Box dont le format est le même que celui vu en cours est similaire à la méthode d'écriture. `readBox()` récupère le contenu du fichier sous forme de string avec la méthode `readContent()`, et convertit la chaîne de caractère en Box si le format est correctement respecté. Dans le cas contraire, une exception est levée, précisant l'erreur du format.



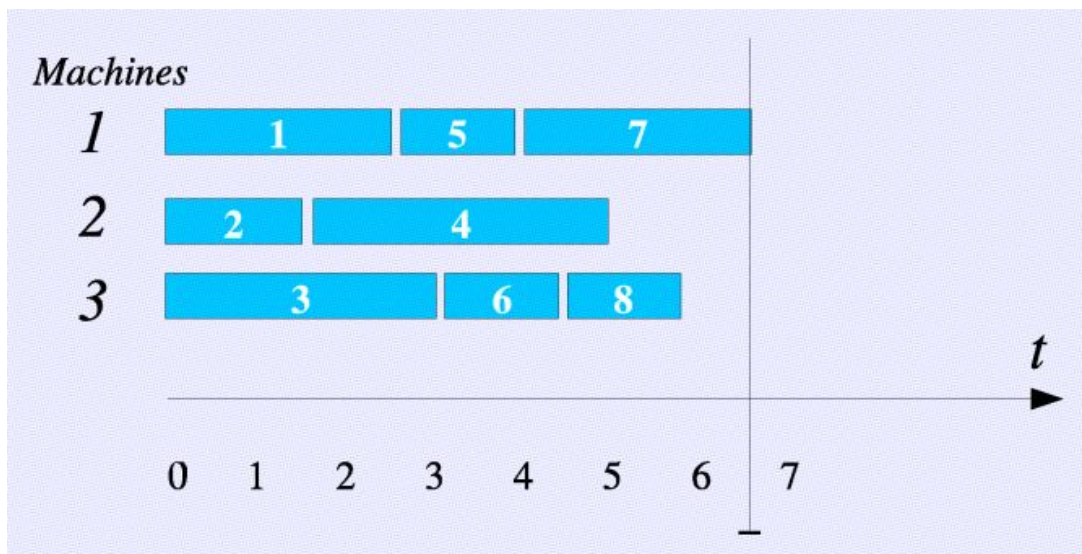
## Description des méthodes de scheduling

### Préambule :

Nous avons conçu une classe Timeline qui facilite la visualisation des tâches allouées pour chaque. Nous l'avons ensuite étendue à un affichage console puis à un affichage graphique à partir de JavaFX.

Ce que nous appellerons temps de réalisation, temps de la Timeline etc est le temps total que prendra la réalisation de toutes les tâches sur les machines à disposition.

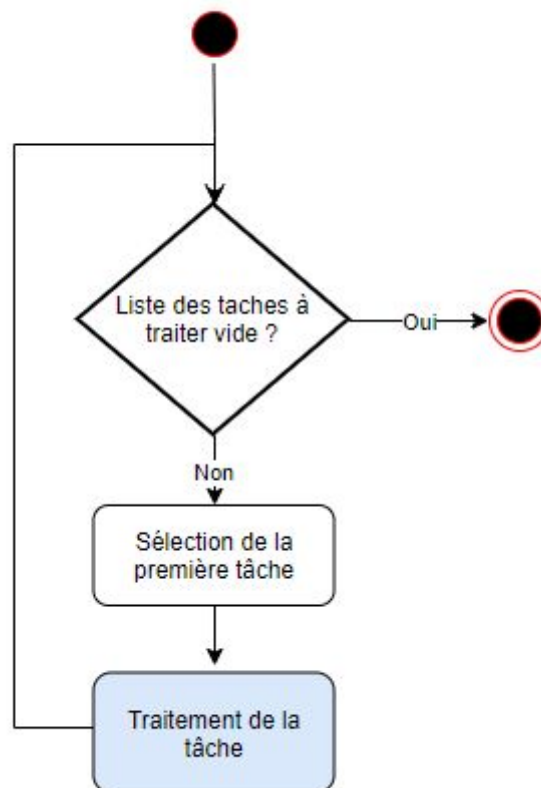
Dans l'exemple ci-dessous il est égal à 6,5 secondes.



## Version GAMMA

### Description

Cet algorithme va prendre les tâches dans l'ordre dans la liste de tâches. Les tâches prédécesseurs seront ainsi toujours effectuées étant donné que les prédécesseurs sont pris dans les tâches définies avant. Nous allons alors traiter la tâche choisie : en trouvant la ligne de la timeline la plus optimale, c'est à dire qui permettra de prendre le moins de temps possible (comme pour l'algorithme BETA).

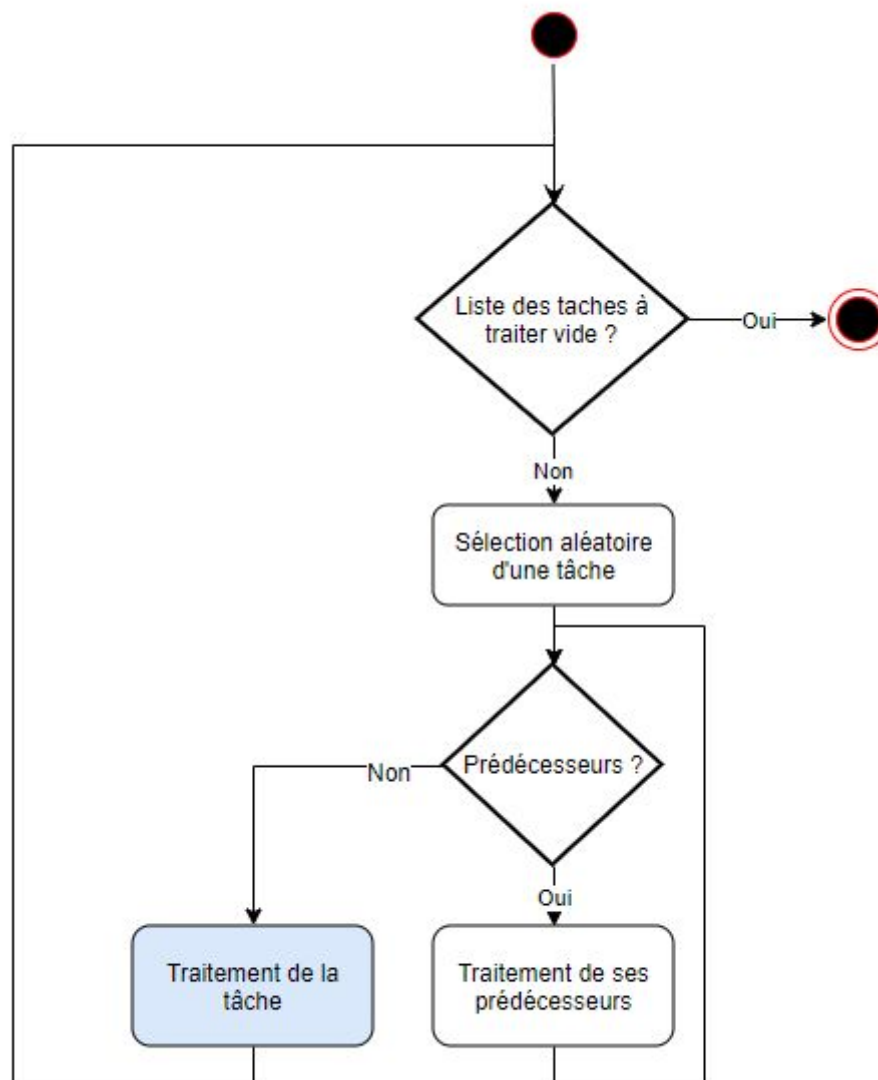


## Version ALPHA

### Description

Cet algorithme va prendre aléatoirement une tâche de notre liste de tâches puis va :

- S'assurer que ses tâches prédécesseurs ont été effectuées : si ce n'est pas le cas on traite les tâches.
- Traiter la tâche choisie : en trouvant la ligne de la timeline la plus optimale, c'est à dire qui celle qui permettra de prendre le moins de temps possible (comme pour l'algorithme BETA).



La pertinence de cette méthode est aléatoire : parfois cela peut se rapproche de beta(néanmoins très rarement). Mais parfois cela peut aller beaucoup plus loin (cf Benchmarking).

### Exemple

Prenons l'exemple suivant :

```
Servers
  CPU = [40G, 10G, 8G]
  GPU = [25T, 11T]
  I/O = [2G, 2G, 5G]
Job 1 = [T12, T10, T3, T5, T11, T14, T8, T9, T1, T4, T2, T6, T7, T15, T13, T16]
  T1 = CPU, 400G, []
  T2 = CPU, 500G, []
  T3 = GPU, 100G, [T1, T2]
  T4 = I/O, 2G, [T3]
  T5 = I/O, 20G, [T3]
  T6 = I/O, 30G, [T3]
  T7 = CPU, 150G, [T3]
  T8 = CPU, 160G, [T3]
  T9 = CPU, 170G, [T3, T7]
  T10 = CPU, 180G, [T7]
  T11 = CPU, 210G, [T4]
  T12 = CPU, 300G, [T5, T4, T3]
  T13 = CPU, 200G, [T3, T2]
  T14 = I/O, 10G, [T5]
  T15 = I/O, 20G, [T7, T12, T13]
  T16 = I/O, 50G, [T1, T15]
Job 2 = [T2, T3, T4, T1]
  T1 = GPU, 200G, []
  T2 = CPU, 1T, [T1]
  T3 = I/O, 7G, [T2]
  T4 = I/O, 5G, [T1]
```

Dans cet exemple (la même configuration que l'algorithme BETA que nous verrons juste après), on atteint 158 sec alors que la méthode BETA permettra, nous le verrons, d'atteindre toujours 79,5 sec.

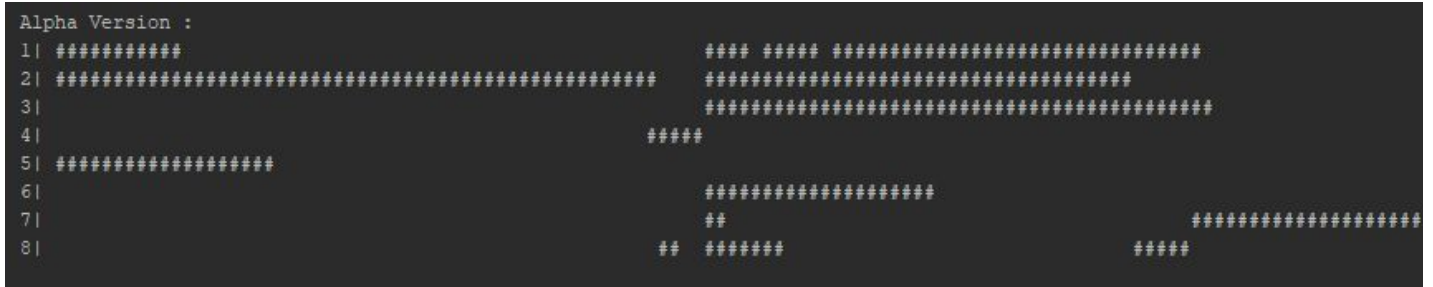
Au bout d'une vingtaine d'essais nous sommes parvenus à obtenir une seule fois 98 sec. Mais la méthode manque de stabilité.

Attribution des tâches :

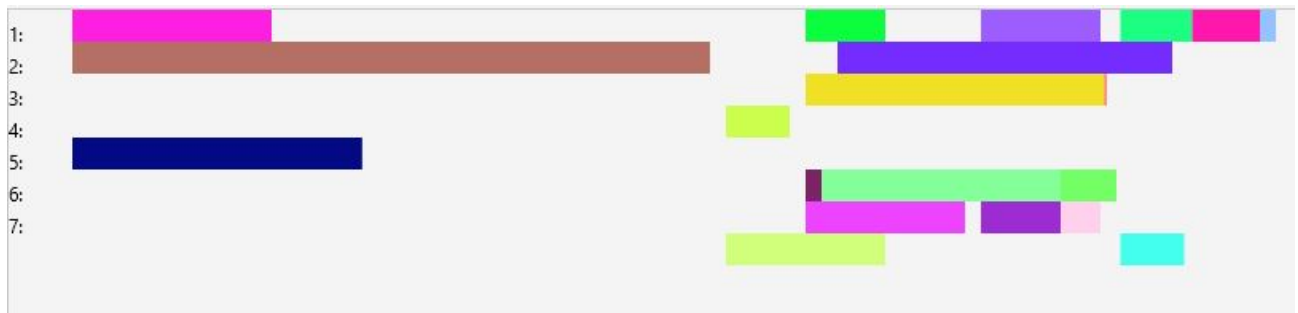
```
Timeline
1|{CPU | 400G | Time : 10.0}{CPU | 150G | Time : 3.75}{CPU | 170G | Time : 4.25}{CPU | 300G | Time : 7.5}{CPU | 400G | Time : 10.0}{CPU | 300G | Time : 7.5}{CPU | 210G | Time : 5.25}
2|{CPU | 500G | Time : 50.0}{CPU | 1T | Time : 0.099998474}{CPU | 150G | Time : 15.0}{CPU | 200G | Time : 20.0}
3|{CPU | 160G | Time : 20.0}{CPU | 180G | Time : 22.5}
4|{GPU | 100G | Time : 4.0}
5|{GPU | 200G | Time : 18.181818}
6|{I/O | 20G | Time : 10.0}{I/O | 10G | Time : 5.0}{I/O | 5G | Time : 2.5}
7|{I/O | 2G | Time : 1.0}{I/O | 50G | Time : 25.0}
8|{I/O | 7G | Time : 1.4000015}{I/O | 30G | Time : 6.0}{I/O | 20G | Time : 4.0}
```

En version console :





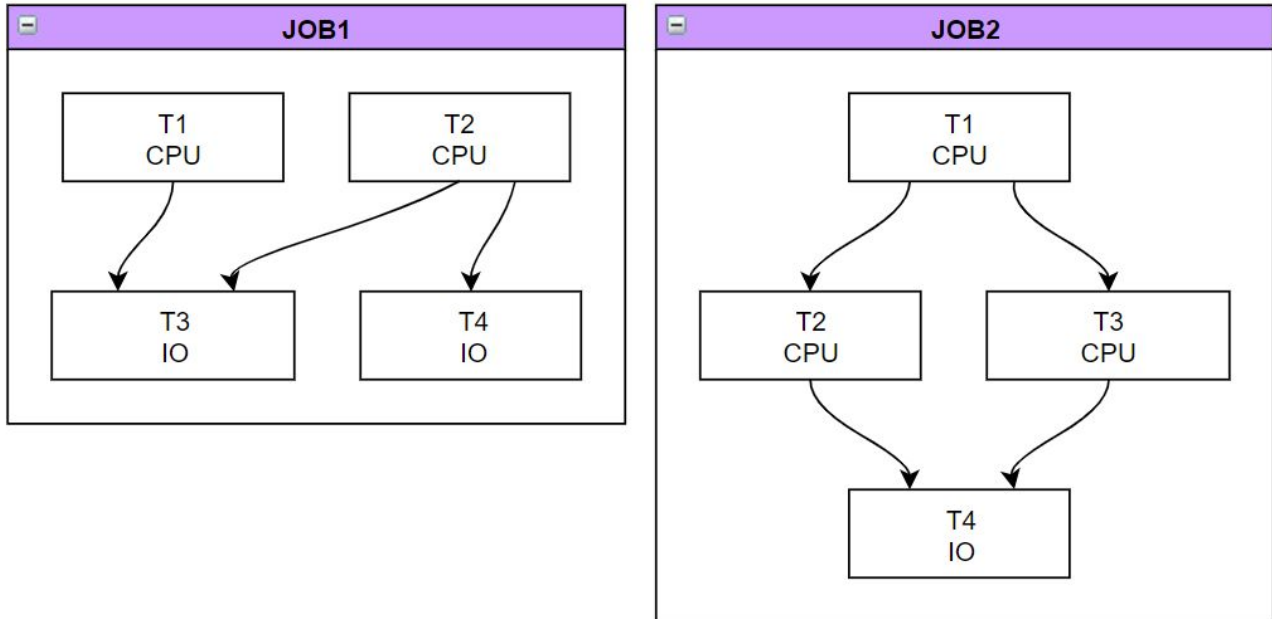
En version graphique :



## Version BETA

### Description à partir d'un exemple concret

Voici 2 JOB :



Nous considérons un cluster de 3 machines CPU et de 2 machines I/O.

CP3 est 2 fois plus puissante que CP1.

Voici le temps d'exécution de chaque tâche sur les machines.

	CP1	CP2	CP3
J1T1	10	15	20
J1T2	5	7.5	10
J2t1	3	4.5	6
J2T2	15	22.5	30
J2T3	4	6	8

	IO1	IO2
J1T3	3	6
J1T4	2	4
J2T4	5	10

**Objectif** : On gagnera du temps à exécuter les tâches qui ont le plus de successeurs. C'est à dire les tâches qui ne peuvent s'exécuter que si les tâches qui la précèdent sont finies (si la plus longue est finie).

Par exemple : dans le JOB1, la tâche 1 à 3 successeurs (2 directs et 1 indirect) donc elle doit être réalisé en premier pour permettre aux autres de s'exécuter.

On calcule pour cela une "priorité" pour chaque tâche de la manière suivante : temps du processus de la tâche + priorité de toutes les tâches qui succèdent. On peut de cette manière le calculer de façon récursive.

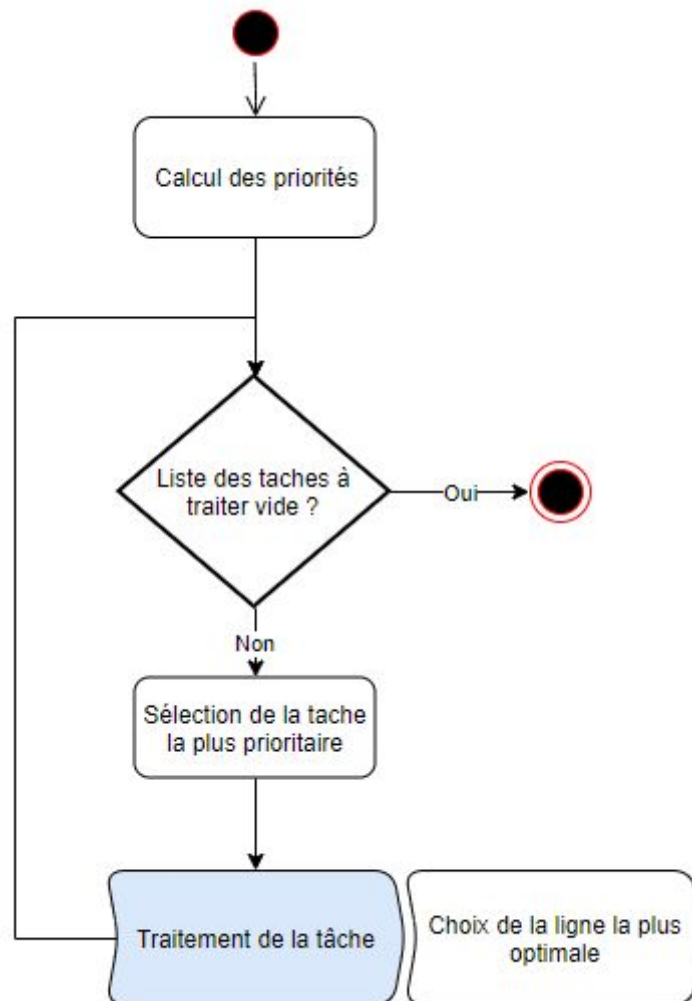
Tâche	Priorité
J1T1	13
J1T2	10
J1T3	3
J1T4	2
J2T1	27
J2T2	20
J2T3	9
J2T4	5

Ainsi, on exécutera la tâche J2T1 dans un premier temps, puis la tâche J1T1 etc...

### Description en schéma

Voici la manière dont sont traitées chaque tâches :

- On va traiter à chaque fois la tâche la plus prioritaire dans notre liste de tâches.
- On détermine la machine sur laquelle doit s'exécuter notre tâche = la ligne ou le temps des tâches déjà exécutées plus le temps d'exécution sur la machine est le plus court.



### Interprétation de cette méthode

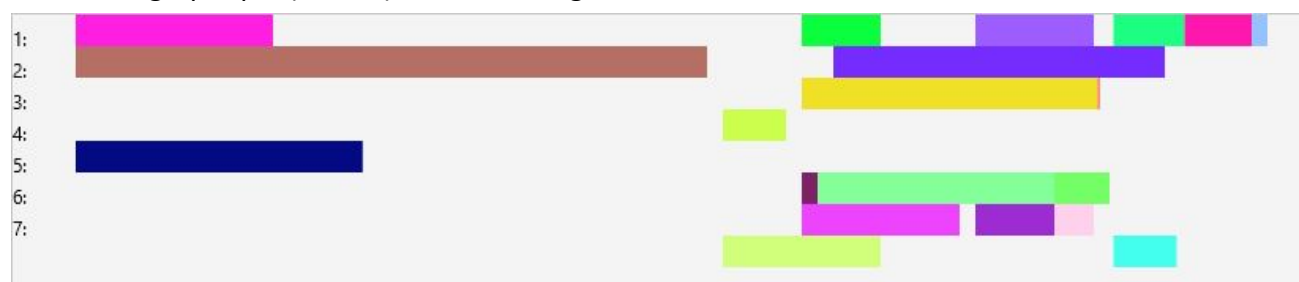
Voici la configuration que nous avons testé (la même que pour ALPHA):

```
Servers
  CPU = [40G, 10G, 8G]
  GPU = [25T, 11T]
  I/O = [2G, 2G, 5G]
Job 1 = [T12, T10, T3, T5, T11, T14, T8, T9, T1, T4, T2, T6, T7, T15, T13, T16]
  T1 = CPU, 400G, []
  T2 = CPU, 500G, []
  T3 = GPU, 100G, [T1, T2]
  T4 = I/O, 2G, [T3]
  T5 = I/O, 20G, [T3]
  T6 = I/O, 30G, [T3]
  T7 = CPU, 150G, [T3]
  T8 = CPU, 160G, [T3]
  T9 = CPU, 170G, [T3, T7]
  T10 = CPU, 180G, [T7]
  T11 = CPU, 210G, [T4]
  T12 = CPU, 300G, [T5, T4, T3]
  T13 = CPU, 200G, [T3, T2]
  T14 = I/O, 10G, [T5]
  T15 = I/O, 20G, [T7, T12, T13]
  T16 = I/O, 50G, [T1, T15]
Job 2 = [T2, T3, T4, T1]
  T1 = GPU, 200G, []
  T2 = CPU, 1T, [T1]
  T3 = I/O, 7G, [T2]
  T4 = I/O, 5G, [T1]
```

Voici la Timeline en quantum de temps générée :  
En version console :

```
Beta Version :
1| #####
2| #####
3| #####
4| #####
5| #####
6| #####
7| #####
8| #####
```

En version graphique (JavaFX) afin de distinguer les différentes tâches :

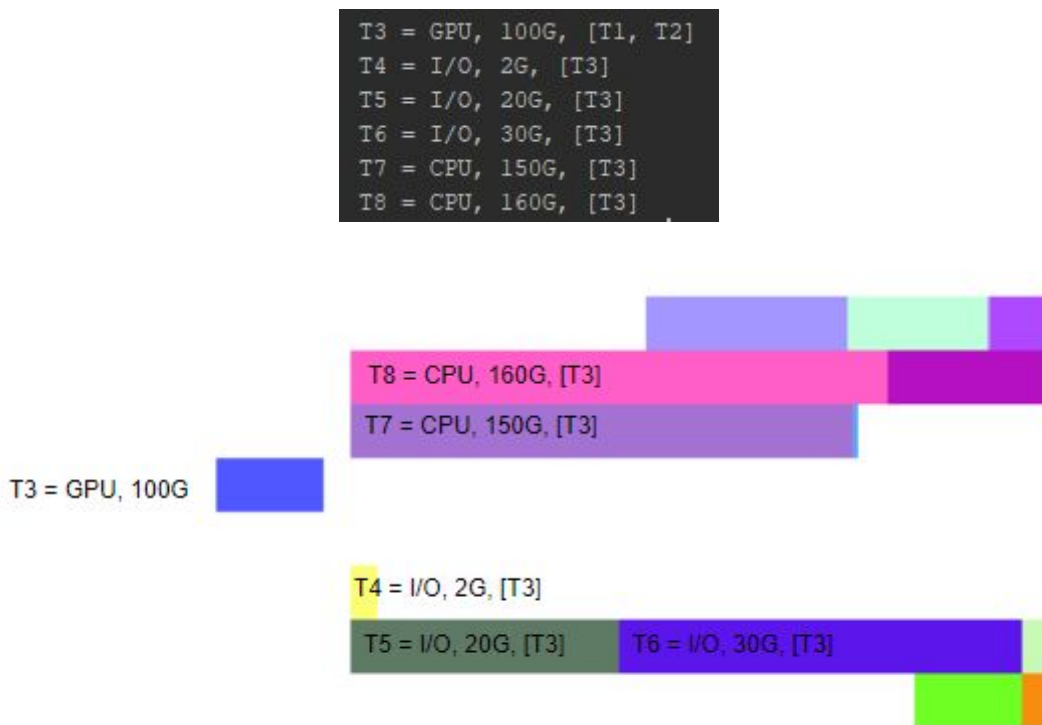


Le temps total renvoyé par cette méthode est de 79,5.

Voici l'affectation des tâches correspondantes :

```
Timeline
1|{CPU | 500G | Time : 12.5}{CPU | 200G | Time : 5.0}{CPU | 300G | Time : 7.5}{CPU | 180G | Time : 4.5}{CPU | 170G | Time : 4.25}{CPU | 160G | Time : 4.0}
2|{CPU | 400G | Time : 40.0}{CPU | 210G | Time : 21.0}
3|{CPU | 150G | Time : 18.75}{CPU | 1T | Time : 0.125}
4|{GPU | 100G | Time : 4.0}
5|{GPU | 200G | Time : 18.181818}
6|{I/O | 2G | Time : 1.0}{I/O | 30G | Time : 15.0}{I/O | 7G | Time : 3.5}
7|{I/O | 20G | Time : 10.0}{I/O | 10G | Time : 5.0}{I/O | 5G | Time : 2.5}
8|{I/O | 50G | Time : 10.0}{I/O | 20G | Time : 4.0}
```

On constate que cette méthode beta, **conserve bien l'exécution des tâches après leur prédécesseurs**. Prenons l'exemple de la tâche T3 du Job1. L'image précédente permet de relier la timeline obtenue avec l'affectation des tâches. On peut ainsi vérifier :



Remarque : cette timeline ci-dessous a été générée par une version de Beta moins optimisée, mais l'argument est exactement le même.

## LOAD BALANCING

Nous avons par la suite mis en place une méthode de Load Balancing suivante.

Lorsque nous déterminons la ligne la plus optimale, si nous constatons que cette dernière est beaucoup plus chargée déjà qu'une autre, nous basculons l'attribution sur cette dernière. Ce choix se base sur le calcul d'un ratio entre le temps total déjà alloué de la tâches optimale et de celle qui pourrait la remplacer. Plus ce ratio est grand (supérieur à 3, 5 ou 10 selon le choix), il devient apportant de basculer sur la nouvelle ligne.

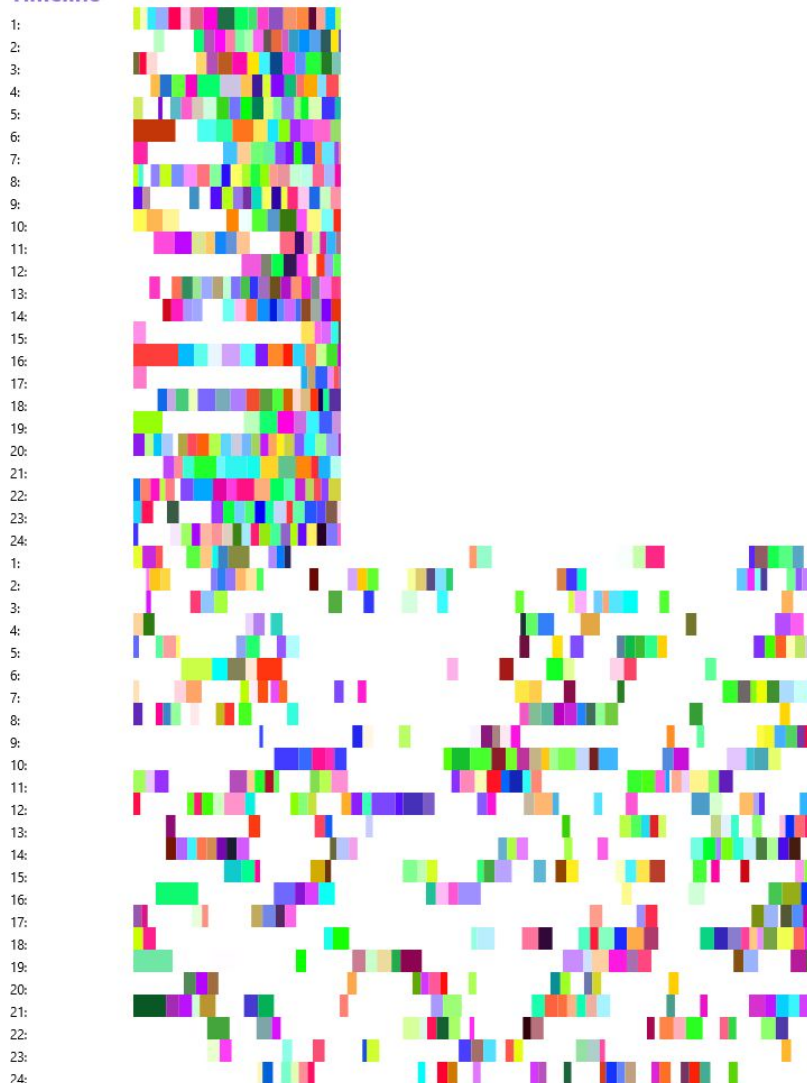
## Benchmarking des méthodes

### Résumé des résultats obtenus sur l'exemple :

Avant de comparer les temps d'exécution, veuillez à exécuter une première fois à blanc le programme.

	Gamma	Alpha	Beta
Temps total de la timeline	129,75	Entre 85 et 158 environ	79,5
Temps de génération	0 à 5 ms	1 à 9 ms	6 à 9 ms

Timeline



### Comparaison de la répartition des tâches

Voici un exemple généré aléatoirement de ce à quoi on peut s'attendre comme répartition de tâches. Les 24 premières lignes correspondent à BETA et les 24 suivantes à ALPHA. On constate que BETA est beaucoup plus compacte et "comble" davantage les vides entre les tâches.

## Comparaison asymptotique des Timeline obtenues

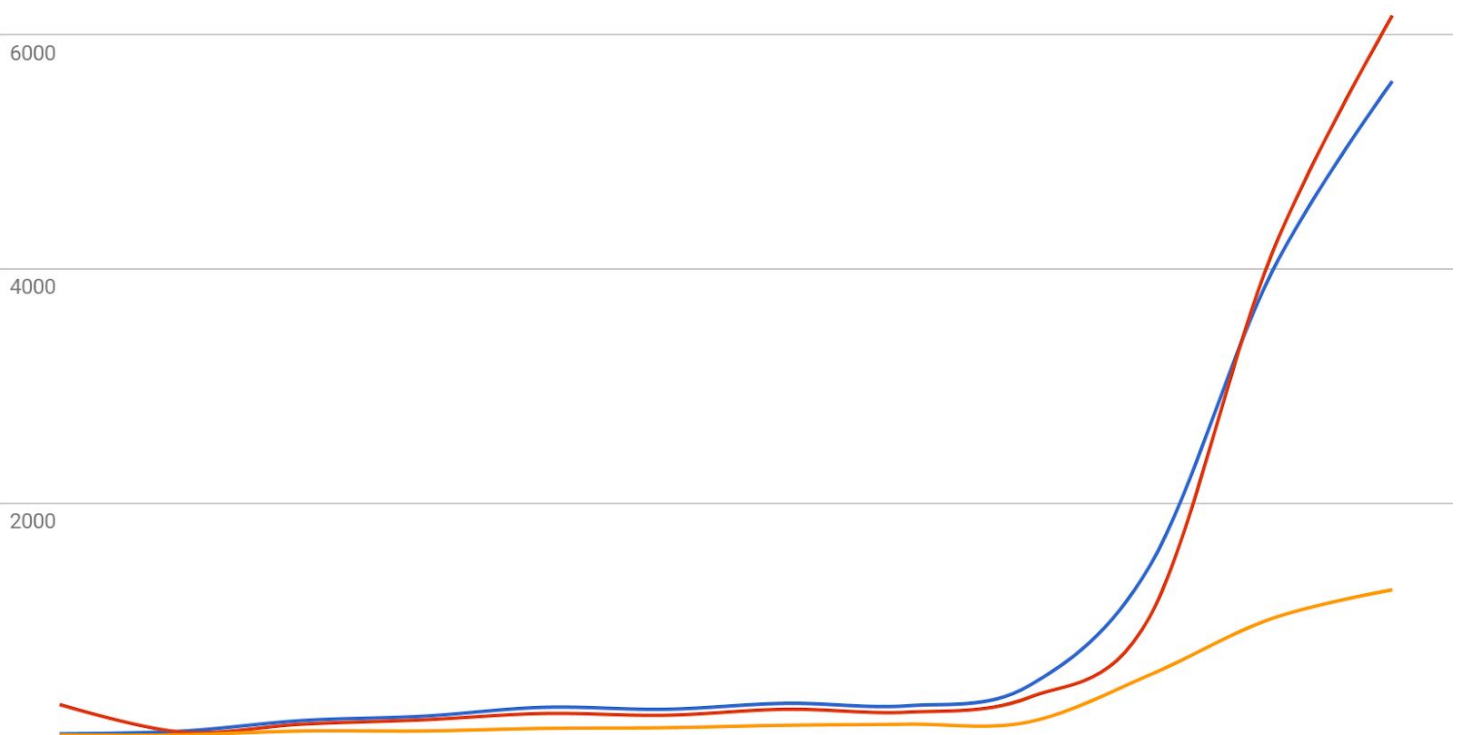
Le temps total de réalisation de toutes les tâches (que l'on peut déterminer en considérant la ligne de la timeline la plus longue) dépend :

- du nombre total de tâches
- de leur temps de réalisation (si la capacité est en T ou en G)
- du nombre de machines disponibles
- de la dépendance entre les tâches (si une tâche longue par exemple précède un grands nombre de tâches)

Tous ces facteurs s'ils sont grands, tendent à augmenter le temps total de la timeline. Le graphique suivant permet de démontrer que plus le temps total tends à grandir (quels qu'en soient les facteurs impliqués), plus la méthode Beta (en rouge) devient stratégiquement meilleur en terme de temps total trouvé sur la timeline que la méthode alpha (en bleu). Cela laisserait supposer que asymptotiquement, la méthode BETA sera toujours de plus en plus meilleur.

### Temps de réalisation

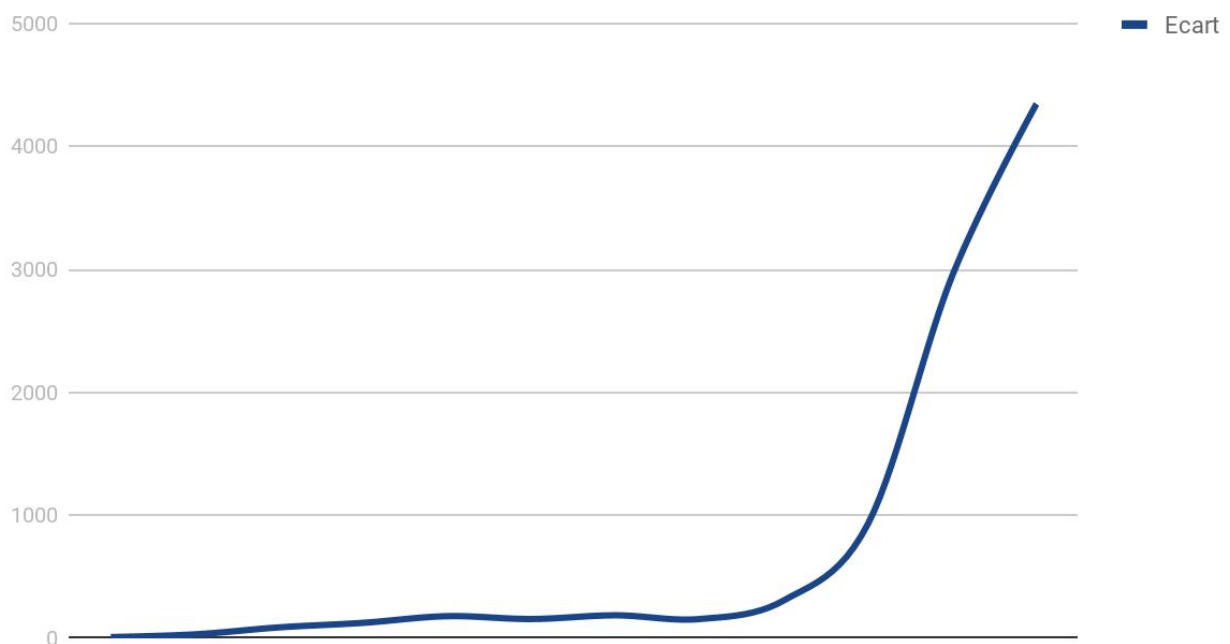
■ Gamma ■ Alpha ■ Beta





	Gamma	Alpha	Beta
320 Tasks	27	278	17
693 Tasks	50	43	18
2428 Tasks	141	110	52
3276 Tasks	178	147	52
4983 Tasks	254	201	74
2971 Tasks	237	186	80
4843 Tasks	288	237	101
1592 Tasks	267	212	110
3443 Tasks	438	338	129
3094 Tasks	1466	1033	532
4451 Tasks	3962	4116	1011

## Écart entre Gamma et Beta

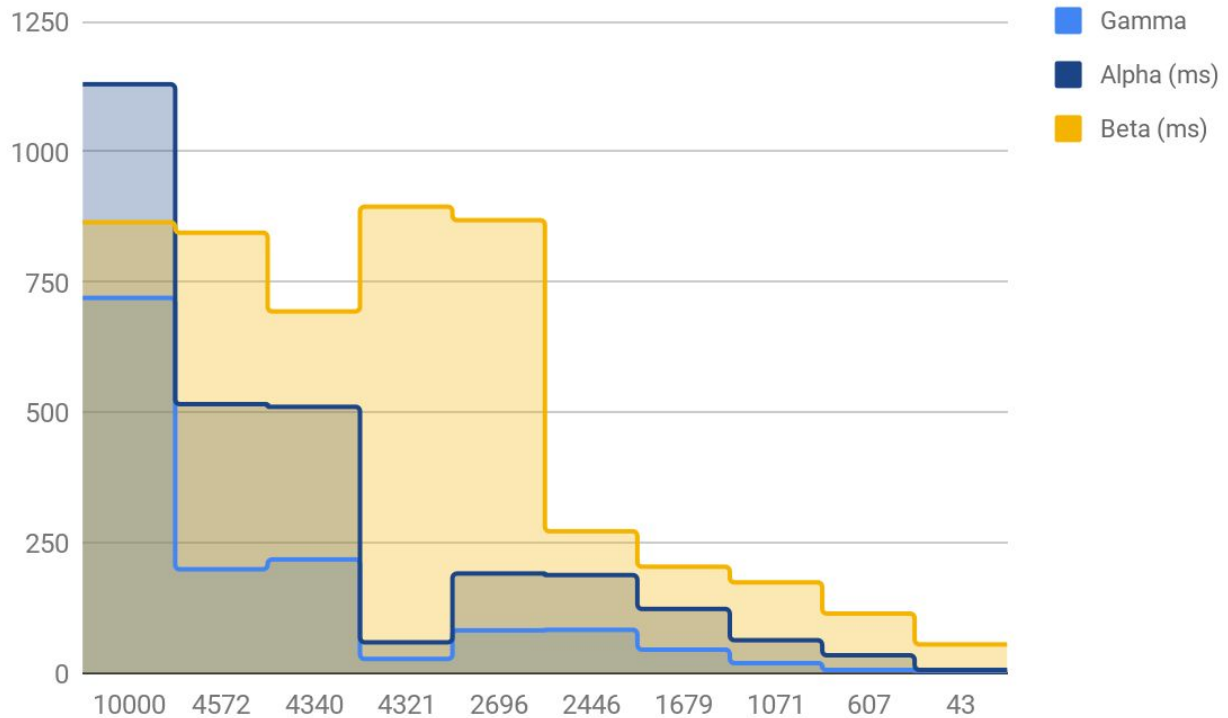


**(ces chiffres ont été calculés avant l'amélioration effectuée sur Beta qui a permis de baisser considérablement les résultats obtenus. Cependant l'argument sur l'écart bien qu'il soit bien plus supérieur, reste le même)**

*Remarque :* Ici, nous avons choisi de désigner l'axe des abscisses par rapport au nombre de tâches, mais comme vous pouvez le constater, ce nombre a une influence partielle sur le temps de la Timeline. Il serait trop long de saisir tous les paramètres influenceurs. Nous les avons donc rangés par ordre croissant du temps de réalisation total.

## Comparaison asymptotique des temps de génération des méthodes

Remarque : le temps de génération de la Timeline pour chaque méthode dépend également de l'ordinateur utilisé.



	Gamma	Alpha (ms)	Beta (ms)
10000	720	1130	865
4572	199	516	845
4340	218	511	694
4321	27	59	895
2696	82	191	869
2446	83	188	272
1679	45	123	204
1071	19	63	174
607	6	34	114
43	4	6	55

Cas exceptionnels : Voici quelques cas exceptionnels obtenus après beaucoup d'essais de configuration (avec beaucoup de dépendances notamment) où la fonction Beta est 4 fois plus lente.

```
Number total of tasks = 3963  
Total time Beta = 52.96667  
Execution Time Beta : 10796 ms  
Total time Alpha = 132.3343  
Execution Time Alpha : 410 ms  
Total time Gamma = 174.44412  
Execution Time Gamma : 161 ms
```

```
Number total of tasks = 4403  
Total time Beta = 42.052635  
Execution Time Beta : 17252 ms  
Total time Alpha = 101.01372  
Execution Time Alpha : 420 ms  
Total time Gamma = 132.04605  
Execution Time Gamma : 227 ms
```

### Limites de Beta :

Beta est en soit un algorithme glouton pour cette résolution du problème NP-Complet de l'ordonnancement.

Les algorithmes gloutons présentent l'avantage d'une conception plus ou moins aisée à mettre en oeuvre et moins coûteuse qu'un algorithme brute de force. Puisqu'il consiste à choisir des solutions locales optimales du problème dans le but d'obtenir une solution optimale globale au problème.

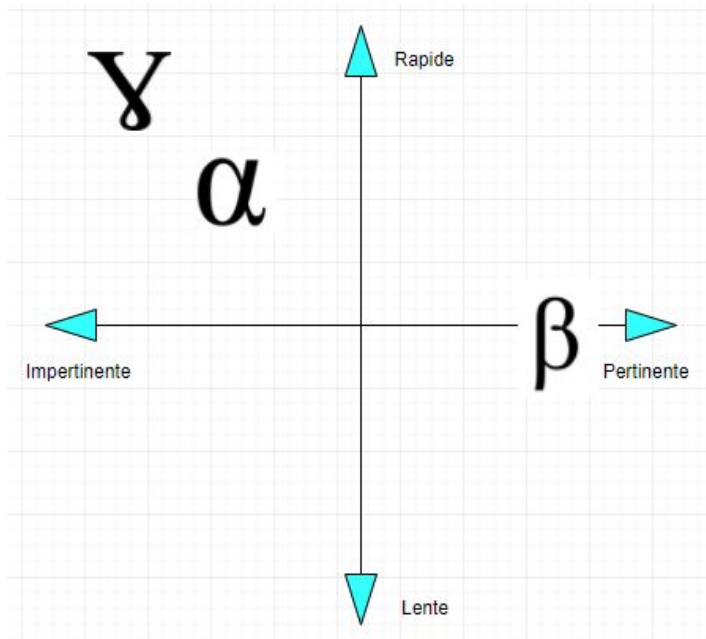
Cependant, le prix à payer est qu'ils ne fourniront pas toujours la solution optimale au problème donné. Pour l'instant, BETA est la solution la plus optimale trouvée bien que plus coûteuse, cependant il serait possible de l'améliorer afin de la rendre plus optimale.

En effet, dans ce cas précis, la solution "locale" choisie ne sera pas forcément la meilleure solution à long terme. En effet, par exemple décider à court terme de mettre une tâche sur une machine plus rapide pour la réaliser plus vite, pourrait bloquer la possibilité d'aller plus vite par la suite au niveau des tâches suivantes: actuellement nous ne pouvons pas nous assurer que la solution optimale localement aura un impact plus positif qu'une autre globalement.

### Résumé

	Rapidité	Optimalité	Stabilité
Alpha	++++	++	+
Beta	++	++++	+++++ (Stable)
Gamma	+++++	+	+++++ (Stable)

Meilleur résultat



## ANNEXE : Classes développées

