**INF8775 – Analyse et conception d’algorithmes**

TP3 – Hiver 2023

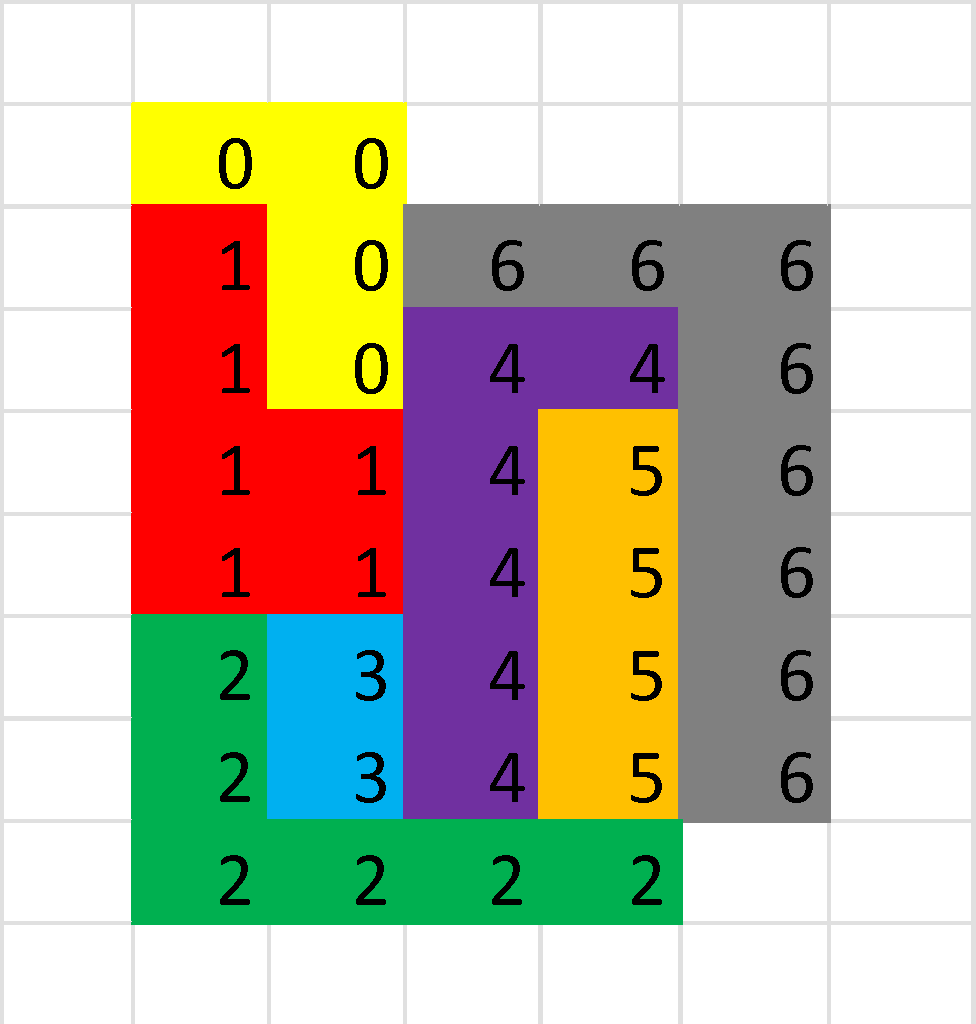
| **Nom, prénom, matricule des membres** | Laurin, Charles-Antoine, 2070729  Yang,Hee-Min , 1961471 |
| --- | --- |
| **Note finale / 20** | **0** |

**Informations sur la correction**

* Répondez directement dans ce document. La correction se fait à même le rapport.
* La date limite pour rendre ce TP est :
  + 18 avril à 23h55 pour le groupe B2
  + 24 avril à 23h55 pour le groupe B1
* Vous devez faire une *remise électronique sur Moodle* en suivant les instructions suivantes :
  + Le dossier remis doit se nommer matricule1\_matricule2\_tp1 et doit être compressé sous format zip.
  + À la racine de ce dernier, on doit retrouver :
    - Ce rapport au format .odt ou docx
    - Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L’interface du script est décrite à la fin du rapport.
    - Le code source et l'exécutable.
* Vous avez le choix du langage de programmation utilisé. Notez que le code et l'exécutable soumis seront testés sur les ordinateurs de la salle L-4714 et doivent être compatibles avec cet environnement. En d’autres mots, tout doit fonctionner correctement lorsque le correcteur exécute votre script *tp.sh* sur un des ordinateurs de la salle.
* La commande *chmod +x mon\_script.sh* rendra le script *mon\_script.sh* exécutable. Pour l’exécuter il s’agira de faire *./mon\_script.sh*

Ce dernier travail pratique se fera dans le cadre du concours du meilleur algorithme pour la session d’hiver 2023. Le travail demandé consiste à concevoir et implanter un algorithme de votre cru pour résoudre un problème combinatoire. Le classement des équipes déterminera votre note pour la *qualité de l'algorithme*. Votre algorithme sera exécuté sur 3 exemplaires de notre choix pendant 2 minutes chacun.

Le but du TP est de placer des enclos dans un zoo pour minimiser le temps de déplacement des clients entre les enclos. Ainsi, chaque enclos a un poids différent pour tous les autres, représentant la quantité de gens qui veulent se rendre d'un enclos donné à un autre. On peut le représenter par un graphe pondéré fortement connexe G avec n sommets (les enclos) et les poids des arêtes E sont le nombre de gens qui veulent se rendre d’un sommet u à v.  
  
Les enclos peuvent avoir des tailles variables de 2 à 20 cases (carrés de 1 de côté), placés sur une  
grille cartésienne 2D infinie. Les cases d’un enclos doivent être orthogonalement adjacentes, de sorte que les enclos soient contigus, mais toutes les formes sont autorisées. Par exemple :



Le terme à minimiser est le suivant :

Où la distance(u,v) est la distance de manhattan entre les deux cases les plus près des enclos u et v (i.e. distance de 1 pour des enclos adjacents), et poids\_arête(u,v) est la valeur de l’arête entre u et v.

*Dans l’exemple ci-haut, la distance entre les enclos 0 et 1 (ou 0 et 6) serait 1, et la distance entre 0 et 2 serait de 4.*

Il y a un autre aspect à prendre en compte dans votre placement des enclos: placer un ensemble donné d'enclos à une distance maximale les uns des autres ajoute à l'attrait du zoo (par exemple, un thème félins). On a un sous-ensemble *S* du graphe G, de taille m. Si tous les enclos de *S* se trouvent à une distance inférieure à *k* les uns des autres, la condition est remplie et l’on obtient un attrait supplémentaire égale au carré du nombre d’enclos dans le sous-ensemble.

On obtient donc la contrainte suivante :

*Par exemple, pour les sous-ensembles et avec une distance max de 3, on obtient la valeur ajoutée de 9 pour car distance(1,6) = 2, mais pas 16 pour car distance(0,2) = 4*

L’attrait total du zoo à maximiser est donc :

**Jeu de données**

Nous vous fournissons un générateur d’exemplaires qui fonctionne comme suit :

python inst\_gen.py -n 10 -m 5

Arguments :

* n = nombre d’enclos (nombre de sommets)
* m = taille du sous-ensemble à placer à proximité (<= n)

Un fichier au format ex\_n\_m.txt est enregistré dans le répertoire d’exécution. Par exemple :

A picture containing arrow

Description automatically generated

La première ligne correspond au nombre d’enclos n, suivi du d’nombre d’enclos m dans le sous-ensemble S, et de la distance maximale à respecter k pour les enclos de ce sous-ensemble.

La 2e ligne donne la liste des enclos à placer à une distance de moins de k pour obtenir le bonus V=m^2.

Les n lignes suivantes donnent la taille de l’enclos i pour i allant de 0 à n-1,

Finalement, il y a n lignes supplémentaires qui représentent les poids de l’enclos i vers les n autres enclos. (poids de 0 pour la distance avec soi-même)

Vous devez générer des exemplaires de la taille de votre choix, voici un ordre d’idée pour les valeurs : nombre d’enclos **n** entre 100 et 1000, taille max du sous-ensemble d’enclos **m** entre 10 et 500.

**Q1 – Description de votre algorithme**

*Décrivez en quelques phrases votre algorithme.*

| **0** | / 2 pt |
| --- | --- |

Notre algorithme est un algorithme qui utilise l’heuristique de recuit simulé pour améliorer la solution générée. Nos exemplaires sont générés de façon pseudo-aléatoire : on choisit une case et ajoute des cases choisies aléatoirement tant que la taille voulue n’est pas atteinte.

Pour l’amélioration, nous choisissons un certain nombre d’enclos aléatoire, le retirons du tableau et on le régénère une nouvelle fois. Ce nombre d’enclos est inversement proportionnel au nombre d’itérations que l’algorithme a déjà fait. Nous faisons alors plusieurs itérations d’exploration de configurations aléatoires et remplaçons notre *meilleure solution* avec une nouvelle si celle-ci a un bonus plus élevé. Bref, nous avons implémenté un algorithme de recuit simulé pour trouver une réponse relativement optimale.

**Q2 – Présentation**

*Sous forme de pseudo-code et incluant une analyse de complexité théorique des principales fonctions. Si vous préférez écrire vos équations en Latex, vous pouvez ajouter un pdf à la remise avec la réponse à cette question et le mentionner ici. Pas besoin de faire une analyse empirique de la complexité.*

| **0** | / 4 pt |
| --- | --- |

Voici les variables utilisées pour les expressions de complexités:

**n** : nombre d’enclos à placer

**a** : taille d’un enclos

**t** : taille du tableau complet

**m** : nombre d’éléments du sous-ensemble m

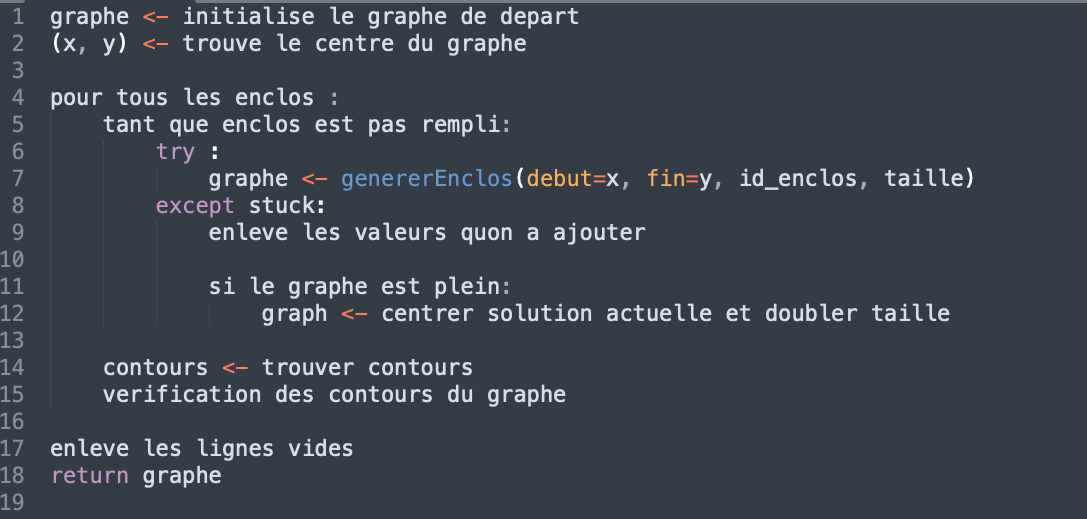
**Recuit simulé:**

La complexité de l’algorithme de recuit simulé dépend surtout des méthodes appelées. Comme l’algorithme accède souvent à la taille complète du tableau de taille txt, la fonction est souvent dans cet ordre de grandeur. Comme nous le verrons plus tard en détail, cette fonction:

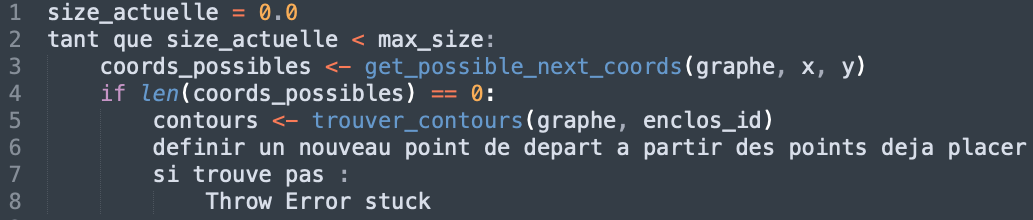
1. Génère une solution initiale
2. Trouve une nouvelle configuration
3. si elle est meilleur, prend cette solution pour la prochaine itération
4. Sinon il y a des chances de prendre la solution sous optimale

**Génération des solutions possibles:**

1- Génération de tous les enclos: Ω(n \* ( a + t^2)), O(n\*t^2 (1+a)) en pire cas

****

1.1- Génération de 1 enclos : Ω(a) dans le meilleur cas, O (a \* t^2) dans le pire

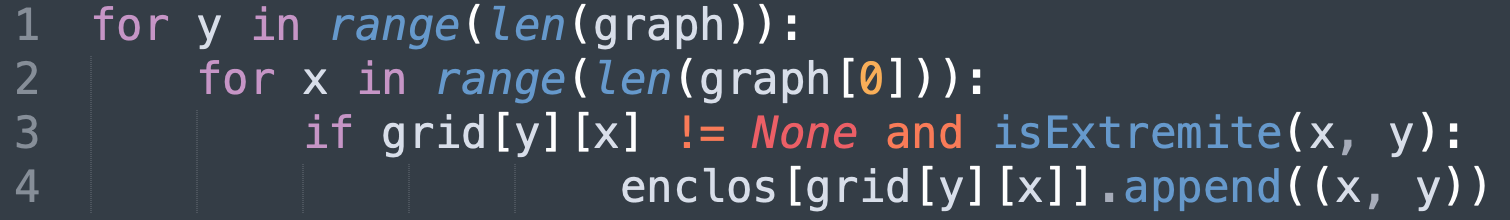
****

En meilleurs cas, générer un enclos se fait en O(a) : trouver les cases possibles consiste à visiter les voisins immédiats de la case évaluée ce qui se fait en O (1). Cependant, si on ne trouve aucun point valide, on doit trouver un autre point de contours ayant des voisins vides. L’opération de trouver les contours de l’enclos se fait en O(a^2).

**Calculs du pointage: O(n^2 \* a^2)**

Le calcul du pointage se fait en 2 étapes distinctes : calculer les distances et calculer les points. Le calcul des distances est un peu plus complexe et représente le plus gros du travail. Voici à quoi ressemble le pseudo code pour le calcul des distances:

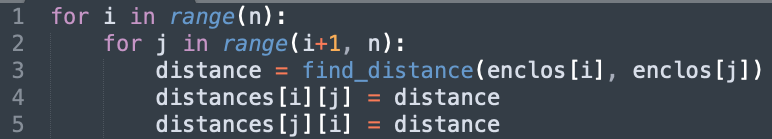
1- Trouver tous les contours d’enclos dans le graphe: Ө(t^2)



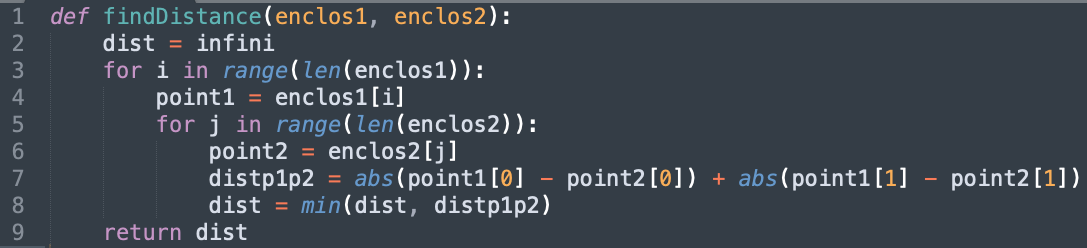
La fonction isExtremite se fait en temps constant : on vérifie si la valeur des voisins immédiats de la case(x,y) est la même que cette dernière.

2- Trouver les distances entre chaque enclos: Ө(n^2 \* a^2)

2.1- Itérer à travers tous les enclos: Ө(n^2)



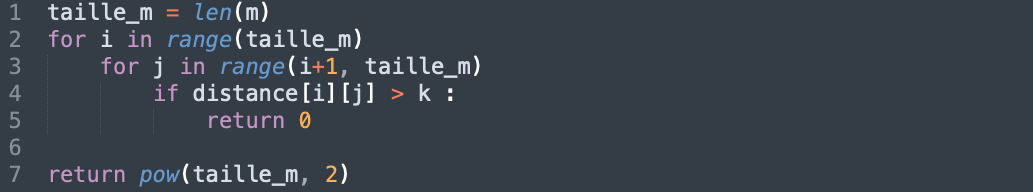
2.2- Trouver la distance entre 2 enclos: Ө(a^2)



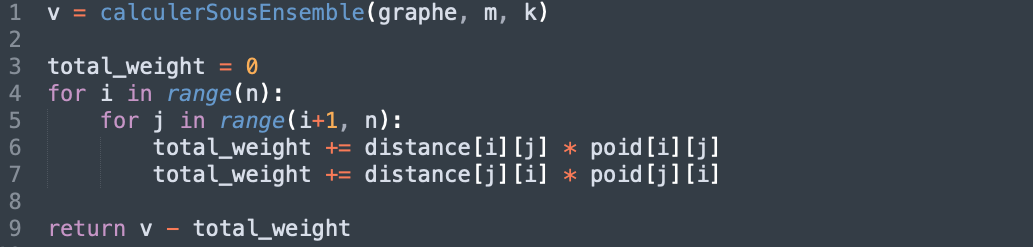
\*La variable a représente ici le nombre de case composant les enclos.

3- Calculer les points:Ө(n^2)

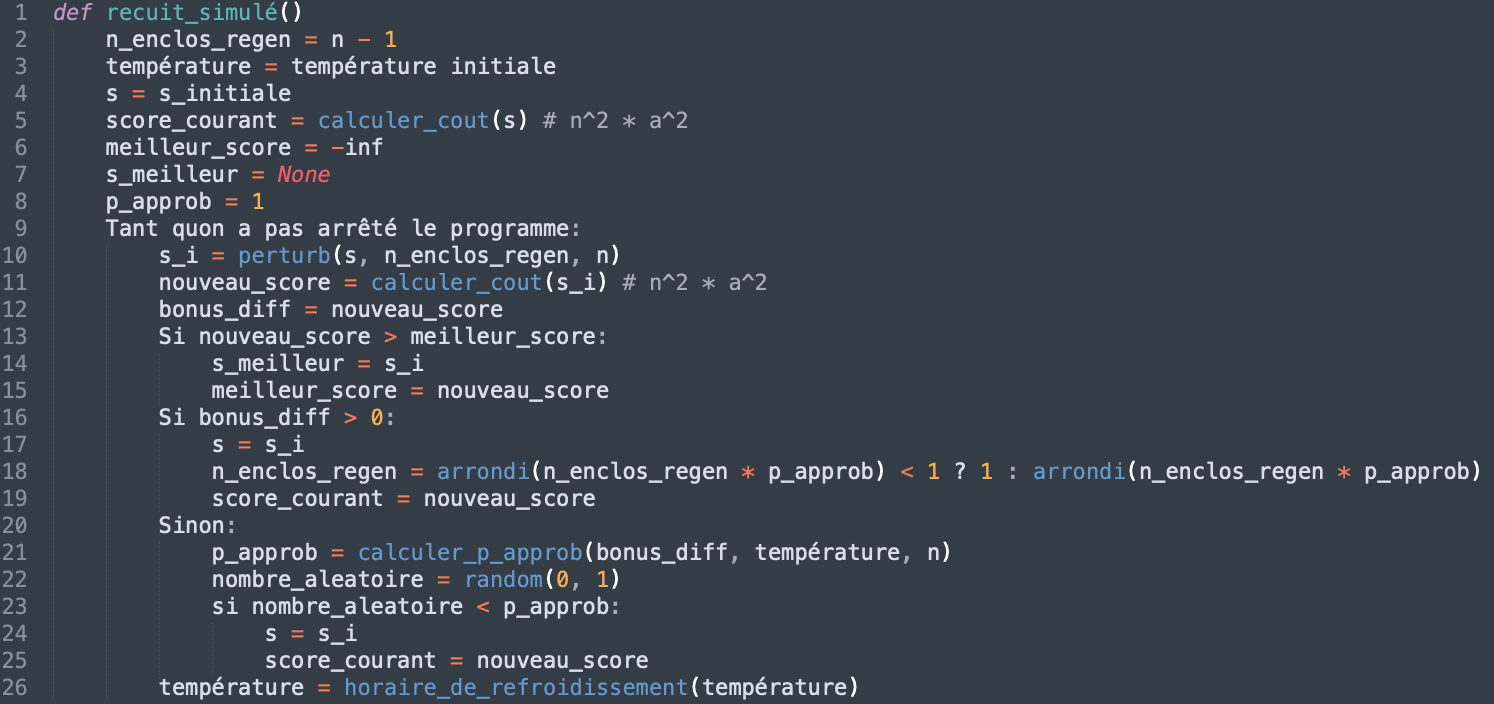
3.1- Calculer le sous-ensemble m est respecté: Ө(m^2)



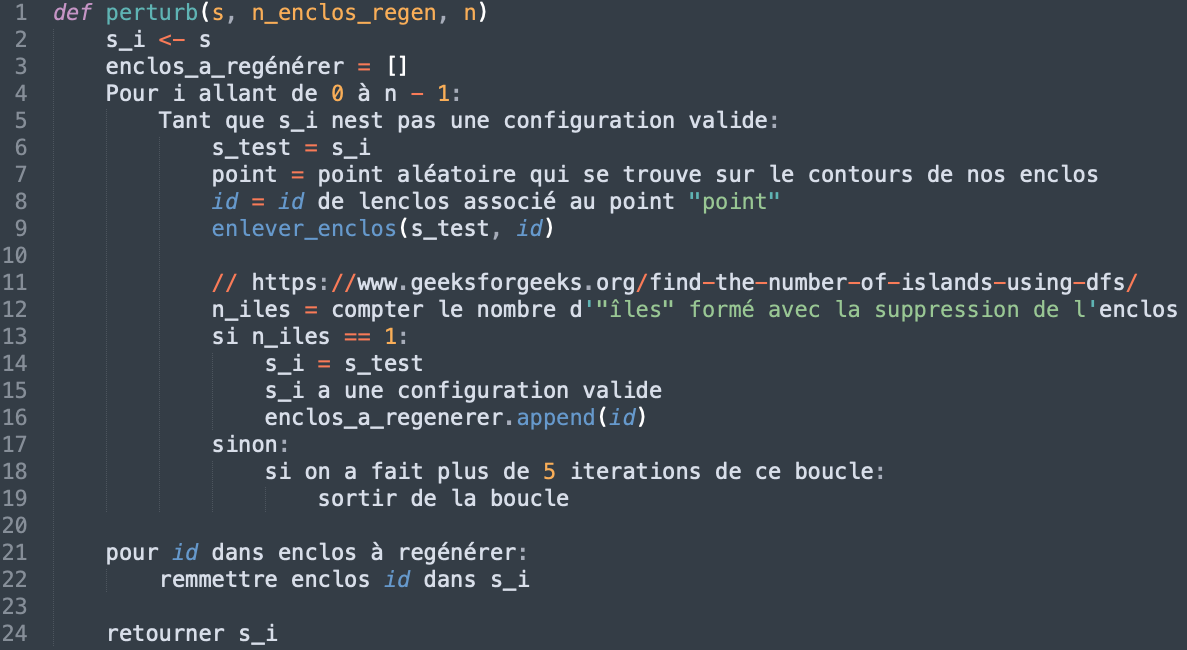
3.2- Calculer le poid du graphe: Ө(n^2)



**Amélioration de la solution:** Ө(n^2 \* a^2 + t^2)

1- Recuit simulé:O(n^2 \* a^2 + t^2)****

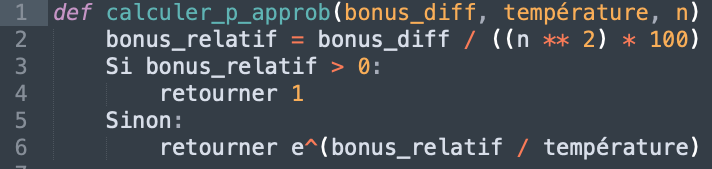
2- Changement de voisinage: Ө(t^2)

****

3- Probabilité d’approbation: Ө(1)



4- Horaire de refroidissement: Ө(1)

****

**Q3 – Justification de l’originalité de vote algorithme**

*La conception de votre algorithme sera jugée avec les critères suivants :*

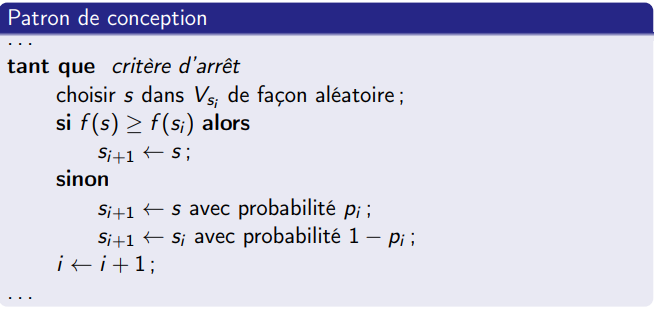
* *Lien avec le contenu du cours*
* *Originalité*
* *Initiative*

| **0** | / 4 pt |
| --- | --- |

On nous a posé comme problème une situation où plusieurs configurations sont possibles sans toutefois savoir si nous avions obtenu la réponse optimale. À cause de cela, nous avons décidé de procéder par métaheuristique. Effectivement, en commençant avec une configuration aléatoire et en faisant des changements sur cette solution initiale, on peut tenter de générer une solution qui se rapproche à la solution optimale. Cette approche nous permet alors de tester une grande quantité de voisinage sans avoir besoin de garde trop de configurations en mémoire comme le ferait l’algorithme génétique dans ce cas-ci.

## Lien avec le cours

Comme nous l’avons vu en cours, le patron de conception de l’algorithme recuit simulé suit le pseudo-code démontré dans la figure ci-dessous:

*Figure 1. Patron de conception de recuit simulé du chapitre 8.*

Dans notre algorithme, nous avons le voisinage exprime une configuration d’enclos. La fonction *f* est une multiplication des poids des distances entre les enclos ainsi que leurs distances l’un de l’autre. Lorsque nous décidons de remplacer notre solution courante avec une nouvelle solution, nous le faisons en choisissant un nombre d’enclos à enlever de notre configuration pour ensuite les placer de nouveau de manière aléatoire. Nous obtenons alors une diversité réellement aléatoire favorisant la découverte de divers voisinages.

## Originalité

### Génération d’enclos

Dans le cas des enclos, nous avons décidé de générer le voisinage initiale de manière complètement aléatoire: Nous procédons enclos par enclos et pour chaque enclos, on place un enclos en prenant une case aléatoire de l’enclos et en choisissant une direction vers lequel il n’existe pas d’enclos. Cette approche est unique puisque nous aurions pu prendre une approche où les enclos avaient des formes prédéterminées tels que des rectangles ou des carrés. Par contre, une approche complètement aléatoire nous permet de faire une exploration réelle de solutions qui seraient difficiles à imaginer.

### Probabilité d’approbation de solution

En plus de cela, notre fonction pour déterminer la probabilité d’approbation de la nouvelle solution nous est aussi unique. Effectivement, nous prenons en considération la taille de la matrice (soit nxn). Sachant que pour des tailles plus grandes de matrices en entrée que la différence entre le bonus d’une nouvelle solution et d’une ancienne solution monte exponentiellement selon n, nous avions défini notre probabilité d’approbation comme suit:

Nous avons décidé de mettre un facteur de 100 après expérimentation de plusieurs facteurs de probabilités qui nous donnaient les meilleurs résultats expérimentaux.

Nous réalisons aussi que dans notre formule d’approbation que nous n’avons pas le signe négatif. Nous faisons ceci puisque notre problème cherche à maximiser le coût de notre solution.

### Changement de voisinage

Nous avons aussi un algorithme de changement de voisinage unique. Soit un chiffre n qui représente le nombre d’enclos. Pour changer le voisinage, on enlève un nombre *c* d’enclos pour alors les replacer de manière aléatoire. Au début, nous commençons avec

Mais au fur et à mesure que nous trouvons une solution de plus en plus optimale, nous voulons faire de moins en moins d’explorations possible de solutions pour convertir vers la solution optimale globale. Sachant que la *température* est une variable de notre algorithme qui devient de plus en plus petite au fur et à mesure que notre algorithme roule, nous réévaluons *c* selon la formule suivante:

## Initiative

Bien qu’il existe sûrement des méthodes de trouver la solution optimale théorique, il est certain qu’une approche comme le backtracking prendrait beaucoup trop de temps pour des tailles d’exemplaires relativement grandes. Il fallait alors procéder avec une méthode qui nous permettait de trouver une solution relativement bonne en une quantité de temps minime. Nous avons d’abord commencé avec l’approche génétique, mais avons rapidement réalisé que l’utilisation d’un tel algorithme pour un problème où la taille en mémoire des différentes solutions devenait rapidement très grande. Il fallait alors procéder avec l’approche recuit simulé. Cette solution permet de commencer avec une solution aléatoire et de faire des itérations afin d’améliorer notre solution.

**Q4 – Votre algorithme est-il assuré de trouver la solution optimale ?**

*Répondez simplement “oui” ou “non”. Aucune justification requise.*

| **0** | / 1 pt |
| --- | --- |

Non

**Autres critères de correction**

**Respect de l’interface tp.sh**

| 0 | / 2 pt |
| --- | --- |

Utilisation :

tp.sh -e [chemin\_vers\_exemplaire] -p

Lorsque le script est exécuté sans le paramètre -p, le programme affiche uniquement l’attrait total du zoo, à chaque fois qu’une meilleure solution est trouvée. Votre programme est sensé s’exécuter tant et aussi longtemps qu’il n’est pas manuellement interrompu.

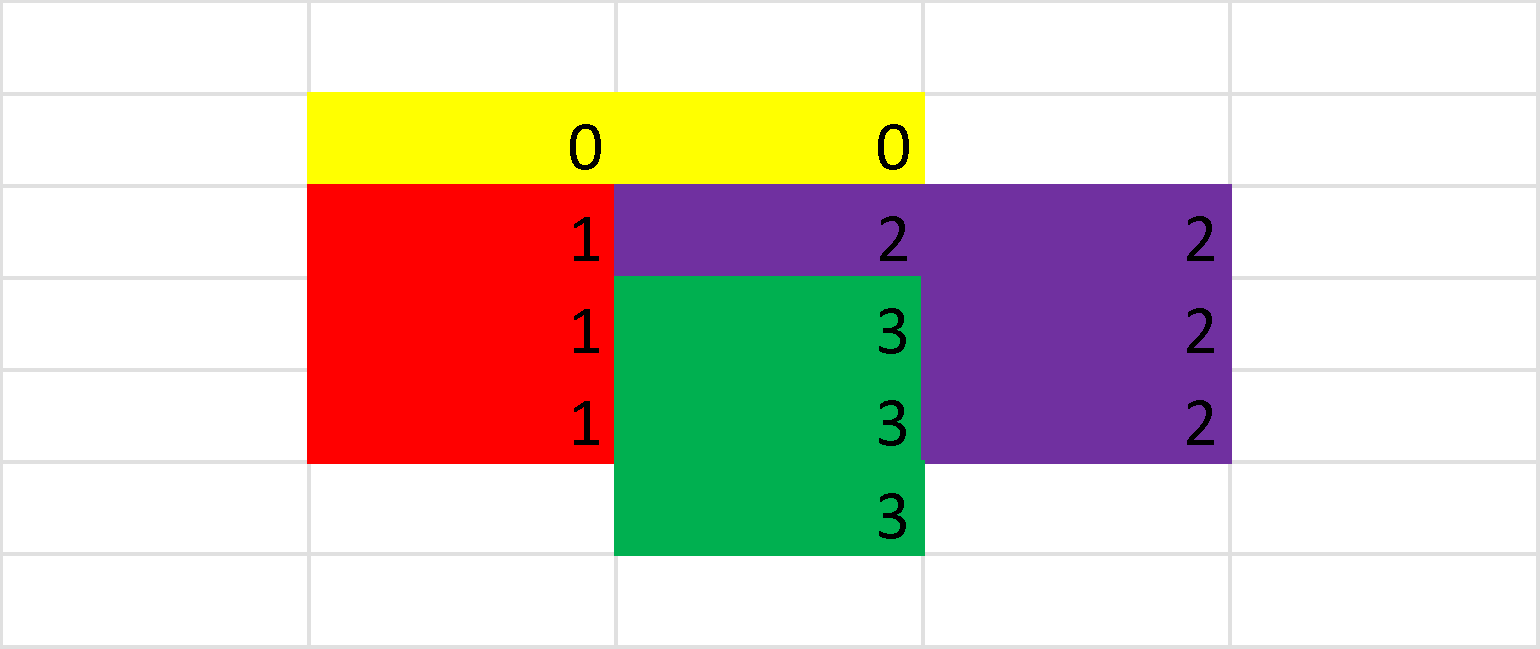
**Argument optionnel** :

-p Chaque fois qu’une meilleure solution est trouvée, le programme affiche cette nouvelle solution (**au lieu** d’afficher le nombre d’arêtes retirées). Le format de cette solution est décrit ci-dessous

**Important** : l’option -e doit accepter des fichiers avec des chemins absolus.

Le script tp.sh ne vous est pas fourni, mais vous pouvez facilement adapter celui du TP2.

**Format de la solution** : voici un exemple d’affichage. Le zoo correspondant est le suivant :



Note : ici, la case la plus haute de l’enclos 3 a les coordonnées (0,0) mais il n’y a pas de contrainte là-dessus. Les coordonnées négatives sont autorisées.

-1 2 0 2

-1 1 -1 0 -1 -1

0 1 1 1 1 0 1 -1

0 0 0 -1 0 -2

-1 2 0 2

-1 1 -1 0 -1 -1

0 1 1 1 1 0 1 -1

0 0 0 -1 0 -2

Chaque enclos occupe une ligne, avec les coordonnées de tous les cases de l’enclos selon la convention (x,y) d’un plan cartésien, séparées d’un espace. Entre chaque nouvelle solution trouvée, **vous devez laisser une ligne vide**.

Un script de vérification de solution vous est fourni (check\_sol.py, les instructions d’utilisation sont dans le code source). Ce script vous indiquera si l’affichage est correct, si votre solution est valide, ainsi que la valeur de votre solution. C’est ce script qui sera utilisé pour la correction, donc assurez-vous qu’il reconnaisse vos solutions.

**Qualité de l’algorithme**

*Les points de cette question sont répartis comme suit :  
Nous allons exécuter votre code sur 3 exemplaires de notre choix. Pour chaque exemplaire la sortie de votre code sera envoyée vers le script de vérification, et nous classerons sur chaque exemplaire les différents binômes.  
Pour chacun des 3 exemplaires, le quart des groupes ayant la meilleure solution remportera 1 point, le deuxième quart 0.75, le troisième quart 0.5 et le dernier quart 0.25 points.*

*Un bonus de 2 point est donné si vous trouvez une solution* ***meilleure qu’une certaine baseline obtenue par un algorithme de base*** *dans le temps imparti.*

*Cela signifie que si vous avez une solution relativement bonne mais un algorithme peu performant vous aurez quand même 2.75/5.*

| 0 | / 5 pt |
| --- | --- |

**Qualité du code**

| 0 | / 1 pt |
| --- | --- |

**Présentation générale (concision, qualité du français, etc.)**

| 0 | / 1 pt |
| --- | --- |

**Pénalités**

| 0 |
| --- |

* Retard : -1 pt / journée de retard, arrondi vers le haut. Les TPs ne sont plus acceptés après 3 jours.
* Autres : Le correcteur peut attribuer d’autres pénalités (par exemple si les exécutables sont manquants, etc.)