FACULTÉ
DES SCIENCES
UNIVERSITÉ D'ANGERS

Algorithmes intelligents pour l'aide à la décision

Timothée Afonso

11 Février 2024

Table des matières

1 Introduction **2**
1.1 Présentation des algorithmes évolutionnaires 2
1.2 Présentation du problème One-Max 3

2 Présentation des trois méthodes étudiées **4**
2.1 Algorithme génétique steady state 4
2.2 Algorithme à estimation de distribution 7
2.3 Algorithme génétique compact 8

3 Analyse expérimentale **11**
3.1 Algorithme génétique steady state 11
3.2 Algorithme à estimation de distribution 13
3.3 Algorithme génétique compact 14

4 Sélection automatique de l'opérateur de mutation **15**
4.1 Roulette adaptative 15
4.2 UCB 15

5 Analyse expérimentale **16**
5.1 Roulette adaptative 16
5.2 UCB 18

6 Autres problèmes binaires **19**
6.1 One-Max avec masque sur la fonction d'évaluation 19
6.2 Leading-Ones 20

7 Conclusion **21**

8 Bibliographie **22**

1 Introduction

1.1 Présentation des algorithmes évolutionnaires

Les algorithmes évolutionnaires s'inspirent de la théorie de l'évolution pour résoudre des problèmes d'optimisation principalement. Son principe est de faire évoluer un ensemble de solutions, appelé population, afin de trouver les meilleurs résultats. Les individus d'une population sont uniques et plus ou moins adaptés à son environnement. De la même manière que l'évolution des êtres vivants, les caractéristiques des meilleurs individus auront tendance à se propager plus rapidement chez les descendants. Un algorithme évolutionnaire fait donc évoluer une population d'individus et la qualité d'un individu est associé à la fonction de fitness. Les opérateurs de l'algorithme permettent de manipuler les individus afin de faire évoluer la population. On retrouve les opérateurs de sélection (sélection et remplacement) et de variation (croisement et mutation). Un algorithme évolutionnaire suit les étapes suivantes :

- Génération d'une population initiale
- Itération jusqu'à ce qu'un critère d'arrêt soit atteint
- Sélection d'un ensemble d'individus dans la population
- Croisement pour générer de nouveaux individus
- Mutation des nouveaux individus
- Remplacement de certains individus par les nouveaux individus

Tout d'abord, une population initiale d'individus est générée aléatoirement ou par des méthodes spécifiques. Ensuite, à chaque itération, l'algorithme évalue les individus de la population selon un critère et continue le processus jusqu'à ce qu'un critère d'arrêt soit atteint, comme un nombre maximum d'itérations ou l'atteinte d'une solution satisfaisante. Dans chaque itération, une sélection d'individus est effectuée en fonction de leur performance, avec un mécanisme de roulette ou de tournoi par exemple. Ensuite, les individus sélectionnés sont croisés, c'est-à-dire que des parties de leurs gènes sont échangées pour créer de nouveaux individus. Après le croisement, une opération de mutation est appliquée à certains des nouveaux individus pour introduire une diversité génétique dans la population. Enfin, les individus mutés et les descendants sont utilisés pour remplacer certains membres de la population initiale, généralement les moins performants, assurant ainsi une évolution vers des solutions potentiellement meilleures à chaque itération. Le choix des opérateurs (favorisant l'exploitation ou l'exploration) est essentiel pour obtenir la meilleure population possible, le but étant de maximiser la valeur de la fonction objective, à l'aide des opérateurs.

1.2 Présentation du problème One-Max

Le problème du One-Max est un problème simple permettant d'introduire efficacement les algorithmes génétiques. On souhaite définir une liste de bit (0 ou 1) de taille N. La valeur de cette liste (valeur de fitness) correspond à la somme de chacun de ces éléments. Une liste correspond à un individu, la valeur de fitness d'un individu n est $fitness(n) = \sum_{i=0}^{N} n_i$. Le but du problème est de déterminer la valeur maximale que l'on peut obtenir à partir d'un individu. On souhaite maximiser la valeur de fitness. Voici une illustration de ce problème.

Soit une liste de taille $N = 10$ initialisée aléatoirement :

0 1 0 0 1 1 1 0 0 0 $\longrightarrow$ Fitness = 4

La solution optimale pour une liste de taille N=10 est donc :

1 1 1 1 1 1 1 1 1 1 $\longrightarrow$ Fitness = 10

Un algorithme génétique devrait donc, à la fin de son exécution, retourner des individus dont la somme est égale à N. Ces individus devraient être composés uniquement de 1.

2 Présentation des trois méthodes étudiées

Afin de résoudre le problème du One-Max, nous avons implémenté différents algorithmes évolutionnaires.

2.1 Algorithme génétique steady state

L'algorithme steady state vise à sélectionner deux individus à chaque génération. Celui-ci peut être généralisé avec plusieurs individus. L'initialisation des bits de chaque individu peut être fixée (tous à zéros) ou fait aléatoirement. Une fois la population générée, le processus de sélection peut se faire de différentes manières, telles que (pour N individus à sélectionner) :

- par roulette : chaque individu a une probabilité de sélection proportionnelle à sa valeur de fitness.
- par tournoi : un tournoi est initialisé avec une taille k. C'est-à-dire, que N tournois de k individus choisis aléatoirement sont créés. Les N gagnants (individu avec la meilleure fitness parmi les k) seront les individus sélectionnés pour subir un croisement et une mutation.
- par classement : les N meilleurs individus sont choisis.
- aléatoire : les N individus sont choisis aléatoirement.

Les individus sélectionnés vont subir une opération de croisement. Il existe plusieurs types de croisement tels que :

- $mono-point$: Un point de croisement est choisi aléatoirement sur les parents. Les parties situées après le point de croisement sont échangées entre les deux parents pour créer des descendants.
- $deux-points$: Deux points de croisement sont choisis aléatoirement sur les parents. Les parties situées entre les deux points de croisement sont échangées entre les deux parents pour créer des descendants.
- $multi-point$: Plusieurs points de croisement sont choisis aléatoirement sur les parents. Les parties situées entre les différents points de croisement sont échangées entre les deux parents pour créer des descendants.
- $uniforme$: Chaque bit d'un descendant est choisi indépendamment pour être hérité de l'un des parents. Chaque bit a généralement une chance égale d'être hérité de l'un ou l'autre parent.

Suite au croisement, chaque individu a une probabilité P de subir une mutation. Comme pour le croisement, il existe plusieurs types de mutation comme :

— $bit - flip$: chaque bit d'un individu a une probabilité P d'être muté.
— $one - flip$: un bit est choisi aléatoirement pour être muté.
— $k - flip$: k bit sont choisi aléatoirement pour être muté.

Enfin, l'insertion des nouveaux individus dans la population peut se faire selon différents critères, tels que l'âge des individus existants (les plus anciens peuvent être remplacés) ou la performance (les moins performants peuvent être remplacés). L'algorithme se termine selon un critère d'arrêt choisi (nombre de générations, solution trouvées, etc.).

Une analyse comportementale de l'algorithme génétique steady state a été réalisé en faisant varier la taille de la population et les opérateurs de sélection, de mutation et de croisement. Les résultats sont disponibles dans la troisième partie du rapport.

Ci-dessous, le pseudo code de l'algorithme génétique steady state.

Algorithm 1: Algorithme génétique steady state

```
// Initialisation
operateur_croisement : opérateur de croisement;
operateur_mutation : opérateur de mutation;
operateur_selection : opérateur de sélection;
fonction_remplacement : fonction de remplacement des anciens individus
 par les nouveaux;
for i ← 1 to NB_RUNS do
    Initialiser une population aléatoire de taille POPULATION_SIZE ;
    Évaluer la fitness de chaque individu de la population;
    generationCounter ← 0;
    while generationCounter < MAX_GENERATIONS do
        generationCounter ← generationCounter + 1;
        // Sélection des parents
        Sélectionner x individus de la population avec operateur_selection;
        // Croisement
        Appliquer le croisement avec operateur_croisement selon une
         probabilité P_CROSSOVER;
        // Mutation
        Appliquer la mutation avec operateur_mutation selon une
         probabilité P_MUTATION;
        // Évaluation de la fitness
        Mettre à jour la fitness des individus modifiés;
        // Insertion des nouveaux individus dans la population
        Insérer les nouveaux individus dans la population avec
         fonction_remplacement;
    end
end
```

2.2 Algorithme à estimation de distribution

L'algorithme à estimation de distribution vise à estimer la distribution de probabilité des solutions dans l'espace de recherche. L'algorithme commence par une initialisation aléatoire d'un vecteur de probabilité de taille N (correspondant à la taille d'un individu). Chaque élément du vecteur représente la probabilité qu'un bit prenne la valeur 1 ou 0. Cette distribution de probabilité est utilisée pour générer une population. Elle sera ensuite ajustée au fur et à mesure des générations pour générer une population de meilleurs individus à chaque fois. La mise à jour du vecteur de probabilité est effectuée en fonction des individus évalués. À chaque itération, les K meilleurs individus de la population sont choisis. On effectue ensuite la moyenne de chaque bit sur les K individus pour mettre à jour le vecteur de distribution. $Nouvelle\ probabilite\ du\ bit\ i\ =\ somme\ des\ bits\ i\ des\ K\ individus\ /\ N$. L'algorithme se termine lorsqu'un critère d'arrêt est atteint. Cette approche diffère de l'algorithme génétique étudié précédemment. En effet, à chaque itération, l'entièreté de la population est régénérée en fonction du vecteur de probabilité. Il n'y a donc pas d'opérateur de sélection, ni d'opérateur de croisement et de mutation. Voici le fonctionnement général pour $N = 5$ et $K = 2$:

- Vecteur de distribution initiale : 0.2 0.4 0.8 0.3 0.5
- Population initiale :
1) 0 0 1 0 1 $\longrightarrow$ Fitness = 2
2) 1 1 1 0 1 $\longrightarrow$ Fitness = 4
3) 0 1 1 1 1 $\longrightarrow$ Fitness = 4
4) 0 0 1 1 1 $\longrightarrow$ Fitness = 3
- Sélection des K meilleurs individus $\longrightarrow$ 2) et 3).
- Mis à jour du vecteur de distribution en calculant la valeur moyenne des bit pour les K individus.
- Nouveau vecteur de distribution : 0.5 1 1 0.5 1
- Génération d'une nouvelle population.
- Itération jusqu'au critère d'arrêt.

Une analyse comportementale de cet algorithme a été réalisée en faisant varier le nombre d'individus sélectionné. Les résultats sont disponibles dans la troisième partie du rapport. Ci-dessous, le pseudo code de l'algorithme à estimation de distribution.

Algorithm 2: Algorithme génétique à estimation de distribution

```
// Génération de la distribution initiale
Function genere_distib_initiale() :
    Initialiser une liste distrib de taille ONE_MAX_LENGTH avec des
     nombres aléatoires entre 0 et 1;
    return distrib;

// Génération de la population selon la distribution
Function genere_population_distribution(population, distrib) :
    for chaque individu dans la population do
        for chaque position de l'individu do
            Générer aléatoirement un nombre r entre 0 et 1;
            if r < distrib[position] then
                distrib[position] = 1;
            end
            else
                distrib[position] = 0;
            end
        end
    end

// Mise à jour de l'estimation de la distribution
Function maj_estimation_distribution(population, distrib, k) :
    Sélectionner les k meilleurs individus de la population;
    for chaque bit des k individu do
        Mettre à jour la valeur de distrib[position] en fonction de la moyenne des
         bits obtenue;
        Attribuer une valeur minimal P_min à distrib[position] pour éviter une
         probabilité de zéro;
    end

// Algorithme principal
for i de 1 à NB_RUNS do
    Initialiser une population aléatoire de taille POPULATION_SIZE;
    Initialiser la distribution initiale;
    generationCounter ← 0;
    while generationCounter < MAX_GENERATIONS do
        generationCounter ← generationCounter + 1;
        Générer la population selon la distribution actuelle;
        Évaluer la fitness de chaque individu de la population;
        Mettre à jour l'estimation de la distribution;
    end
end
```

2.3 Algorithme génétique compact

L'algorithme génétique compact est similaire à l'algorithme génétique à estimation de distribution. Il initialise un vecteur de probabilité de taille N (correspondant à la taille d'un individu) à 0,5. Cette distribution de probabilité est utilisée pour générer une population de deux individus. Elle sera ensuite ajustée au fur et à mesure des générations pour générer une population de deux meilleurs individus à chaque fois. À chaque itération, on détermine le meilleur individu (winner) et le moins bon individu (loser). On compare ensuite bit à bit les valeurs des deux individus. S'ils sont égaux, la probabilité pour ce bit ne change pas. Cependant, si les bits diffèrent, on ajoute ou on retire $1/N$ à la probabilité. *Nouvelle probabilite du bit* i $=$ *probabilite du bit* $i + 1/N$, *si le bit i du winner est egal a* 1 *sinon nouvelle probabilite du bit* i $=$ *probabilite du bit* i $- 1/N$. L'algorithme se termine lorsqu'un critère d'arrêt est atteint. Voici le fonctionnement général pour $N = 5$:

- Vecteur de distribution initiale : 0.5 0.5 0.5 0.5 0.5
- Population initiale :
1) 0 0 1 1 0 $\longrightarrow$ Fitness = 2
2) 1 0 1 0 1 $\longrightarrow$ Fitness = 3
- Sélection du *winner* et du *loser* $\longrightarrow$ 1) et 2).
- Mis à jour du vecteur de distribution en calculant la valeur moyenne des bit pour les K individus.
- Nouveau vecteur de distribution : 0.7 0.5 0.5 0.3 0.7
- Génération d'une nouvelle population
- Itération jusqu'au critère d'arrêt.

Une analyse comportementale de cet algorithme a été réalisée en comparant l'évolution de la fitness moyenne et de la fitness max avec celle de l'algorithme génétique à estimation de distribution. Les résultats sont disponibles dans la troisième partie du rapport. Ci-dessous, le pseudo code de l'algorithme génétique compact.

Algorithm 3: Algorithme génétique compact

```
// Génération de la distribution initiale
Function genere_distib_initiale() :
    Initialiser une liste distrib de taille ONE_MAX_LENGTH à 0,5;
    return distrib;

// Génération de la population selon la distribution
Function genere_population_distribution(population, distrib) :
    for chaque individu dans la population do
        for chaque position de l'individu do
            Générer aléatoirement un nombre r entre 0 et 1;
            if r < distrib[position] then
                distrib[position] = 1;
            end
            else
                distrib[position] = 0;
            end
        end
    end

// Mise à jour de l'estimation de la distribution
Function maj_estimation_distribution(population, distrib) :
    Détermine le meilleur individu (winner) et le pire individu (loser) de la
     population;
    for chaque position dans la distribution do
        if winner[position] ≠ loser[position] then
            if winner[position] = 1 then
                distrib[position] += 1/ONE_MAX_LENGTH;
            end
            else
                distrib[position] -= 1/ONE_MAX_LENGTH;
            end
        end
    end
    return distrib;

// Algorithme principal
for i de 1 à NB_RUNS do
    Initialiser une population aléatoire de taille POPULATION_SIZE;
    Initialiser la distribution initiale;
    generationCounter ← 0;
    while generationCounter < MAX_GENERATIONS do
        generationCounter ← generationCounter + 1;
        Générer la population selon la distribution actuelle;
        Évaluer la fitness de chaque individu de la population;
        Mettre à jour l'estimation de la distribution;
    end
end
```

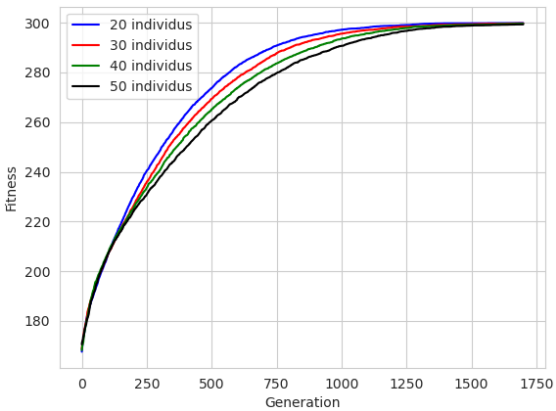
3 Analyse expérimentale

Pour analyser le comportement des algorithmes génétiques, nous allons faire varier un paramètre à la fois et étudier son impact sur l'algorithme. Pour cela, des choix ont été réalisés pour définir des valeurs par défaut à nos paramètres :

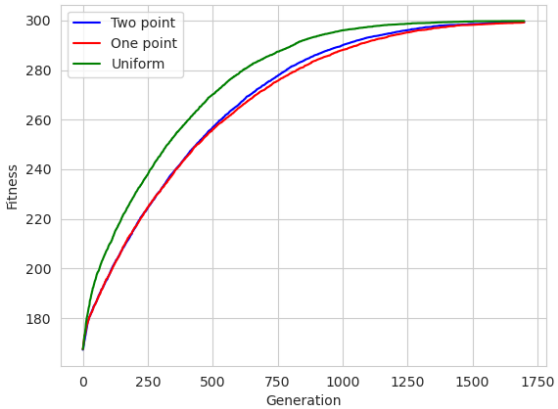
- Taille d'un individu fixé à 300.
- Taille de la population fixée à 20.
- Probabilité de réalisé un croisement sur les individus sélectionné fixé à 1 (croisement obligatoire).
- Probabilité de réalisé une mutation sur les individus sélectionnée fixée à 1 (mutation obligatoire).
- Nombre de run fixé à 30.
- Initialisation aléatoire des individus.
- Nombre d'individus sélectionné fixé à deux (10% de la population).
- Les opérateurs choisis par défaut (on ne fait varier qu'un opérateur à la fois) sont un tournoi pour la sélection (de taille 3), un croisement uniforme pour le croisement et $bit - flip$ pour la mutation.
- Remplacement selon l'age (les moins bons individus sont remplacés par les nouveaux).

3.1 Algorithme génétique steady state

Nous avons étudié l'impact de la taille de la population et des opérateurs de sélection, de mutation et de croisement sur l'algorithme génétique steady state.



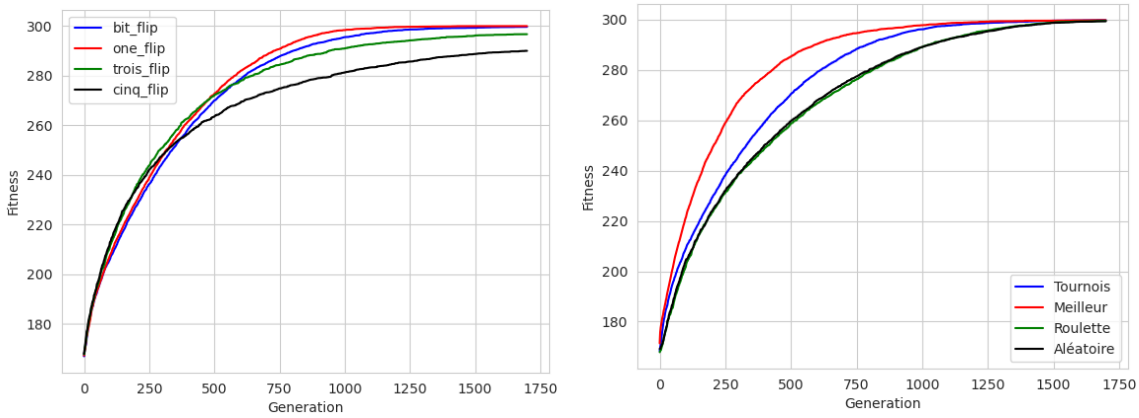
(a) Fitness moyenne en fonction de la taille de la population sur 30 runs



(b) Fitness moyenne en fonction de l'opérateur de croisement sur 30 runs

FIGURE 1 – Impact de la taille de la population et de l'opérateur de croisement

On peut observer que plus la taille de la population diminue, plus l'on converge rapidement vers une solution optimale. En effet, lors de la sélection, plus la population est petite, plus on augmente nos chances de choisir les meilleurs individus et donc de transmettre leurs caractéristiques aux générations suivantes. Si l'on s'intéresse aux opérateurs de croisement, le croisement uniforme semble être le plus performant. On n'observe pas de réelle différence entre le croisement mono-point et deux-points.



(a) Fitness moyenne en fonction de l'opérateur de mutation sur 30 runs

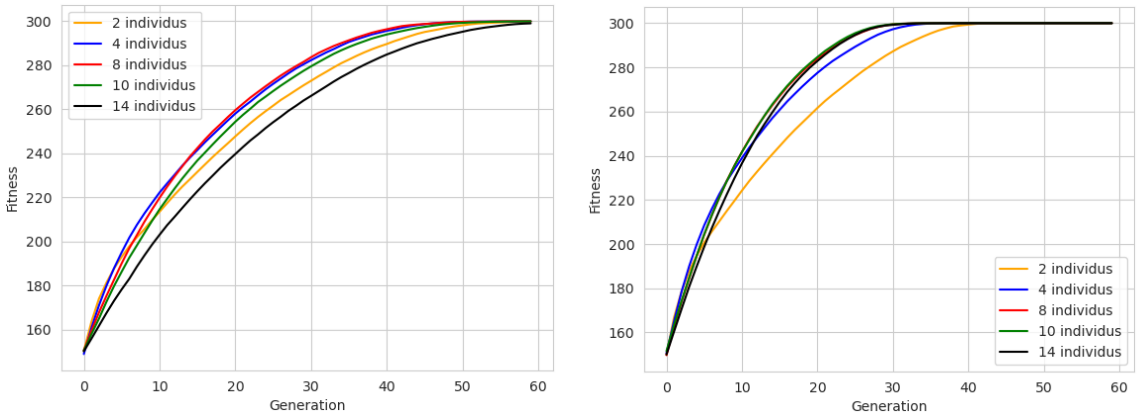
(b) Fitness moyenne en fonction de l'opérateur de sélection sur 30 runs

FIGURE 2 – Impact des opérateurs de mutation et de sélection

L'opérateur de mutation a un impact important sur notre population. L'opérateur cinq-flip est efficace au début car il a de grandes chances d'améliorer la solution. Cependant, lorsqu'il ne reste que peu de bit à zéro, il aura beaucoup de chance de faire régresser la solution. L'opérateur bit-flip et one-flip ont l'effet inverse. Concernant les opérateurs de sélection, on observe une meilleure performance pour le tournoi qu'une sélection aléatoire ou par roulette. La sélection des meilleurs individus (en rouge) est logiquement plus efficace que les autres opérateurs de sélections.

3.2 Algorithme à estimation de distribution

Avec l'algorithme à estimation de distribution, il est possible de faire varier le nombre de meilleurs individus sélectionné (parents). Ceux-ci ont un impact sur la vitesse de résolution du problème. Une moyenne bit à bit sera effectuée sur les parents pour mettre à jour le vecteur de probabilité. On peut également faire varier la taille de la population pour voir si celle-ci influence le nombre d'individus a sélectionné.



(a) Population de taille 30

(b) Population de taille 80

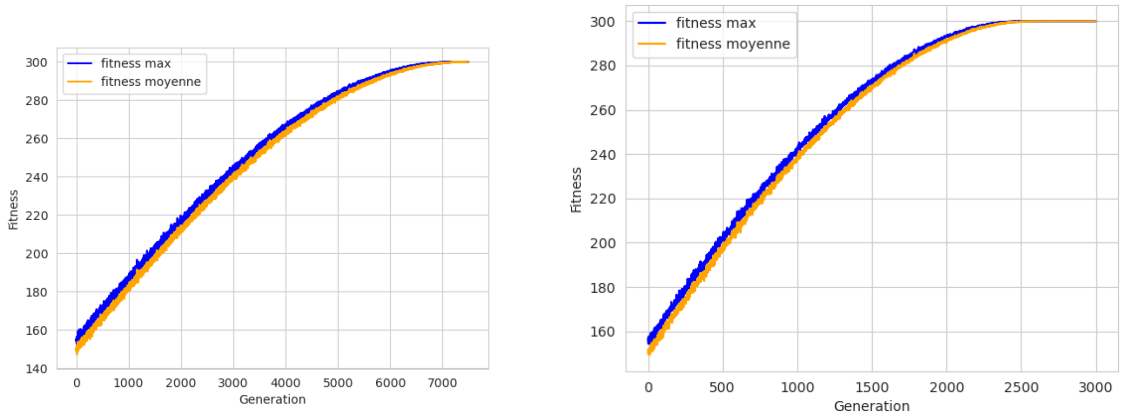
FIGURE 3 – Fitness moyenne en fonction du nombre d'individus sélectionné sur 30 runs

On observe qu'avec notre population par défaut de taille 30, la sélection de 4 à 8 individus est plus performant. Durant les premières générations, la sélection de quatre individus est plus efficace, car on a moins de chance d'utiliser des individus mauvais comme parent. À partir d'un certain nombre de générations, de plus en plus d'individus ont une fitness équivalente à la fitness maximum de la population. Il devient donc plus intéressant de sélectionner huit parents pour générer un nouveau vecteur de probabilité. La sélection de 14 parents n'est pas efficace car certains auront une mauvaise fitness.

Avec une population plus grande, on obtient rapidement un grand nombre d'individus à avoir une fitness équivalent à la fitness maximum de la population. C'est pourquoi, avec une population de taille 80, la sélection de 14 parents devient possible et plus intéressantes que deux ou quatre parents.

3.3 Algorithme génétique compact

Pour deux individus (un winner et un loser) de taille N, le calcul de la probabilité du bit i dans le vecteur de probabilité ω est $\omega_i = \omega_i + \frac{1}{N}$, si $winner_i == 1$; $\omega_i = \omega_i - \frac{1}{N}$, sinon. Afin d'accélérer l'apprentissage, il peut paraître intéressant d'ajouter un coefficient α qui est par défaut égal à 1, $\omega_i = \omega_i + \alpha * \frac{1}{N}$



(a) Avec $\alpha = 1$ (b) Avec $\alpha = 3$

FIGURE 4 – Fitness maximum et moyenne avec en fonction des générations sur 30 runs

Avec un coefficient α égal à un, le nombre de générations nécessaire pour obtenir une solution optimale est très important (environ 7000). On converge très lentement vers la solution optimale. En augmentant, α le nombre de générations nécessaire diminue fortement. La valeur du coefficient dépend fortement de la fonction objective. Il est également nécessaire de spécifier une valeur minimale P_min pour les probabilités du vecteur de probabilité. Sans cette valeur P_min, notre algorithme pourrait rester bloqué dans un optimum local si la probabilité d'un bit est à 0.

4 Sélection automatique de l'opérateur de mutation

L'algorithme génétique steady state peut-être amélioré. En effet, nous avons utilisé par défaut l'opérateur de mutation $bit - flip$. Or, celui-ci peut ne pas être l'opérateur optimal durant les premières générations (Figure 2.a). Notre amélioration consiste donc à permettre à l'algorithme de choisir dynamiquement le type de mutation à appliquer à chaque individu en fonction de son comportement et de l'évolution de la population. Deux types d'algorithme adaptatif ont été implémentés, la roulette adaptative et UCB.

4.1 Roulette adaptative

La méthode de la roulette adaptative utilise un vecteur d'opérateur $\Omega = \{o_1, ..., o_n\}$, avec o_i un opérateur de mutation. À chaque opérateur de Ω est associée une probabilité de sélection enregistrée dans le vecteur θ. L'utilité d'un opérateur au temps t est calculée selon la formule suivante, $u_i^t = (1-\alpha)u_i^{t-1} + \alpha * g(o_i, s^0, .., s^{t-1})$, avec $u_i^0 = 0$ et $g = max(0, nouvellefitness - anciennefitness)$ le gain de l'opérateur. La probabilité d'un opérateur au temps t+1 est calculé de la façon suivante, $\sigma_i^{t+1} = p_{min} + (1 - n * p_{min}) * \frac{u_i^{t+1}}{\sum_{k=1}^{n} u_k^{t+1}}$.

4.2 UCB

La méthode UCB est similaire à la roulette adaptative, elle utilise un vecteur d'opérateur Ω et un vecteur de probabilité θ associés. Chaque opérateur enregistre ses x dernier gain obtenu (fenêtre glissante) dans un vecteur G. La probabilité d'un opérateur est ensuite calculée de la façon suivante, $\sigma_i^t = UCB(\sigma_i, t)/M$, avec M la moyenne des gains de chaque opérateur. On calcule $UCB(\sigma_i, t) = g_i^t + c * \sqrt{\frac{2*log(t)}{nb_i^t}}$, avec g_i^t la moyenne des récompenses obtenues en exécutant l'opérateur i à la génération t, c une constante et nb_i^t le nombre de fois que l'opérateur i a été choisi jusqu'à la génération t. L'UCB permet, grâce à la partie droite de la fonction, de forcer la réutilisation d'un opérateur si ce dernier n'a pas été utilisé depuis longtemps.

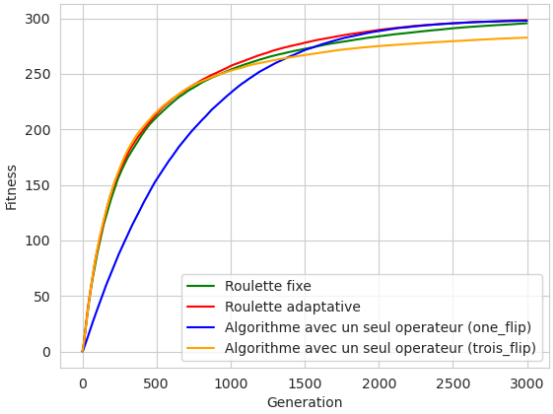
Une analyse comportementale de la roulette adaptative et de l'UCB ont été réalisés. Les résultats sont disponibles dans la cinquième partie du rapport.

5 Analyse expérimentale

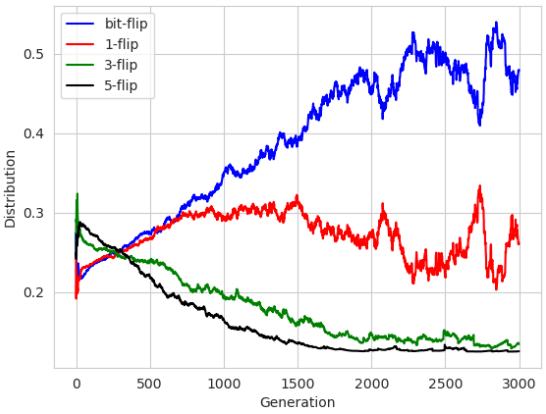
La valeur par défaut des paramètres de nos algorithmes génétique sont les mêmes que dans la première phase expérimentale. Cette fois-ci, on initialise nos individus à zéro afin de mieux comprendre quels sont les opérateurs les plus efficaces selon la fitness de la population. Ainsi, chaque run aura la même solution initiale, les résultats seront donc indépendants de la solution initiale et pourront être comparé plus facilement.

5.1 Roulette adaptative

Nous allons étudier la distribution de probabilité des opérateurs bit-flip, one-flip, trois-flip et cinq-flip avec l'algorithme de la roulette adaptative. Ce dernier sera également comparé à des algorithmes non adaptatifs. Le paramètre α est fixé à 0,1 et p_{min} est fixé à 0,125.



(a) Fitness moyenne en fonction des algorithmes génétiques



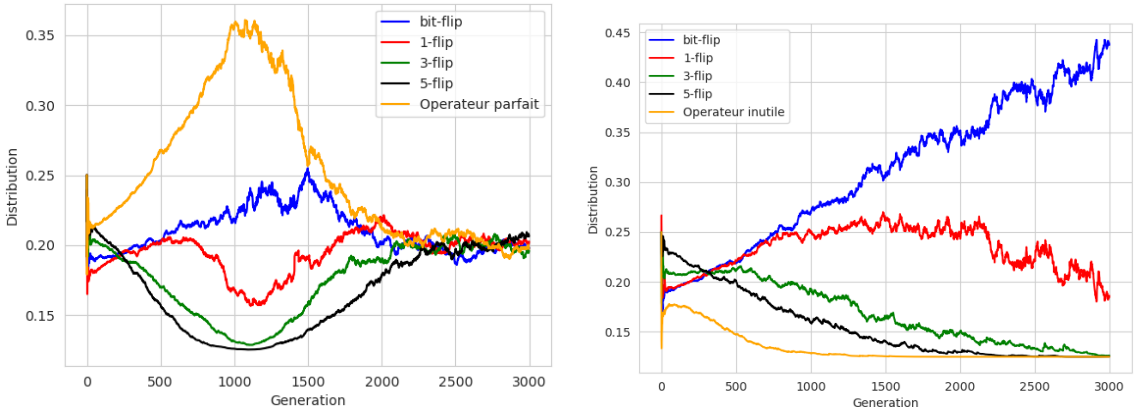
(b) Distribution des opérateurs de mutation au cours des générations

FIGURE 5 – Observation de l'impact de la roulette adaptative sur l'algorithme génétique Steady State sur 30 runs

Sur la figure 5.a, on observe que l'algorithme utilisant la roulette adaptative est le plus performant tout au long des générations. Il est autant performant que l'algorithme utilisant trois-flip au départ et autant performant que l'algorithme utilisant one-flip à la fin. On peut donc supposer qu'il utilise l'opérateur trois-flip sur les premières générations puis one-flip sur les dernières. Il est également supérieur à la roulette fixe (quatre mêmes opérateurs que la roulette adaptative, mais avec une distribution fixe de 0,25).

Notre observation se confirme sur la figure 5.b. Sur les environ 350 premières générations (correspondant à une fitness inférieur à 200), les opérateurs 3-flip et 5-flip sont plus efficaces et ont donc une plus grande probabilité d'être utilisé. Après 350 générations, ce sont les opérateurs 1-flip et bit-flip qui sont les plus efficaces car la probabilité de changer un zéro en un est faible (inférieur à 1/3 car la fitness est supérieur à 200 donc moins de 100 bit à zéro).

Pour vérifier le bon fonctionnement de notre roulette adaptative, on peut ajouter un opérateur parfait (change cinq bit aléatoirement en un) qui doit normalement être le plus souvent utilisé. On peut utiliser la même méthode avec un opérateur inutile (change cinq bit aléatoirement en zéro).



(a) Distribution des opérateurs de mutation avec l'ajout d'un opérateur parfait

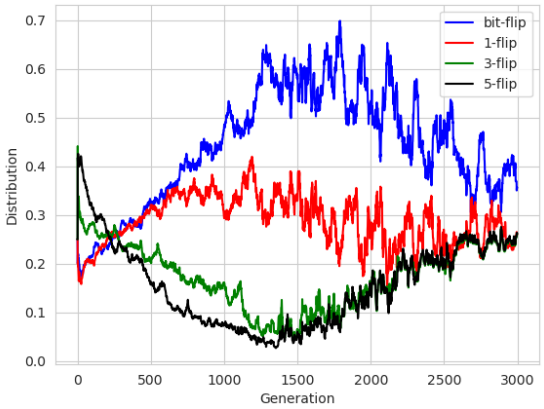
(b) Distribution des opérateurs de mutation avec l'ajout d'un opérateur inutile

FIGURE 6 – Ajout d'opérateur de mutation pour vérifiant le bon fonctionnement de la roulette adaptative

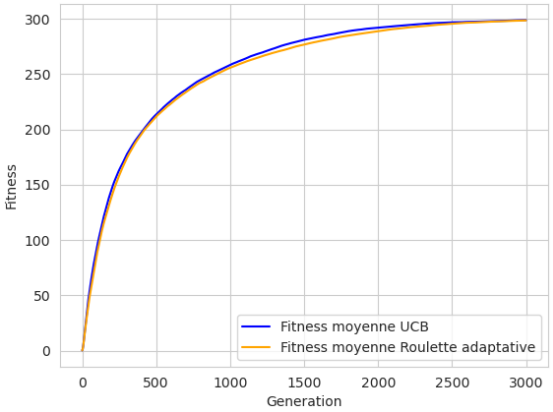
On observe que l'opérateur inutile a rapidement une probabilité quasiment nulle. À l'inverse, l'opérateur parfait à une probabilité très forte d'être utilisé. En 1000 itérations une solution optimale est trouvée avec l'opérateur parfait. Après cela, les opérateurs ne peuvent plus améliorer la solution, les distributions des opérateurs convergent donc toutes vers la même valeur.

5.2 UCB

Comme pour la roulette adaptative, nous pouvons analyser la distribution de probabilité des opérateurs bit-flip, one-flip, trois-flip et cinq-flip avec l'algorithme UCB. Le coéficient c est fixé à 0.01 et la fenêtre glissante est de taille 10.



(a) Distribution des opérateurs de mutation au cours des générations sur 30 runs



(b) Fitness moyenne au cours des générations sur 30 runs

FIGURE 7 – Comparaison de la roulette adaptative avec l'UCB

La distribution des opérateurs est similaire à celle obtenue avec la roulette adaptative. Les opérateurs trois-flip et cinq-flip sont plus utilisés durant les premières générations. Ce sont ensuite les opérateurs one-flip et bit-flip qui sont plus utilisés. La fitness est équivalente que l'on utilise l'UCB ou la roulette adaptative.

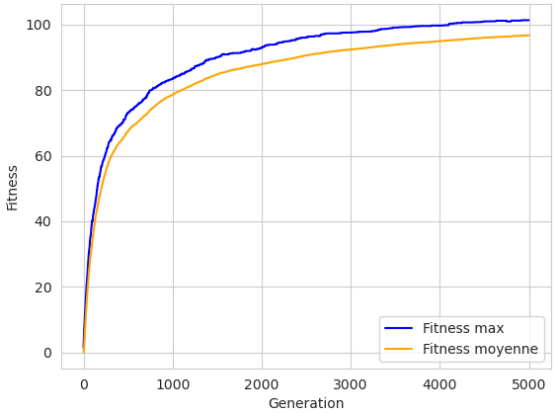
6 Autres problèmes binaires

Il est possible d'appliquer ces algorithmes sur d'autres problèmes binaires simples. La roulette adaptative a été adaptée pour le One-Max avec un masque sur la fonction d'activation ainsi que sur le problème du Leading-Ones.

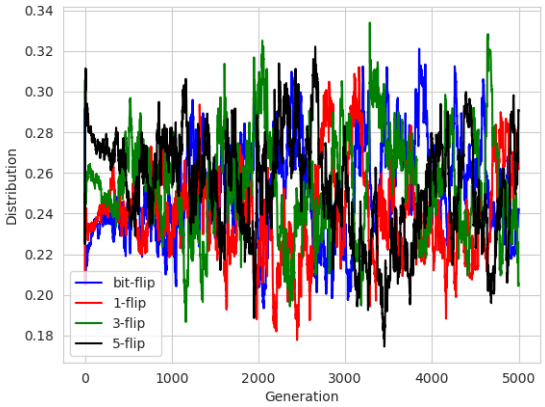
6.1 One-Max avec masque sur la fonction d'évaluation

Cette approche utilise un masque sur la fonction d'évaluation. Le masque m est un vecteur binaire de taille N avec N la taille d'un individu. La nouvelle fonction d'évaluation pour un individu n est : $\sum_{i=0}^{N} n_i * m_i$.

Masque $= 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1$
Individu $= 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0$
Fitness $= 2$



(a) Fitness moyenne et maximum au cours des générations sur 30 runs



(b) Distribution des opérateurs de mutation sur 30 runs

FIGURE 8 – Utilisation de la roulette adaptative sur le One-Max avec masque

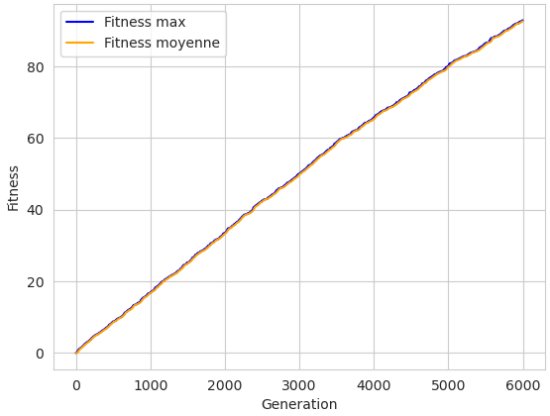
On observe que la roulette adaptative ne parvient pas à trouver une solution. Elle stagne à une fitness de 100, soit 1/3 des bits à un. La distribution des opérateurs est chaotique, l'algorithme ne parvient pas à identifier quels opérateurs est efficaces.

6.2 Leading-Ones

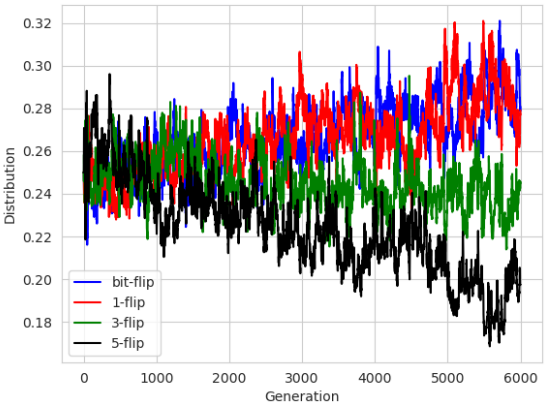
Le problème du Leading-Ones est une seconde variante du One-Max. La fonction d'évaluation réalise la somme des bit à un consécutif depuis le début de l'individu et s'arrête au premier zéro rencontré.

1 1 1 0 0 1 $\longrightarrow$ Fitness = 3

Nous avons appliqué notre algorithme adaptatif utilisant la roulette adaptative pour résoudre ce problème.



(a) Fitness moyenne et maximum au cours des générations sur 30 runs



(b) Distribution des opérateurs de mutation sur 30 runs

FIGURE 9 – Utilisation de la roulette adaptative sur le Leading-Ones

La roulette adaptative parvient à s'adapter à ce problème. Cependant, le temps de résolution est beaucoup plus lent que le problème One-Max classique. L'intervalle de distribution des opérateurs est plus petit ([0.20,0.30] pour 30 000 générations) que celui du One-Max ([0,0.55] pour 30 000 générations), l'algorithme a du mal à déterminer quel opérateur est le plus performant.

7 Conclusion

En conclusion, les algorithmes génétiques sont une approche efficace pour résoudre de nombreux problèmes combinatoires. Ceux-ci peuvent être améliorés afin de s'adapter aux problèmes durant la résolution. Certains algorithmes tels que la roulette adaptative ou l'UCB permettent de rendre l'algorithme génétique adaptatif et donc de mieux s'adapter au problème. Nos expérimentations sont centrées sur le problème du One-Max. On observe des limites si la fonction d'évaluation des individus est trop aléatoire comme c'est le cas pour le One-Max avec masque. De plus les expérimentations sont focalisées que sur certains paramètres de l'algorithme génétique, d'autre choix aurait pu être fait pour analyser les performances des algorithmes.

8 Bibliographie

1. Adrien Goëffon, Frédéric Lardeux, "Toward Autonomous Search with Island Models", *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, ResearchGate, 2012.
2. John H. Holland, "Genetic algorithms", Scholarpedia, 2012.
3. Mark Hauschild, Martin Pelikan, "An introduction and survey of estimation of distribution algorithms", *Swarm and Evolutionary Computation*, Elsevier, 2011.
4. Aguston Eiben, Zbigniew Michalewicz, Marc Schoenauer, Jim Smith, "Parameter Control in Evolutionary Algorithms", HAL, 2007.