Etude des probabilités de croisement et de mutation dans les algorithmes génétiques via l'exemple du Parkour

May 29, 2024

Abstract

Étude de l'impact des probabilités de mutations et de croisements sur un algorithme génétique pour l'optimisation d'un enchaînement de figures à une compétition Freestyle de Parkour ainsi que la comparaison de la performance de cet algorithme avec un autre algorithme génétique ayant une fonction de score semblable au premier abord à celle utilisée pour le Parkour.

1 Introduction

Le Parkour a gagné beaucoup de popularité chez les jeunes depuis les années 90s, notamment mis en lumière par des films comme "Yamakasi" ou "Banlieue 13". Ce sport fût mon coup de coeur dès lors que j'ai commencé à en pratiquer **à** depuis mes 14ans, ce qui justifie l'importance que j'accorde à ce TIPE.

Le Parkour est un sport sans réelles règles définies au préalable comme pourrait l'être le volleyball. Il s'agit d'entrainements à la maîtrise de son corps à travers des figures comme des saltos dans différents environnements, en ville tout comme en salle de gymnastique. Cette liberté au niveau des règles à permis l'émergence d'une diversité de compétitions selon différentes règles comme celle qui nous intéresse aujourd'hui : *La compétition de Parkour Freestyle.*

Pour cette compétition, l'athlète dispose de 70 secondes ainsi qu'une scène avec des murets, murs et barres en métal pour réaliser le meilleur enchainement de figures possible. Un score lui est ensuite attribué selon la difficulté des figures, la fluidité de l'enchainement ainsi que la variété des figures.

On cherche alors à optimiser l'enchainement de figures pour obtenir le meilleur score possible. Pour cela, on utilisera un algorithme génétique qui aura pour but de trouver le meilleur chemin sur le terrain pour un athlète donné. Dès lors, nous étudierons sa cohérence avec la réalité et l'impact des probabilités de croisement et de mutation sur la performance de cet algorithme à converger vers un chemin optimal. De plus, nous comparerons ces résultats à l'étude de Kalyanmoy Deb et Samir Agrawal [1] sur une fonction de score similaire à notre problème.

2 Définitions et modélisation

On va définir dans cette section les termes et concepts importants pour la compréhension de ce TIPE.

2.1 L'algorithme génétique

L'algorithme génétique est une méthode d'optimisation qui s'inspire de la théorie de l'évolution. Il est composé de 4 étapes principales : la génération, l'évaluation, la sélection et la reproduction (Figure 1).

L'objectif est de trouver la meilleure solution, c'est à dire maximiser une fonction de score, à un problème donné en faisant évoluer une population d'individus au fil des générations.

Définition : Chromosome / Gènes

On appelle un *gène* une variable d'état d'un individu, un *chromosome* est alors une suite de gènes $(g_n)_n$.

Définition : Individu

Pour notre problème, on appelle un individu, et moins formèlement un athlète, le couple d'un *chromosome* et de son *niveau de maîtrise*.

Définition : Fonction de score

Une fonction de score est une fonction qui, à un individu, attribue un score.

Nous la notons ici $f : \left((g_n)_n, k\right) \mapsto f(g,k) \in \mathbb{R}$

L'opération d'évaluation revient à appliquer la fonction de score à chaque individu de la population.

La sélection est l'opération qui consiste à choisir les d meilleurs individus afin de les reproduire.

La mutation et le croisement sont des opérations ayant des probabilités p_m et p_c de s'exécuter. La mutation consiste à modifier un gène de l'individu tandis que le croisement consiste à échanger des gènes entre deux individus sélectionnés.

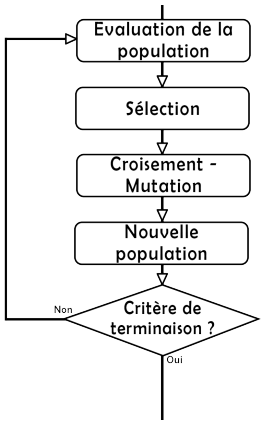

Figure 1: Schéma de l'algorithme génétique

Définition : Génération et Exécution

On appelle *génération* un tour de boucle et une *exécution* de l'algorithme génétique l'ensemble des générations jusqu'à validation du critère de terminaison.

Définition : Critère de terminaison

Le critère de terminaison est une condition qui, si elle est vérifiée, arrête l'exécution de l'algorithme génétique. Dans notre cas, l'arrêt intervient si le score de l'individu le plus performant à atteint un seuil s (dépendant du niveau de l'athlète) ou si le nombre maximal d'évaluation est atteint.

Définition : Performance

La performance d'un algorithme génétique est la moyenne des exécutions qui ont terminées car le seuil s a été atteint par un athlète (et non car le nombre d'évaluation maximal a été atteint) par rapport au nombre total d'exécutions.

2.2 Choix de modélisation

On explique ici les choix de modélisation pour notre problème de Parkour Freestyle.

2.2.1 Le terrain

Le terrain est modélisé par un graphe planaire de taille 10×40 où chaque sommet admet un type :

- 0 : sol
- 1 : mur
- 2 : barre

Nous choisissons de baser notre graphe sur le terrain de la compétition Freestyle à Sofia en 2021.

Figure 2: Scène de la compétition Freestyle de Parkour à Sofia en 2021, © FIG

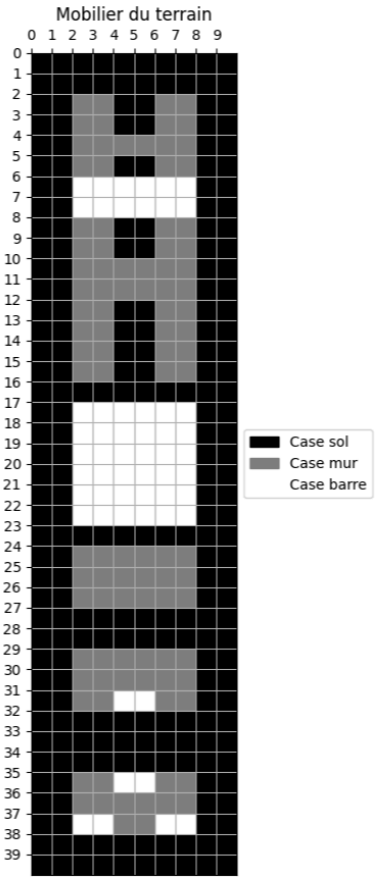

Figure 3: Graphe planaire modélisant le terrain

2.2.2 Les individus

En adéquation avec la définition donnée plus haut, nous définissons plus formèlement la suite $(g_n)_n$ et le niveau de maîtrise d'un individu :

Définition : Gènes et maîtrise

On définit la suite de gènes, ou *chromosome*, d'un individu comme une suite de 3 indices $(g_n)_n$ tels que :

- $(g_{3n})_n$: La suite des abscisses de réalisation des figures
- $(g_{3n+1})_n$: La suite des ordonnées de réalisation des figures
- $(g_{3n+2})_n$: La suite des figures faites

Nous définissons le niveau de maîtrise d'un individu comme un entier entre 0 et 10, 0 étant la personne n'en ayant jamais fait et 10 un niveau inatteignable réellement.

Donc tous les 3 indices, on à les informations d'un point sur notre terrain, ces chromosomes représentent alors un *chemin* dans notre graphe avec comme information supplémentaire à la position : la figure faite, voir Figure 4.

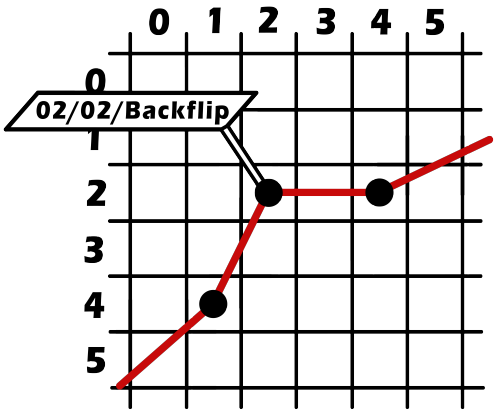

Figure 4: Représentation d'un bout de chemin sur un graphe, chaque point est un triplet de gènes : $(g_{3n}, g_{3n+1}, g_{3n+2})$ = (abscisse, ordonnée, figure)

2.2.3 La fonction de score

La notation d'un athlète est régie par des règles et des points accordables décidés par la *FIG* (Fédération internationnale de Gymnastique) dont voici les documents officiels :

Points	Reference Elements	Example
0	Running	
0.5	Parkour classics, handsprings	
1	Basic flips, baby giants	
1.5	180, gaet flip, pistol-set backflip, ping back	Regrasp-90
2	360, cast backflips, giant, inward flips	Regrasp-0, cork
2.5	540	
3	720	double cork
3.5	900	
4	1 ½ flips, double swing gainer	
4.5	Double flips, 1080	
5	*More difficult moves than 4.5 or reached with connection-upgrade*	

Figure 5: Extrait des points par figure pour la compétition féminine de Parkour Freestyle [2]

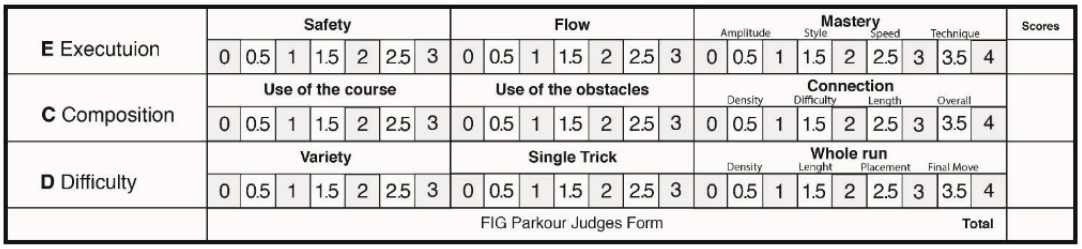

Figure 6: Extrait de la fiche jury [3]

Donc pour noter un individu, on va appliquer les règles données par Figure 5 et Figure 6 sur le chemin représenté de l'individu.

- Pour les sous catégories de la notation subjectives pour les jurys (ie *Mastery*, *Connection*, ...), on les modélise par des fonctions linéaires de la forme :

$$n(x) = \frac{M}{10}x$$

  Avec x le niveau de maîtrise de l'individu et M le nombre de points maximum pour cette catégorie. Pour un individu de niveau de maîtrise 8, la formule calculant le score de *Connection* est alors $8 \times \frac{4}{10} = 3.2$ sur 4.

- Pour les sous catégories de la notation moins subjectives (ie *Variety*, *Use of the obstacles*, ...), on les modélise par des fonctions calculant le nombre de cases différentes dans le chemin ou le nombre de figures différentes réalisées.

Voir implémentation de la fonction de score : Section A.1.

2.2.4 Critère de terminaison

On remarque que l'implémentation de notre fonction de score (Section A.1) donne un maximum de $15 + \frac{15}{10}x$ avec x le niveau de maîtrise de l'individu ainsi que les scores sont attribués par demi-points (Figure 6) . On en déduit la fonction de seuil :

Définition : Seuil

On définit le seuil comme la fonction $s : \mathbb{N} \to \mathbb{R}$ telle que $s(x) = 14.5 + \frac{15}{10}x$ pour un individu de niveau de maîtrise x.

2.3 Les opérations de croisement et de mutation

On définit dans cette section les opérations de croisement et de mutation pour le problème du Parkour.

2.3.1 Croisement

Soient deux individus A et B avec des chromosomes $(g_n)_n$ et $(h_n)_n$, s'il y a croisement entre ces deux individus, on vérifie s'ils ont deux points du chemin en commun sans prendre en compte les figures.

Ce qui revient à regarder s'il existe i_1 et j_1 tels que $g_{3i_1} = h_{3j_1}$ et $g_{3i_1+1} = h_{3j_1+1}$, de même pour i_2 et j_2 différents de i_1 et j_1.

Dès lors, on pose l'opération de croisement qui prend deux chromosomes et les croise en prenant les points en commun pour former deux nouveaux chromosomes :

Opérateur : Croisement

Soit deux individus $A = \left((g_n)_n, k\right)$ et $B = \left((h_n)_n, k\right)$ de niveau égal, on définit l'opération de croisement par la fonction $C : \left(\mathbb{N}^{\mathbb{N}}\right)^2 \to \left(\mathbb{N}^{\mathbb{N}}\right)^2$ suivante, en reprenant les mêmes notations que précédemment:

$$C\left((g_n)_n, (h_n)_n\right) = \begin{cases} \begin{pmatrix} (g_1,\dots,g_{3i_1-1},h_{3j_1},\dots,h_{3j_2-1},g_{3i_2},\dots,g_n) \\ (h_1,\dots,h_{3j_1-1},g_{3i_1},\dots,g_{3i_2-1},h_{3j_2},\dots,h_n) \end{pmatrix} & \text{s'il existe deux sommets en commun} \\ \begin{pmatrix} (g_1,\dots,g_n) \\ (h_1,\dots,h_n) \end{pmatrix} & \text{s'il n'existe pas deux points en commun} \end{cases}$$

Cette opération est illustrée par Figure 7.

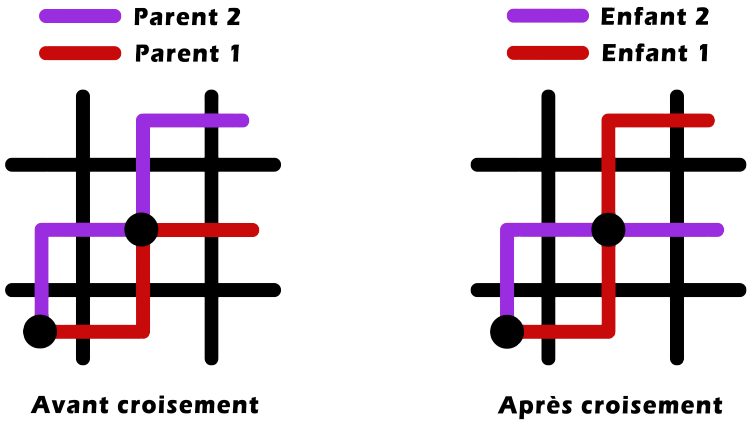

Figure 7: Schéma de l'opération de croisement. Le croisement du parent 1 et 2 donne les enfants 1 et 2 tel que l'enfant 1 ait le début du parent 1 et la fin du parent 2, l'enfant 2 le chemin complémentaire.

Voir implémentation de l'opération de croisement : Section A.2.

2.3.2 Mutation

En reprenant les mêmes individus A et B définis à la section précédente, on définit l'opération de mutation qui, à un chromosome, modifie un de ses gènes (représentée par Figure 8).

Afin de respecter l'aspect réalisable, on définit une zone de déplacement maximale pour chaque point du chemin (ici les points noirs) correspondant à la distance maximale que l'athlète peut parcourir en une seconde, on prend pour valeur 6m/s (soit 21km/h).

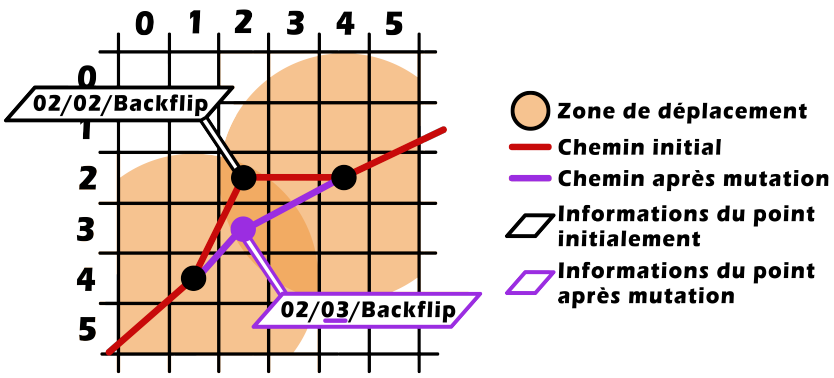

Figure 8: Schéma de l'opération de mutation

Pour muter un gène g_i, on l'incrémente ou le décrémente en fonction de si le nouveau gène est toujours réalisable (ie dans la zone de déplacement, toujours dans les limites du terrain ou avec un identifiant de figure valide).

Opérateur : Mutation

Soit un individu $A = \left((g_n)_n, k\right)$, on définit l'opération de mutation par la fonction $M : \mathbb{N}^{\mathbb{N}} \times \{-1, 1\} \to \mathbb{N}^{\mathbb{N}}$ qui après choix aléatoire d'un gène g_i et d'une valeur ε tel que $\varepsilon \in \{-1, 1\}$, retourne le chromosome modifié :

$$M\left((g_n)_n, \varepsilon\right) = \begin{cases} (g_1, ..., g_{i-1}, g_i + 1, g_{i+1}, ..., g_n) \text{ si } g_i + \varepsilon \text{ est réalisable} \\ (g_1, ..., g_{i-1}, g_i - 1, g_{i+1}, ..., g_n) \text{ sinon si } g_i - \varepsilon \text{ est réalisable} \\ (g_1, ..., g_n) \text{ sinon} \end{cases}$$

Cette opération est illustrée par Figure 8.

Remarque : Si la mutation n'est pas réalisable, on ne modifie pas le gène et on mute le gène de l'individu suivant.

Voir implémentation de la mutation : Section A.3.

3 Optimisation du code

Afin de pouvoir comparer les performances de notre algorithme génétique avec celui discuté dans Section 5, nous allons devoir effectuer un grand nombre d'exécutions de notre algorithme (autour de 4000). L'optimisation du temps de calcul est donc primordiale.

Le choix de Python ne semble pas être le plus adapté pour ce genre de calculs, mais permettait une implémentation beaucoup plus rapide que tous les concepts abordés précédemment avec de la programmation orientée objet. Nous cherchons donc à optimiser notre code malgré cette contrainte.

3.1 Parallélisme

Les différentes exécutions de notre algorithme génétique étant complétement indépendantes, nous choisissons de les exécuter en parallèle en utilisant notamment du *multiprocessing*.

Le choix du multiprocessing est dû au fait que le *multithreading* n'est pas adapté pour les calculs intensifs en CPU et n'utilise qu'un seul coeur de notre processeur. Le *multiprocessing* permet d'utiliser plusieurs coeurs et donc d'accélérer les calculs.

L'implémentation du parallélisme étant sommaire, nous n'en détaillerons pas le code ici. La seule information intéressante est que l'on calcule 8 processus en parallèle ayant comme but l'exécution de 50 exécutions d'algorithmes génétiques pour chacun des 5 couples (p_c, p_m) pour une taille de population, tous définis dans Section 5.2.

3.2 Mutation Clock Operator

Afin de se rapprocher de l'implémentation de l'algorithme génétique implémenté par l'étude de DEB [1] sur laquelle on base notre comparaison plus tard, nous décidons d'implémenter le calcul des prochaines mutations de manière plus efficace que la façon naïve :

```
if random.randint(0, 100)/100 < cross_prob:
    # Opération de mutation
```

Pour cela, nous nous basons sur les travaux de Kalyanmoy et Debayan DEB [4] qui introduisent le concept de *Mutation Clock Operator*.

Il s'agit de déterminer le prochain gène d'un prochain individu à muter depuis l'individu actuel à l'aide d'une probabilité exponentielle :

$$p(t) = p_m e^{-p_m t}$$

Dès lors, nous choisissons un nombre u aléatoire entre 0 et 1 et calculons l tel que :

$$u = \int_0^l p(t)dt = \int_0^l p_m e^{-p_m t} dt$$

$$\iff \qquad l = \frac{1}{p_m} \ln(1-u)$$

Ainsi, si nous venons de muter le i-ième du k-ième individu d'une population de n individus, le prochain gène que l'on mutera sera le $(k+l)\%n$-ième gène du $\frac{k+l}{n}$-ième individu.

Cette opération permet, dès lors, de ne générer plus qu'un seul nombre aléatoire u pour toute une *exécution* au lieu de n pour chaque *génération*.

Les auteurs DEB ont d'ailleurs montré dans cette étude que cette méthode était :
- Plus efficace en terme de temps de calcul
- Plus efficace en terme de performance de l'algorithme génétique.

Donc tout ce que l'on recherche pour notre problème. Après implémentation de cette méthode, nous avons pu réduire le temps d'exécution de notre algorithme de 1 jour à 1h30 pour 4000 excéutions.

4 Cohérence de la notation

Nous cherchons à vérifier si notre implémentation de la fonction de score est cohérente avec les scores attribués dans de vraies compétitions.

Pour cela, on va se baser sur l'enchainement de figures réalisé par la championne du monde de Parkour Freestyle en 2021, Lilou Ruel, lors de la compétition de Sofia [5]. On va implémenter son enchainement comme un chemin et le noter avec notre fonction de score.

Après notation, nous avons les résultats suivant :

Catégories	Score Réel	Score Calculé
Execution	7	8.6
Composition	7	9.2
Difficulté	9	7.2
Total sur 30	23	25

Table 1: Comparaison des scores de Lilou Ruel, extraits de mon programme et des résultats de la compétition [6]

On voit que notre l'implémentation de la fonction n'est pas cohérente avec les scores réels (2 points d'écarts dans chaque catégorie). Donc notre modèle n'est pas forcément le plus adapté, notamment pour les raisons suivantes :

- La notation est grandement subjective et dépend de l'appréciation des jurys
- L'implémentation de l'enchainement de figures depuis la vidéo [5] n'est pas viable car on n'a pas la liste des figures réalisées par Lilou Ruel. J'ai dû juger par moi même les figures qu'elle faisait. De plus, si celles ci ne sont pas référencées dans la liste de figure dans Figure 5, je dois choisir la figure la plus proche visuellement et donc on s'éloigne de la notation des juges.
- La modélisation du terrain est surement trop simpliste pour représenter toutes les possibilités de figures que permet le véritable terrain.

Je pense donc que cette fonction de score (qui à un chemin associe un score) n'est pas semblable à la notation réelle. Cependant, nous étudierons quand même dans la suite de ce document l'impact des probabilités de croisement et de mutation sur la *performance* de notre algorithme génétique avec cette fonction de score.

5 Comparaison des performances

Malgrè la cohérence discutable de notre fonction de score avec la réalité, nous cherchons tout de même à comparer les performances de notre algorithme génétique avec un de ceux proposés par Kalyanmoy DEB et Samir AGRAWAL dans leur étude : *Understanding Interactions among Genetic Algorithm Parameters* [1].

5.1 Fonction de Rastrigin

Dans l'étude de DEB et AGRAWAL, les auteurs présentent différentes fonctions de score et les différentes performances de l'algorithme génétique sur celles ci selon les probabilités et les tailles de population différentes.

Nous choisissons ici de comparer la performance de l'algorithme génétique avec notre fonction de score à la fonction de Rastrigin. Cette fonction à la particularité d'avoir un minimum global en $(0, 0)$ ainsi que de nombreux extrémas locaux pouvant être des pièges pour des algorithmes d'optimisations. Ce pourquoi elle est souvent utilisée pour tester la performance de ces algorithmes.

Définition : Fonction de Rastrigin

Soit $x = (x_1, ..., x_n) \in [-5.12, 5.12]^n$, la fonction de Rastrigin est définie en dimension n par :

$$f((x_1, ..., x_n)) = 10n + \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i)), \forall x \in [-5.12, 5.12]^n$$

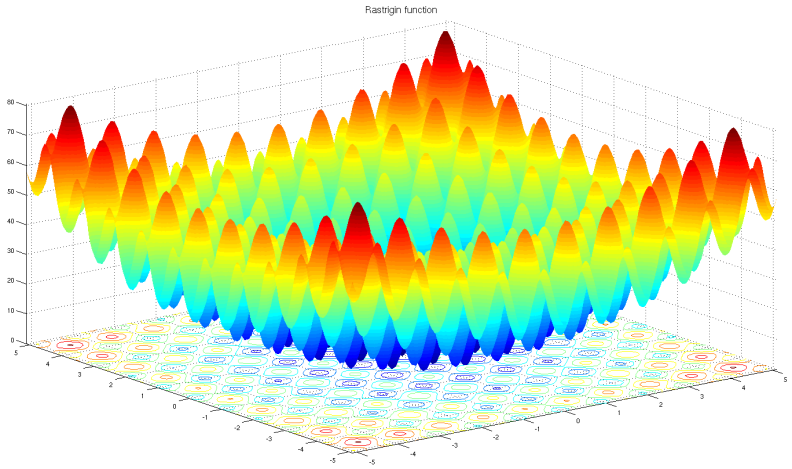

Figure 9: Représentation graphique en dimension 2 de la fonction de Rastrigin

J'ai choisi de comparer notre fonction de score à cette fonction étant donné que, de même, notre fonction de score admet beaucoup d'extrémas et qu'en particulier il existe beaucoup de chemins qui peuvent effectivement mener au maximum de score en ayant des déplacements sur le terrain différents.

Comparer notre fonction de score à une fonction n'ayant pas tant d'extremas locaux ne me paraissait pas pertinent pour notre modèle.

5.2 Comparaison

De même que pour l'étude [1], nous avons choisi un nombre d'évaluation de 45 000.

Après avoir fait 50 excéution de chaque couple (p_m, p_c) pour une taille de population variant de 2 à 2000, pour une durée totale de 1h30 (contre 1jour avant optimisation), nous obtenons les résultats suivants :

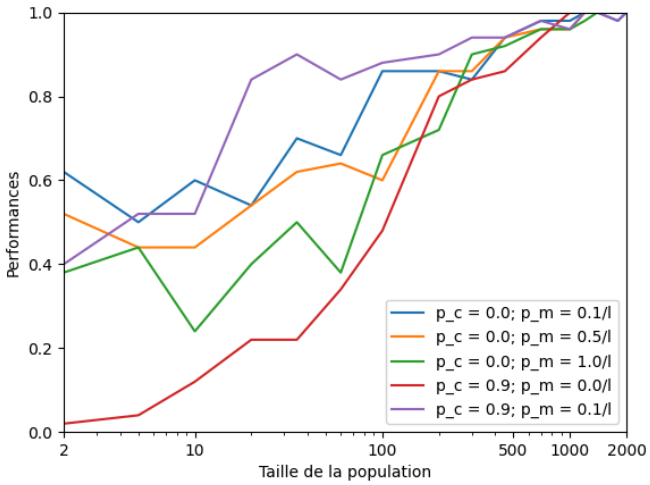

Figure 10: Performance sur notre fonction de score

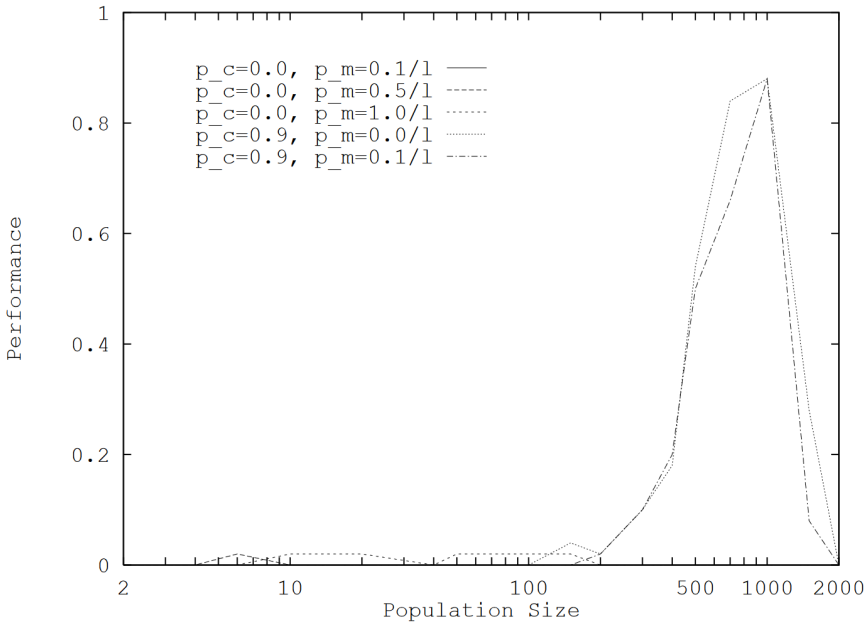

Figure 11: Performance sur la fonction de Rastrigin, tiré de l'étude [1]

Il est alors intéressant de noter que nos résultats Figure 10 sont complétement contraire à ceux obtenus par DEB et AGRAWAL Figure 11. En effet, ils développent dans leur étude que la probabilité de croisement p_c est **nécessaire** pour la performance de l'algorithme sur ces types de fonctions, la probabilité de mutation p_m n'ayant pas d'impact significatif sur la performance.

Alors qu'en revanche, nous obtenons des résultats inverses : la probabilité de croisement semble avoir un effet néfaste sur la performance, du moins sur les faibles populations, alors que la probabilité de mutation semble avoir un effet tendancieux positif sur la performance sur un faible population.

Mais ce qui choque énormément est que la performance de notre algorithme, peu importe les paramètres, converge toujours vers 1 autour de 1000 exécutions. Cela est très étonnant compte tenu du fait que les performances de l'algorithme de DEB et AGRAWAL admet un pic autour de 1000 individus pour ensuite redescendre à 0 autour de 2000 individus.

Ceci peut s'expliquer par la génération quasi-systématique d'un chemin déjà optimal pour notre fonction de score parmis une population de 1000 individus. En effet, étant donné que notre fonction de score admet beaucoup d'extremas locaux **et** globaux, il est possible que l'algorithme génétique tombe sur un chemin optimal dès la première génération.

6 Conclusion

Après implémentation de la notation d'une compétition de Parkour Freestyle, nous avons pu voir que celle ci n'était pas cohérente avec les scores réels mêmes si les scores calculés étaient cohérents entre eux. De plus, les différentes probabilités de croisement et de mutation n'ont pas eu l'effet escompté sur la performance de notre algorithme génétique dû à la présence inhérente du grand nombre d'antécédents du maximal global de notre fonction de score.

Ainsi, ces probabiltés ne joue qu'un rôle mineur dans la convergence d'un algorithme génétique vers la solution optimale pour une fonction avec beaucoup d'extremas globaux pour une faible population. Pour une forte population, une solution optimale semble être trouvée dès la première génération, ce qui pourrait être une piste pour la recherche d'un algorithme probabiliste pour trouver une solution optimale et la comparaison de ces deux méthodes d'approximation selon un temps et/ou un espace mémoire limité.

Bibliography

[1] Kalyanmoy Deb and Samir Agrawal, *Understanding Interactions among Genetic Algorithm Parameters*

[2] Parkour Commission, *PK CODE OF POINTS 2022-2024 - TABLE OF TRICKS 2023*, (n.d.)

[3] Fédération internationale de Gymnastique, *CODE DE POINTAGE 2019-2021*, (n.d.)

[4] Kalyanmoy Deb and Debayan Deb, *Analyzing Mutation Schemes for Real-Parameter Genetic Algorithms*

[5] Fédération internationale de Gymnastique, *2022 Sofia Parkour World Cup* (n.d.)

[6] Fédération internationale de Gymnastique, *FIG Parkour World Cup Sofia - Results Book*, (n.d.)

APPENDIX A Implémentation

Note : Nous implémentons les gènes avec la subtilités des 3 indices tous les 6 indices afin de pouvoir représenter les gènes par une chaine de caractères (une variable est encodé sur 2 caractères (un entier de 0 à 99)), voir Section 2.2.2.

A.1 Code de la fonction de score

```
def calc_fitness(self) -> int:
    """Calcule le score de l'athlète"""
    score =  {
        "execution": {"safety": 3,"flow": 0,"mastery": 0},
        "composition": {"use_of_space": 0,"use_of_obstacles": 0,"connection": 0},
        "difficulty": {"variety": 0,"single_trick": 0,"whole_run": 0},
    }

    # Liste des figures faites
    nb_figure = len(self.genes)//6
    tricks = [Figure.figures[int(self.genes[6*i+4: 6*i+6])]
              for i in range(nb_figure)]
    field = self.athlete.field
   cases = [field.getCase((int(self.genes[6*i:6*i+2]), int(self.genes[6*i+2: 6*i+4])))
      for i in range(nb_figure)]

    # Calcul de la sureté des figures
    score["execution"]["safety"] = (score["execution"]["safety"])*self.athlete.xp/10

    # Calcul du flow
    # Compte le nb de fois qu'on s'est arreté
    score["execution"]["flow"] = 3 - tricks.count(FIGURES["do_nothing"])

    # Calcul de la maitrise
    score["execution"]["mastery"] = 4*self.athlete.xp/10

    # Calcul de l'utilisation de l'espace
    # Compte le nb de cases différentes utilisées
    l = []
    for case in cases:
        if case.id not in l: l.append(case.id)
    score["composition"]["use_of_space"] = len(l)


    # Calcul de l'utilisation des obstacles
    # Compte le nb de types de cases différents utilisés
    l = []
    for case in cases:
        if case.name not in l: l.append(case.name)
    score["composition"]["use_of_obstacles"] = len(l)

    # Calcul de la connexion entre les obstacles
    score["composition"]["connection"] = 4*self.athlete.xp/10

    # Calcul de la variété
    # Ajoute 1 pts a chaque figure de complexité >= 2
    score["difficulty"]["variety"] = sum([1 for trick in tricks
```

```
                                        if trick.complexity >= 2])

    if score["difficulty"]["variety"] > 3: score["difficulty"]["variety"] = 3

    # Calcul de la difficulté d'un trick
    # Ajoute 1 pt par trick de complexité >= 3
    score["difficulty"]["single_trick"] = sum([1 for trick in tricks
                                               if trick.complexity >= 3])

    if score["difficulty"]["single_trick"] > 3: score["difficulty"]["single_trick"]=3

    # Calcul de la difficulté d'un run
    score["difficulty"]["whole_run"] = 4*self.athlete.xp/10

    # Calcul du score final
    self.fitness = sum([sum(score["execution"].values()),
                        sum(score["composition"].values()),
                        sum(score["difficulty"].values())])

    if self.fitness < 0: self.fitness = 0
    return self.fitness
```

A.2 Code du croisement

```
def crossover(parents: list, probs) -> list:
    """
    Crée les enfants de la prochaine population
    On choisit 2 parents et on les on prend 2 moins communs aux deux
    chemins (s'il y a) et on échange les chemins entre ces deux points

    Si les deux parents sont en réalité le même, on le copie tel quel.

    Params:
        parents (AthleteChromosome list): liste d'athlètes

    Returns:
        children (AthleteChromosome list): liste d'athlètes enfants
    """
    children = []
    CROSSOVER_PROB, _ = probs
    for i in range(0, len(parents)-1, 2):
        c1, c2 = get_point_communs(p1, p2)

        if c1 != -1 and c2 != -1\
                        and randint(0, 100)/100 < cross_prob:

            # Premier enfant, avec un premier croisement des combos
            child1 = Athlete(p1.athlete.xp)
            child1.setField(p1.athlete.field)
            child1.combos = from_string_to_combos(p1.genes[:c1]+p2.genes[c2:])
            childChro1 = AthleteChromosome(child1)

            # Deuxieme enfant, avec le croisement complémentaire au premier
            child2 = Athlete(p2.athlete.xp)
            child2.setField(p2.athlete.field)
```

```
            child2.combos = from_string_to_combos(p2.genes[:c2] + p1.genes[c1:])
            childChro2 = AthleteChromosome(child2)

            children.append(childChro1)
            children.append(childChro2)

        else:
          children.append(copy_chromosome(p1))
          children.append(copy_chromosome(p2))

    return children
```

A.3 Code de la mutation

```
def mutation(population:list, l: int) -> list:
    """
    Fait muter la `population`, en ajoutant 1 ou -1 à un gène aléatoire
    (position x, position y ou l'indentifiant de la figure) selon le dernier muté
    et du paramètre `l` (Mutation Clock operation) et la cohérence de
    ce changement avec le modèle réel

    Params:
        population (AthleteChromosome list): liste d'athlètes
        l (int): nombre associé à une probabilité selon la 2nd étude sur les GAs

    Returns:
        population (AthleteChromosome list): liste d'athlètes enfants

    """
    global k, i
    L = 6*70 #Nombre de variables maximal représentant un gène

    has_mutated = mutation_individual(population[i], k)

    if not has_mutated:
        i = (i+1)%len(population)

    else:
        k = int((k+l)%L)
        i = (i + ceil((k+l)/L))%len(population)

    return population

def mutation_individual(athleteChromosome: AthleteChromosome, k:int):
    """
    Mutation en place de l'athlete `athleteChromosome` passé en paramètre.
    Le caractère du gene à modifier est imposé par le paramètre `k` contenu
    entre 0 et 419 inclus.
    """
    # k = Indice du caractère à modifier
    # i = Indice du gene contenant la variable
    k = k%len(athleteChromosome.genes)
    i = (k - k%6)//6

    # Etat associé au gène
```

```
e = athleteChromosome.genes[i*6: (i+1)*6]

# Positions et figure associé à l'état
x = int(e[0:2])
y = int(e[2:4])
f = int(e[4:6])

modifieur = choice([-1, 1])

# Récupération des états précédant et succédant l'état à l'étude
if i == 0 :
    e1 = e
else:
    e1 = athleteChromosome.genes[(i-1)*6: i*6]

if i >= len(athleteChromosome.genes)//6 - 1:
    e3 = e
else:
    e3 = athleteChromosome.genes[(i+1)*6: (i+2)*6]

has_mutated = True

# Match sur la composante qui va être modifiée
match (k%6)//2:
    case 0 : # Si on modifie la variable de l'abscisse
        # Le modifieur étant choisi avant le match, on vérifie que
        # la modification apporté à l'abscisse n'enfreint aucune
        # des conditions de bons fonctionnements comme :
        # 0 <= x < SIZE_X et que le déplacement à cette case depuis
        # la case précédente e1 est possible et le deplacement vers e3,
        # assuré par le renvoie "true" de la fonction coherence_suite_etats
        e2 = from_combo_to_string(
            [((x+modifieur, y), Figure.getFigureById(f), 0)])

        if x + modifieur >= 0 and x + modifieur < SIZE_X:
            if coherence_suite_etats(e1, e2, e3):
                x += modifieur

            else:
                if x-modifieur >= 0 :
                    e2_recovery = from_combo_to_string(
                            [((x-modifieur, y), Figure.getFigureById(f), 0)])

                    if coherence_suite_etats(e1, e2_recovery, e3):
                        x -= modifieur

                    else: has_mutated = False

        else:
            if x-modifieur >= 0 and x-modifieur < SIZE_X:
                e2_recovery = from_combo_to_string(
                    [((x-modifieur, y), Figure.getFigureById(f), 0)])

                if coherence_suite_etats(e1, e2_recovery, e3):
                    x -= modifieur
```

```
                else: has_mutated = False
            else: has_mutated = False


    case 1: pass # Sensiblement la même chose que précédemment mais pour l'ordonné

    case 2: # Cas du changement de la figure
        if f + modifieur >= len(FIGURES) or f+modifieur < 0 :
            f -= modifieur
        else:
            f += modifieur

# Reconstruction du gène
gene = athleteChromosome.genes[0: i*6] +\
        from_combo_to_string([((x, y), Figure.getFigureById(f), 0)])+\
        athleteChromosome.genes[(i+1)*6:]

# Modification en place de l'athlete
athleteChromosome.genes = gene
athleteChromosome.athlete.combos = from_string_to_combos(gene)

return has_mutated
```