TP 2 - Choix 4
Laurent SAVIVANH et Vandy FATHI

I) Donnez et expliquez (motiver) l'importance de ces algorithmes(illustrez la réponse avec exemples)

Ces différents algorithmes sont des algorithmes d'ordonnancement qui permettent à l'ordonnanceur (ou scheduler en anglais) du système d'attribuer de façon optimale les ressources du processeur pour exécuter des tâches (ou processus).
Par exemple, l'ordonnanceur permet aux systèmes d'effectuer le multitasking sur un seul processeur.
Le but de ces algorithmes est de maximiser le débit (nombre de tâches traitées par unité de temps), le taux d'utilisation des ressources du processeur, diminuer le temps d'attente et d'exécution d'une tâche. Ils permettent ainsi à ce que toutes les ressources aient leur temps d'exécution.

Prenons l'exemple d'un cas simple : le **Cloud Computing**.
C'est une technologie qui est de plus en plus utilisée aujourd'hui, elle permet aux utilisateurs d'accéder à des services (software, hardware, platform ...etc) à travers internet. Elle peut donc servir des milliers voir des millions d'utilisateur simultanément.
C'est à ce moment là qu'interviennent les algorithmes d'ordonnancements car en effet ils ont un impact très positif (si l'algorithme est bien codé et bien choisi) sur les performances du service proposées aux utilisateurs lorsqu'il est soumis à une grande charge de travail (par exemple beaucoup d'utilisateur envoient des requêtes lourdes en calcul en même temps).

Considérons ces 5 algorithmes d'ordonnancement :

- First Come First Serve (FCFS)
- Minimum Completion Time (MCT)
- Longest Cloudlet Fastest Processing Element (LCFP)
- Shortest Cloudlet Fastest Processing Element (SCFP)
- Genetic Algorithm (GA)

a) **FCFS:**

Cet algorithme suit le principe du "premier arrivé premier servi". Ce qui définit l'ordre d'exécution des processus est le temps d'arrivée. Cet algorithme est le plus intuitif et le plus naturel.
Dans un restaurant lorsque plusieurs client passe une commande, le premier à recevoir sa commande et le premier à avoir fait sa commande, le second de même et ainsi de suite.

Voyons dans le cas de processus ce qu'il en est.

Processus	Temps d'exécution	Temps d'arrivé du processus
P1	6	2
P2	5	7
P3	3	0
P4	4	1
P5	8	4

On peut voir qu'il y a plusieurs processus P1 à P5. "Arrival time" est l'unité de temps auxquelles les processus arrivent. Burst time est le temps d'exécution du processus correspondant.

- Lorsque P3 arrive à t=0, son exécution commence.
- A t=1, P4 arrive mais il ne peut être exécuté car P3 est en cours d'exécution. Il sera ajouté à la queue qui sera la file d'attente
- A t=2, P1 arrive et sera aussi ajouté à la queue.
- A t=3, P3 a fini d'être exécuté. P4 qui est le suivant est exécuté.
- A t=4, P5 est ajouté à la queue.
- A t=7, P2 est ajouté à la queue. P4 a fini d'être exécuté, P1 est exécuté.
- A t=13, P1 est exécuté, c'est à présent P5 qui est exécuté.
- A t=21, P5 est exécuté, c'est enfin à P2 d'être exécuté.
- A t=26, P2 est exécuté, tous les processus ont été exécutés les uns après les autres.

Si l'algorithme ne s'occupait pas de la gestion des processus en asynchrone, ce serait l'anarchie car tous les processus occupent les mêmes processeurs. Il serait compliqué de tout exécuter en même temps pour des raison de techniques (exemple l'espace de mémoire pour les processeur)

b) Minimum Completion Time (MCT):

Cet algorithme se base sur le temps d'achèvement le plus petit. On l'utilise dans le cas du cloud computing. Concrètement une tâche peut être assignée à plusieurs machines virtuelles, cependant cette allocation sera en fonction du temps d'achèvement de la tâche dans une machine virtuelle. Une tâche sera attribuée à la machine dont le temps d'achèvement sera le plus court. Cela signifie que l'on compare le temps de d'achèvement d'une tâche avec toutes les machines virtuelles.

Mais comment définit-on ce temps d'achèvement ?
Il suffit de savoir quand est-ce qu'une tâche est complètement exécutée dans une machine virtuelle. Il faut considérer deux paramètres:

- Le temps qu'a besoin le processus pour être exécuté **[1]**

- Le temps à partir duquel le processus peut s'exécuter dans la machine virtuelle.**[2]**

[1] Ce temps est intrinsèque à la tâche mais peut varier selon la configuration de la machine virtuelle.
[2] Ce temps dépend du nombre de processus que possède une machine virtuelle. En effet, plus une machine virtuelle possède de tâches, plus le temps d'attente sera grand car il faudra attendre que tous les processus soient exécutés avant de pouvoir exécuter un nouveau processus.
Ainsi on peut définir Cij = Eij + Rj

- Cij = le temps d'achèvement du processus i dans la machine virtuelle j.
- Eij = Le temps d'exécution du processus i dans la machine virtuelle j.
- Rj = Le temps à partir duquel le processus il peut commencer son exécution dans la machine virtuelle j.

Maintenant que l'on sait ce qu'est le temps de d'achèvement, il faut savoir quelle tâche sera allouée à quelle machine virtuelle.
Pour une tâche donnée, on prendra celle dont le temps d'achèvement est le plus court. Ainsi nous savons auprès de quelle machine virtuelle la tâche sera le plus rapidement exécutée et cette machine virtuelle devra exécuter la tâche.

On voit que l'objectif recherché est de faire en sorte d'exécuter chaque tâche le plus rapidement possible.

NB: Lorsqu'une machine virtuelle reçoit une nouvelle tâche à exécuter, son Rj change.

Exemple: Trouver un exemple ou en inventer un mais contacter le prof pour peut être avoir une idée d'exemple pcq ya pas grand chose sur internet.

c) Longest Cloudlet Fastest Processing Element (LCFPE):

Dans une situation où l'homme effectue un projet long et difficile, il choisira la personne la plus compétente. Cet algorithme en fait de même. Quel est le critère de sélection ? Ce critère est le la durée d'exécution d'une tâche. Le but est de minimiser le makespan (le temps qu'il faut pour exécuter toutes les tâches).
Pour ce faire, l'algorithme assignera les tâches les plus longues aux éléments de traitement les plus performants.

d) Shortest Cloudlet Fastest Processing Element (SCFP):

Cet algorithme suit le même principe que **LCFPE** mais effectue l'opposé. Il assigne les tâches les plus rapides à exécuter aux éléments de traitement les plus performants.

e) Genetic Algorithm (GA):

Les algorithmes génétiques sont des algorithme qui se basent sur ce qu'a toujours fait la nature depuis la nuit des temps: **La Sélection Naturelle**
L'algorithme génétique fait partie de la famille des algorithmes évolutionniste (ou évolutionnaire). Les algorithmes évolutionnistes font évoluer un ensemble de solutions lié à un problème. L'objectif est d'obtenir la solution la plus adaptée (voire la meilleure solution). **On utilise ce genre d'algorithme lorsque nous n'avons pas de réponse directe à un problème d'optimisation ou d'estimation**.

Pour comprendre l'algorithme, revoyons les bases de la génétique.
Chaque être vivant possède un génome (un ensemble de gènes qui définissent cet individu). Le meilleur individu est celui qui possède le génome le plus adapté à son environnement, soit celui qui possède les gènes qui composent ce génome.
Au cours de l'évolution de la vie, les espèces se sont adaptées à leurs environnements et seules les espèces de ceux qui possédaient les gènes nécessaires à cette adaptation ont pu se reproduire. Ainsi s'est fait une sélection naturelle. Cependant lorsque ces individus se sont reproduits, il s'est passé ce qu'on appelle un "cross-over". C'est-à-dire qu'il y a eu un mélange du code génétique entre deux individus qui se sont reproduit. Concrètement la génération suivante est composée des gènes des générations précédentes.
Un événement à ne pas négliger dans l'évolution des espèces est la mutation génétique. Qu'est-ce que la mutation génétique ? C'est un phénomène aléatoire qui va modifier le(s) gène(s) d'un individu. Cette mutation peut lui être désavantageuse tout comme elle peut l'aider à être un individu encore meilleur.
Maintenant que l'on a une vision de l'évolution des espèces, essayons d'en faire une allégorie.
L'environnement serait le problème.
Les individus seraient des solutions à ce problème.
Le génome serait le critère qui compose une solution.
Les gènes seraient ce qui caractérise une solution (les critères sur lesquels on se basera pour déterminer si une solution est pertinente).
L'algorithme génétique suit exactement le même principe.

Voici le Pseudocode de l'algorithme génétique:

```
START
1-Generate the initial population
2- Compute fitness
3-REPEAT
  4-Selection
  5-Crossover
  6-Mutation
  7-Compute fitness
8-UNTIL population has converged
```

STOP

Comme dit plus haut, les algorithmes génétiques se basent sur la sélection naturelle.
[1] C'est-à-dire que l'on va faire générer aléatoirement des solutions
[2] On effectue le ou les calculs qui proposent des solutions au problème
[3] C'est là qu'intervient le principe de sélection naturelle.

[4] On effectue une sélection des solutions les plus satisfaisantes (les solutions les plus optimales)

[5] On fera un Cross-Over entre les individus sélectionnés. Le but étant de propager les meilleurs critères.

[6] Les mutations modifieront aléatoirement les critères des réponses, les rendant plus ou moins pertinentes.

[7] On définit un nouvel ensemble de solutions avec les critères sélectionnés.

[8] En répétant cela un grand nombre de fois, on peut prévoir qu'à la fin notre ensemble de solutions sera composé de solutions n'ayant gardé que les critères les plus pertinents. Ainsi, notre but serait de faire évoluer (converger) nos solutions vers une solution idéale.

La solution que l'on choisirait n'est pas forcément la meilleure des solutions mais elle est la solution la plus adaptée au problème.

II] Identifier des critères de comparaison de ces algorithmes

Les critères de comparaison sont évidemment les critères qui nous ferait choisir un algorithme plutôt qu'un autre. Concernant les algorithme pour le scheduling on a:

- **Makespan** (Temps total pour exécuter toute les tâches)
  Effectivement, le but est d'exécuter toutes les tâches le plus rapidement possible.

- **Temps d'exécution** (Temps exacte pour compléter la tâche)
  Il existe deux types d'ordonnancement: préemptif et non-préemptif.
  Un algorithme non-préemptif va allouer une tâche à un processeur, la tâche est complétée seulement lorsque son exécution est terminée sans interruption.
  Un algorithme préemptif va allouer un temps d'exécution à une tâche. Celle-ci peut être interrompue lors de son exécution.

- **Temps d'attente**
  Ce critère est le temps qu'il faut à une tâche avant de voir son exécution commencer.

- **Performance** (Efficacité globale de l'algorithme d'ordonnancement)

- **CPU utilisation** (% du CPU utilisé)
  Est-ce que l'algorithme consomme beaucoup de CPU ?

  // <u>Pour les algorithmes stochastiques (méthode de recherche "aléatoire"):</u>

- **Nombre de paramètres**
  On peut prendre beaucoup de paramètres nous permettant d'avoir des résultats avec le moins d'erreurs et plus précises

- **Pertinence de la solution proposée**
  Dans les problèmes d'optimisation. On veut la solution la plus adéquate

- **Les performances nécessaire pour faire tourner ce genre d'algorithme**
  Ces algorithmes peuvent être gourmands.

III] Etude de Cas

Nous avons utilisé l'IDE Intellij pour faire nos études de cas.

Cas 1 : FCFS

Nous avons choisi d'étudier l'algorithme FCFS dans le cadre d'utilisation du Cloud Computing en utilisant le langage java et l'outils CloudSim pour pouvoir simuler un cloud (un datacenter, un broker, un ou des machines virtuelles et des tâches).

NB : La plupart des fonctions qui ont été utilisées dans cette étude ont été reprises des différents exemples proposés par CloudSim (par exemple les fonctions de création de datacenter ou de cloudlets…).
De plus, les algorithmes FCFS et RR sont par défaut proposés par CloudSim.

Il y a plusieurs étapes essentielles avant de pouvoir simuler un cloud recevant des cloudlets soumis à un algorithme d'ordonnancement (par exemple FCFS ou MCT....):

Etape 1 : Initialisation du package "CloudSim"
Etape 2 : Création du/des DataCenter(s) et Host
Etape 3 : Création du/des Broker(s)
Etape 4 : Création d'un/des VM(s) et d'un/des Cloudlet(s)
Etape 5 : Envoie des VM et Cloudlet au(x) Broker(s)
Etape 6 : Début de la Simulation
Etape 7 : Fin de la simulation
Etape 8 : Affichage des résultats

Lors de la création des DataCenters et des VMs, il est important de bien avoir un datacenter et un host avec les bonnes caractéristiques pour pouvoir accueillir un certain nombre de VMs.
En effet, si par exemple l'host n'a pas assez de RAM, des VM ne pourront pas être créer en conséquence (il y a bien sûr d'autre paramètre à prendre en compte).

Pour nos deux cas, nous avons utilisés les caractéristique suivants:

Tableau 1 : Caractéristiques Host

Host	1
RAM	8000 MB
Storage	1 000 000 MB

bw	10 000
MIPS	1000

Tableau 2 : Caractéristiques du Datacenter

architecture	x86
os	Linus
VMM	Xen
time_zone	10
cost	3

Tableau 3 : Caractéristiques d'un VM

Taille	10 000 MB
RAM	512 MB
MIPS	200
bw	1000
Nombre de CPU	1

NB : Dans le code, nous ne pouvons pas créer plus de 5 VM car le datacenter a besoin de plus de ressources. Si vous voulez faire plus de VM, il faut changer les caractéristiques du datacenter pour qu'il puisse supporter plus de 5 VM.
Ou sinon vous pouvez baisser les caractéristiques des VM.

Tout d'abord nous allons tester l'algorithme FCFS avec une machine virtuelle et 10 cloudlets avec pour BurstTime, un temps aléatoire (ici elle correspond au paramètre Time)

```
========== OUTPUT ==========
Cloudlet ID    STATUS    Data center ID    VM ID    Time    Start Time    Finish Time    Response Time
    0        SUCCESS           2            0       3,79       0,1        3,89
    1        SUCCESS           2            0       4,47       3,89       8,36
    2        SUCCESS           2            0       0,57       8,36       8,93
    3        SUCCESS           2            0       3,97       8,93       12,9
    4        SUCCESS           2            0       3,4        12,9       16,3
    5        SUCCESS           2            0       4,09       16,3       20,39
    6        SUCCESS           2            0       0,73       20,39      21,11
    7        SUCCESS           2            0       1,45       21,11      22,56
    8        SUCCESS           2            0       1,09       22,56      23,66
    9        SUCCESS           2            0       4,79       23,66      28,45
```

Tableau 4 : Résultat pour une machine virtuelle et 5 cloudlets

L'algorithme FCFS étant non-préemptive, les tâches s'exécutent successivement une par une.
Dans cet exemple, nous avons choisi des cloudlets ayant un temps d'exécution aléatoire entre 0 et 10 secondes.

NB : Datacenter_0 à pour id 2. En effet, les id 0 et 1 sont automatiquement attribués au processus d'arrêt de la simulation et pour les informations concernant la simulation.

Testons maintenant avec plusieurs machines virtuelles par exemple 5 ainsi qu'une vingtaine de cloudlets.

```
========== OUTPUT ==========
Cloudlet ID    STATUS    Data center ID    VM ID    Time    Start Time    Finish Time
    0        SUCCESS           2            0       5         0,1          5,1
    1        SUCCESS           2            1       5         0,1          5,1
    2        SUCCESS           2            2       5         0,1          5,1
    3        SUCCESS           2            3       5         0,1          5,1
    4        SUCCESS           2            4       5         0,1          5,1
    5        SUCCESS           2            0       5         5,1          10,1
    6        SUCCESS           2            1       5         5,1          10,1
    7        SUCCESS           2            2       5         5,1          10,1
    8        SUCCESS           2            3       5         5,1          10,1
    9        SUCCESS           2            4       5         5,1          10,1
```

Nous pouvons voir que les cloudlets ont été assignés à un VM lorsque ce dernier n'était pas occupé.
Ainsi les 5 VMs ont été assignées une cloudlet à l'instant t = 0, puis suivant l'algorithme FCFS, dès que le VM a fini d'exécuter son cloudlet, il relâche les ressources qui ont été attribuées à ce cloudlet pour accueillir un nouveau cloudlet.
On remarque que les cloudlets ont été exécutés bien plus plus rapidement car ils ont été exécutés par les différents VM disponibles.

En conclusion, nous avons vu que dans un environnement de cloud computing, l'algorithme d'ordonnancement est assez efficace pour attribuer les ressources de l'host aux cloudlets Même si cela n'est pas fait de façon optimale (en comparaison avec d'autre algorithmes qui peuvent être plus efficace selon , l'algorithme FCFS montre l'importance de ces algorithmes d'ordonnancement dans un environnement de Cloud Computing.

Cas 2: Genetic Algorithm

Dans cette étude de cas, nous avons étudié le problème du timeTable (d'emploi du temps) résolu avec l'algorithme génétique

NB: Le TP a été réalisé en Java..

Voici notre situation:

Je suis étudiant et j'ai une liste de matières. Je dois choisir un créneaux sur chaque matière, cependant j'ai deux types contraintes à respecter.

Les contraintes imposées:

- Il faut que tous les cours figurent sur mon emploi du temps.
- Je ne peux pas avoir deux fois le même cours dans la semaine.

Les contraintes personnelles:

- Je désire passer le moins de jours à l'école
- Je désire avoir le moins d'heures vides entre deux matières.

On suppose qu'il y a cours que du lundi au vendredi de 9h à 17h

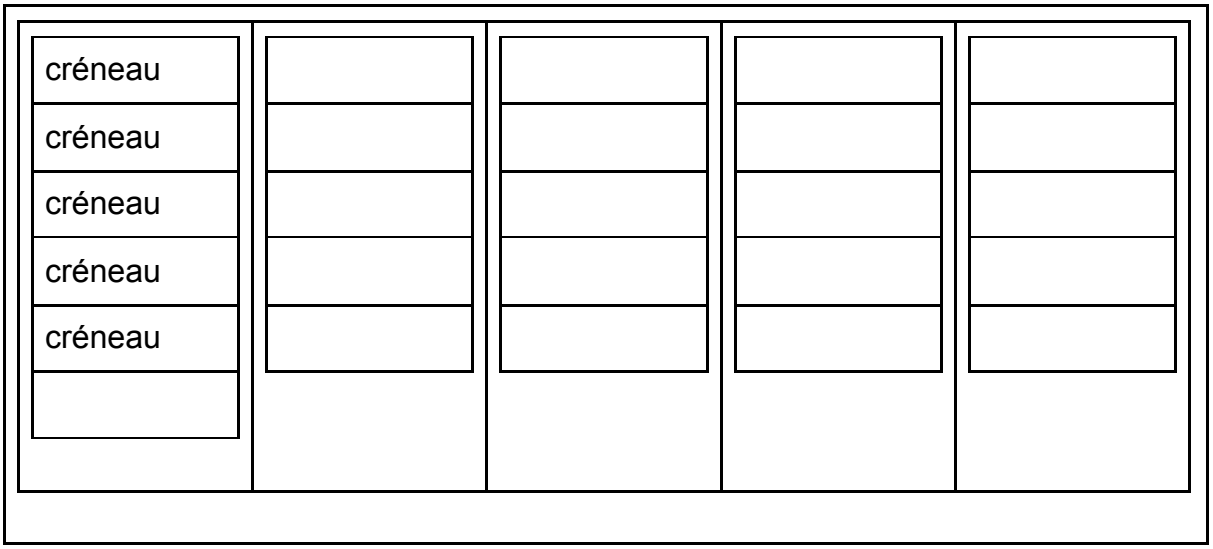
Objectif: Je recherche l'emploi du temps qui répond le plus à mon problème.

Voici un tableau montrant comment ont été pensées les relations.

Un emploi du temps est relié à plusieurs jours, qui sont eux-mêmes reliés à des créneaux.

Emploi du temps

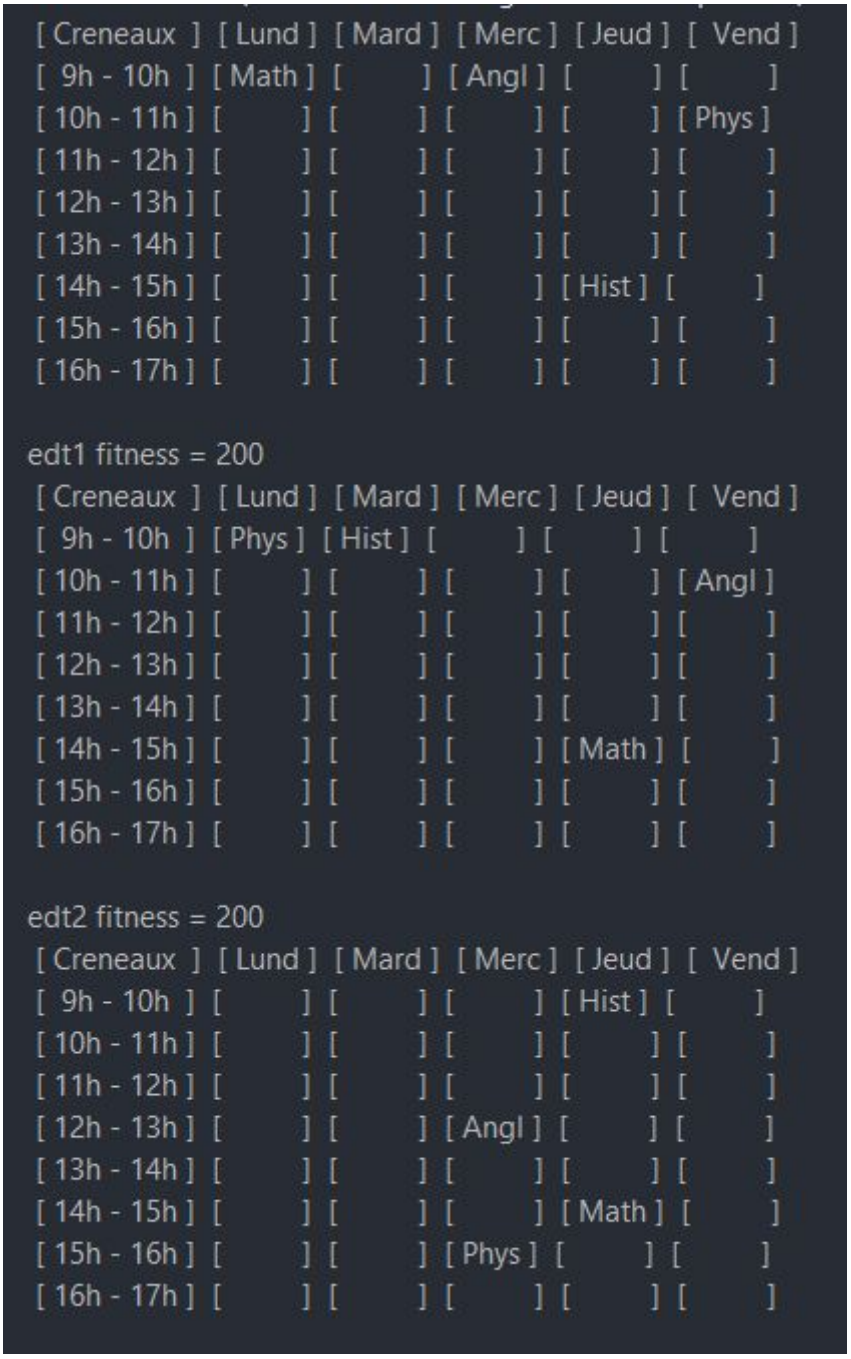
Jour	Jour	Jour	Jour	Jour
créneau				
créneau				
créneau				



Nous avons, de façon totalement arbitraire, établie la list des matières et des créneaux disponibles:

- Mathématiques:
    - Lundi: 9h-10h:
    - Mardi: 14h-15h
    - Jeudi: 9h-10h
    - Jeudi: 14h-15h
- Physique:
    - Lundi: 9h-10h
    - Lundi : 12h-13h
    - Mercredi: 15h-16h
    - Vendredi: 10h-11h
- Anglais:
    - Mardi: 11h-12h
    - Mercredi: 9h-10h
    - Mercredi: 12h-13h
    - Vendredi: 10h-11h
    - Vendredi: 14h-15h
- Histoire:
    - Lundi: 14h-15h
    - Mardi: 9h-10h
    - Mercredi: 9h-10h
    - Jeudi: 11h-12h
    - Jeudi: 14h-15h

Voici l’exemple de quelque emploi du temps.



A savoir nous sommes avons utilisé qu'une seule population initiale.

<u>Détermination du Fitness:</u>

Nous nous sommes dit qu'il est préférable de passer moins de temps à l'école au dépend des heures vides entre les cours. Ainsi nous avons apporté un malus au critère du nombre de jour en cours.

Nous avons finalement:

Fitness = ∑nombreJourCours * PénalitéJoursCours + ∑NombreHeureVide * PénalitéHeureVide

Nous avons fixé PénalitéJoursCours = 50;
PénalitéHeureVide = 5;

Ainsi plus il y a de jour de cours, plus le fitness est élevé, cependant cela ne signifie pas que c'est meilleur.
Dans notre cas, plus le fitness est petit, plus la solution est appropriée.

```
public int fitness(){
    int fitness=0;
    int PenaliteJourDeCours = 50;
    int PenaliteCreneauxVide = 5;
    for (int i = 0; i <5 ; i++) {
        fitness += CreneauxVideJour(i) * PenaliteCreneauxVide;
    }
    fitness += JoursDeCours() * PenaliteJourDeCours;
    return fitness;
}
```

C'est donc à partir de ce fitness que nous allons pouvoir faire la sélection de nos solutions.

La Sélection:

Cette étape consiste à garder les meilleurs solutions actuelles.
Ici, nous avons décidé de trier dans un tableau les solutions par ordre croissant de fitness. Les meilleurs solutions se trouveront donc dans les premières solutions.
Mais nous sélectionnons aussi les solutions valides.
En effet il se peut que parmis nos solutions, certaines ne sont pas valides (elles possèdent des cours en moins ou en trop).
Une fois la sélection faite nous nous retrouvons à présent avec une population réduite.

```java
public void Selection(){
    EmploiDuTemps edtr;
    //Ce compteur sert à sélectionner les 4 individus
    int ct=0;
    for (int i = 0; i < this.solution.length; i++) {
        if(!this.solution[i].edtNonValable()){
            edtr = this.solution[i];
            this.solution[ct] = edtr;
            ct++;
        }
        if(ct==4) return;
    }
}
```

NB: Dans le cadre du TP, la population était de 8 emplois du temps, et la sélection prenait 4 solutions sur les 8.
Nous avons à présent besoin de repeupler notre population. C'est la qu'intervient le CrossOver.

Le CrossOver:

Cette étape est primordiale pour la survie des bons gènes! En effet, c'est en transmettant son code génétique que les générations suivantes auront un bagage les favorisant à leur environnement. Comme nous avons nos 4 premières du tableau de solution, il suffisait de modifier les 4 dernières du tableau de solution.
Pour ce faire, nous avons décider de prendre les moitiés des tableaux de les joindres pour donner les tableaux enfant.
Ainsi soit 2 tableau parents, tab1 et tab2: (les proportions ont été réduites mais le principe est le même)

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

tab2:

2	2	2	2
2	2	2	2
2	2	2	2
2	2	2	2

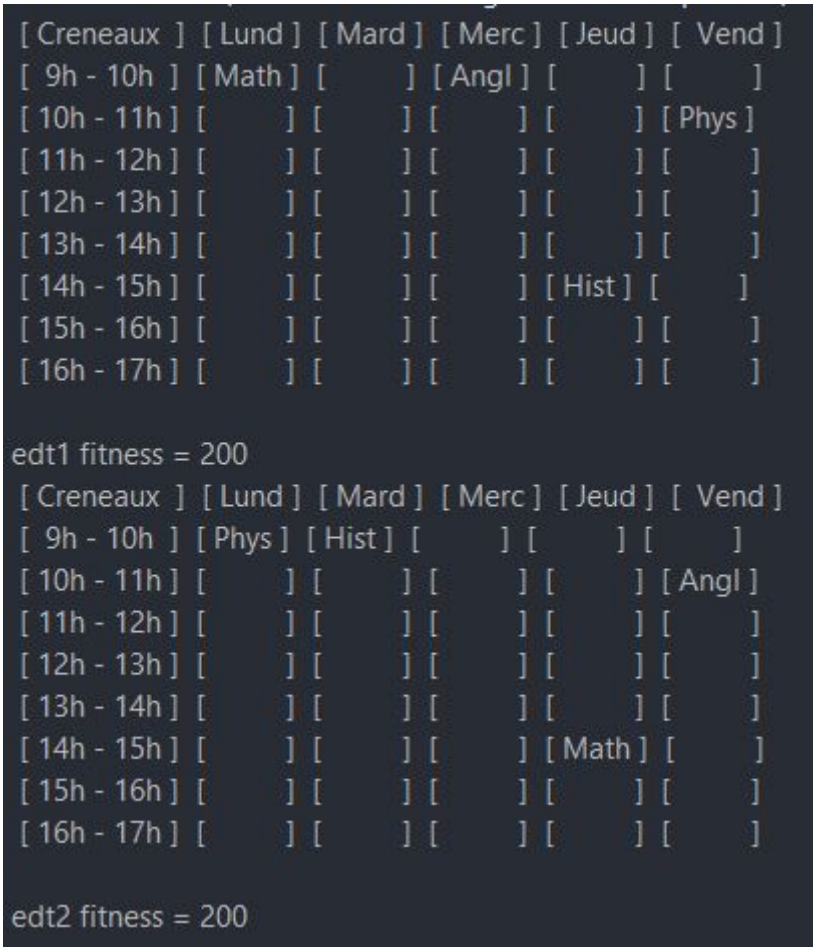
Ainsi on pouvait soit avoir CrossOver(tab1,tab2) =

1	1	2	2
1	1	2	2
1	1	2	2
1	1	2	2

Ou bien CrossOver(tab2, tab1) =

2	2	1	1
2	2	1	1
2	2	1	1
2	2	1	1

Cependant nous avons un cas particulier. Il se peut que malgré toutes les combinaisons faites entre les parents, la plupart des unions donnent naissances à des solutions non valides.
Prenons deux emploi du temps de notre population de solution initiale.



Si l'on fait CrossOver(tab1, tab2) on a une solution de la forme:

lundi	mardi	mercredi	jeudi	vendredi
math				
				Angl
			math	

On voit que l'union donne quelque chose d'invalide. Il manque des cours mais en plus il y a des cours en trop.

De même avec CrossOver(tab2,tab1):

lundi	mardi	mercredi	jeudi	vendredi
phys	hist	Angl		
				Phys
			hist	

Nous rencontrons de nouveau ce problème. Il se peut que la sommes des solutions valides issue d'une union soit inférieur à 4. Hors nous avons impérativement besoin de repartir d'une population de 8. Nous avons donc décidé de générer aléatoirement les solutions restantes.

La génération de solutions aléatoires:
Nous allons vous parler de l'importance l'utilisation de HashMap.
Dans un HashMap nommé "horaireAuto" est stocké tous les créneaux disponibilités des jours et des matières concernés.
Le HashMap se présente ainsi:
[matière -> [jours -> créneaux]]

```
private  final HashMap<String, HashMap<String, String>> horaireAuto = new
HashMap<>();
```

Dans le HashMap principal, les clés sont les matières et les valeurs sont des HashMaps.
Ces HashMaps ont pour clés les jours où se trouvent les matières et ont pour valeurs les créneaux dans les journées auquel avait lieu les cours.

Ainsi, il suffirait de choisir une une matière et on pouvait déterminer l'ensemble des jours et des créneaux associé à ces matières.
L'autre avantage était que l'ensemble des créneaux était représentait par des indices de 1 à 8, et les jours par des indices de 0 à 4.

Prenons le cas des cours de mathématique comme donné en page 11.

[“Math” → { [0 → 1]; [1 → 6]; [3 → 1#6] }]

Keys → Values,Keys→ Values
Matières → Jours → Créneaux

Ce qui affiché ci dessus représente la liste des créneaux et des jours disponible pour la matière “Math”.
Nous n’avions plus besoin de nous soucier de l’indice de créneaux du Cours de Mathématique pour un jour particulier. Toutes les relations étaient déjà faites.
Donc lorsqu’on choisissait (en clé du grand HashMap) “Math”, on avait (en valeurs du grand HashMap) les jours. Il suffit de piocher aléatoirement parmis ces valeurs pour choisir un jour parmis lesquelles il y a mathématique.
Mais Les valeurs du grands HashMaps sont les Clé des sous HashMaps, et ce sont ces sout HashMaps qui lient les jours aux créneaux.
En piochant aléatoirement jours (parmis les clé des sous HashMaps) on pouvait choisir les créneaux (soit les valeurs des sous HashMaps) correspondant.

On peut voir que la dernière valeur se présente comme: [3 → 1#6]
Cela veut dire que pour le jour 3 (soit jeudi) il y a 2 créneaux. Le créneaux “1” (9h-10h) ou le créneaux 6 (14h-15h)
On les a séparé par un # car nous nous sommes inspiré du TP précédent. Il faut choisir aléatoirement parmis ces deux créneaux nous avons donc “split()” et fait un choix aléatoire.

Conclusion de cette partie:

Pour générer un emploi du temps aléatoirement, il faut piocher aléatoirement les jours et les créneaux pour toutes les matières (sinon on aurait un solution non valide)

Si on veut faire une modification sur une matière il suffit de savoir la matière que l’on va modifier.

Hors une mutation c’est justement une modification précise aléatoire.

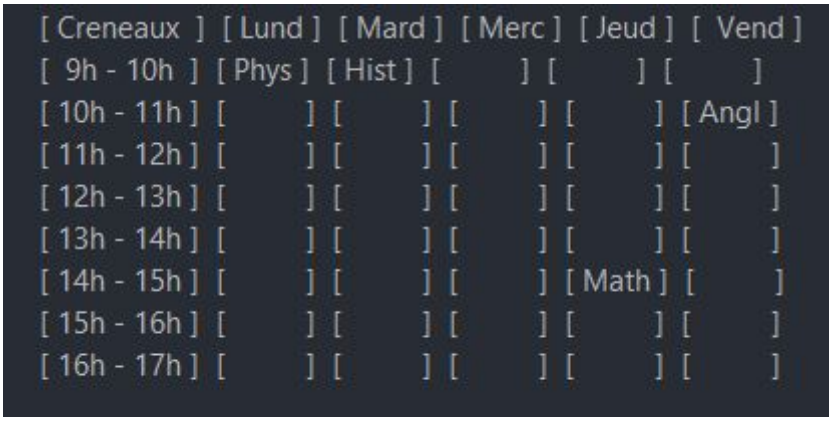
<u>La Mutation:</u>
Cette étape peut paraître farfelue mais c’est la plus importante de l’algorithme.
En effet une mutation peut soit se passer, soit ne pas se passer.
Dans le cas où rien ne se passe le cours de choses continue.
Dans le cas où il y a une mutation 3 possibilités en ressortent:

- 1) soit la modification n’a pas un si grand impact qu’elle est négligeable
- 2) soit elle a un impact mauvais que l’on l’individu (la solution) qui en ressort n’est pas adapté à son environnement et finira pas être rejeté par la sélection.
- 3) soit elle a un impact positif rendant l’individu (la solution) plus approprié à son environnement.

Nous recherchons justement cette troisième possibilité. Afin d'en tirer le meilleur des solutions, le facteur chance des mutations a un impact clairement pas négligeable.

Mais que pourrait représenter une mutation dans le cadre de l'emploi du temps ?
Une emploi du temp subit une mutation si une de ces matières change de créneaux de façon aléatoire.

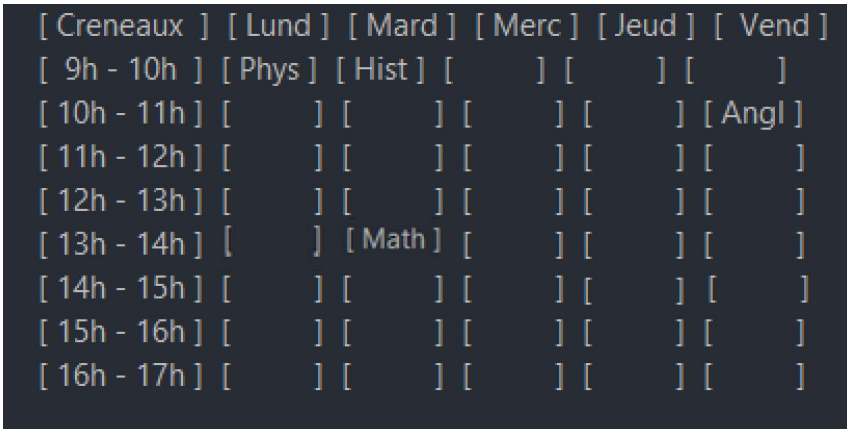
Prenons cet emploi du temps



Son fitness = $'4_{\text{nombre de jours}} * 50_{\text{PenalitéJoursCours}} = 200$
Notre algorithme va piocher un emploi du temps et une journée aléatoirement. Si cette journée est vide, aucune mutation ne sera faite.
Si elle n'est pas vide. Elle piochera aléatoirement parmis les matières présentes dans la journée.
Ce créneaux sera vidé, et un nouveau créneaux sera aléatoirement choisis grâce au HashMap.

NB: Il se peut que le nouveau créneaux choisis soit celui précédemment retiré, dans ce cas on considère que l'emploi du temps n'a subit aucune mutation

Supposons que la mutation soit celle-ci:



Le cours de mathématique du jeudi 14h s'est déplacé au mardi 14h. Cette mutation a eu pour conséquences:
fitness = $3_{NombreJours}$ * $50_{PénalitéJoursCours}$ + $3_{NombreHeurevide}$ * $5_{PénalitéHeureVide}$ = 165
Cette mutation à rendu meilleur cette cet emploi du temps.

La solution finale:

Il y a deux façons de déterminer la solution:

- Soit on se base sur le nombre de fois que l'on effectue l'algorithme génétique.
- Soit on pose une condition sur le fitness.

Nous nous sommes naturellement dirigés vers la première possibilité puisque nous pouvons savoir une valeur pertinente pour la fitness de telle sorte à poser une condition dessus.

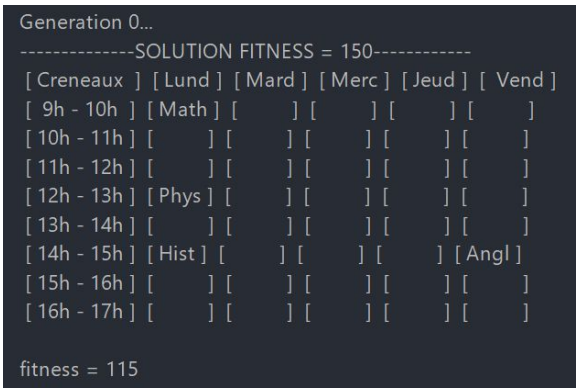
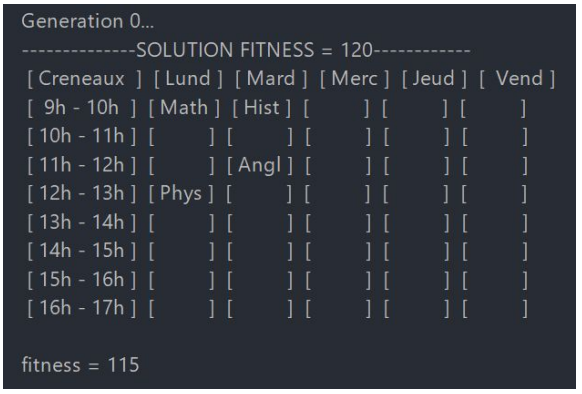
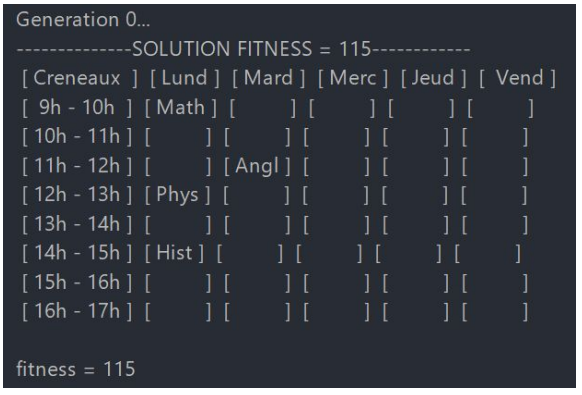
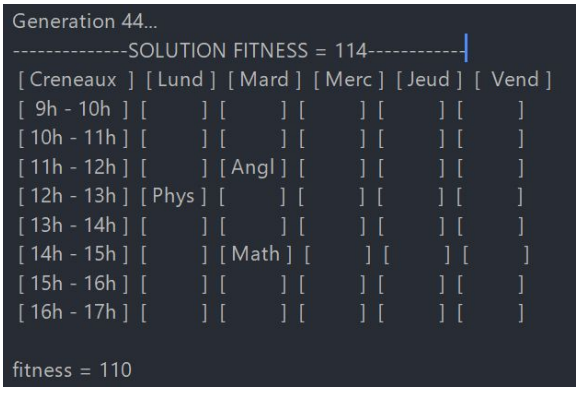
Pour l'ensemble des tests effectués, nous avons gardé la même population initiale

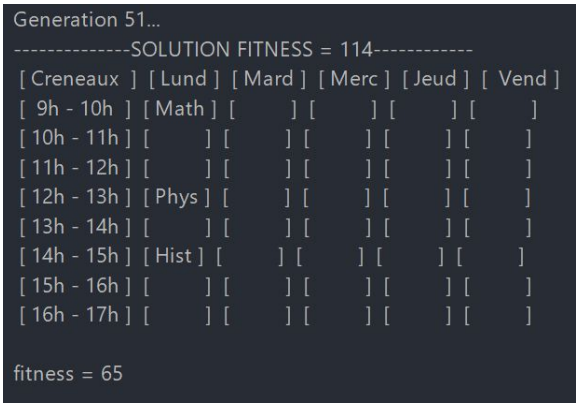
Génération	Solution
5 Le fitness est relativement bas.	----------SOLUTION 5 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [Math] [Hist] [ ] [ ] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [ ] [11h - 12h] [ ] [Angl] [ ] [ ] [ ] [12h - 13h] [Phys] [ ] [ ] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [ ] [ ] [ ] [ ] [ ] [15h - 16h] [ ] [ ] [ ] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 115
20 Cet emploi du temps a le même fitness que le précédent mais n'est pas le même	----------SOLUTION 20 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [ ] [Hist] [ ] [ ] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [ ] [11h - 12h] [ ] [Angl] [ ] [ ] [ ] [12h - 13h] [Phys] [ ] [ ] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [ ] [Math] [ ] [ ] [ ] [15h - 16h] [ ] [ ] [ ] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 115

50 On peut estimer une certaine convergence de la fitness mais les emplois du temps restent différents.	----------SOLUTION 50 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [Math] [ ] [ ] [ ] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [ ] [11h - 12h] [ ] [ ] [ ] [ ] [ ] [12h - 13h] [Phys] [ ] [ ] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [Hist] [ ] [ ] [ ] [Angl] [15h - 16h] [ ] [ ] [ ] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 115
100 La fitness est plus élevé mais l'emploi du différents	----------SOLUTION 100 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [ ] [ ] [ ] [Math] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [ ] [11h - 12h] [ ] [ ] [ ] [ ] [ ] [12h - 13h] [ ] [ ] [Angl] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [ ] [ ] [ ] [Hist] [ ] [15h - 16h] [ ] [ ] [Phys] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 130
100 Avec la même population initiale on a une fitness différentes et une solution différente.	----------SOLUTION 100 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [ ] [Hist] [ ] [ ] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [Phys] [11h - 12h] [ ] [Angl] [ ] [ ] [ ] [12h - 13h] [ ] [ ] [ ] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [ ] [Math] [ ] [ ] [ ] [15h - 16h] [ ] [ ] [ ] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 115
100 De même	----------SOLUTION 100 GENERATION---------- [Creneaux] [Lund] [Mard] [Merc] [Jeud] [Vend] [9h - 10h] [Phys] [Hist] [ ] [ ] [ ] [10h - 11h] [ ] [ ] [ ] [ ] [ ] [11h - 12h] [ ] [Angl] [ ] [ ] [ ] [12h - 13h] [ ] [ ] [ ] [ ] [ ] [13h - 14h] [ ] [ ] [ ] [ ] [ ] [14h - 15h] [ ] [Math] [ ] [ ] [ ] [15h - 16h] [ ] [ ] [ ] [ ] [ ] [16h - 17h] [ ] [ ] [ ] [ ] [ ] fitness = 115

On constate que les valeurs des fitness convergent vers 115 (de façon expérimentale). On remarque aussi que malgré le nombre de génération, on a pas tout le temps la même solution.

Maintenant qu'on a une idée de valeur pour la fitness on a essay l'autre méthode.

Fitness	Solution
150 La solution est trouvé à la génération 0	
120 De même mais la solution est différente	
115 De même que 120	
114 On obtient une solution invalide, cependant il a fallut 44 génération	

114 Solution invalide cependant il a fallut 51 génération	

On remarque que dans notre contexte, se baser sur le fitness n'est pas la façon la plus approprié. Nous avons du dépendre des générations pour connaître une valeur de fitness quelque peu pertinente.
De plus, on remarque qu'en dessous de 115, on obtient des solutions qui sont invalides. On peut comprendre qu'ici, la valeur du fitness induit en erreur.*

Conclusion:

On voit qu'effectivement l'algorithme génétique est très efficace dans la résolution de problèmes d'optimisation et d'estimation. Peut-être que nous, humains, aurions pû déduire ces résultats, cependant si nous devions le faire avec 10 matières ce serait beaucoup trop long. C'est la que nous nous faisons devancer par cet algorithme. Cependant on peut comprendre qu'à cause du facteur de mutation, on a pas toujours les mêmes solutions finales, mais on remarque que souvent les mêmes solutions apparaissaient et cela est du à la population initiale surtout au contraintes imposées.

IV] Avantages et Désavantage

Nous avons choisi les algorithmes FCFS (First Come First Served) et génétique

Algorithme	Avantages	Désavantages
FCFS	Implémentation Simple Facile à comprendre	● Non-préemptive ● Simple mais pas efficace ● Le temps moyen d'attente est élevé. ● Pas de priorité selon les tâches ● Consommation des ressources pas très optimale
Algorithme Génétique	- Permet d'avoir une solution adaptée - Peut prendre en compte un grande nombre de paramètres - Simple à comprendre et à implémenter - Adapté pour résoudre des problèmes d'optimisations et d'estimations	- La quantité de calculs à faire avant d'avoir une réponse peut prendre du temps en fonction d'un contexte (le temps pour qu'il y ait une convergence) - La réponse final dépend de la population initiale - On ne peut pas garantir que la solution obtenue soit la solution optimal - Les solutions ne sont pas toujours pareils. On peut avoir une solution lié un contexte local (un minimum local)

Référence

Algorithme d'Ordonnancement :
Ordonnancement dans les systèmes d'exploitation — Wikipédia
Scheduling (computing)
Tahani Aladwani , "Types of Task Scheduling Algorithms in Cloud Computing Environment**"**

CloudSim :
Mr. E. Rajesh, Mr. J. Mahalakshmi, "Optimization of Resource Allocation using FCFS Scheduling in Cloud Computing"

Algorithme pour l’algorithme génétique:
https://www.youtube.com/watch?v=FCVWXkSZlNU&t=282s