# Recherche Operationnelle Projet:

## Programme de recherche tabou et algorithme génétique

BLANCHARD Edouard
BLASSELLE Louis

# Introduction

Ce projet a pour but de développer un programme utilisant une recherche avec tabous et un algorithme génétique afin de résoudre un problème d'optimisation combinatoire. Des fichiers texte nous sont donnés en entrée et nous devrons les importer dans notre programme afin de les traiter.
Ces fichiers sont composés de la manière suivante:
la première ligne possède le nombre de tâche à ordonnancer
la ou les lignes suivantes possèdent le temps de traitement de chacune des tâches.
La ou les lignes suivantes représentent les temps de traitement
La ou les lignes qui suivent représent une matrice de temps de réglage entre les différentes tâches.

# Première étape: création de la structure du programme

Nous avons tout d'abord choisi notre langage de programmation: le JAVA nous semblait approprié. Il a donc fallu créer notre projet sur IntelliJ, puis créé un espace de travail gitHub afin de pouvoir travailler en parallèle.
Nous avons appliqué un pattern MVC, dans le cadre d'une amélioration future (mise en place d'une IHM par exemple).

# Deuxième étape, création d'un parser fonctionnel

Le développement d'un parser nous permet de lire directement les fichiers entrés en paramètre. Il y a une fonction pour chaque élément à extraire du fichier afin de simplifier les code et permettre une possible évolution.

# Troisième étape, développement des deux algorithmes en parallèle.

## Algorithme de recherche tabou

Pour réaliser notre algorithme de recherche avec tabous, nous allons tout d'abord générer une solution aléatoire sous la forme d'un ArrayList<Integer> représentant l'ordre des tâches à effectuer.

On appelera voisin toute solution que l'on peut trouver en intervertissant deux tâches.
Ex : {1,2,3,4,5} et {1,3,2,4,5} sont voisins car les tâches 2 et 3 on été switchés.

On générera ensuite un tableaux d'entiers à deux dimensions contenant la différence de coût entre la solution actuelle et tous les voisins.
Ex :

| | tâche 1 | tâche 2 | tâche 3 |
|---|---|---|---|
| tâche 1 | 0 | 27 | -11 |
| tâche 2 | 27 | 0 | 2 |

| | | | |
|---|---|---|---|
| tâche 3 | -11 | 2 | 0 |

Ici, on améliore la fonction objective de 27 si on interverti la tâche 1 et 2, mais on la dégrade de 11 si on interverti la tâche 2 et 3.

On va alors sélectionner le meilleur duo qui n'est pas présent dans la liste tabou.
La liste tabou est une liste FIFO de taille maximale 7, contenant des duo de deux tâches.

On ajoute le duo à la liste tabou.

Si la nouvelle solution est meilleure que l'ancienne, on la place dans la variable bestSolution.

Les résultats obtenus sont très satisfaisants :

```
rech tab 401: 1512 avec [12, 7, 13, 3, 9, 10, 2, 1, 5, 11, 6, 4, 14, 15, 8]
La meilleur solution a été trouvée en 2137 itérations

rech tab 408: 1654 avec [7, 8, 5, 3, 13, 4, 11, 15, 1, 6, 14, 2, 9, 10, 12]
La meilleur solution a été trouvée en 1102 itérations

rech tab 501: 2625 avec [16, 1, 4, 22, 5, 14, 13, 15, 11, 9, 24, 23, 21, 12, 19, 25, 18, 3, 7, 17, 8, 2, 6, 20, 10]
La meilleur solution a été trouvée en 17292 itérations

rech tab 601: 3477 avec [20, 30, 5, 33, 21, 32, 22, 15, 28, 27, 35, 3, 34, 10, 11, 25, 12, 16, 8, 26, 19, 1, 23, 4, 24, 13, 18, 14, 31, 17, 6, 7, 29, 9, 2]
La meilleur solution a été trouvée en 81794 itérations

rech tab 605: 3581 avec [4, 8, 20, 17, 14, 23, 5, 27, 2, 24, 31, 6, 19, 11, 15, 28, 22, 35, 16, 18, 9, 7, 12, 3, 21, 33, 1, 30, 32, 25, 29, 13, 34, 26, 10]
La meilleur solution a été trouvée en 46385 itérations

rech tab 701 : 4547 avec [2, 29, 40, 44, 13, 12, 22, 17, 7, 23, 10, 19, 33, 9, 20, 32, 39, 41, 36, 35, 3, 1, 34, 16, 37, 14, 30, 11, 21, 6, 31, 38, 15, 43,
La meilleur solution a été trouvée en 88024 itérations

Process finished with exit code 0
```

## Algorithme génétique

Pour l'algorithme génétique nous avons pris quelques décisions:
La première sélection des individus utilisera la sélection par tournoi.
Il y aura 50 tournois de 100 individus, chaque tournoi renverra le meilleur individus
On effectuera 10 000 itération de reproduction.
Le facteur de mutation est placé à 50.
On a déclaré toute nos variable au début de la fonction.
Le fitness sera calculé de la manière suivante: plus le temps est petit, meilleur est l'individu.

```
/*Les trois variables ci dessous permettent de configurer notre algorithme génétique
 * en définissant la taille des tournois, le nombre de mutations et le facteur de mutation */
int nombreTournoi=50; // une population de 100 individus suffit largement
int adversaireParTournoi=100;
int nombreIterationGenetique=100000;
int facteurMutation=50;

/* on déclare ensuite les tableaux que nous allons utiliser */
int tempsOptimal[]=new int[ensembleTache.getnbTaches()];
int tempOptimalBeforeChange[]=new int[ensembleTache.getnbTaches()];
int tabIndividus[][]= new int[nombreTournoi][ensembleTache.getnbTaches()];
int tabEnfants[][]= new int[nombreTournoi][ensembleTache.getnbTaches()];
int tableauParentEnfant[][]= new int[nombreTournoi*2][ensembleTache.getnbTaches()];
int tabTampon[]= new int[ensembleTache.getnbTaches()];

int tempVal; // une variable tampon
int random; // une variable qui va etre utilisé pour les brassages dans le tournoi
```

La phase de tournoi se situe dans la boucle ayant pour paramètre "tournoi", on va alors effectuer le nombre voulu de tournoi contenant n "selection" (les participant), seul celui avec le meilleur fitness sera gardé

```
for(int tournoi=0; tournoi<nombreTournoi;tournoi++)
{
    for(int i=0;i< ensembleTache.getnbTaches();i++)
    {
        tempsOptimal[i]=i+1; // pour chaque tournoi on repart de l'ordre de base
    }
    // pour chaque tournoi on selectionne adversaireParTournoi éléments au hasard, on ne gardera que le meilleur à chaque fois
    for(int selection=0;selection<adversaireParTournoi;selection++)
    {
        for(int parcoursTab=0;parcoursTab<ensembleTache.getnbTaches();parcoursTab++)
        {
            tempOptimalBeforeChange[parcoursTab]=tempsOptimal[parcoursTab];
        }
        for(int i=0;i<ensembleTache.getnbTaches();i++)
        {
            //brassage aléatoire: on échange au hasard deux valeurs dans le tableau et ce nbTaches fois
            random= ThreadLocalRandom.current().nextInt( origin: 0, ensembleTache.getnbTaches());
            tempVal=tempsOptimal[random];
            tempsOptimal[random]=tempsOptimal[i];
            tempsOptimal[i]=tempVal;
        }
        if(ensembleTache.calculerTempTraitement(tempsOptimal)>ensembleTache.calculerTempTraitement(tempOptimalBeforeChange))
        {
            // si l'element est moins bon que le précédent, on ne le garde pas
            for(int i=0;i<ensembleTache.getnbTaches();i++)
                tempsOptimal[i]=tempOptimalBeforeChange[i];
        }
    }
    for(int i=0;i<ensembleTache.getnbTaches();i++)
    {
        tabIndividus[tournoi][i]=tempsOptimal[i];
    }
```

Le bloc qui suit contient le corps principal de l'algorithme génétique: Nous allons créer une génération enfant grâce au mélange génétique des parents. Pour cela, pour chaque enfant nous effectuons deux cassures dans la première et deuxième moitié du gène, puis nous allons sélectionner trois parents au hasard qui transmettront chacun une partie de leur gène aux enfants. Il faudra alors ici bien s'assurer qu'un numéro de tâche n'est pas présent deux fois ( nous utilisons les boucles verifDoublons qui scanneront la partie précédente du gène enfant pour savoir si le numéro de la tâche parent à insérer n'est pas déjà présente).

```
for(int genetique=0;genetique<nombreIterationGenetique;genetique++)
{
    cassure1=ThreadLocalRandom.current().nextInt( origin: 1,  bound: (ensembleTache.getnbTaches()/2)-1);
    cassure2=ThreadLocalRandom.current().nextInt( origin: (ensembleTache.getnbTaches()/2)+1,  bound: ensembleTache.getnbTaches()-2);

    for(int compteur=0;compteur<nombreTournoi;compteur++)
    {
        //On choisit au hasard des element qui vont composer l'enfant voulu
        placeElementAChanger1=ThreadLocalRandom.current().nextInt( origin: 0, nombreTournoi);
        placeElementAChanger2=ThreadLocalRandom.current().nextInt( origin: 0, nombreTournoi);
        placeElementAChanger3=ThreadLocalRandom.current().nextInt( origin: 0, nombreTournoi);

        // Ensuite on remplis le tableau d'enfant avec les elements pris au hasard
        for(int compteurCasssure1=0; compteurCasssure1<cassure1;compteurCasssure1++)
        {
            tabEnfants[compteur][compteurCasssure1]=tabIndividus[placeElementAChanger1][compteurCasssure1];
        }
        indentation=0;
        for(int compteurCasssure2=cassure1; compteurCasssure2<cassure2;compteurCasssure2++)
        {

            for(int verifDoublon=0;verifDoublon<cassure1;verifDoublon++)
            {
                // n va utiliser un % au cas ou on devrait retourner au debut du tableau chercher la valeur suivante
                if(tabEnfants[compteur][verifDoublon]==tabIndividus[placeElementAChanger2][(compteurCasssure2+indentation)%(ensembleTache.getnbTaches())])
                {
                    // si on trouve un doublon, on passe au chiffre d'apres et on retest depuis le début
                    indentation++;
                    verifDoublon=-1;
                }
            }
            tabEnfants[compteur][compteurCasssure2]=tabIndividus[placeElementAChanger2][(compteurCasssure2+indentation)%(ensembleTache.getnbTaches())];
        }
```

```
for(int compteurFinTaches=cassure2;compteurFinTaches<ensembleTache.getnbTaches();compteurFinTaches++)
{

    for(int verifDoublon=0;verifDoublon<cassure2;verifDoublon++)
    {
    // n va utiliser un % au cas ou on devrait retourner au debut du tableau chercher la valeur suivante
        if(tabEnfants[compteur][verifDoublon]==tabIndividus[placeElementAChanger3][(compteurFinTaches+indentation)%(ensembleTache.getnbTaches())])
        {
            indentation++;
            verifDoublon=-1; // car quand on va sortir de l'iteration de boucle verifDoubon va augmenter a 0
        }
    }
    tabEnfants[compteur][compteurFinTaches]=tabIndividus[placeElementAChanger3][(compteurFinTaches+indentation)%(ensembleTache.getnbTaches())];
}
}
```

La phase suivante est de réorganiser les individus parents et enfants, alors présent dans un gros tableau regroupant les deux générations. Grâce à la méthode du tri à bulle, seul les individus les plus performants seront conservé puis replacé dans le tableau individus.

```
for(int i=0;i<nombreTournoi;i++)
{
    for(int j=0;j<ensembleTache.getnbTaches();j++)
    {
        tabIndividus[i][j]=tableauParentEnfant[i][j];;
    }
}
```

Enfin, avant de recommencer une itération, nous allons effectuer quelques mutation de certains gènes, cela permet de sortir d'un éventuel minimum local:

```
for(int mutation=0; mutation<facteurMutation;mutation++)
{
    int individuMutable=ThreadLocalRandom.current().nextInt( origin: nombreTournoi/2, nombreTournoi); // on ne mute pas le meilleur element
    int j=ThreadLocalRandom.current().nextInt( origin: 0, ensembleTache.getnbTaches());
    int k=ThreadLocalRandom.current().nextInt( origin: 0, ensembleTache.getnbTaches());
    int tampon=tabIndividus[individuMutable][j];
    tabIndividus[individuMutable][j]=tabIndividus[individuMutable][k];
    tabIndividus[individuMutable][k]=tampon;
}
```

Pour finir, lorsque le nombre d'itérations voulu à été effectué, nous retournons le meilleur résultat du tableau d'individus (la mutation étant codé pour ne pas agir sur le meilleur individu, la performance optimal sera conservée).

Cet algorithme est vraiment efficace et peut le devenir si on sait bien quels paramètres initiaux appliquer.

Lorsque l'on lance l'algorithme avec comme paramètre:

```
int nombreTournoi=100;
int adversaireParTournoi=100;
int nombreIterationGenetique=1000;
int facteurMutation=20;
```

```
1526
15 8 7 1 5 11 3 9 10 2 14 12 13 6 4
Temps d'execution: 2960 Millisecondes
```

On obtient le résultat précédent en à peine 3 secondes. En jouant avec les paramètre on observe que le facteur déterminant semble être le nombre d'individus parents Lorsque ce chiffre est faible l'algorithme trouve des résultats moins bons que lorsqu'il est élevé. Le nombre d'itération génétique est également concerné.

```
1524
5 4 14 15 11 6 2 1 9 10 12 7 13 3 8
Temps d'execution: 356 Millisecondes
```

Plus on augmente les paramètre, plus l'algorithme met du temps mais plus les résultats sont proche du minimum global:

```
int nombreTournoi=100; // une population de 100 individus suffit largement
int adversaireParTournoi=100;
int nombreIterationGenetique=10000;
int facteurMutation=20;
```

```
1513
12 5 11 6 4 14 15 8 7 13 3 9 10 2 1
Temps d'execution: 26328 Millisecondes
```

Meilleurs résultats trouvés (un seul lancé):

Fichier 401:

```
1513
12 5 11 6 4 14 15 8 7 13 3 9 10 2 1
Temps d'execution: 26328 Millisecondes
```

Fichier 408:

```
temps optimal: 1657 avec l'ordre: 10 3 12 8 13 4 11 15 1 7 5 6 14 2 9
Temps d'execution: 36147 Millisecondes
```

Fichier 501:

```
temps optimal: 2631 avec l'ordre: 20 10 14 15 22 23 21 12 16 3 7 17 8 6 9 25 4 24 13 18 11 2 1 19 5
Temps d'execution: 55956 Millisecondes
```

Fichier 601:

```
temps optimal: 3498 avec l'ordre: 31 8 2 23 30 27 35 3 22 9 5 24 33 29 15 28 26 4 34 10 12 19 1 17 6 7 16 21 32 20 11 25 13 18 14
Temps d'execution: 133893 Millisecondes
```

Fichier 605:

```
temps optimal: 3601 avec l'ordre: 4 17 31 23 1 35 22 9 18 14 28 26 19 2 13 34 24 30 10 16 25 27 6 8 12 29 11 15 20 7 32 3 5 21 33
Temps d'execution: 66022 Millisecondes
```

Fichier 701:

```
temps optimal: 4579 avec l'ordre: 43 29 27 26 44 13 2 25 23 34 38 15 32 28 14 10 3 30 16 37 24 22 12 19 33 6 40 18 45 1 36 35 9 20 42 8 17 39 41 11 21 5 4 31 7
Temps d'execution: 93309 Millisecondes
```

## Conclusion

Ce projet nous a permis de mieux comprendre le fonctionnement des deux algorithmes que nous avons eu à étudier. Ces derniers sont très efficaces et sur les fichiers d'exemple arrivent à trouver une solution proche du minimum assez rapidement.

Comme tout code, le notre est largement perfectible, on pourrait par exemple songer à réduire le nombre de boucles dans l'algorithme génétique en optimisant certains paramètres (en plaçant les valeur dans un tableau à 3 dimension pouvant contenir les individu et leur fitness pour le pas avoir à le recalculer à chaque fois par exemple), ou même faire du multithreading. Nous sommes quand même satisfait de nos algorithmes et des résultats qu'ils produisent.

## Mode d'emploi :

Pour faire tourner un des algorithmes sur un fichier, il suffit d'ouvrir une fenêtre de commande puis de taper "java -jar MonPrograme.jar FichierVoulu.
Il faudra ensuite choisir und es deux algorithme et spécifier les valeurs de paramétrage voulus.