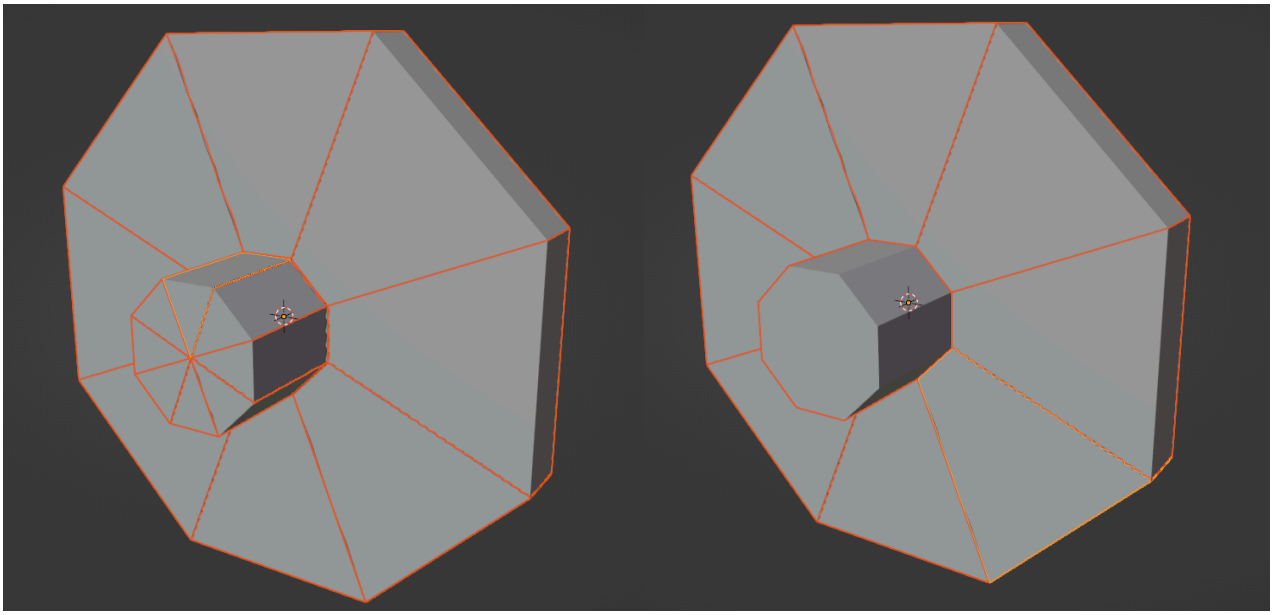
Rapport de projet

Fusion de polyèdres pour les jeux vidéo et logiciels 3D



Étudiants :

FONTAINE Antoine
DUBOIS Jérôme

Encadrant :

M. LEPAGNOT Julien

Année : 2024
Formation : Master 1 Informatique et Mobilité

Sommaire

I. Introduction

Le projet "Fusion de Polyèdres Convexes" a pour objectif de développer un algorithme destiné à fusionner de manière itérative des polyèdres convexes élémentaires, dans le but de minimiser le nombre final de polyèdres convexes résultants. Ce processus repose sur l'identification, à chaque étape de l'algorithme, de l'ordre optimal de fusion des polyèdres, afin d'atteindre un résultat où le nombre de polyèdres est réduit au minimum possible.

Dans un premier temps, nous allons adopter une approche exhaustive en utilisant la force brute pour tester toutes les combinaisons possibles de fusion des polyèdres. Cette méthode devient impossible lorsque le nombre de polyèdres convexes élémentaires augmente, en raison du temps de calcul excessivement long. Par conséquent, dans une seconde phase, une recherche et implémentation de métaheuristique sera mise en place pour remplacer cette recherche exhaustive par une méthode plus efficace et rapide.

Ainsi, ce projet vise à finaliser et améliorer un algorithme existant, tout en explorant des solutions optimisées pour la fusion de polyèdres convexes en trois dimensions, en utilisant des techniques d'optimisation.

II. Structure du projet

L'ensemble des travaux réalisés durant le projet est structuré de la façon suivante : le dossier principal du projet comporte l'ensemble des classes utilisées pour l'algorithme de fusion de polyèdre ainsi que le programme principal main.cpp. Plusieurs dossiers sont présents et expliqués ci-dessous

- *genetic* contient l'ensemble des différentes sélections, croisements et mutations implémentés ainsi que le code de l'algorithme génétique.
  Les sélections implémentées sont la sélection élitiste, la sélection par tournoi ainsi que par roulette. Le croisement en N points ainsi que la mutation par insertion sont présents.

- *Tests* contient différents dossiers regroupant les tests effectués sur le programme principal et l'algorithme de fusion de polyèdres comme la convexité, la fusion, l'écriture et la lecture de polyèdres dans des fichiers *.obj* ou blender.

- Le sous-dossier *generated* contient l'ensemble des 'runs' effectués par le programme. Lorsque le programme se termine, un dossier 'run_' suivi du nombre de polyèdre final, de la date et de l'heure est créé. Dans celui-ci on retrouve le fichier *.obj de la solution finale ainsi que des graphiques et un fichier texte.

- *scripts* contient un sous-dossier *pythonGenerateGraph*. A l'intérieur on retrouve des fichiers Python permettant de créer et d'afficher des graphiques avec matplotlib, ainsi qu'un dossier *dist* contenant l'exécutable de ces fichiers compilés, directement utilisé par le code C++. L'exécutable prend en paramètres le chemin du répertoire ainsi que le nom du fichier. Si une modification du fichier python est réalisée, il faut recompiler l'exécutable avec une commande présente en commentaire dans le code python.

Un diagramme de classe se trouve dans le répertoire *Documentation/* du projet. De plus, le **README.md** et la documentation html située dans le GITHUB donnent un point de vue global sur les outils nécessaires au bon fonctionnement du projet.

III. Ecriture / Lecture d'objet 3D

L'algorithme de fusion permet de lire un fichier au format *.obj en entrée, fusionner au maximum les polyèdres entre eux et écrire un fichier *.obj correspondant à cette solution. La lecture et l'écriture se font donc à l'aide de la classe *OBJFileHandler*.

- La lecture permet de lire un fichier *.obj, de créer dynamiquement et de stocker les objets présents dans le fichier selon la syntaxe obj. Une liste de mots clés est présente dans la classe, associant la syntaxe obj avec un vocabulaire plus parlant. Toute la syntaxe présentée ci-dessous est reconnue dans la liste mais n'est pas forcément implémentée.

  Chaque ligne du fichier est parcourue et selon son mot clé associé, une action est réalisée :
  - '#' correspond à un commentaire, on ne traite pas la ligne
  - 'v' est un sommet, on traite la ligne contenant les coordonnées du sommet en créant un objet Point et en le stockant
  - 'o' est un objet, on traite la ligne contenant le nom de l'objet en créant un objet Polyedre pour le moment vide avec un id unique
  - 'f' est une face, on traite la ligne contenant les ids des sommets en créant un objet Face avec la liste des sommets et un id unique. Cette face est ajoutée au vecteur faces du dernier polyèdre courant
  - Le reste de la syntaxe n'est pas traité, mais a été développé pour être ajouté facilement.

  Une fois les objets créés, la convexité de chaque polyèdre est testée et marquée. Après la lecture, on retrouve la liste des polyèdres, des faces et des sommets stockées dans la classe algorithme.

- L'écriture permet d'écrire un fichier *.obj pouvant être correctement visualisé. Pour écrire un fichier *.obj, la liste des sommets et des polyèdres ainsi que le nom du fichier doivent être passés en paramètres. La surcharge de l'opérateur << pour les sommets, les polyèdres et les faces permet de les écrire facilement. On commence par écrire tous les sommets puis tous les polyèdres.

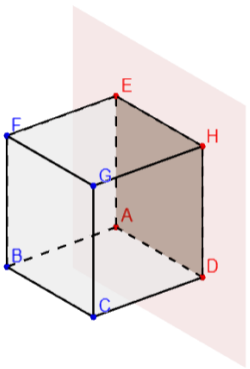
IV. Test de convexité

Un polyèdre est convexe si, pour toute paire de points situés à l'intérieur du polyèdre (ou sur une des faces), le segment reliant ces deux points est entièrement contenu à l'intérieur ou sur la surface du polyèdre.

1. Convexité d'un polyèdre

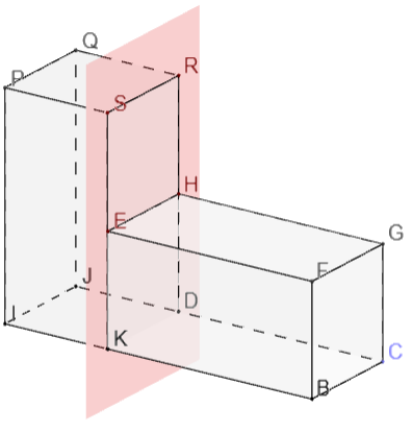
En pratique, pour chaque face, on vérifie que tous les sommets sont du même côté du plan prolongeant la face, ou sur le plan lui-même.

Exemple d'un **polyèdre convexe** (un cube) :



- On commence par la face **EHDA**. On construit un plan à partir de 3 points de cette face, puis on vérifie que tous les sommets sont soit à droite soit à gauche du plan. On constate que tous les sommets sont soit à gauche, soit sur le plan, donc il faut tester d'autres faces.
- On répète cette étape pour les 5 autres faces. Toutes les faces vérifient la condition, donc on conclut que le cube est convexe.

Exemple d'un polyèdre **non convexe** :



- Pour la face **SRHE**, on construit un plan qui prolonge cette face. On constate que les sommets P, Q, I et J sont à gauche du plan, alors que les sommets F, G, C et B sont à droite du plan.
- Il y a au moins une face qui ne vérifie pas la condition (on aurait aussi pu prendre l'exemple de la face **EHGF**), donc le polyèdre n'est pas convexe.

Voici les étapes pour vérifier si des sommets sont tous du même côté d'un plan :

1. <u>Création du plan</u>

- On calcule 2 **vecteurs directeurs** ***u*** et ***v*** du plan à partir de 3 sommets de la face.
- On effectue un **produit vectoriel** entre ***u*** et ***v*** pour obtenir le **vecteur normal** ***n*** (*a, b, c*) du plan
- Calcul de **l'équation cartésienne** du plan $ax + by + cz + d = 0$. Les paramètres a, b et c sont donnés par les composantes de la normale. d est calculé en remplaçant x, y et z par les coordonnées d'un point du plan et en résolvant l'équation.

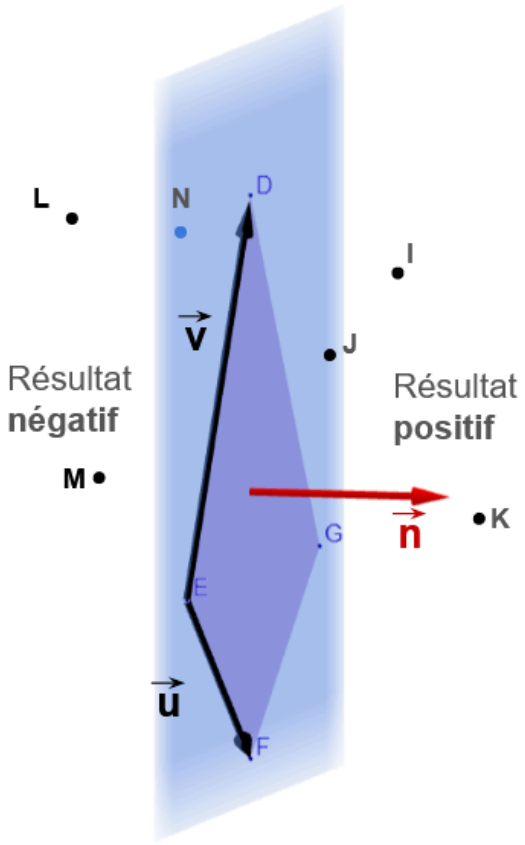
2. Vérifier de quel côté du plan est un point

- On résout l'équation cartésienne du plan en remplaçant *x*, *y* et *z* par les coordonnées du point en question.
- Si le résultat est nul, cela signifie que le point est sur le plan. Si le résultat est négatif ou positif, le point est d'un côté ou de l'autre du plan.

3. Tester tous les points

- On applique l'étape 2 à tous les points.
- Si tous les résultats de l'équation sont soit tous négatifs ou nuls, soit tous positifs ou nuls, alors tous les points sont du même côté.
- Sinon, il y a au moins un point qui n'est pas du même côté.

Exemple : avec une face (polygone) **EDGF**. Les points *L*, *M* et *N* sont à gauche du plan et les points *I*, *J* et *K* sont à droite du plan.

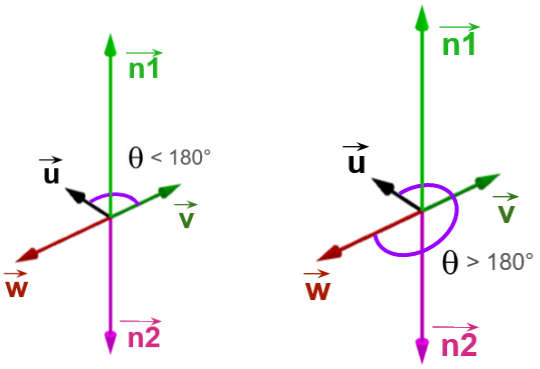


2. Convexité d'un polygone

Le projet traite des polyèdres, mais pour gérer un cas particulier qui est détaillé dans la partie "Algorithme de fusion", nous avons besoin de vérifier la convexité d'un polygone.
Le principe est différent du test de la convexité d'un polyèdre. On peut considérer qu'un polygone est convexe si tous ses angles intérieurs sont inférieurs à 180 degrés, ou si tous ses angles extérieurs sont supérieurs à 180 degrés.

Rappel sur le produit vectoriel :

Soit trois vecteurs **coplanaires u**, **v** et **w**.
Le vecteur **n1** est le vecteur normal résultant du produit vectoriel entre **u** et **v**.
Le vecteur **n2** est le vecteur normal résultant du produit vectoriel entre **u** et **w**.



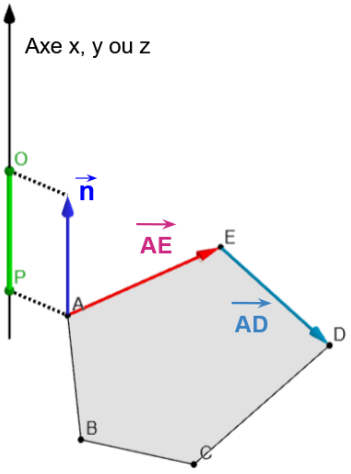
On constate que les vecteurs **n1** et **n2** ont un **sens opposé**. C'est ce principe que nous allons utiliser pour vérifier de quel côté d'une droite se situe un point dans un espace en trois dimensions. Le sens de la normale nous indique si l'angle **thêta** entre les deux vecteurs est supérieur ou inférieur à 180 degrés.

Voici comment déterminer si un polygone est convexe ou pas, illustré par un exemple :

On va parcourir toutes les arêtes du polygone deux à deux pour vérifier qu'elles forment soit toutes un angle supérieur à 180 degrés, soit toutes un angle inférieur à 180 degrés. En effet, selon les vecteurs choisis, on teste soit les **angles extérieurs** soit les **angles intérieurs**.

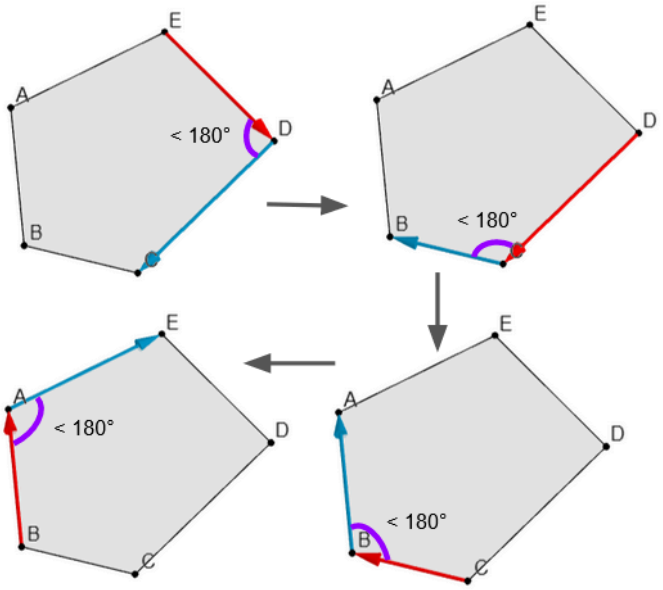
Exemple avec un polygone convexe :

Soit le polygone convexe **AEDCB**. Nous allons commencer l'algorithme de vérification par l'arête *AE.*

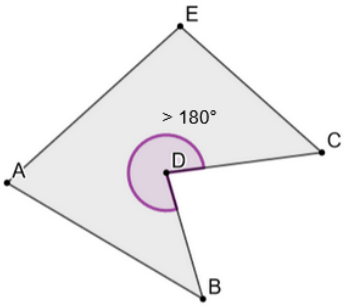


- Construire les vecteurs *AE* et *ED*
- Effectuer un produit vectoriel entre les vecteurs *AE* et *ED* pour obtenir la normale **n**
- Pour déterminer le sens du vecteur normal **n**, il faut le projeter sur un axe défini (*x*, *y* ou *z*) en faisant un produit scalaire (représenté en vert sur le schéma).
  Par exemple, pour projeter **n** sur l'axe *y*, on fait le produit scalaire entre **n** et (0, 1, 0).
- Le résultat sera un nombre positif, négatif ou nul si les points *A*, *E* et *D* sont alignés.
  Le **signe** de ce nombre représente si l'angle entre les vecteurs *AE* et *ED* est inférieur ou supérieur à 180 degrés (pas besoin de calculer la valeur de l'angle) .

Après l'arête *AE* et *ED*, il faut tester toutes les autres paires d'arêtes.
Ensuite, on parcourt des autres paires d'arêtes en refaisant les étapes précédentes.



<u>Exemple d'un polygone non convexe :</u>



V. Algorithme de fusion

1. Fusion de deux polyèdres convexes

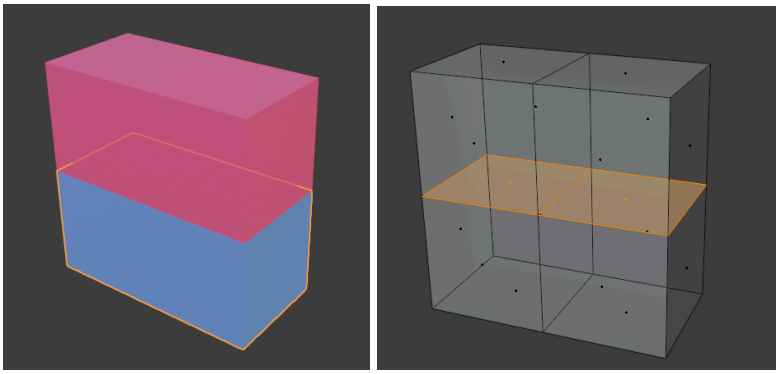
a. Cas classique

Pour fusionner les polyèdres entre eux, nous avons implémenté un algorithme de fusion réparti en différentes étapes :

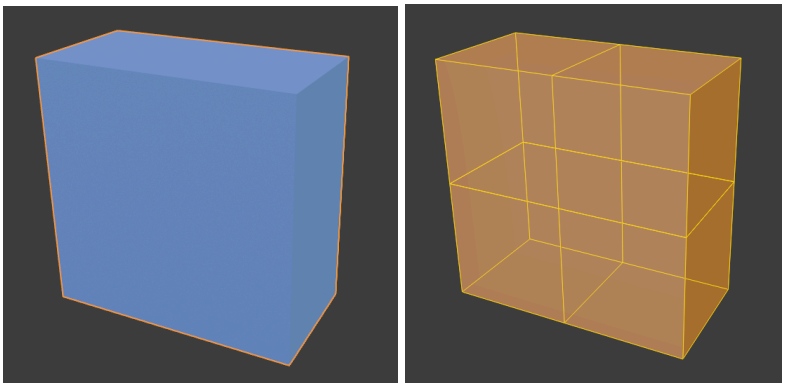
- (1) Vérifier que les 2 polyèdres sont convexes (cf. partie calcul **convexité polyèdre**)
- (2) Chercher des **faces communes** entre les 2 polyèdres (parcours exhaustif)
- (3) Fusion des polyèdres : Créer un nouveau polyèdre avec toutes les **faces** des 2 polyèdres sauf celles qui sont communes
- (4) Calculer la convexité du polyèdre résultant de la fusion (cf. partie calcul **convexité polyèdre**).

Selon le sujet, s'ils n'ont aucune face en commun ou si leur fusion n'est pas convexe, alors la fusion n'est pas réalisée.

Exemple :
Avant fusion : 2 polyèdres avec 2 faces communes



Après la fusion : 1 seul polyèdre, sans les 2 faces à l'intérieur qui étaient communes



b. Fusion de polyèdres avec un volume nul

Un polyèdre avec un volume nul est un polyèdre avec une seule face, qui peut s'apparenter à un polygone. Il y a deux cas de figure possibles :

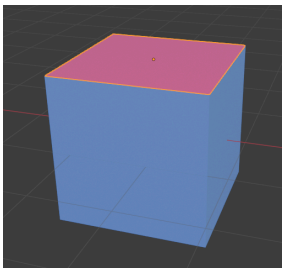
- Seul un des deux polyèdres a un volume nul
- Les deux polyèdres ont un volume nul

1. **Un seul des deux polyèdres a un volume nul** : alors on peut appliquer la méthode classique

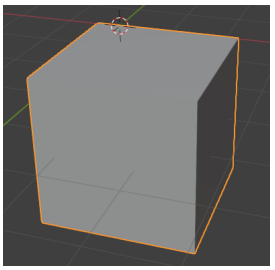
Exemple :

Un polyèdre rouge avec un volume nul et un polyèdre bleu (cube). Ils partagent une face qui est la face du dessus pour le cube, et l'unique face pour le polyèdre rouge.

Avant fusion :



Après fusion



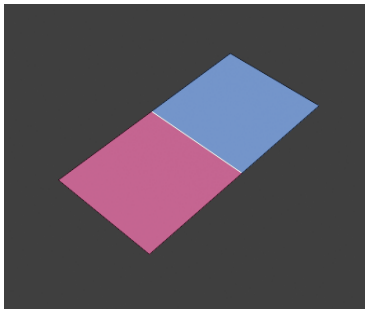
2. **Les deux polyèdres ont un volume nul**

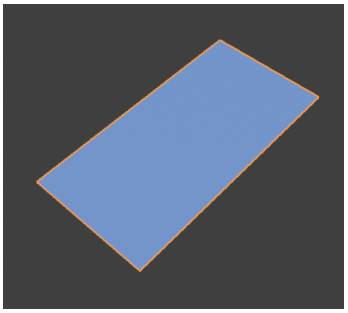
Dans ce cas, on va traiter les 2 polyèdres comme des polygones et reprendre les étapes du cas classique

- (1) Vérifier que les 2 polyèdres sont convexe (cf. partie calcul **convexité polygone**)
- (2) Chercher des **arêtes communes** aux 2 polyèdres
- (3) Fusion des polyèdres : Créer un nouveau polyèdre avec une seule face : composée des **arêtes** des 2 polyèdres sans celle(s) commune(s).
  En pratique, la structure de données des polyèdres ne représente que les faces et les sommets, donc pas les arêtes. Ainsi cette étape se fait en enlevant un sommet et en ré-ordonnant les sommets des 2 polyèdres. Un **exemple détaillé** de cette étape est illustré dans le document ***Algo_Fusion_Polygone.pdf*.**
- (4) Vérifier que le polyèdre résultant de la fusion est convexe (cf. partie calcul **convexité polygone**).

Exemple :

Avant fusion





Après fusion

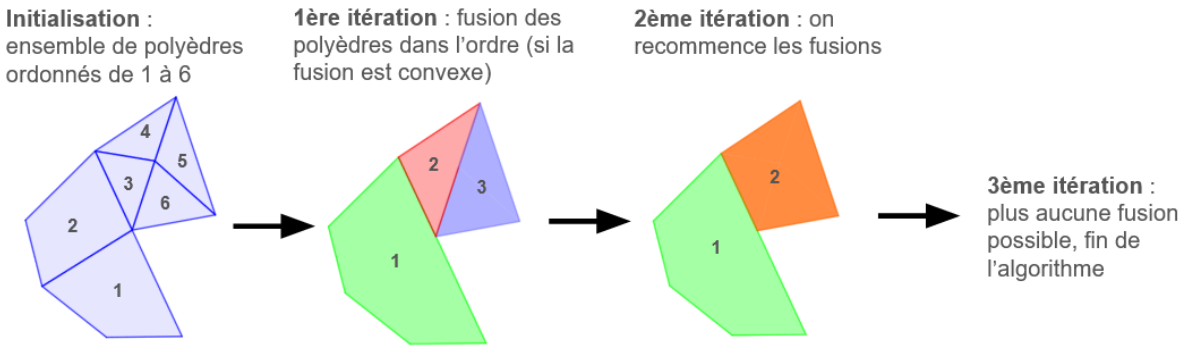
2. Fusion d'un ensemble de polyèdres

On parcourt la liste de polyèdres dans un ordre fixé (c'est ce qu'on appelle une **solution**). On commence par essayer de fusionner les deux premiers polyèdres. Si la fusion est convexe, on continue en ajoutant le troisième polyèdre, puis les suivants, jusqu'à ce que le polyèdre résultant de la fusion ne soit plus convexe. À ce moment-là, on repart du dernier polyèdre à partir duquel la fusion a cessé d'être convexe et on reprend le processus en parcourant la liste.

Algorithme en pseudo-code

```
Entrée : listePoly  // une liste ordonné de polyèdres

Tant qu'on arrive à faire de nouvelles fusions
   // Le bloc de code suivant correspond à une itération dans l'exemple ci-dessous

   polyCourant ← premier polyèdre de la liste listePoly
   listePolyFusion ← liste vide  // liste de polyèdres après tests de  fusion

   Pour chaque polyèdres (polySuivant) de la liste, sauf le premier
         polyFusionné ← fusion(polyCourant, polySuivant)
         si polyFusionné convexe
               polyCourant ← polyFusionné
         sinon
               ajouter polyCourant dans la liste listePolyFusion
               polyCourant ← polySuivant
         finsi
   finpour
   listePoly ← listePolyFusion
fintant

Sortie : listePolyFusion
```

Exemple de l'algorithme de fusion sur un exemple avec 6 polygones (même principe qu'avec des polyèdres) :



Pour la **première itération**, on teste la fusion de [1] avec [2]. La fusion est convexe, donc on essaye avec [3]. Le polygone formé par [1], [2] et [3] n'est pas convexe donc on reprend les fusions à partir de [3], et on répète cette méthode jusqu'à la fin.

Lors de la **troisième itération**, on teste la fusion de [1] avec [2]. Comme ce n'est pas possible, l'itération se termine sans avoir fait de nouvelle fusion, donc l'algorithme s'arrête.

VI. Graphe des fusions convexes

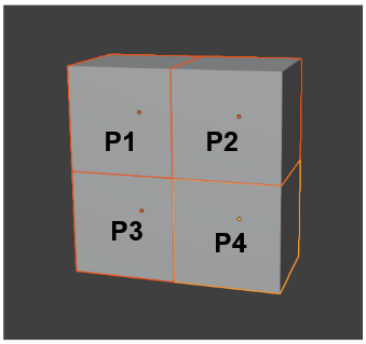
Nous avons utilisé un **graphe non orienté** pour modéliser les fusions entre les différents polyèdres qui sont convexes. Les polyèdres sont représentés par les sommets, et les arêtes représentent les fusions convexes. Ce graphe permet à la fois de réduire le nombre de calculs de fusion et aide aussi à évaluer les fusions (cf. partie *Fonction objectif*).

Après la lecture d'un fichier *.obj, un graphe est **pré-calculé** pour les fusions de toutes les paires de polyèdres possibles.

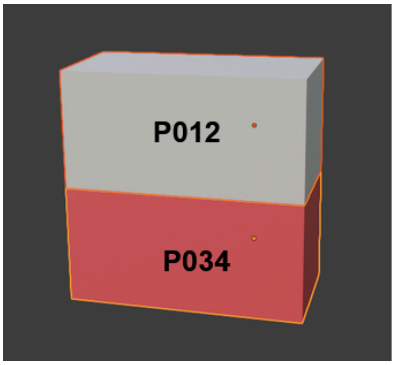
L'algorithme de fusion vérifie d'abord si la fusion a déjà été testée et si elle présente dans le graphe, auquel cas on peut directement récupérer le polyèdre résultant de la fusion. Sinon il calcule la fusion selon l'algorithme présenté précédemment, puis met à jour le graphe dynamiquement.

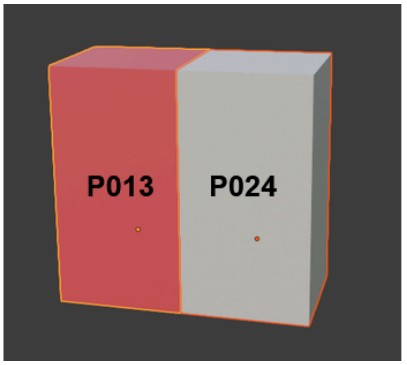
Les polyèdres ont tous un **identifiant** qui va de 1 jusqu'au nombre de polyèdres dans le fichier. Les polyèdres résultants de fusions ont un identifiant préfixé de "*0*", suivi des identifiants des polyèdres qui ont servi à faire la fusion, dans un ordre croissant.

Exemple :
4 polyèdres convexes (cubes) P1, P2, P3, P4

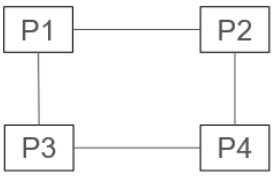


Après la lecture du fichier, pré-calcul du graphe pour les fusions possibles de 2 polyèdres. Ainsi les fusions ci-dessous sont déjà calculées.

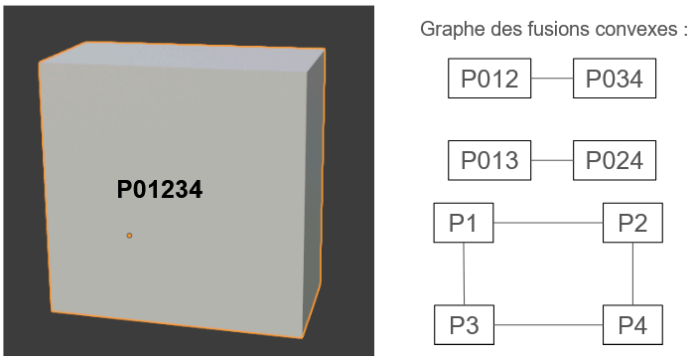




Graphe des fusions convexes :



Lorsque l'algorithme de fusion essaye de fusionner **P012** et **P034** ou **P013** et **P024** ensemble, il calcule la fusion résultante puis met à jour le graphe.

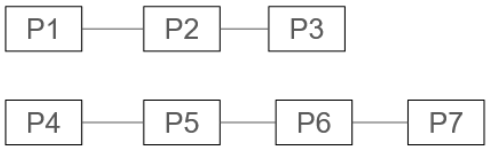


Graphe des fusions convexes :

La distance minimale du graphe est fixée à **1** (distance entre deux sommets voisins).

Pour obtenir la distance maximale, il faut calculer le diamètre du graphe (cf. la méthode *Graph::calculateDiameter*), c'est-à-dire la distance entre les deux sommets les plus éloignés, à laquelle on ajoute 1 pour obtenir la distance maximale.

Exemple :
Soit le graphe des fusions convexes ci-dessous :



Distance entre **P1** et **P2** = **1**
Distance entre **P4** et **P7** = **3** (c'est le diamètre du graphe)
Distance maximale = **3** + 1 = **4**
Distance entre **P1** et **P4** = distance maximale = **4**

VII. Algorithme Brute-force

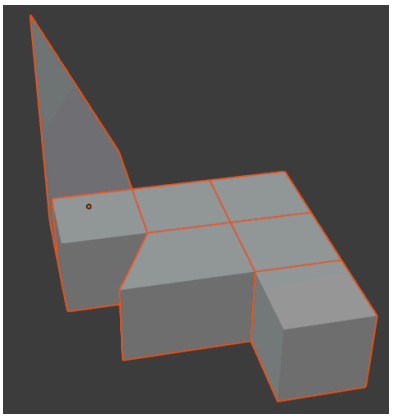
La première étape du projet a été de réaliser un algorithme de force brute, réalisant toutes les permutations possibles. L'algorithme de force brute implémenté dans ce code suit le principe de tester toutes les permutations possibles des polyèdres pour trouver des solutions optimales de fusion. Après avoir lu depuis un fichier *.obj et créé dynamiquement les polyèdres, l'algorithme génère toutes les permutations possibles (dont le nombre est factoriel de n) et applique un algorithme de fusion à chaque permutation pour trouver une solution complète. Pour chaque permutation, si une solution complète est obtenue, le nombre de polyèdres dans cette solution est comparé à une taille minimale stockée et mise à jour si cette solution est plus petite ou égale au précédent minimum. Cela permet de conserver uniquement les solutions optimales en termes de taille.

De plus, l'algorithme implémente un mécanisme d'arrêt prématuré en utilisant le paramètre *minNbPolySolution* dans la fonction *mergeAlgorithm*. Si, au cours du processus de fusion, une solution intermédiaire dépasse la taille de la meilleure solution trouvée jusqu'à présent, l'algorithme arrête immédiatement cette permutation, évitant ainsi l'utilisation inutile de ressources.

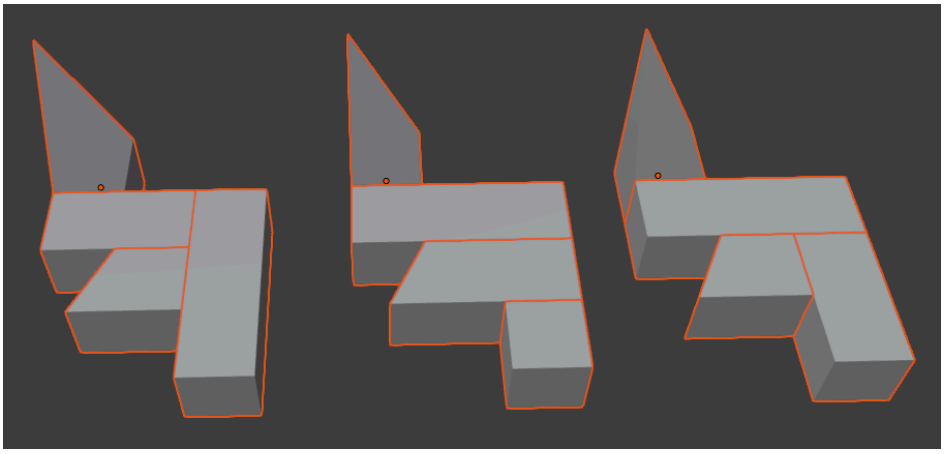
Afin de s'assurer que chaque solution est unique et éviter les doublons, une vérification est effectuée via la fonction *isSolutionAlreadyFinded*. Cette fonction compare la nouvelle solution trouvée avec toutes les solutions déjà enregistrées et ne la stocke que si elle n'a pas été trouvée précédemment. Elle pourrait être réécrite et optimisée en utilisant le graphe des fusions convexes, car elle a été écrite avant. Les solutions optimales sont ensuite sauvegardées sous forme de fichiers *.obj, et diverses statistiques, telles que le nombre de permutations effectuées, le nombre de solutions complètes et uniques, et le nombre optimal de polyèdres fusionnés, sont affichées pour évaluer les performances de l'algorithme.

L'algorithme utilise également un graphe pour optimiser le processus de fusion, représenté par l'attribut *d_mergeGraph* dans la classe de base *Algorithm*. Le graphe aide à déterminer les fusions possibles entre les polyèdres, ce qui peut réduire considérablement le nombre de combinaisons à tester et améliorer l'efficacité de l'algorithme global. En d'autres termes, chaque fusion entre deux polyèdres n'est calculée qu'une seule fois au cours de l'algorithme.

Exemple : Exécution de l'algorithme brute-force pour un fichier avec 7 polyèdres (cf. fichier *exemple3.obj*)



Après exécution de l'algorithme brute-force, on obtient le nombre minimal de polyèdres convexes possibles, c'est-à-dire 4. Les 3 résultats optimaux différents possibles sont écrits dans des nouveaux fichiers .obj.



L'algorithme Brute-force permet de trouver le ou les résultats optimaux (et donc les solutions qui conduisent à ces résultats, car plusieurs solutions peuvent donner les mêmes fusions) pour chaque problème. L'inconvénient est que sa **complexité algorithmique** est de l'ordre de ***n!*** **(factorielle de n**, **n** étant le nombre de polyèdres du fichier lu), cela signifie que le temps d'exécution de l'algorithme croît très rapidement en fonction **n**. Plus précisément, si **n** augmente, le nombre d'opérations nécessaires pour terminer l'algorithme augmente de manière exponentielle. Par exemple pour le fichier *exemple.obj* qui contient **32 polyèdres**, il y a environ **2.631×10^35 permutations possibles** à tester. L'algorithme Brute-force n'est donc utilisable que sur des exemples avec très peu de polyèdres (plus ou moins 10 polyèdres maximum selon les capacités de calcul pour un temps raisonnable).

VIII. Métaheuristiques

Nous avons testé le recuit simulé et l'algorithme génétique parce que ce sont des méta-heuristiques couramment utilisées lorsque l'on ne peut pas obtenir de solutions exactes ou correctes rapidement. Cela permet de trouver des solutions approximatives en un temps raisonnable. De plus, le recuit simulé et les algorithmes génétiques sont deux types de méta-heuristiques différentes.

Le **recuit simulé** (simulated annealing ou SA en anglais) fonctionne par exploration successive de voisinage. C'est-à-dire qu'il y a une modification progressive d'une solution actuelle pour explorer ses voisins.

L'algorithme génétique fonctionne par exploration simultanée de solutions différentes. Cette méthode travaille avec une population de solutions en parallèle. Chaque solution est un "individu" dans la population, et les opérations génétiques permettent d'explorer plusieurs directions simultanément.

Ces deux algorithmes ont été testés sur le fichier *exemple.obj* qui est une roue composée de 32 polyèdres (donc impossible à résoudre avec l'algorithme Brute-force).

1. Fonction objectif

Le principe d'une fonction objectif est d'évaluer la qualité d'une solution, plus elle est proche de la solution optimale plus la valeur de son évaluation est faible et inversement. Le tout est de trouver des critères pertinents qui permettront d'orienter les recherches dans l'espace des solutions possibles. Dans notre cas la fonction objectif est à minimiser.

a. Calcul de la récompense

La formule de la récompense implémentée dans notre algorithme permet d'avantager les solutions qui contiennent des grosses fusions, et est définie par la formule suivante :

$$\frac{2\left(\sum_{i=1}^{\text{nbPoly}}(\text{taille polyèdre}_i)^{1.35}\right)}{\text{Nombre de polyèdres} - 1}$$

- **taille polyèdre** : nombre de fusions effectuées pour obtenir le polyèdre i
  exemple : si on fusionne 3 polyèdres ensemble, vaut **2**

- **Nombre de polyèdres - 1** : nombre de polyèdres restants après avoir effectué les fusions moins 1 (correspond à la taille de polyèdre maximale)

On met la somme des tailles à la puissance **1.35** pour avantager légèrement les solutions qui contiennent des grosses fusions. La puissance valait **2** initialement, mais cela désavantageait trop les solutions qui contenaient beaucoup de petites fusions (qui peuvent être intéressantes). Le numérateur est multiplié par 2 pour équilibrer le résultat par rapport au calcul de l'objectif. Une piste d'amélioration est de faire évoluer le calcul de la récompense au cours de l'algorithme, par exemple pour favoriser les fusions nombreuses plus petites au début puis les grosses fusions quand l'algorithme est plus avancé.

b. Calcul de la pénalité

La formule de la pénalité implémenté dans notre algorithme permet de désavantager les solutions qui essayent de fusionner des polyèdres qui sont éloignés au sens du graphe des fusions convexes et est défini par la formule suivante :

$$\frac{\left(\frac{\sum_{i=2}^{\text{nbPolyInit}} \text{distance}(i-1,i)}{\text{distance max} \times \text{nbPolyInit}}\right)^2}{3}$$

- **distance(i-1, i)** : distance dans le graphe entre les polyèdres i-1 et i
- **distance max** : distance maximale du graphe
- **nbPolyInit** : nombre de polyèdres dans la solution initiale (avant les fusions)

La division par 3 permet d'équilibrer le résultat par rapport au calcul de l'objectif.

c. Calcul de l'objectif

L'objectif, qui représente **l'évaluation globale de la solution**, prend en compte le nombre final de polyèdres après avoir effectué les fusions, auquel on ajoute une pénalité et une récompense calculés précédemment, c'est-à-dire des distances entre les polyèdres qu'on essaye de fusionner, du nombre et de la taille des fusions effectuées. Pour rappel, on cherche à minimiser l'objectif, une valeur plus faible représente une meilleure solution.

La formule de la fonction objectif implémenté dans notre algorithme est la suivante :

$$1 + \left(\frac{\text{Nombre de polyèdres après fusion}}{\text{Nombre de polyèdres}}\right) + \text{pénalité} - \text{récompense}$$

- **Nombre de polyèdres après fusion** : nombre de polyèdres après avoir effectué l'algorithme de fusion sur la solution

- **Nombre de polyèdres** : nombre de polyèdres initial de la solution

2. Recuit simulé

a. Principe

Le recuit simulé est basé sur l'analogie avec le refroidissement lent d'un matériau solide. Lorsque ce matériau est chauffé à une température élevée, ses atomes peuvent se déplacer librement, ce qui leur permet d'explorer diverses configurations énergétiques. En refroidissant lentement, les atomes se réorganisent progressivement pour atteindre une configuration de basse énergie, idéalement la plus stable.

Cet algorithme est plutôt simple à implémenter, il permet de sortir d'optima locaux et s'adapte assez bien à notre problème. Cependant il nécessite un ajustement précis de ses paramètres pour une exploration efficace. De plus, les performances dépendent beaucoup de la qualité des solutions voisines générées.

b. Implémentation

1. Initialisation

- Choisir une solution initiale aléatoire *S0*.
- Définir une température initiale élevée
- Définir un facteur de refroidissement de la température
- Fixer un nombre maximal d'itérations
- Nombre limite d'itérations infructueuses avant d'augmenter la température

2. Exploration

A chaque itération, on va générer une **solution voisine *S'*** à la **solution actuelle *S***. On génère le voisin par permutation de la solution S (on peut considérer que c'est une mutation, donc un permutations de gènes).

Voici les différentes permutations augmentée :

- Permutation aléatoire de n éléments (une ou plusieurs fois d'affilé)
- Permutation de 2 éléments côtes à côtes (une ou plusieurs fois d'affilé)
- Nombre de permutations variable selon la température.

La permutation sur peu d'éléments fonctionne mieux, car de petits changements peuvent avoir de grandes conséquences et cela évite de chercher des voisins complètement différents. De plus, la permutation aléatoire semble plus pertinente dans le contexte de notre projet que celle d'éléments côtes à côtes.

3. Évaluation et acceptation

On évalue les solutions **S** (courante) et **S'** (voisine) avec la fonction objectif et on calcule la différence entre ces deux solutions.

Si la solution voisine est meilleure que la solution actuelle, on l'accepte.
Si on accepte toujours les meilleures solutions, on peut se retrouver piéger dans des minima locaux. Pour éviter cela, on va définir un critère d'acceptation qui nous permettra d'accepter parfois des solutions qui sont plus mauvaises.

On considère la solution acceptée si un nombre généré aléatoirement est supérieur au nombre généré par les formules suivantes (sachant que l'évaluation voisine sera toujours supérieure à celle courante, car sinon elle est acceptée d'office).

Chaque critère d'acceptation :

- La formule génère un nombre compris entre 0 et 1, **S** correspond à la **solution courante** et **S'** correspond au **voisin**
- Lorsque la **température diminue**, la **probabilité d'accepter** une solution plus mauvaise **diminue** et inversement.
- Lorsque que **l'écart** entre la solution courante et la solution voisine **augmente** (donc pire est la solution voisine par rapport à celle courante), la **probabilité d'accepter** le voisin **diminue**.

Critères d'acceptation testés :

Formule 1

Acceptation si un nombre généré aléatoirement entre 0 et 1 est **inférieur** au résultat.

$$\exp\left(\frac{|\text{évaluation } S' - \text{évaluation } S|}{\text{température courante}}\right) - 1$$

- Le résultat varie trop fortement en fonction de la température et pas assez en fonction de l'écart entre l'évaluation des solutions.

Formule 2

Acceptation si un nombre généré aléatoirement entre 0 et 1 est **supérieur** au résultat.

$$\exp\left(-\frac{|\text{évaluation } S' - \text{évaluation } S|}{\text{température courante}}\right)$$

- Ce critère donne des résultats cohérents, mais la probabilité d'acceptation est trop faible, notamment quand la température n'est pas très élevée (cf. *Test 2* dans la partie *Tests)*

Formule 3 (retenue)

Acceptation si un nombre généré aléatoirement entre 0 et 1 est **supérieur** au résultat.

$$1 - \left(\frac{0{,}01}{|\text{évaluation } S' - \text{évaluation } S|} \times \frac{\text{température courante}}{\text{température initiale}}\right)$$

- Ce critère s'adapte le mieux au paramétrage de notre recuit simulé, elle permet de sortir rapidement des minimum locaux et d'explorer beaucoup de solutions. Malgré tout, la probabilité d'acceptation semble être un peu trop élevée.

Dans l'idéal il faudra une formule qui s'adapte encore plus au comportement de l'algorithme.

4. Refroidissement

Réduction de la température au cours de l'algorithme en fonction d'un **facteur de refroidissement**. Toutes les **24** itérations(définit arbitrairement dans le code) la température est réduite en étant multipliée par le facteur de refroidissement (généralement compris entre 0.90 et 0.99.

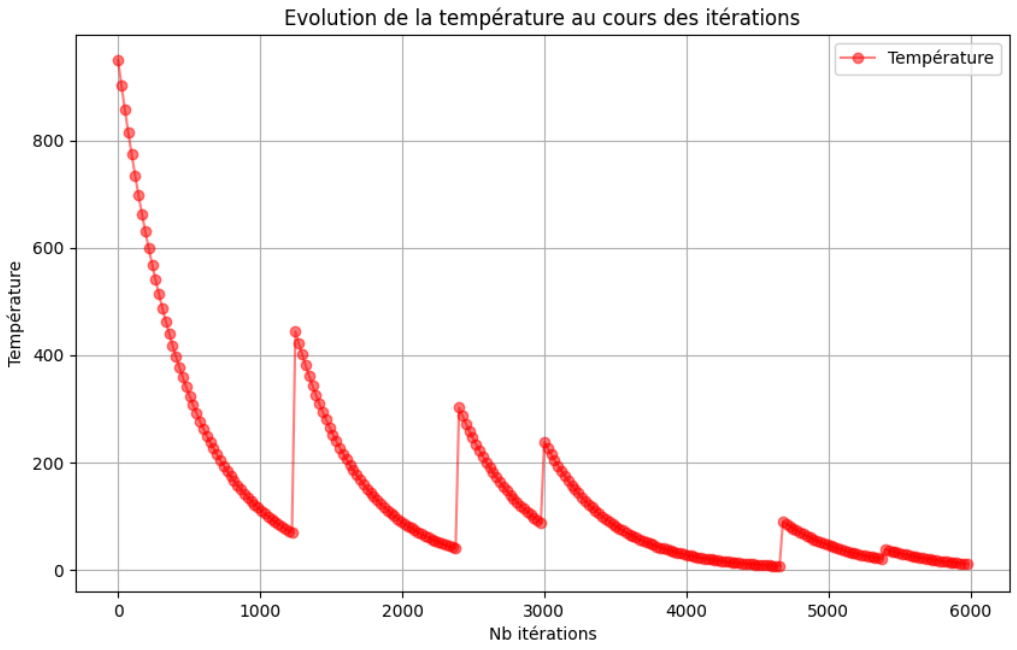
Cependant, si **l'algorithme stagne** pendant un certain temps, il est judicieux de faire **remonter la température**. En effet l'exploration et l'acceptation des solutions sont liées à la température, donc la faire augmenter peut permettre de sortir d'un minimum local en allant chercher des solutions plus éloignées.

Formule

$$\left(\exp\left(-\frac{\text{itération courante}}{\text{nombre total d'itérations}}\right) - \exp(-1)\right) \times \text{température initiale}$$

Cette formule permet d'augmenter la température en fonction du nombre d'itérations courantes. Plus on se rapproche de la fin de l'algorithme, moins la température est augmentée.

Exemple de l'évolution de la température au cours des itération :



5. Critère d'arrêt

L'algorithme s'arrête soit quand le nombre maximum d'itération est atteint. A la base on avait mis en place un mécanisme d'arrêt prématuré si l'algorithme stagne plusieurs fois d'affilée. Cependant au vu des performances actuelles de notre algorithme, il n'est pas bon de se priver d'itérations car la convergence est assez lente.

c. Tests

Cette section présente des tests représentatifs (c'est-à-dire qu'on devrait généralement retrouvé le comportement observé avec la même configuration) qui ont été réalisés pour montrer le comportement de l'algorithme en fonction de différents paramètres

Paramètres utilisés pour tous les résultats suivants (**sauf indication contraire, en gras**) :

- Température initiale : **1000**
- Facteur de refroidissement : **0.95**
- Nombre maximum d'itérations : **6000**
- Nombre limite d'itérations infructueuses avant d'augmenter la température : **600**
- Permutation : permutation de **2 éléments côtes à côtes, 3 fois**
- **Critère d'acceptation** :

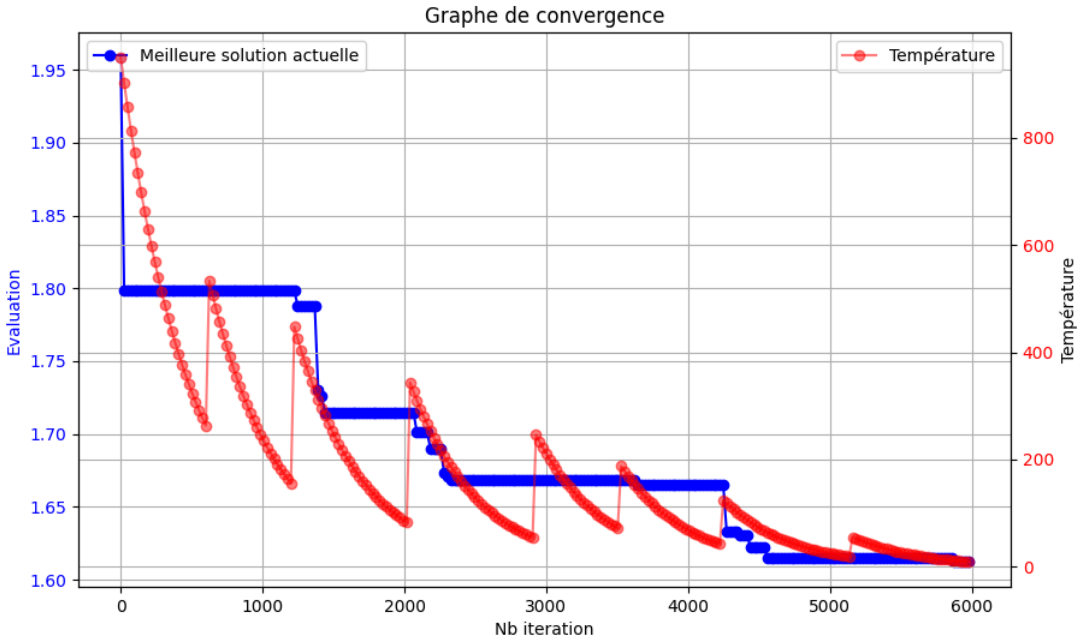
$$1 - \left(\frac{0{,}01}{\Delta(\text{évaluation } S' - \text{évaluation } S)} \times \frac{\text{température courante}}{\text{température initiale}} \right)$$

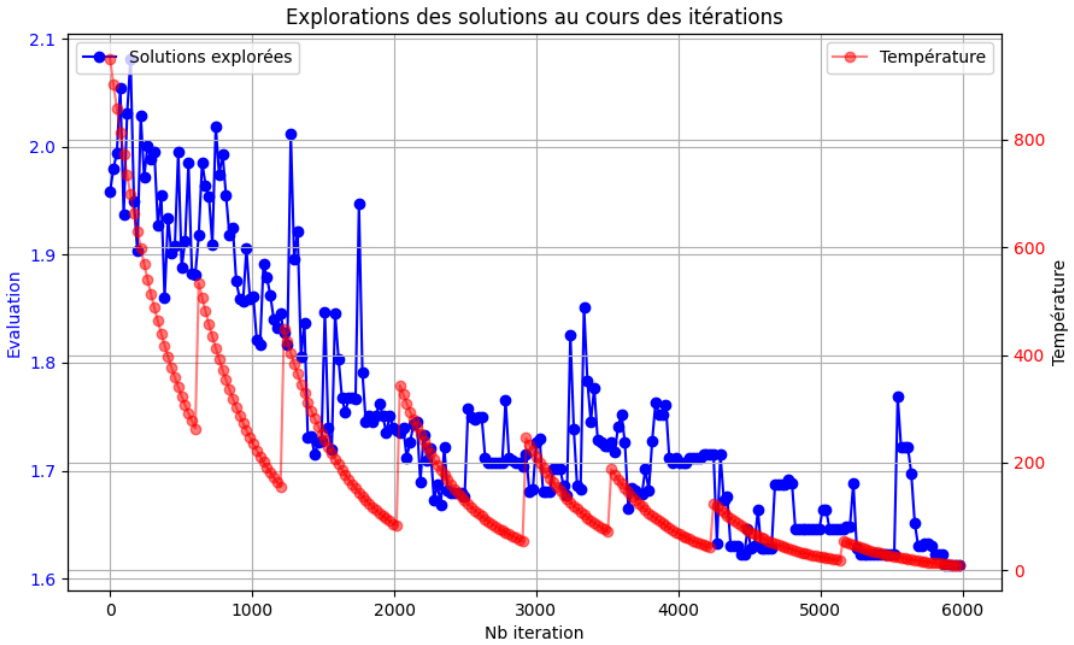
Interpréter les graphiques :

- Les **graphiques de convergence** représentent l'évolution de la meilleure solution trouvée au cours des itérations. La température est affichée pour comprendre l'impact de cette dernière sur la convergence (sortir d'un optimum local par exemple)
- Les **graphiques "d'exploration de solutions au cours des itérations"** représentent l'évolution des solutions acceptées (cf. partie Implémentation) au cours de l'algorithme.

Test 1

Interprétation : La meilleure solution trouvée converge de manière assez régulière même si l'exploration des solutions a l'air un peu chaotique. Cependant l'allure de la courbe se rapproche de ce qu'on recherche.



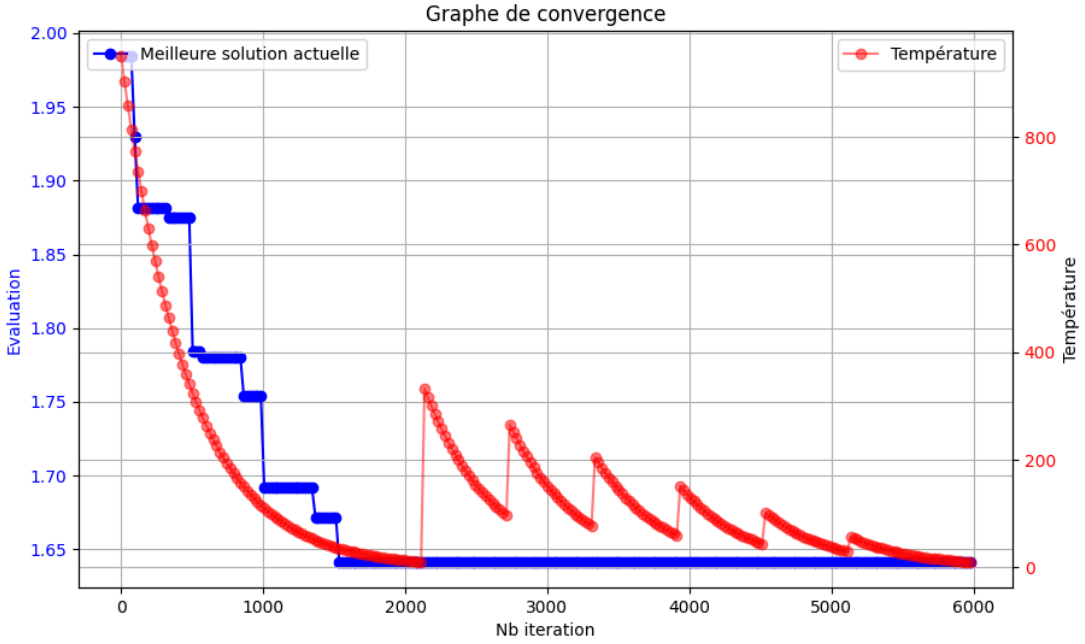


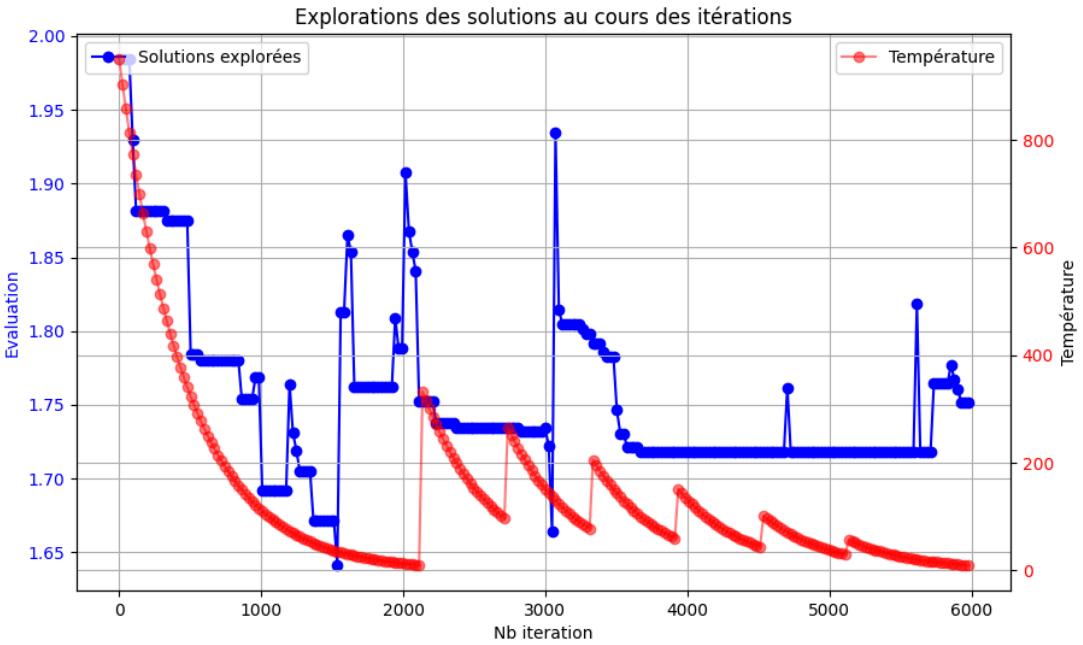
Test 2

Critère d'acceptation (cf. *formule 2*):

$$\exp\left(-\frac{|\text{évaluation } S' - \text{évaluation } S|}{\text{température courante}}\right)$$

Interprétation : La probabilité d'acceptation est trop faible, l'algorithme stagne beaucoup et on peine à sortir des minima locaux quand la température est inférieure à 400.

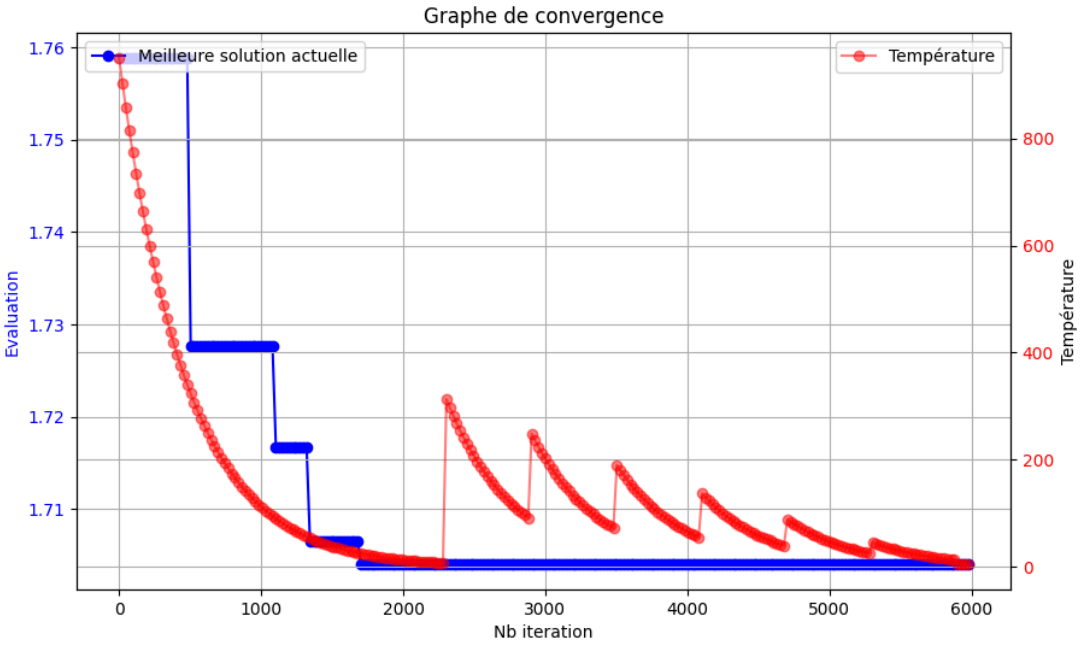


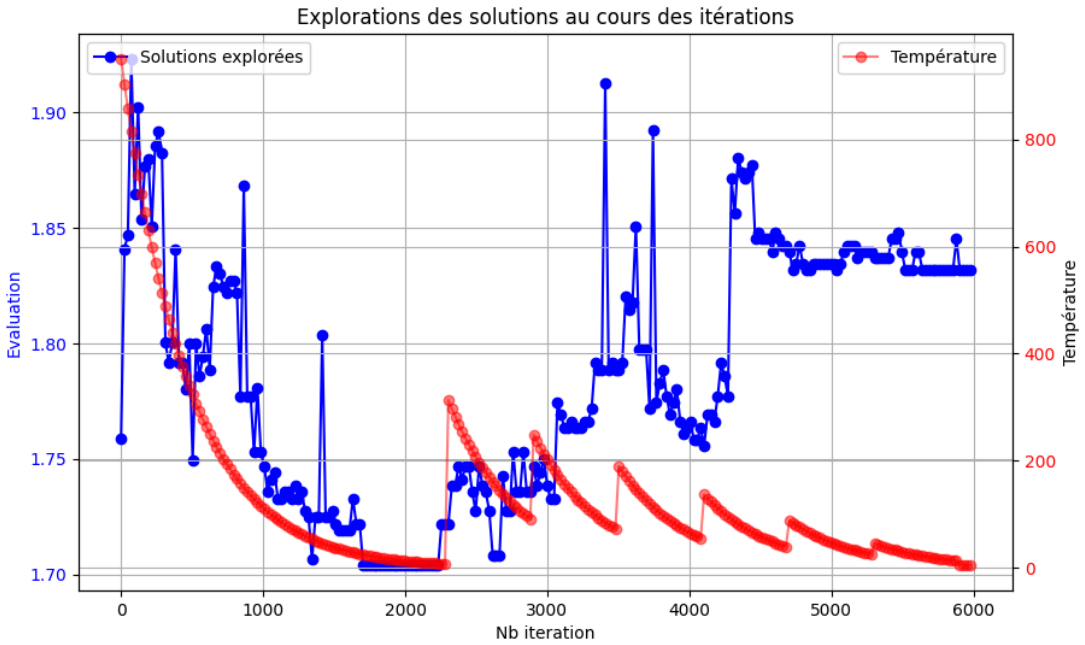


Test 3

Permutation : une seule permutation de 2 éléments côtes à côte

Interprétation : Avec les paramètres actuels, une seule permutation n'est pas suffisante pour améliorer l'évaluation avant que la température n'augmente. L'exploration a tendance à empirer.



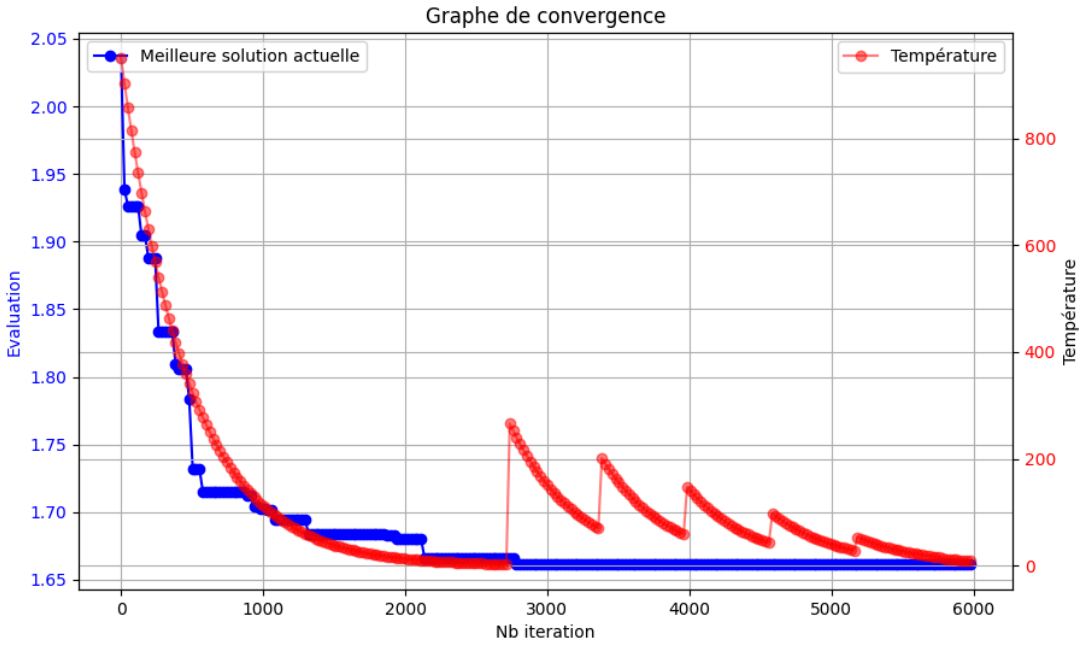


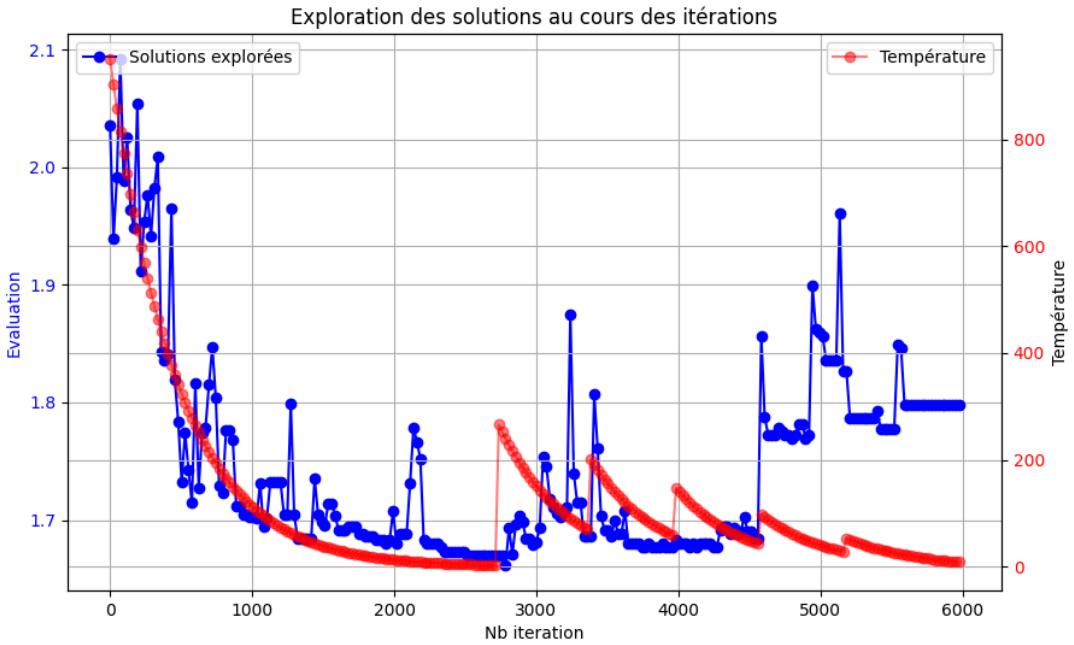
Test 4

Nombre de permutations : varie linéairement selon la température de 2 à 5, exemple :

- 1000°C : permute 2 éléments côtes à côte **5** fois
- 0°C : permute 2 éléments côtes à côte **2** fois

Interprétation : Semble intéressant, cependant on constate que l'exploration empire à partir de l'itération 4500. Il pourrait être pertinent de faire varier le nombre de permutations autrement que linéairement, voire avec un ou plusieurs autres critères en plus ou à la place de la température.

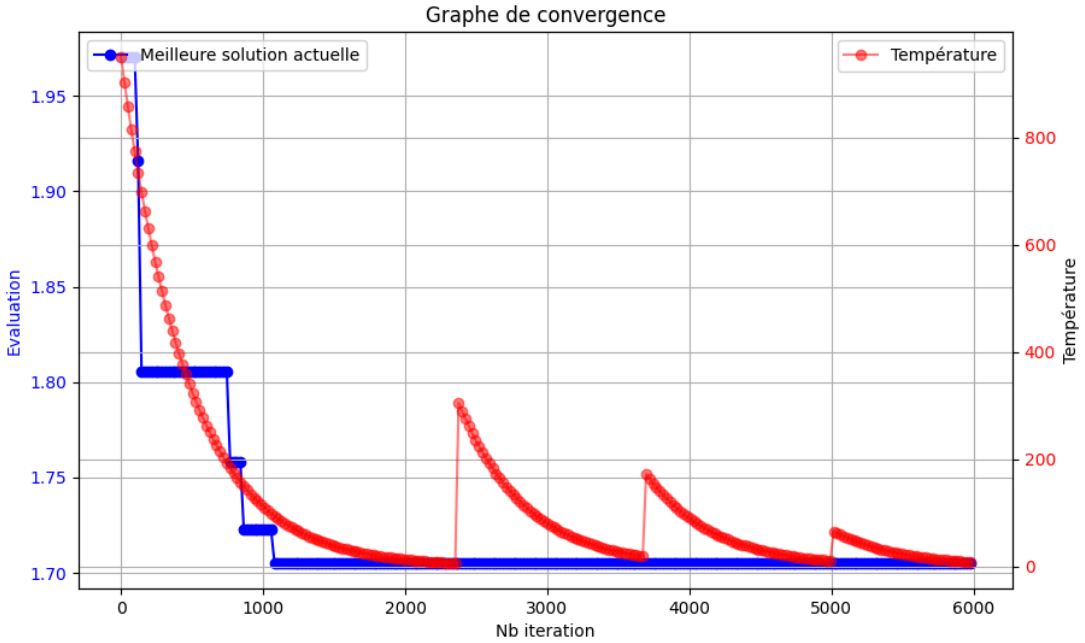


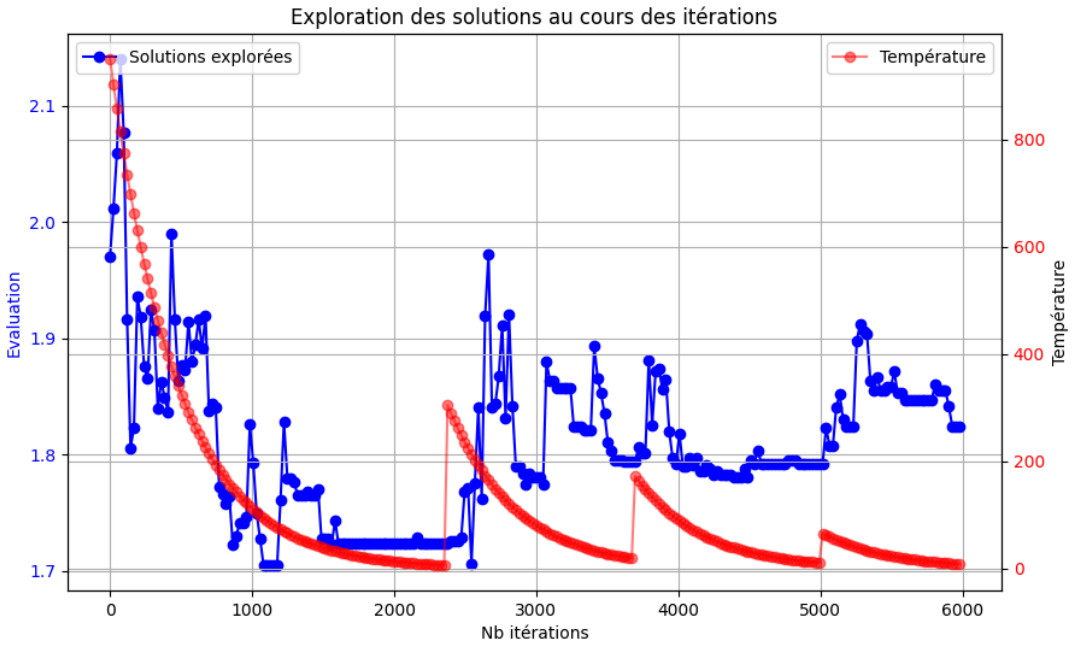


Test 5

Nombre limite d'itérations infructueuses avant d'augmenter la température : 1300

Interprétation : Même si la température descend plus lentement et qu'il y a plus d'itérations avant de remonter, un optimum local est rapidement atteint. On peut en conclure qu'avec le paramétrage actuel, il est inutile d'avoir une limite d'itérations infructueuses trop grande.

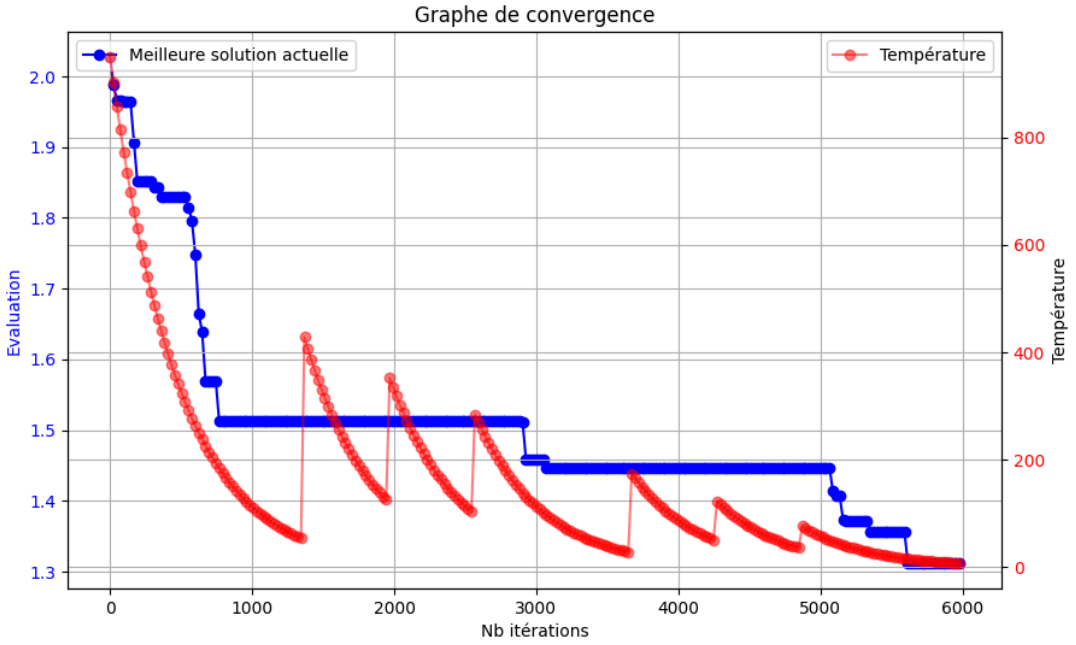


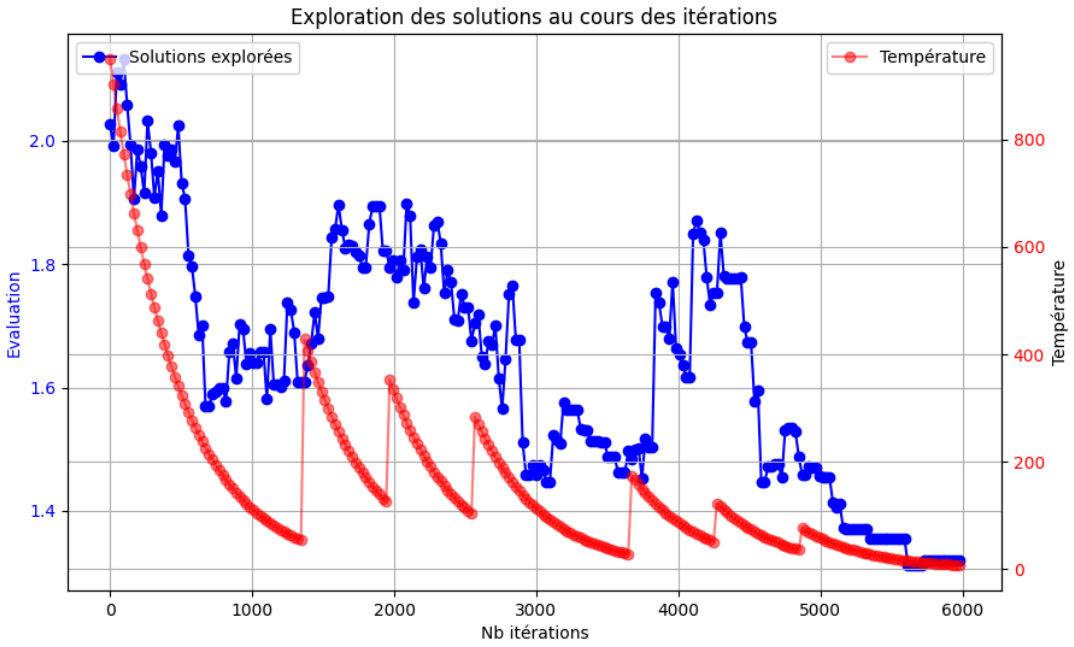


Test 6

Permutation de 3 éléments choisis aléatoirement

Interprétation : C'est un des résultats les plus prometteurs obtenus. Bien que la convergence est lente, on arrive tout de même à sortir des optima locaux.

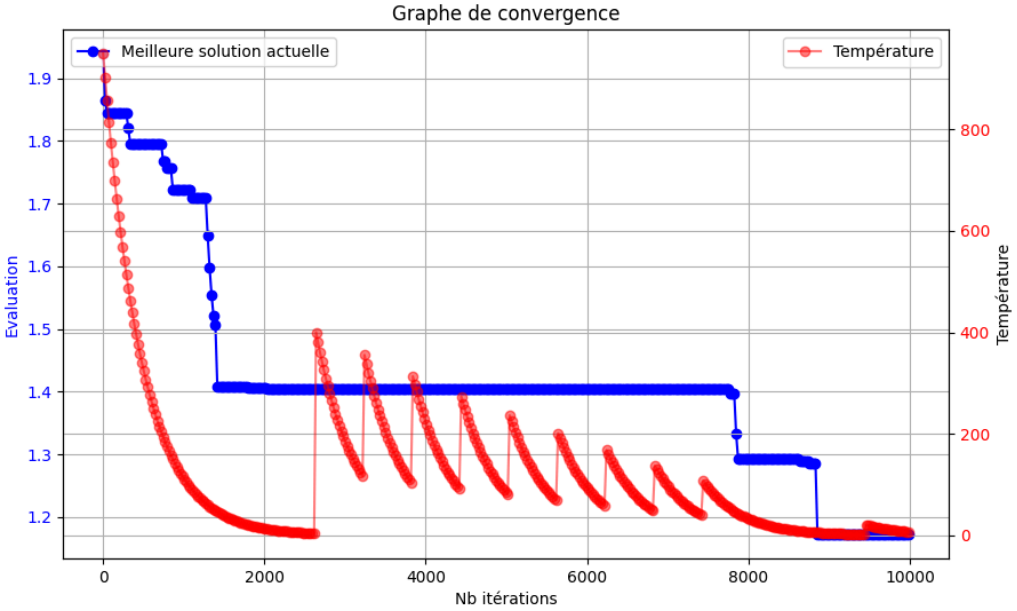


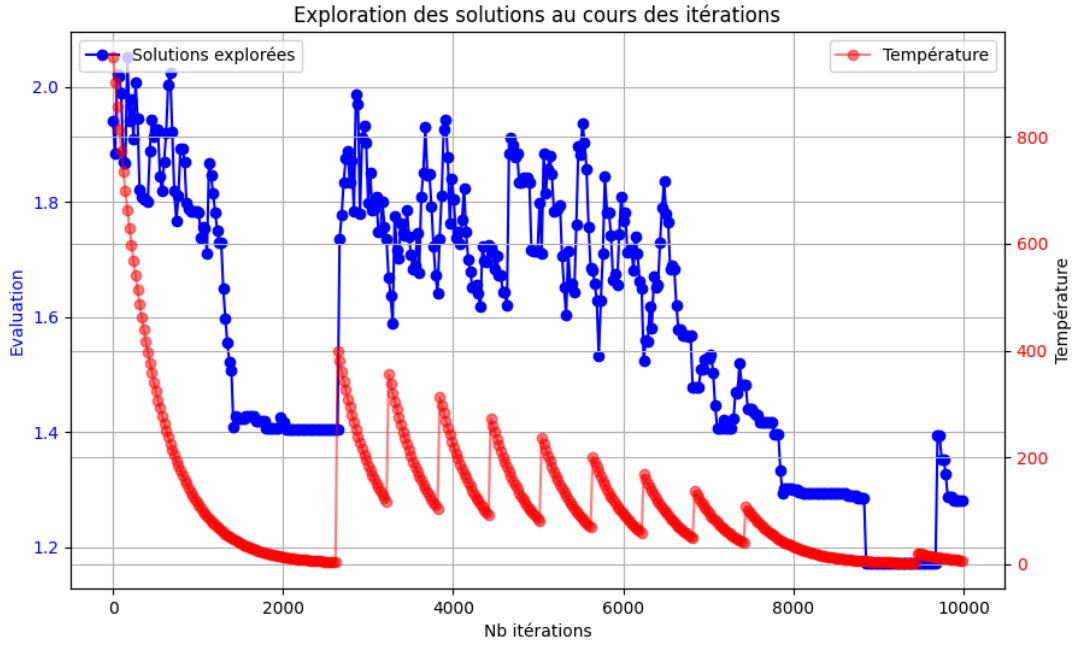


d. Conclusion vis à vis des tests

A priori, de ce qui a été testé, le meilleur comportement est obtenu en permutant 3 éléments aléatoirement, et en faisant remonter la température après 600 itérations à stagner (600 itérations impliquent de faire diminuer la température rapidement, avec un facteur de refroidissement de 0.95 par exemple). C'est ce qu'on constate avec le **test 6**.

L'exemple ci-dessous (*page suivante*) a été testé avec les mêmes paramètres que le **test 6** mais avec une limite de **10000 itérations**. On constate que cette configuration permet de sortir d'un minimum local, mais cela prend beaucoup d'itérations, ce qui signifie que l'algorithme met beaucoup de temps à converger. Il y a donc encore bien des points de l'algorithme à améliorer, mais c'est le meilleur résultat trouvé dans la limite impartie du projet.


Graphe de convergence
Meilleure solution actuelle
Température
Evaluation
Température
Nb itérations
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
800
600
400
200
0
0
2000
4000
6000
8000
10000


Exploration des solutions au cours des itérations
Solutions explorées
Température
Evaluation
Température
Nb itérations
2.0
1.8
1.6
1.4
1.2
800
600
400
200
0
0
2000
4000
6000
8000
10000

3. Algorithme génétique

a. Implémentation

Un algorithme génétique est une méthode d'optimisation inspirée de la sélection naturelle, où des populations de solutions potentielles évoluent à travers des processus de sélection, croisement et mutation. Il est utilisé pour résoudre des problèmes complexes en recherchant des solutions proches de l'optimal dans un espace de solution vaste. Nous avons choisi d'implémenter cette méta-heuristique car elle est couramment utilisée. Un algorithme génétique est séparé en plusieurs étapes importantes :

- La création de la population initiale
- L'évaluation de cette population
- La sélection des individus utilisés comme parents pour la nouvelle population
- Les croisements
- Les mutations
- Le remplissage de la nouvelle population
- L'évaluation des nouveaux individus

Dans le cadre du projet, un individu est représenté par un vector<int>, soit une liste de permutations d'index de polyèdre. Une population est constituée de N individu.

Notre implémentation possède plusieurs caractéristiques techniques permettant une modularité de sélection, croisement et mutation. La classe *GeneticAlgorithm* permet de construire un objet correspondant à l'algorithme génétique, en prenant divers paramètres en compte tels que la taille de la population, la probabilité de mutation, le nombre d'itérations maximum, une sélection, un croisement et une mutation.

Les classes *Selection*, *Crossover*, *Mutation* sont des classes abstraites permettant de créer et d'intégrer facilement sa propre implémentation. Pour cela il suffit de faire hériter sa nouvelle classe de *Selection*, *Crossover* ou *Mutation* et redéfinir respectivement les méthodes *select*, *cross* ou *mutate.* La classe *Population* contient actuellement une méthode pour initialiser la population, et est pensée pour pouvoir en ajouter facilement.

- Sélection : la méthode *select* prenant en paramètre la population et le score de la population est à redéfinir pour chaque type de sélection. La classe *Selection* possède une liste de vector<int>, correspondant aux individus parents, une fonction pour retourner cette liste et une fonction pour effacer cette liste. Elle possède aussi un vector<int> d_parent_1 et d_parent_2.

- Croisement : la méthode *cross* prend en paramètre le premier et le deuxième parent, ainsi que des variables pour stocker les premiers et les seconds enfants générés à la fin du croisement. Cette méthode doit être redéfinie pour chaque type de croisement.

- Mutation : la méthode mutate prend en paramètre un individu (vector<int>) sur lequel la mutation est appliquée, et elle doit être redéfinie. L'individu muté est une variable de sortie.

Voici les différentes méthodes d'initialisation, sélection, croisement et mutation implémentées dans le cadre du projet. Nous nous sommes aidés d'un document ci-joint pour les différentes méthodes. La première méthode implémentée est l'initialisation aléatoire de la population : N individu sont initialisés aléatoirement en faisant en sorte de ne pas avoir de doublons d'individus ni de doublons de gènes.

Trois méthodes de sélection sont implémentées :

- La sélection par score (*BestScoreSelection*), où chaque individu de la population est trié en fonction de son score. Le but étant de minimiser la fonction objectif, les meilleurs individus sont placés au début de la population avec un score faible. Les individus les moins bons sont placés à la fin de population. Cette méthode sélectionne un nombre pair de parents, équivalent à environ la moitié de la population. Ce paramètre peut être changé.

- La sélection par tournoi (*TurnamentSelection*) réalise N fois des tournois dans un sous ensemble d'individus appartenant à la population (de taille *nbIndivInTurnament*) , avec un seul gagnant. Chaque gagnant est ajouté à la liste des parents et retiré de la population pour éviter les doublons entre les N tournois.

- La sélection par roulette (*WheelSelection*) réalise N fois une sélection aléatoire d'un individu présent dans la population. Cet individu est ensuite ajouté à la liste des parents et retiré de la population pour éviter les doublons.

Une méthode de croisement est implémentée. Le croisement en N points (*NXCrossover*) permet de croiser deux individus entre N gènes et de récupérer deux enfants. Le nombre de points de croisement doit être inférieur au nombre de gènes. Chaque point de croisement est choisi aléatoirement et sans doublons. Les gènes des parents passés en paramètres se croisent entre les points de croisement.

Une méthode de mutation est implémentée (*InsertionMutation*). Cette méthode prend en paramètre un individu, choisit aléatoirement une position d'un gène et le remplace par une nouvelle valeur aléatoire, sans créer de doublons. L'individu passé en paramètre est donc modifié.

b. Fonctionnement

Pour faire fonctionner correctement l'algorithme, il faut au préalable créer un objet *GeneticAlgorithm* avec le nom du fichier à lire, la taille de la population, la probabilité de mutation, le nombre d'itération maximum ainsi qu'un objet *Selection*, *Crossover* et *Mutation.* Le lancement de l'algorithme se fait avec la méthode run. Une fois l'algorithme lancé, un chronomètre se lance suivi de l'initialisation de la population et de l'évaluation de celle-ci. Si la taille de la population choisi est plus grande que le nombre de permutations possible, on exporte la meilleur solution trouvé pendant l'initialisation et on arrête l'algorithme, sinon, tant que le nombre d'itérations réalisés est inférieur au nombre d'itérations maximum, on réalise ses actions :

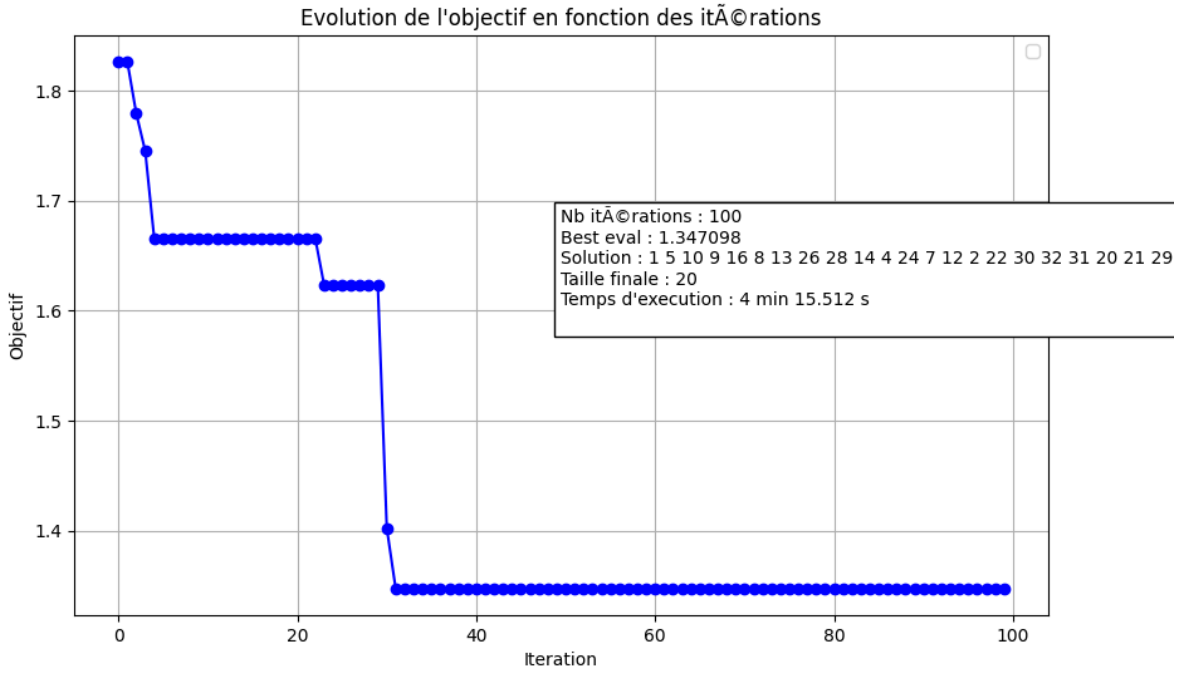
- Sélection de parents dans la population
- Croisement des parents, mutation possible sur les enfants
- Remplissage de la moitié de la population avec les enfants
- Croisement des individus les plus mauvais de la population pour de la diversité
- Remplissage de la deuxième moitié de la population avec ses enfants
- Evaluation de la nouvelle population

Enfin, la meilleure solution est exportée dans un dossier unique avec des informations sur la run réalisé.

c. Résultats

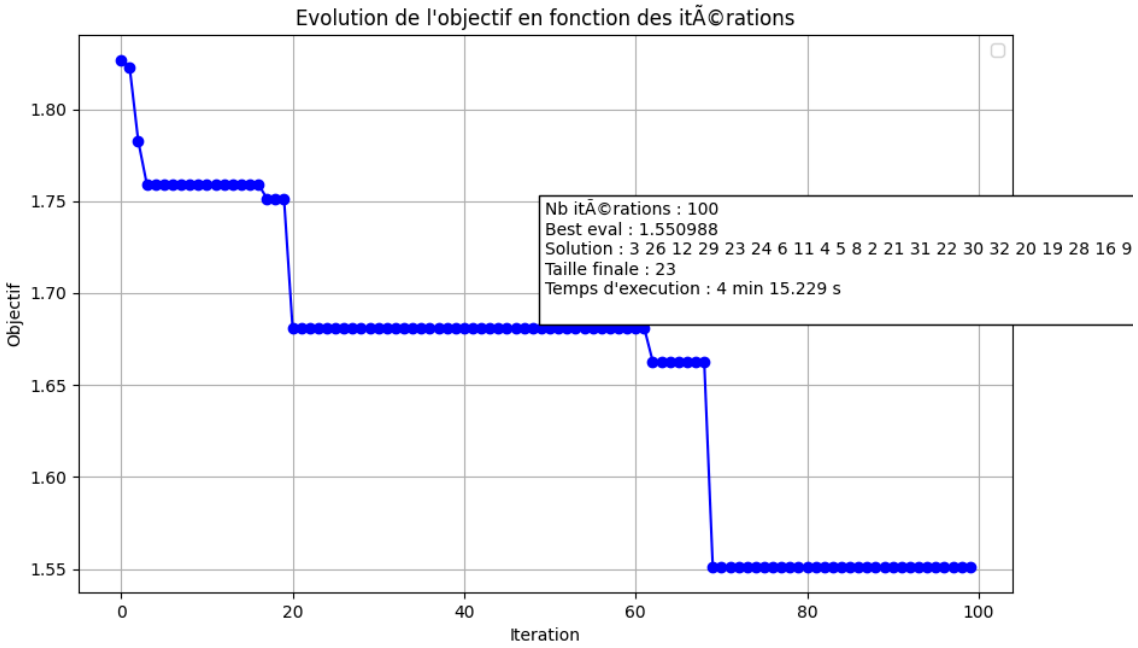
Chaque test de méta paramètres est réalisé selon un même protocole. Les résultats présentés dans ce document sont donc tous issus d'un algorithme génétique avec une taille de population fixé à 100, un nombre maximum d'itérations fixé à 100, une probabilité de mutation fixée à 0.8 ainsi qu'un crossover en 7 points. Tous les résultats présentés ont été obtenus en utilisant l'exemple de la roue, qui comprend 32 polyèdres à fusionner. Enfin, chaque test a été réalisé avec la même graine pour l'aléatoire. Le but de l'algorithme est de minimiser la fonction objectif.

Méthode de sélection par tournoi (TurnamentSelection)

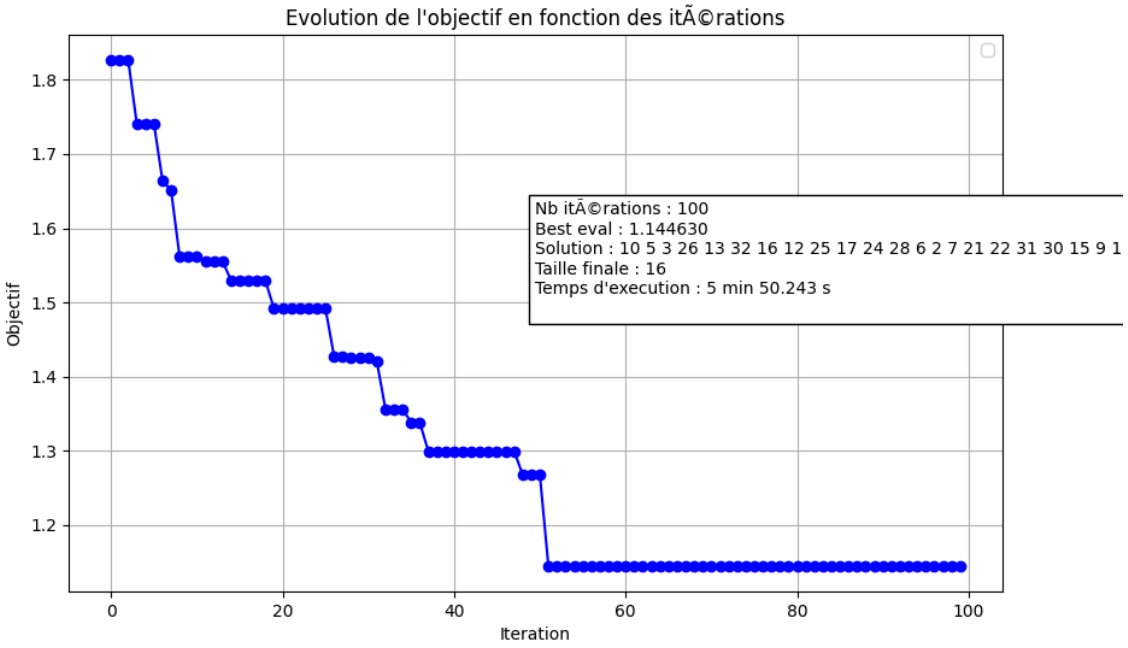


Ici, on peut remarquer que l'objectif se coince sur des sortes de plateaux à certains endroits et stagne pendant longtemps. Il pourrait s'agir d'un manque de diversité dans la population qui pousse l'algorithme à ne plus explorer assez loin et rester sur des solutions médiocres. Le minimum atteint est 1.34 soit une taille finale après fusion de 20 polyèdres sur les 32 présents au début de l'algorithme.

Méthode de sélection par roue (WheelSelection)



Méthode de sélection élitiste (BestScoreSelection)



De même pour la sélection par roue, l'objectif a tendance à se coincer dans des plateaux / minima locaux, avec un moins bon score que la sélection par tournoi.

La sélection élitiste, en revanche, semble plutôt bien fonctionner en arrivant à sortir de ses plateaux, jusqu'à un certain seuil, 1.14. La méthode de **sélection élitiste est celle qui fonctionne le mieux** d'après les résultats obtenus, avec une taille totale après fusion de ***16 polyèdres*** au lieu de 32 polyèdres au départ.

Il est important de noter que les tests montrés ci-dessus ne regroupent pas l'ensemble des tests réalisés. Le choix du nombre de points de croisement a été testé et un croisement à 7 points pour une taille de 32 polyèdres s'est avéré plus optimal que n'importe quel autre nombre de points de croisement.

Le remplissage de la prochaine population a aussi été un point que nous avons cherché à optimiser. Nous sommes arrivés à la conclusion que remplir la première moitié de la nouvelle population avec les enfants issues des parents sélectionnés puis remplir la deuxième moitié aléatoirement était moins efficace que remplir la deuxième moitié avec les enfants issues des moins bons individus. Le fait de croiser avec les moins bons individus permet d'avoir une certaine diversité. Nous pensons que de meilleurs résultats pourraient être obtenus en jouant sur la proportion des enfants issus de la sélection élitiste et des enfants issus du croisement des individus moins performants. Dans notre cas, nous avons uniquement testé avec une population divisée en deux parties.

Dans ce projet nous avons testé principalement différentes sélection ainsi que divers moyens de remplir la nouvelle population car il s'agit de deux étapes importantes pour l'algorithme génétique, en témoigne les résultats. Cependant, si nous avions eu plus de temps, il aurait été judicieux d'explorer également différentes méthodes de croisement et de mutation..

IX.Conclusion

Tout au long du projet, nous avons pu réaliser différentes tâches, en passant par la réalisation de code pour l'écriture et lecture de fichiers *.obj, la réalisation de l'algorithme de fusion, son optimisation ainsi qu'un algorithme brute-force testant toutes les permutations possibles. Une fois cette étape passée, nous nous sommes renseignés sur les méta-heuristiques possibles pour notre problème et un algorithme génétique ainsi qu'un recuit simulé étaient les solutions les plus pertinentes et intuitives.

D'après tous les résultats obtenus sur l'exemple de la roue contenant 32 polyèdres, (image en page de garde), nous pouvons conclure que notre recuit simulé est plus efficace et fournit de meilleures solutions que notre algorithme génétique, compte tenu des résultats obtenus. Cependant c'est vrai uniquement avec les paramètres testés.

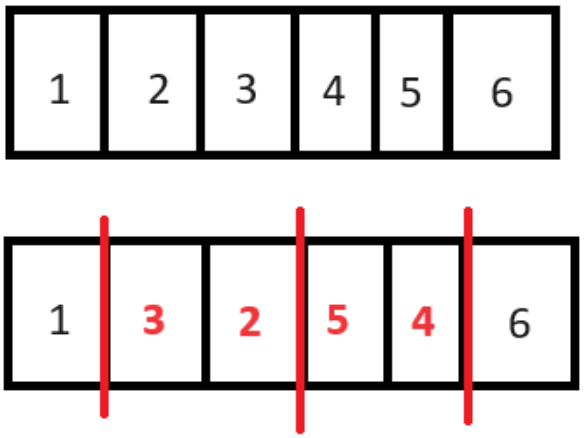
X. Perspectives

Nous pensons que ce projet peut être poussé encore plus loin, avec plusieurs points d'améliorations à différents niveaux.

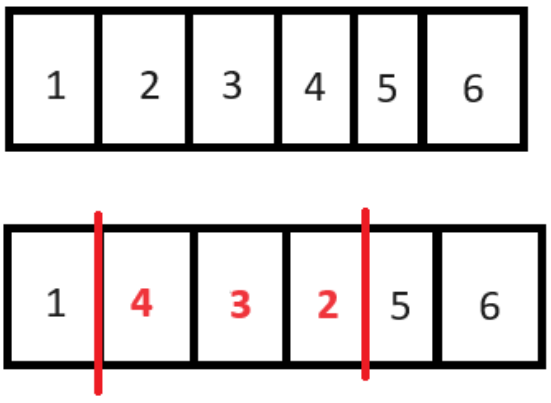
Au niveau de l'optimisation de l'algorithme et du temps de calcul par exemple, il serait intéressant de paralléliser des morceaux de code afin d'accélérer les calculs. Pour la partie méta-heuristique, il serait aussi peut être intéressant de tester d'autres méthodes pour pouvoir les comparer.

Pour des améliorations plus spécifiques, nous pouvons noter qu'il aurait été intéressant d'implémenter plus de mutations et de croisements. Nous avons envisagé deux mutations en particulier qui pourraient être cohérentes et intéressantes à implémenter expliqué ci dessous :

- Une permutation de 2 gènes côtes à côtes pour un ou plusieurs bloc de deux



- Inverser l'ordre de N gènes côte à côte (**mutation par inversion**).



Cette mutation est intéressante pour notre problème, car elle préserve les fusions au sein du bloc rouge. En effet, *4,3,2* et *2,3,4* donnent les mêmes fusions.

Ainsi, cette mutation pourrait éventuellement diminuer grandement la pénalité de distance (car la distance entre *1* et *2* serait remplacée par la distance entre *1* et *4*. Cette mutation serait d'autant plus pertinente si on s'assurerait que le bloc rouge ne coupe aucune autre fusion. On aurait donc rien à perdre à la tester.

De manière plus générale, il faudrait adapter la mutation et le croisement en fonction des fusions possibles au sein des solutions, tout en diminuant la part d'aléatoire dans l'exploration des solutions

Pour le croisement, on pourrait par exemple croiser les 2 parties des parents ayant le plus ou les plus grandes fusions pour donner un enfant qui aura nécessairement un meilleur score.

Pour les mutations, éviter (avec une faible probabilité) de couper les fusions déjà présentes. Cela pourrait par exemple être conditionné à la température, ou au nombre d'itérations sans amélioration.

XI. Sources

Sites internet :

- https://docs.safe.com/fme/html/FME-Form-Documentation/FME-ReadersWriters/obj/Supported_File_Syntax.htm
- https://leria-info.univ-angers.fr/~jinkao.hao/papers/RIA.pdf
- https://perso.liris.cnrs.fr/christine.solnon/publications/poly.pdf
- https://hal.science/hal-01313162/document
- https://webusers.i3s.unice.fr/~malapert/publications/malapert.jeantet-06b-UPMC.pdf
- https://www.researchgate.net/profile/Mais-Rachid/publication/274718261_Differentes_operateurs_evolutionnaires_de_permutation_selections_croisements_et_mutations/links/5527e8150cf2e089a3a213ee/Differentes-operateurs-evolutionnaires-de-permutation-selections-croisements-et-mutations.pdf?origin=publication_detail&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uRG93bmxvYWQiLCJwcmV2aW91c1BhZ2UiOiJwdWJsaWNhdGlvbiJ9fQ
- https://perso.univ-rennes1.fr/jean-christophe.breton/agreg/AGREG/TEXTES/recuit.pdf
- https://perso.univ-rennes1.fr/jean-christophe.breton/agreg/AGREG/TEXTES/PermutationsAleatoires2015.pdf