
8INF846 - Projet final

Algorithmes génétique

Infinite Monkey Theorem

Avril 2021



Université du Québec
à Chicoutimi

Antoine Bouabana (BOUA25119908)

Lucas Gonzalez (GONL29019907)

SOMMAIRE

Développement	3
Technologies utilisées	3
Organisation des sources	3
Exécutable	4
Introduction	4
Contexte	4
Problématique	5

Algorithme génétique	5
Génome	5
Généricité	5
Cycle de vie de l'algorithme	6
1 - Sélection naturel	6
2 - Évolution	8
Croisement (CrossOver)	9
Mutation	9
3 - Évaluation (Acceptabilité / Fitness)	10
Méthodes d'évaluation	10
Évaluation simple	10
Évaluation avancée	11
Évaluation des générations	11
Expérience génétique	12
Options	12
Résultats	12
Résultats	13
Tests manuels	13
Benchmarking	13
Implémentation	13
Résultats finaux	13
Sélection naturelle (meilleurs génomes retenus)	14
Taux de mutation	15
Fonction d'évaluation	16
Taille de la population	17
Midpoint (croisement)	18
Longueur des génomes	19
Conclusions	20
Paramètres optimaux	21
Limites	21

Développement

Technologies utilisées

Les technologies utilisées sont :

- **Typescript** (Langage de programmation)
- **HTML, CSS** (Interface graphique)
- **Jquery** (Manipulation de l'interface)
- **Chart.js** (Librairie graphes, diagrammes, etc)
- **Webpack** (Bundling de l'application)

Organisation des sources

Tout le code source se situe dans le dossier **sources/src** :

- **scripts** : dossier contenant tous les scripts de l'application
 - **benchmark** : contient toute la logique du benchmark
 - **results/*.json** : les résultats des différents benchmark
 - **Benchmark.config.ts** : configuration du benchmark c'est là que l'on peut personnaliser le benchmark
 - **Benchmark.results.ts** : c'est le fichier qui gère l'affichage des résultats sous forme de graphiques
 - **Benchmark.ts** : la classe côté navigateur qui permet d'appeler le worker et de transmettre les messages
 - **Benchmark.worker.ts** : gère le benchmarking côté worker
 - **components** : contient uniquement des classes utiles à l'affichage en HTML
 - **genetic** : contient toutes les classes de l'algorithme génétique
 - **GeneticExperience.ts** : contient le code de l'algorithme génétique (gère les options, les résultats, le cycle de vie, les populations, etc)
 - **Genome.ts** : modélisation des génomes (générique car c'est une classe template)
 - **fitness** : contient les classes liés au fitness function
 - **index.ts** : permet juste de faciliter l'import de toutes les classes du dossier
 - **FitnessStrategy.ts** : définit l'interface permettant d'implémenter une fitness function (selon le design pattern Strategy)

- **SimpleFitnessStrategy.ts** : implémentation de la fitness function simple (détaillée plus loin)
 - **AdvancedFitnessStrategy.ts** : implémentation de la fitness function avancée(détaillée plus loin)
- **renderers/Renderer.ts** : classe gérant le rendu de la population d'une expérience génétique (implémenté en JQuery)
- **utils** : contient les fonctions utiles
 - **data.json** : fichier contenant 2000 phrases aléatoires en français (générées grâce au site de [Romain Valeri](#))
 - **StringFormat.ts** : formatage des chaînes de caractères dans notre application (retire les accents, etc)
 - **Utils.ts** : fonction utiles diverses
- **Controller.ts** : classe contrôleur qui s'occupe de gérer les expériences, d'initialiser, etc
- **View.ts** : s'occupe de toute la partie gestion de la vue (listeners du formulaire, des boutons, etc)
- **index.ts** : point d'entrée de l'application initialise le contrôleur, la vue et le renderer
- **views/index.html** : le fichier HTML de la vue de l'application

Exécutable

Le projet peut être exécuté en mode développement en installant les packages en tapant la commande **npm install** puis lancer le projet en tapant la commande **npm run dev** (les commandes sont à lancer depuis la racine du projet).

Note : Il faudra au préalable s'assurer d'avoir nodeJs et NPM installés sur le machine.

Sinon le projet compilé sera disponible dans le dossier **dist/index.html**.

Introduction

Contexte

L'objectif principal est de découvrir le domaine des algorithmes génétiques traditionnels, à l'origine développés pour résoudre des problèmes dont l'espace de solution est trop vaste pour qu'une méthode dite "Brute Force" fonctionne.

Problématique

Nous avons choisi de mettre en pratique le problème “**Infinite Monkey Théorème**” pour explorer ce domaine. L'idée est que pour n'importe quel texte de taille n , une personne tape aléatoirement sur un clavier et doit retrouver le texte originale. Si on utilise l'exemple de la table ASCII on a $1 / (128^n)$ (pour une table ASCII de 128 caractères) chances de trouver le bon résultat. Dans notre projet, l'ordinateur remplacera la personne et utilisera un algorithme génétique pour effectuer cette recherche dans un temps raisonnable.

Notre résolution du problème est inspirée du chapitre “The Evolution of Code” du livre “[The Nature of Code](#)” de Daniel Shiffman qui traite des algorithmes génétiques et de ce problème mais sans s'intéresser aux différents paramètres de l'algorithme comme le fait notre projet (voir [Résultats finaux](#)).

Algorithme génétique

Dans cette partie, nous détaillerons comment nous avons implémenté l'algorithme génétique à travers une modélisation objet.

Génome

Les génomes représentent des individus dans notre génétique qui contiennent un tableau de gène. Puisque notre problème implique des phrases comme individus, les gènes seront représentés par des caractères.

Par exemple, dans notre cas nous avons une chaîne de caractères qui est notre objectif “**Je suis un génome**”. Nous représentons le génome comme un tableau de caractères. Chaque caractère est un gène du génome. Dans les sections qui suivent nous expliquerons comment sont évalués et modifiés ces génomes pour atteindre notre objectif.

P	E	K	S	U	M	S	P	U	N	G	G	E	R	O	Z	Q
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(génom initial)

J	E		S	U	i	S		U	N		G	E	N	O	M	E
---	---	--	---	---	---	---	--	---	---	--	---	---	---	---	---	---

(génom cible)

Généricité

Par ailleurs, pour que les génomes soient le plus générique possible, nous leur avons attaché une fonction de génération de gène aléatoire qui leur permet de générer un gène (de cette manière, nous pouvons implémenter notre solution avec des gènes correspondant à des caractères mais il serait possible de le faire avec autre chose puisque les classes génomes sont des classes patrons).

Cycle de vie de l'algorithme

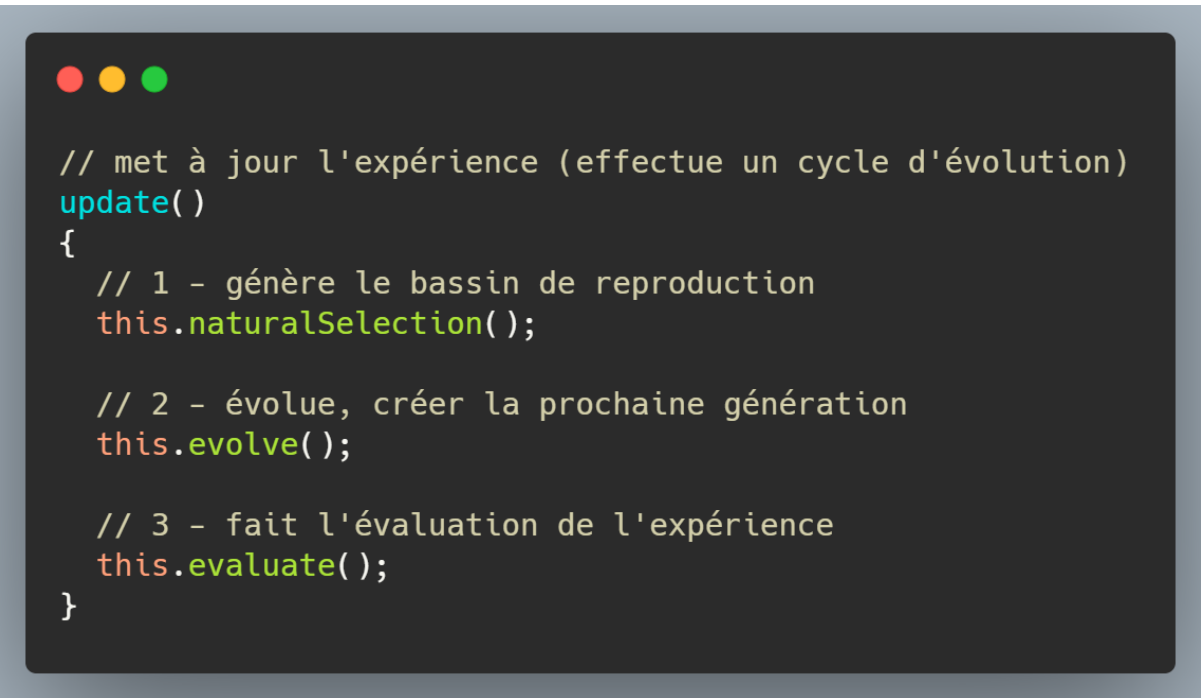
Comme on pourra le voir ci-dessous, la structure de notre algorithme suit celle d'un algorithme génétique traditionnel. Nous avons préalablement généré une population initiale.

À chaque cycle d'évolution :

1. Nous générons un bassin de reproduction en fonction de l'acceptabilité des individus. C'est l'étape de la sélection naturelle.
2. Nous faisons évoluer nos individus par croisements (au sein du bassin de reproduction) et par mutation. C'est l'étape de l'évolution
3. Nous faisons l'évaluation de la nouvelle population générées (en utilisant la "fitness function" pour évaluer le score des génomes)

Chacune de ces étapes sera présentée plus en détails dans les sections qui suivent, leur fonctionnement et comment elles s'adaptent au problème que l'on veut résoudre.

Voici le code derrière ce cycle de vie :



```
// met à jour l'expérience (effectue un cycle d'évolution)
update()
{
  // 1 - génère le bassin de reproduction
  this.naturalSelection();

  // 2 - évolue, créer la prochaine génération
  this.evolve();

  // 3 - fait l'évaluation de l'expérience
  this.evaluate();
}
```

1 - Sélection naturelle

Notre fonction de sélection naturelle peut être décomposée en deux parties :

Premièrement, nous offrons la possibilité à l'utilisateur de choisir le pourcentage de meilleur élément à garder dans la nouvelle population. On trie alors nos génomes en fonction du "fitness score" et détermine l'index final ou nombre de génomes que l'on va garder.



```
// on filtre la population pour ne garder que les keepBest % meilleurs éléments
const sortedPopulation = this.population.sort((a, b) => b.fitness - a.fitness)

// nombre de meilleurs individus à conserver
let count = Math.floor((this.options.keepBest / 100) * sortedPopulation.length)

if (count < 1) count = 1 // s'assure qu'on prend au moins un élément
```

Deuxièmement, on ajoute nos génomes dans le nouveau bassin de reproduction. L'idée est qu'en fonction de son fitness score et du génome qui a le fitness max, un ratio est calculé.

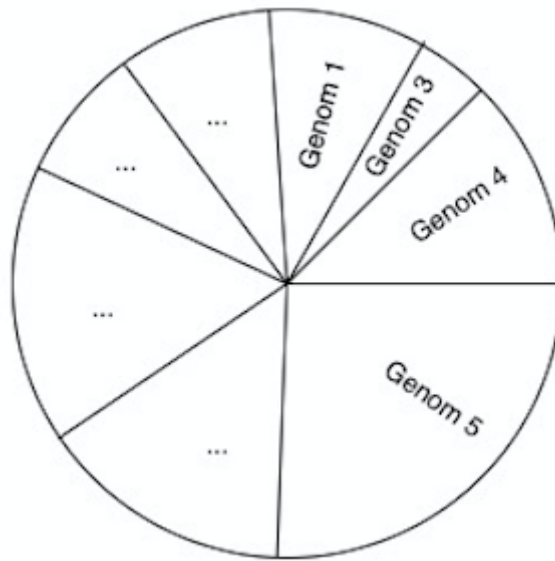
Par la suite, chaque génome est ajouté au bassin un nombre de fois proportionnel à son ratio d'acceptabilité. Plus son score est élevé pour il sera présent dans le bassin et aura de chances d'être choisi dans les étapes qui suivent.



```
// chaque génome est ajouté un nombre de fois proportionnel au poids de son acceptabilité
// tel que plus un génome est acceptable plus il aura de chance d'être prit (et inversement)
for (let i = 0; i < count; i++) {
  // ratio d'acceptabilité du génome
  const fitnessRatio = sortedPopulation[i].fitness / maxFitness

  // calcul du nombre de fois qu'il sera ajouté au bassin de reproduction
  const n = Math.floor(fitnessRatio * 100)

  // ajoute le génome n fois dans le bassin de reproduction
  for (let j = 0; j < n; j++) {
    this.matingPool.push(sortedPopulation[i])
  }
}
```



(Répartition du bassin de reproduction)

2 - Évolution

Notre fonction d'évolution regroupe deux des parties importantes des algorithmes génétiques : le croisement (CrossOver) et la mutation. Pour l'évolution, on itère sur notre bassin de reproduction, on choisit deux génomes aléatoirement, on effectue un croisement entre les deux et possiblement une mutation.

```
// regénère la population à partir du croisement de génomes dans le bassin de reproduction
for (let i = 0; i < this.population.length; i++)
{
  // on tire deux partenaires aléatoires dans le bassin de reproduction
  const partnerA = this.matingPool[
    Math.floor(Math.random() * this.matingPool.length)
  ]
  const partnerB = this.matingPool[
    Math.floor(Math.random() * this.matingPool.length)
  ]

  // on les fait se reproduire (croisement)
  const child = partnerA.crossover(partnerB, this.midpoint)

  // on fait muter l'enfant
  child.mutate(this.options.mutationRate)

  // on assigne le nouveau génome à la population
  this.population[i] = child
}
```


Croisement (CrossOver)

Comme on peut le voir, notre fonction de croisement (présente dans la classe génome) prend en paramètre un partenaire et un point d'intersection. Dans le paramétrage de l'algorithme, nous offrons la possibilité de choisir quel pourcentage du partenaire est gardé lors du croisement. Il est possible de ne pas renseigner ce paramètre auquel cas un point de croisement aléatoire sera choisi.

Voici un exemple concret de croisement :

P	E	K	S	U	M	S	P	U	N	G	G	E	R	O	Z	Q
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

(Genome A)

P	E	Z	D	U	M		P		Z	G	G	A	R	M		Q
---	---	---	---	---	---	--	---	--	---	---	---	---	---	---	--	---

(Génome B)

P	E	K	S	U	M		P		Z	G	G	A	R	M		Q
---	---	---	---	---	---	--	---	--	---	---	---	---	---	---	--	---

(Croisement de A et B avec un midpoint à 30%)

P	E	K	S	U	M	S	P	U	N	G	G	E	R	M		Q
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--	---

(Croisement de A et B avec un midpoint à 82%)

Mutation

Dans un algorithme génétique, il arrive qu'un ensemble de solution stagne et n'arrive plus à évoluer (on dit qu'elles atteignent un maximum local). Ceci est dû au fait que les génomes se reproduisent entre eux et les gènes transmis sont toujours les mêmes. On introduit alors le concept de mutation pour y remédier.

Dans le paramétrage de l'algorithme, nous offrons la possibilité de choisir quel pourcentage de mutation est utilisé. Le principe de mutation est simple : on parcourt le génome et suivant un taux de mutation (pourcentage), on remplace ou non un gène par un nouveau gène aléatoire.

```
// fait muter le génome en passant en paramètre une probabilité (% entre 0 et 100)
mutate(mutationRate: number): void {
  // pour chaque gène
  for (let i = 0; i < this.getSize(); i++) {
    // si on doit faire muter
    if (Math.random() < mutationRate / 100) {
      // remplace le gène par un nouveau gène aléatoire
      this.genes[i] = this.generateGene();
    }
  }
}
```

Voici un exemple concret de mutation :

P	E	K	S	U	M	S	W	U	N	G	G	E	R	M		Q
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--	---

(Croisement de A et B avec un midpoint à 30% avec mutation de 1%)

3 - Évaluation (Acceptabilité / Fitness)

Méthodes d'évaluation

L'évaluation de l'acceptabilité se fait à travers ce que l'on appelle des "fitness function" (ou fonction d'évaluation), elles permettent de calculer l'acceptabilité d'un génome en fonction d'un autre génome cible pour déterminer si il est candidat à la sélection naturelle.

Évaluation simple

Dans notre problème, la façon la plus basique d'évaluer l'acceptabilité d'un génome est de calculer le nombre de caractères communs entre le génome à évaluer et le génome cible puis de diviser ce score par la longueur du génome (tel que l'on obtient un ratio de caractères justes).

Voici un exemple d'évaluation avec cette méthode :

	+1		+1	+1		+1		+1	+1			+1	+1			
P	E	K	S	U	M	S	W	U	N	G	G	E	R	M		Q
↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕	↕
J	E		S	U	i	S		U	N		G	E	N	O	M	E

```

calcFitness(source: Genome<Gene>, target: Genome<Gene>): number
{
  if (source.getSize() !== target.getSize())
    throw "On ne peut évaluer l'acceptabilité d'un génome qu'avec un autre génome de taille identique !";

  let fitness = 0;

  // pour chaque gène commun, incrémente le score d'acceptabilité
  for (let i = 0; i < source.getSize(); i++) {
    if (source.getGenes()[i] === target.getGenes()[i]) fitness++;
  }

  // on divise le score par la longueur du génome
  return fitness / target.getSize();
}

```

Évaluation avancée

Une méthode plus efficace pour faire cela (nous en discuterons plus en détails dans les résultats) est d'appliquer une pondération forte à ce score pour que la différence entre x caractères commun et $x + 1$ caractères commun soit plus impactante. Pour comprendre cela, il suffit d'imaginer une recherche de chaîne de longueur 2000, il faut comprendre que lorsqu'il y a autant de possibilité, d'avoir 200 caractères communs et en avoir 201 est très différents et l'on voudra que 201 soit choisis beaucoup plus que 200. Pour cela, nous portons simplement le score à la puissance 4 pour passer d'une fonction linéaire à une fonction exponentielle.

```

// rend la fonction exponentielle en ajoutant une puissance (4 est choisi arbitrairement)
fitness = Math.pow(fitness, 4);

```

Évaluation des générations

Dans notre algorithme, l'évaluation d'une génération se déroule en deux étapes :

1. Évaluation de l'acceptabilité de tous les génomes
2. Recherche du meilleur génome
3. Enregistrement des données (optionnel)

```
// évalue la génération actuelle
private evaluate(): void {
    // 1 - calcule l'acceptabilité des génomes de la population
    this.calcFitnesses();

    // 2 - trouve le meilleur individu dans la population
    this.evaluateBestMatch();

    // 3 - enregistre les stats de la génération
    this.registerGeneration();
}
```

Expérience génétique

Dans notre modélisation, une “expérience génétique” correspond à un algorithme génétique prenant en entrée des options et un génome cible et retournant en sortie des résultats.

Options

Une expérience génétique est construite avec un ensemble d'options :

- La taille de la population
- Le taux de mutation (voir [Mutation](#))
- La stratégie d'évaluation (voir [Méthodes d'évaluation](#))
- Le taux de meilleurs génomes conservés lors de la sélection naturelle (voir [1- Sélection naturelle](#))
- Le midpoint de croisement (voir [Croisement \(CrossOver\)](#)) sachant qu'on peut le laisser aléatoire en ne le définissant pas

Puis elle est initialisée avec une cible (un génome), on peut ensuite la mettre à jour pour lui faire subir un cycle d'évolution générationnel (voir [Cycle de vie de l'algorithme](#)) jusqu'à l'atteinte d'un résultat ou d'une limite définie.

Résultats

Les résultats d'une expérience génétique sont représentés par :

- Un booléen de succès (si l'expérience a réussie ou si le nombre limite de générations a été atteint)

- Le numéro de la génération finale
- La taille de la cible
- Le génome final trouvé (le meilleur → celui cible dans le cas d'un succès)
- Les options de l'expérience (optionnel : utile pour retrouver les options de recherches)
- Les statistiques d'avancement de la recherche soit un tableau avec pour chaque génération :
 - Le numéro de la génération
 - L'acceptabilité moyenne de la génération
 - L'acceptabilité du meilleur individu de la génération

Résultats

Une fois l'algorithme implémenté et le problème résolu, nous nous sommes attelés à rechercher plus en profondeur l'impact que les différents paramètres de la recherche avaient sur les résultats (et notamment sur le nombre de génération nécessaire à atteindre un résultat concluant).

Tests manuels

Le programme offre la possibilité de personnaliser toutes les options de l'expérience génétique directement via l'interface sans avoir à toucher au code. C'est avec ce type de test que nous avons pu d'abord tatonner sur l'importance des paramètres et sur les supposées meilleures valeurs de chacun de ces paramètres.

Benchmarking

Une fois que nous avons un peu manipuler le projet pour tester les différents paramètres, nous avons mis en place un système élémentaire de benchmarking permettant de tester une multitude d'options pour une multitude de cibles et rapidement (sans l'affichage).

Implémentation

Pour cela nous avons implémenté des [web workers](#) afin de faire du traitement algorithme sur un autre thread (car nous utilisons JavaScript et le code s'exécute donc dans un navigateur).

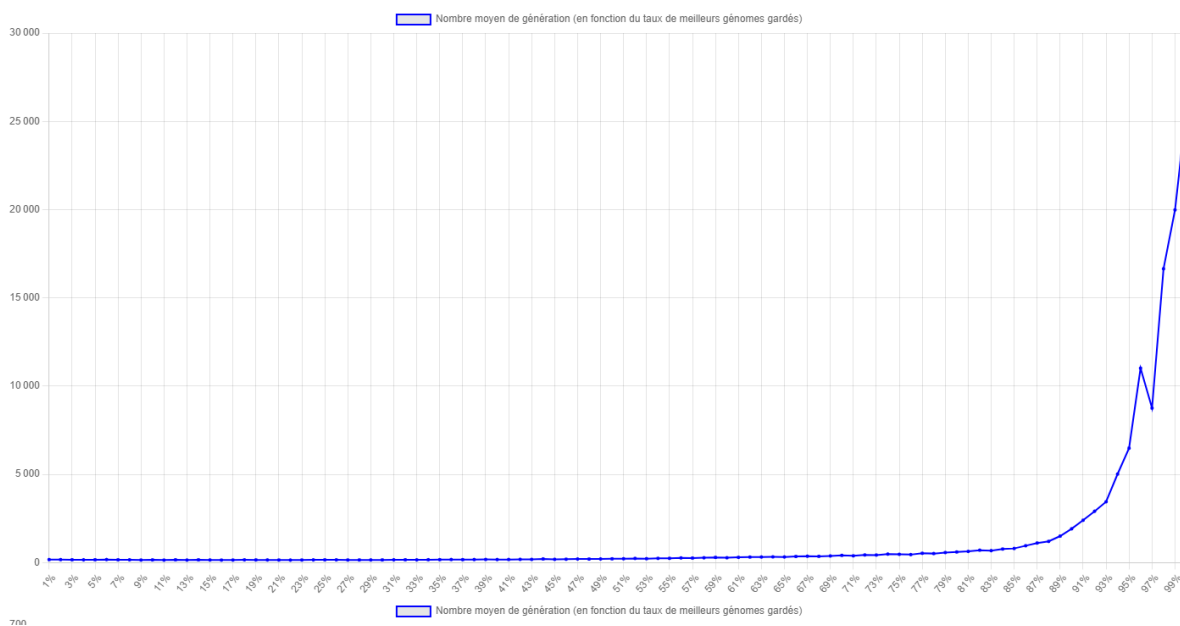
Résultats finaux

Voici les résultats que nous avons obtenus grâce à nos différents benchmarks. Il est à noter que tous les résultats présentés dans cette section sont disponibles directement en bas de page du site.

Sélection naturelle (meilleurs génomes retenus)

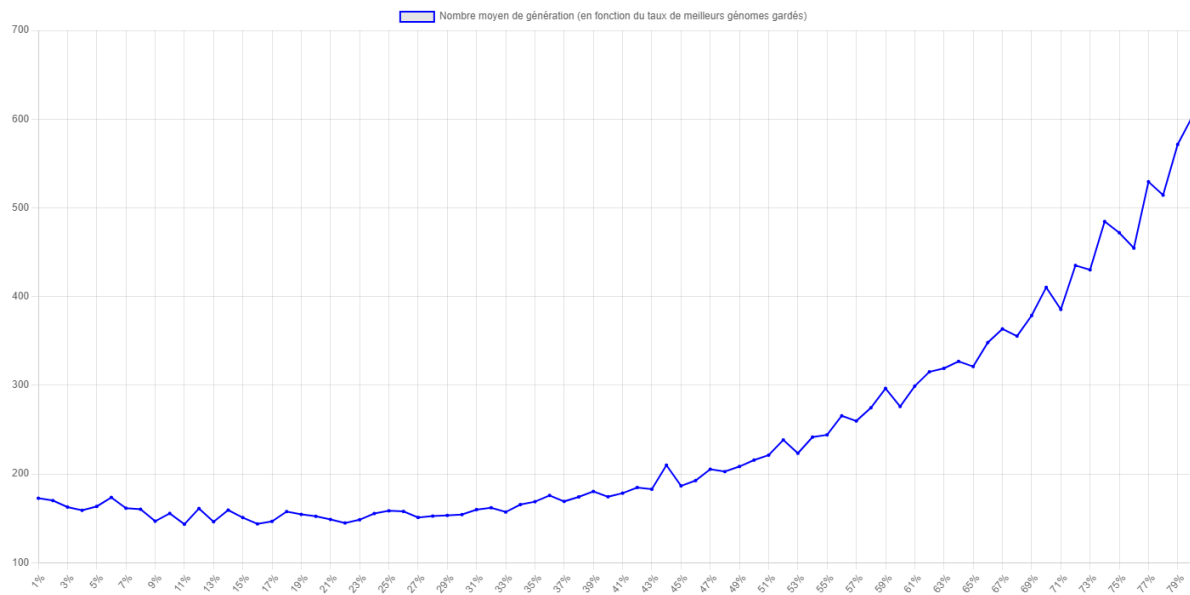
Le premier paramètre que nous avons évalué est celui qui avait le plus d'impact lors de nos expériences manuelles. Il s'agit du pourcentage de meilleurs génomes que l'on conserve lors de la sélection naturelle car bien que dans tous les cas les génomes soient pondérés par leur acceptabilité lors de leur introduction dans le bassin de reproduction, par défaut, tous y sont ajoutés (même les pires). Nous avons donc fait l'expérience suivante :

- Variation du pourcentage de meilleur génome retenus : 1% à 100%
- 5 individus aléatoire de longueur 50 (testés individuellement 10 fois pour chaque option)
- Autres paramètres de recherches optimaux
- Légende : nombre moyen de génération en fonction du % de meilleur génome



Il est ici très clair qu'éliminer les pires génomes a un impact considérable sur le nombre de génération (en éliminant les 20% pires génomes, nous passons de 25.700 générations à seulement 600 générations).

Le graphique suivant montre la courbe entre 1 et 80% pour voir le détail des résultats :

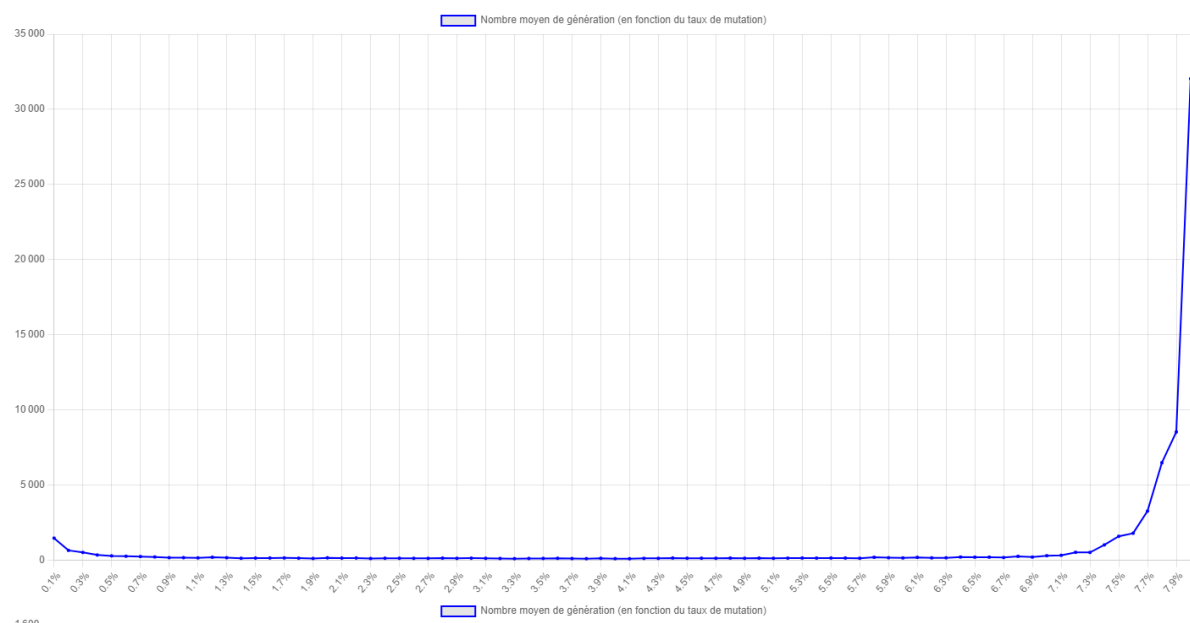


On constate alors que les meilleures valeurs de pourcentage sont essentiellement situées entre 1% et 30% des meilleurs génomes conservés.

Taux de mutation

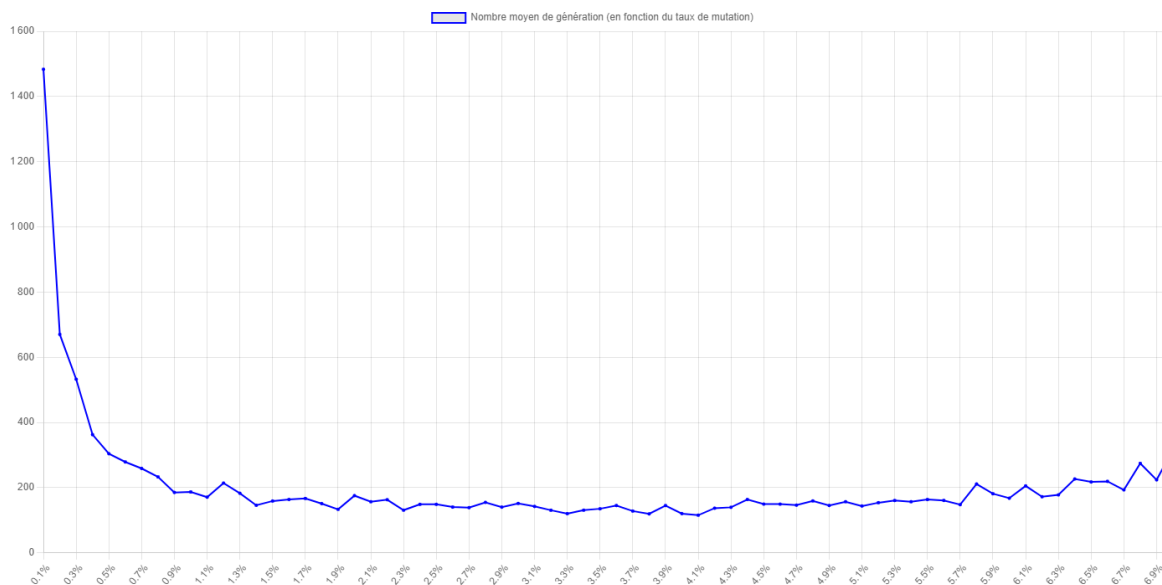
Nous avons testé l'impact du taux de mutation sur l'efficacité de l'algorithme. Nous avons choisi un intervalle de manière manuel en se basant sur des résultats testés avec des valeurs "au jugés" jusqu'à cibler un intervalle de valeur pertinent. Nous avons donc fait l'expérience suivante :

- Variation du taux de mutation de 0.1% à 8% (avec un pas de 0.1%)
- 10 individus aléatoire de longueur 50
- Autres paramètres de recherches optimaux
- Légende : nombre moyen de génération en fonction du % de mutation



On constate qu'il est surtout important de ne pas trop mettre un taux de mutation bas ni de dépasser un certain seuil à partir duquel les résultats deviendront aléatoires et chaotiques.

Avec le même graphique mais plus détaillé :

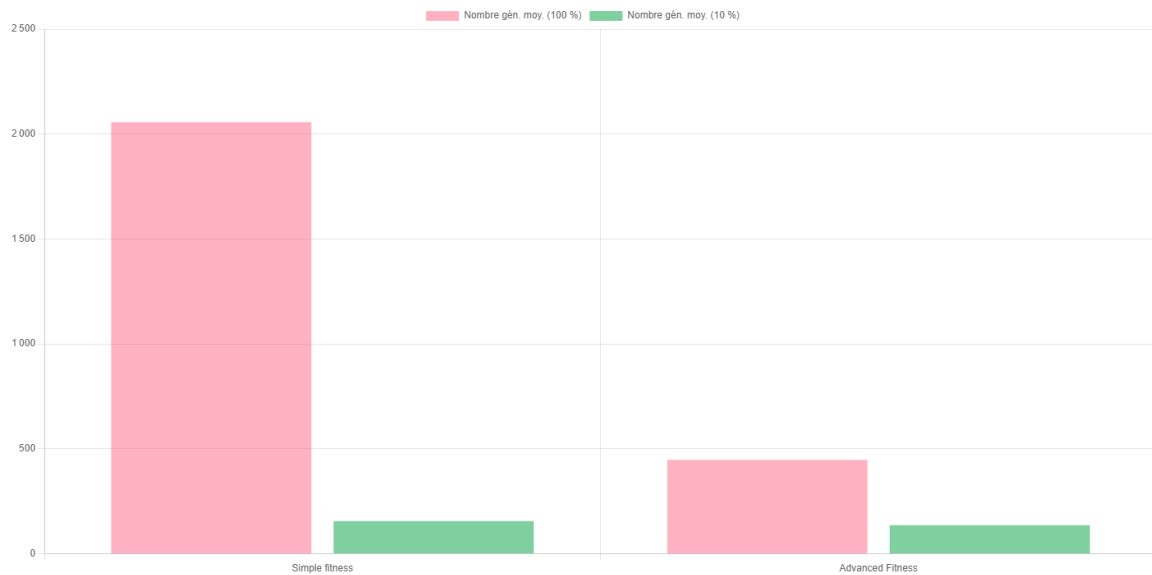


On constate que la première valeur acceptable du taux de mutation semble être à 1% et finit aux alentours de 5%.

Fonction d'évaluation

Nous avons ensuite testé les deux fonctions d'évaluation (voir [Méthodes d'évaluation](#)). Nous avons donc fait l'expérience suivante :

- 100% / 10% des meilleurs individus conservés
- Acceptabilité simple / avancée
- 20 individus aléatoire de longueur 30
- Autres paramètres de recherches optimaux
- Légende
 - Rouge : 100% des meilleurs individus conservés
 - Vert : 10% des meilleurs individus conservés
 - Barres : nombre de génération moyenne
 - Gauche : Acceptabilité simple
 - Droite : Acceptabilité avancée



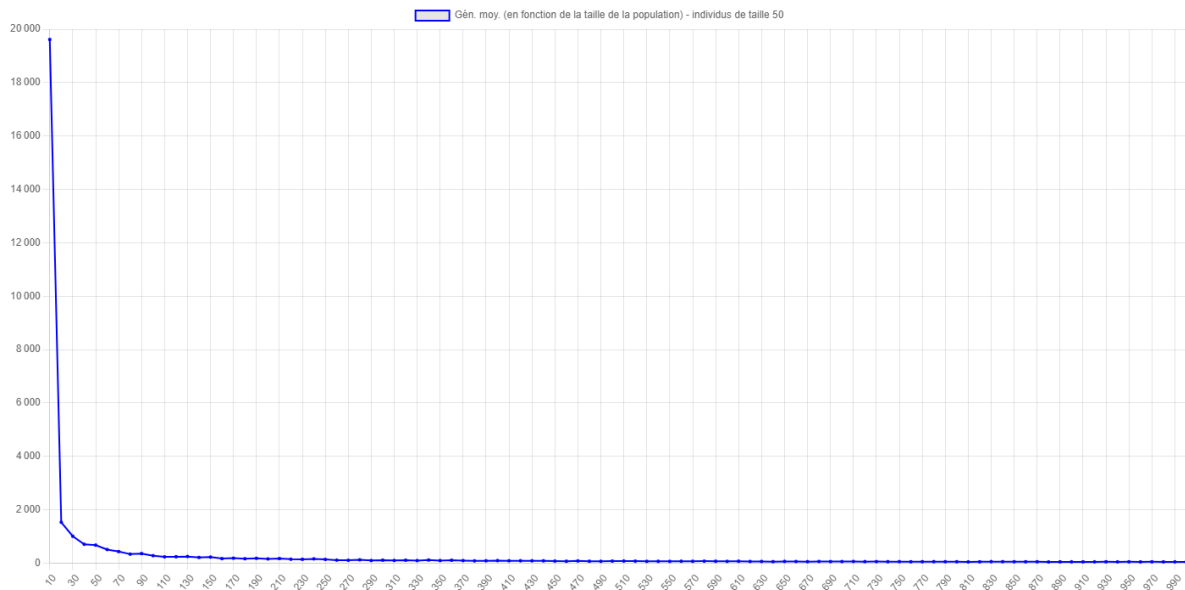
Si l'on regarde les barres rouges on constate bien que celle de droite qui correspond à la fonction avancée est bien plus efficace que pour la fonction simple à gauche (2056 vs 447). On constate également l'efficacité lorsque l'on élimine les pires individus (155 vs 136) bien qu'elle soit moins importante que dans le premier cas.

Taille de la population

Nous nous intéressons maintenant à l'impact de la taille de la population. L'expérience que nous avons fait est la suivante :

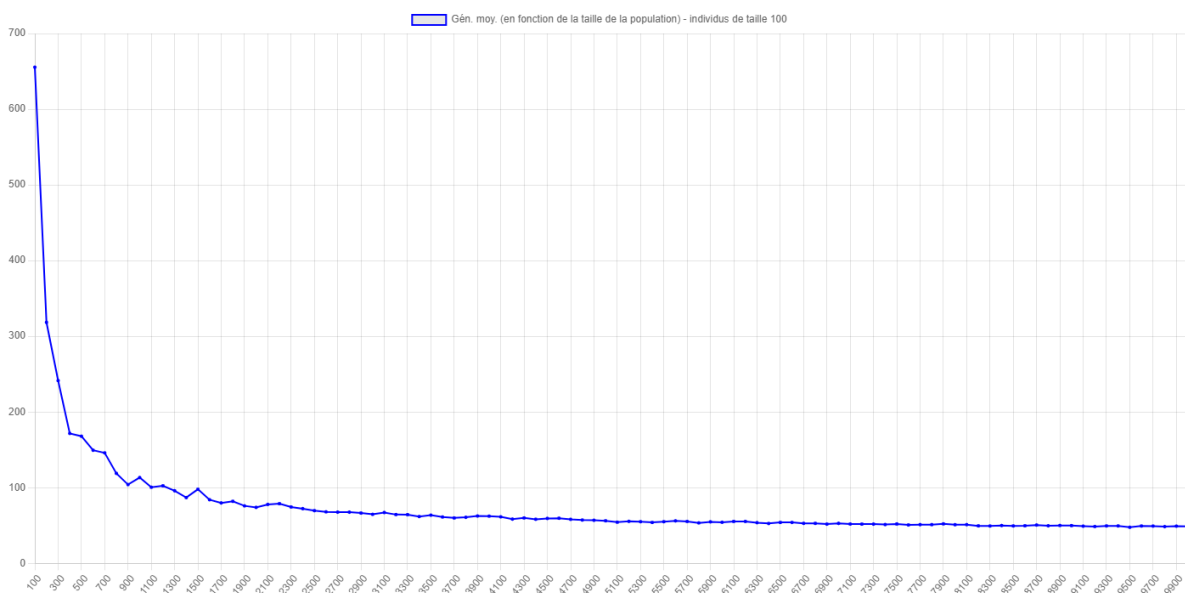
- Génomes de tailles 50
 - Population de taille 10 à 1000 (avec un pas de 10)
- Génomes de tailles 100
 - Population de taille 100 à 10.000 (avec un pas de 100)
- 10 individus aléatoires pour chaque taille
- Autres paramètres de recherches optimaux
- Légende : nombre moyen de génération en fonction de la taille de la population

Voici les résultats pour des individus de taille 50 :



On voit ici que la taille de la population devient acceptable à partir d'environ 100 et que bien que l'efficacité continue à augmenter à mesure que la taille de la population augmente, cela reste assez faible au deçà de 200 (pour 200 : 150 | pour 1000 : 46).

Voici les résultats pour des individus de taille 100 :



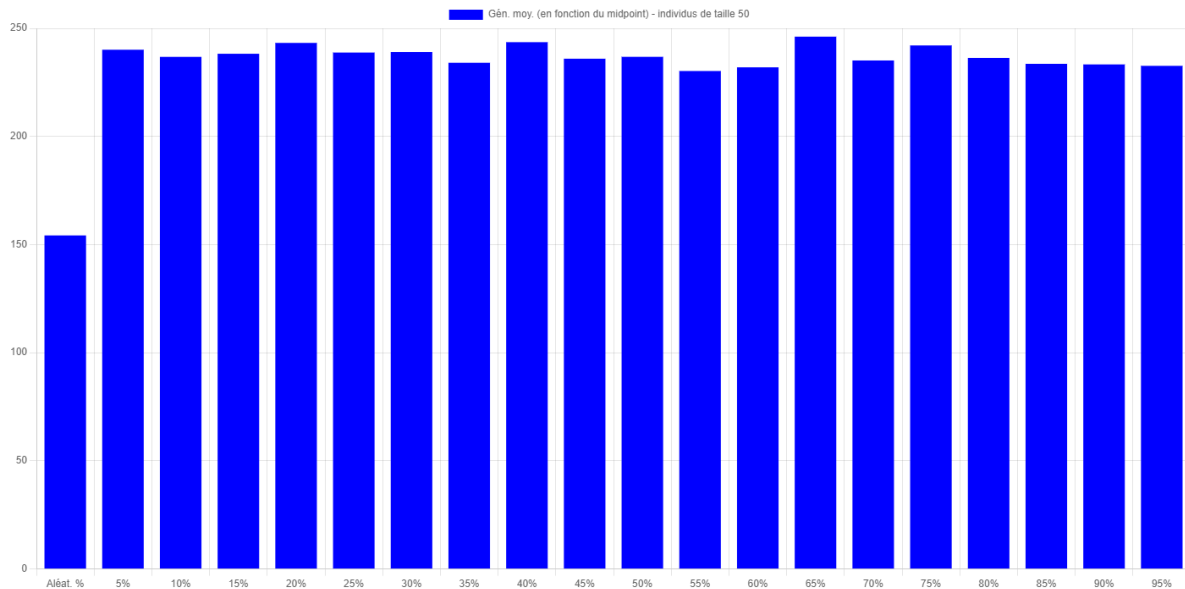
Cette fois ci on voit que la taille de la population devient acceptable à partir d'environ 1000 - 1500 mais continue à réduire à mesure que l'on augmente sa taille.

Cela nous permet de mettre en lumière l'importance de la taille de la population mais surtout que c'est un paramètre qui dépend également de la recherche.

Midpoint (croisement)

Le midpoint est quant à lui le paramètre le moins impactant mais nous avons tout de même tenu à le montrer via notre benchmarking. L'expérience que nous avons fait est la suivante :

- Midpoint de 5% à 95% (avec un pas de 5%)
- Génomes de tailles 50
- 100 individus aléatoires
- Autres paramètres de recherches optimaux
- Légende : nombre moyen de génération en fonction du midpoint

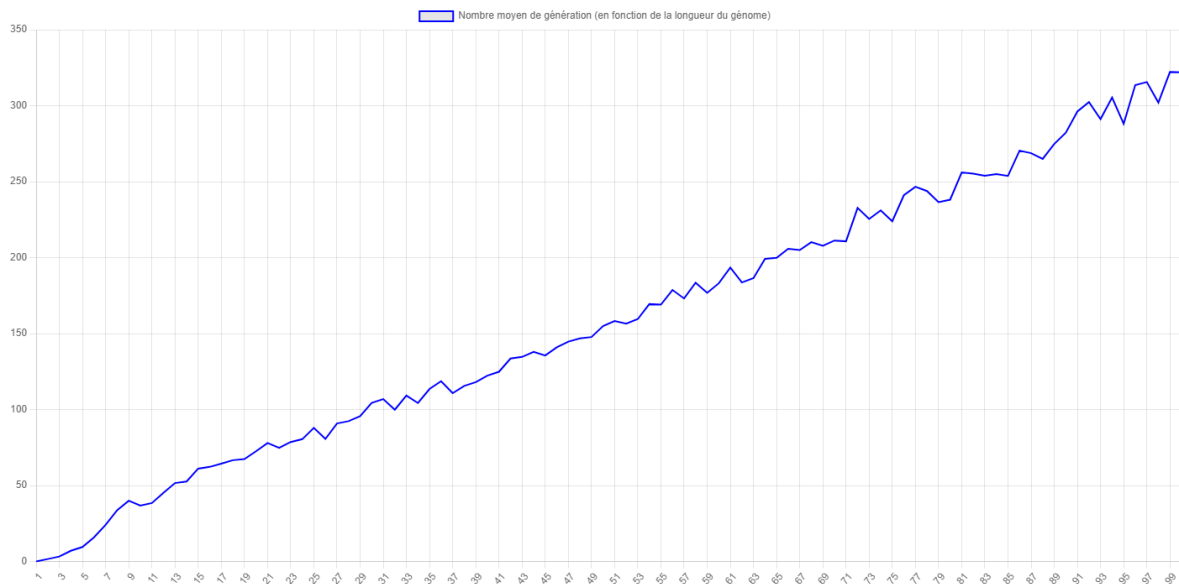


On voit bien ici que le paramètre n'a quasiment aucun impact peu importe sa valeur à l'exception du cas par défaut, lorsque l'on le laisse être aléatoire où l'efficacité est bien meilleure.

Longueur des génomes

Enfin, nous voulions nous assurer que le nombre de génération nécessaire pour résoudre un génome était proportionnel à la longueur de son ADN (avec les mêmes options). Pour cela, nous avons fait l'expérience suivante :

- Variation de la longueur du génome : 1 à 100 caractères
- 10 génomes aléatoire pour chaque longueur
- Paramètres de recherche optimaux
- Légende : nombre de génération moyenne en fonction de la longueur du génome



Les résultats ci-dessus sont ceux attendus, c'est-à-dire une fonction linéaire proportionnelle à la longueur du génome. Il est bien sûr impossible de garantir que cela restera linéaire pour des plus grand nombres (génomés à plusieurs milliers de gènes).

Avertissement : Cela semble effectivement linéaire mais uniquement pour les options de recherche sélectionnées.

Conclusions

Voilà ce que nos résultats nous ont permis de conclure :

- Le pourcentage de meilleur génomes gardés lors de la sélection naturelle (l'élimination des pire génomes) est la manière la plus efficace de réduire le nombre de génération pour résoudre un génome
 - Il est logique que renforcer la difficulté de la sélection naturelle permet à terme d'avoir de meilleurs individus
- Le taux de mutation ne peut pas être nul sinon l'on atteint des maximum locaux mais il ne peut pas être élevé sinon les résultats sont totalement aléatoires
 - Si tous les individus mutent et mutent beaucoup alors on n'obtient une population qui dégénère, si la population ne mute pas alors elle n'évolue pas réellement
- Rendre la fonction d'évaluation exponentielle au lieu de linéaire permet d'obtenir de meilleur résultats particulièrement lorsque le nombre de gène va augmenter
 - Augmenter l'écart d'acceptabilité entre deux individus permet de faire une sélection naturelle plus pertinente
- La taille de la population doit être choisie en fonction de la taille du génome cible (nous n'avons pas dégager de méthodologie pour déterminer cette taille) tout en tenant compte des limites d'une population trop grande (consommation de mémoire, etc)

- Plus d'individus mène à une plus grande diversité génétique et à plus de chance d'obtenir des individus performant
- Le midpoint pour le croisement doit être laissé par défaut, c'est-à-dire aléatoire car le définir fixe ne fait que réduire son efficacité
 - Le croisement entre deux génomes se doit d'être aléatoire comme dans la vraie vie sinon il ne s'agit pas réellement d'un croisement et donc d'évolution

Paramètres optimaux

Nous obtenons donc les paramètres optimaux suivants :

- La taille de la population
 - doit être calculée en fonction de la taille du génome cible (mais il est recommandé de prendre au minimum 2X le nombre de gènes du génome cible)
- Le taux de mutation
 - entre 1% et 5%
- La stratégie d'évaluation
 - exponentielle en fonction de la longueur des génomes
- Sélection naturelle
 - conserver entre 1% et 30% des meilleurs génomes (éliminer au minimum les pires 15% des génomes)
- Midpoint (croisement)
 - le laisser aléatoire

Limites

Même si nous nous sommes efforcés d'obtenir des résultats basés sur plusieurs même calculs afin d'obtenir des valeurs moyennes, nous avons été confrontés à des soucis de durées (certains benchmarks prenaient plus de 3 heures à effectuer).

Il est aussi important de rappeler que toutes les expériences se sont faites avec des génomes de taille inférieure ou égale à 100 et que donc on ne peut pas conclure avec certitude sur les grands génomes mais uniquement les hypothétiser.

Cependant, nous restons satisfaits de ces résultats qui, à défaut d'être précis, donnent une bonne idée générale de l'importance des différentes options dans notre algorithme génétique.