UNIVERSITÉ S'ANGERS

Algorithmes intelligents pour l'aide à la décision

Étude d'un Algorithme Génétique pour la résolution du problème One-Max

Lepinette Cyril

10 février 2020

TABLE DES MATIÈRES

1 Introduction **3**

2 Notion de solution pour le problème one-max **5**

3 Analyse de la sélection des opérateurs **6**

3.1 Méthodologie 6

3.2 Pas de sélection : opérateur 1 flip 7

3.3 Récapitulatif 8

3.4 Oracle Myope 11

3.4.1 Présentation 11

3.4.2 Résultats 12

3.4.3 Conclusion 17

3.5 Roulette fixe 18

3.5.1 Présentation 18

3.5.2 Conclusion 18

3.5.3 Résultats 18

3.6 Adaptive wheel 22

3.6.1 Présentation 22

3.6.2 Résultats 23

3.6.3 Conclusion 24

3.7 Adaptive Pursuit 25

3.7.1 Présentation 25

3.7.2 Conclusion 29

3.8 Upper Confidence Bound 30

3.8.1 Présentation 30

3.8.2 Tentative d'amélioration 35

3.8.3 Conclusion 36

3.9 Exp3 36
3.9.1 Présentation 36
3.9.2 Résultats 37
3.9.3 Conclusion 38

4 Analyse de la sélection des individus **39**
4.1 Critère de score 39
4.2 Critère d'âge 40
4.3 Conclusion 40

5 Conclusion **41**

1 INTRODUCTION

Le problème proposé à la résolution est "one max", dont le but est d'obtenir un individu ne contenant que des '1' à partir d'une population initiale.

1. Initialisation : [0000, 0000]
2. Étape intermédiaire : [0110, 0001]
3. État final : [1111, 1111]

Ce problème est facile à modéliser, cependant il ne contient pas de minimum local. Il n'est donc pas représentatif d'autres problèmes ayant des minimums locaux. De ce fait, il n'est pas général et aurait besoin d'être adapté pour d'autres problèmes tels que ceux étudiés en cours comme le "sport scheduling tournament" ou bien le "social golfer". En effet, nous pouvons obtenir de très bons résultats très rapidement et ce, sans favoriser la diversité, qui est essentielle dans d'autres problèmes afin de pouvoir "revenir en arrière" et essayer de chercher d'autres solutions.

L'algorithme génétique réalisé pour résoudre ce problème est classique. Il est de type steady-state : son nombre d'individu est constant, et seulement 2 sont mutés à chaque génération. Voici sa structure :

1. Initialisation d'une population avec un nombre x d'individus de taille y.
2. Évaluation des individus : $1 - \frac{taille\_individu - nombre\_de\_1}{taille\_individu}$
3. Sélection des individus : meilleurs individus, tournois et individus aléatoires
4. Croisement mono-point et génération des enfants à partir des individus sélectionnés
5. Mutation des enfants : opérateurs 1 flips, 3 flips, 5 flips et bit-flip
6. Suppression des individus dans la population : ancienneté, score, utilité [1]
7. Réinsertion des enfants dans la population

1. L'individu qui n'a pas été amélioré depuis le plus long temps est supprimé

Ce rapport portera essentiellement sur l'analyse de la sélection des opérateurs, à savoir :

— Oracle myope

— Fixed wheel

— Adaptive wheel

— Adaptive pursuit

— Upper confidence bound

— Exp3

Le projet a été réalisé en Python. Les fichiers sources sont disponibles à cette adresse :
`https://github.com/KyrillosL/AlgorithmeGenetique`

2 NOTION DE SOLUTION POUR LE PROBLÈME ONE-MAX

Dans la suite de l'analyse, lorsque j'évoquerai la solution optimale, je parlerai en réalité de la suite d'étapes en un certain nombre d'itérations afin d'arriver à la "solution". En effet, la "solution" est connue et unique : un nombre de '1' de la taille de l'individu.
Dans notre cas, la solution idéale sera donc d'arriver à la "solution" en un nombre minimal d'itération, et donc, de temps.
Voici une solution optimale :

1. 00000
2. Application de l'opérateur 5 flips
3. 11111

Voici une solution non optimale

1. 00000
2. Application de l'opérateur 3 flips
3. 11100
4. Application de l'opérateur 1 flip
5. 11110
6. Application de l'opérateur 1 flip
7. 11111

3 Analyse de la sélection des opérateurs

3.1 Méthodologie

Afin d'étudier la sélection d'opérateurs la population étudiée ne sera que de taille 1. Cela signifie que la sélection d'individus, le croisement et la réinsertion seront désactivés.
Bien que les opérateurs agissent de manière aléatoire sur les individus, ils gardent en mémoire quels sont les bits à changer de la sorte :

1. Appliquer la fonction *compute_score* à tous les opérateurs et garder en mémoire quels bits ont étés changés pour cette itération
2. Appliquer le meilleur opérateur aux bits temporaires mémorisés de l'individu (*mutate*)
3. Réinitialiser la mémoire des bits des autres opérateurs

Ceci nous permet de garder l'aspect aléatoire de nos opérateurs tout en prévoyant le résultat d'une mutation. Cette fonctionnalité sera très utile pour notre opérateur Oracle Myope.

De plus, de la même manière, grâce à la fonction *compute_score* il est possible de n'appliquer un opérateur que s'il ne dégrade pas solution finale.

Note : les résultats peuvent être long en temps du fait de la mise à jour des graphiques en temps réel.

Les résultats seront données sous forme de graphiques.

— La première figure (haut gauche) correspond au score en fonction de la génération.
— La seconde figure (haut droite) correspond à l'ensemble des probabilités des opérateurs.
— Les figures 2,3 et 4 (milieu) correspondent aux probabilités d'un opérateur en fonction de la génération. Pour UCB, il s'agira des rewards.
— La dernière figure (bas) indique quel opérateur a été utilisé pour une génération.

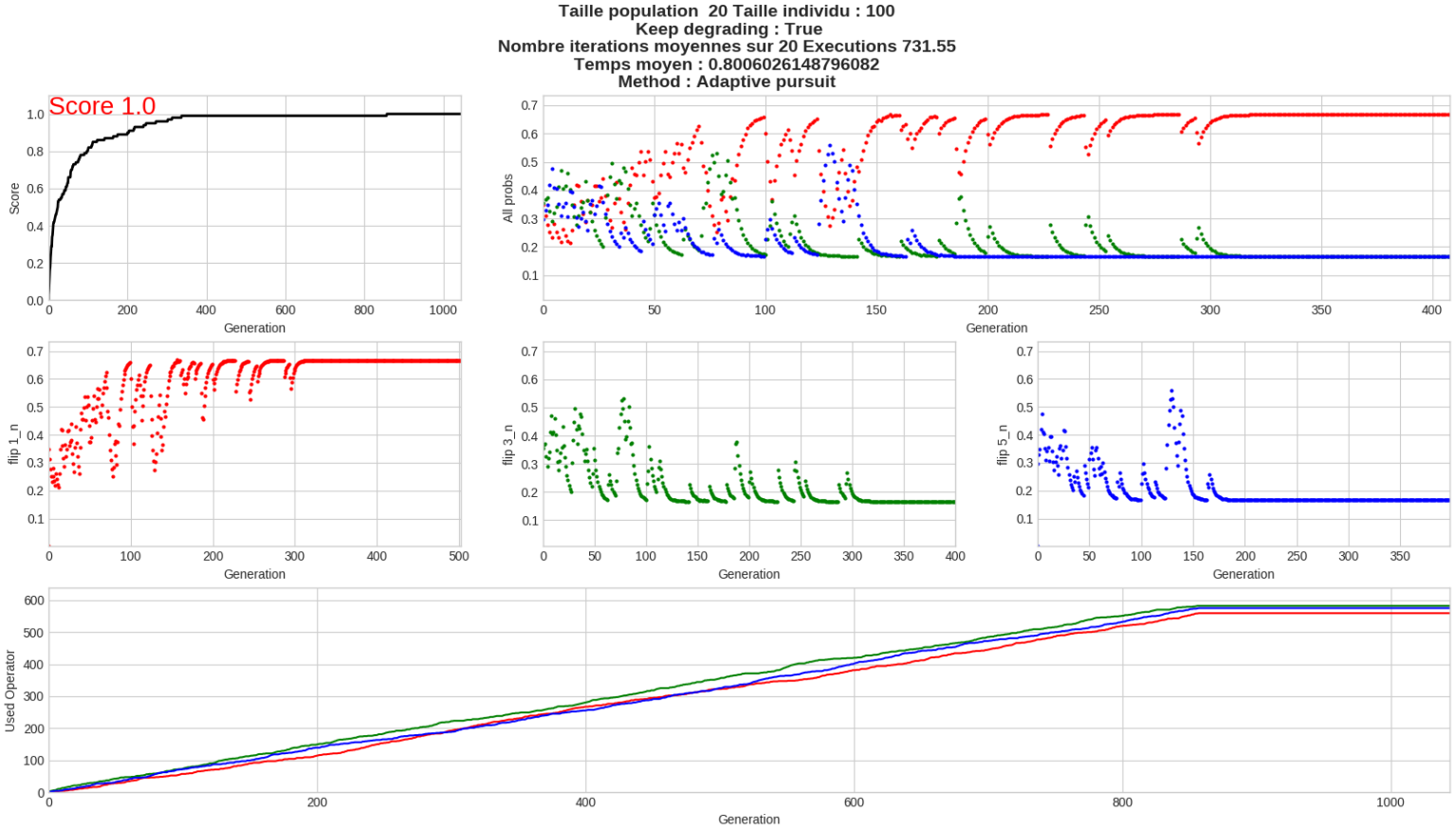

FIGURE 3.1 – Exemple de rendu contenant le score, les probabiltiés d'avoir un opérateur ainsi que leur utilisation en fonction de la génération

3.2 PAS DE SÉLECTION : OPÉRATEUR 1 FLIP

Afin de contrôler l'impact de la sélection des opérateurs une phase de test a été réalisée avec l'opérateur 1 flip. Il est disponible dans le tableau récapitulatif

3.3 RÉCAPITULATIF

Voici deux graphiques représentant le score en fonction du sélecteur d'opérateur. L'un est moyenné sur 2 exécutions et l'autre sur 15. (Les images sont disponibles dans le repo git). Sur 2 exécutions il semble que l'oracle ainsi qu'ucb soient les sélecteurs d'opérateur les plus efficaces. L'adaptive pursuit, bien que peu performante au début semble également faire mieux que l'aléatoire.

Cependant, sur 15 exécutions, nous remarquons que les différences entre les opérateurs diminuent. En n'utilisant que l'opérateur 1flip on remarque qu'il est lent, mais rattrape vite les autres, pour les dépasser. Il obtient un score presque équivalent à l'oracle en terme d'itérations. Notre but en réglant les algorithmes et d'arriver à compenser la lenteur du 1flip au début en le remplaçant par le 5flip pour ensuite repasser au 1flip au bon moment.

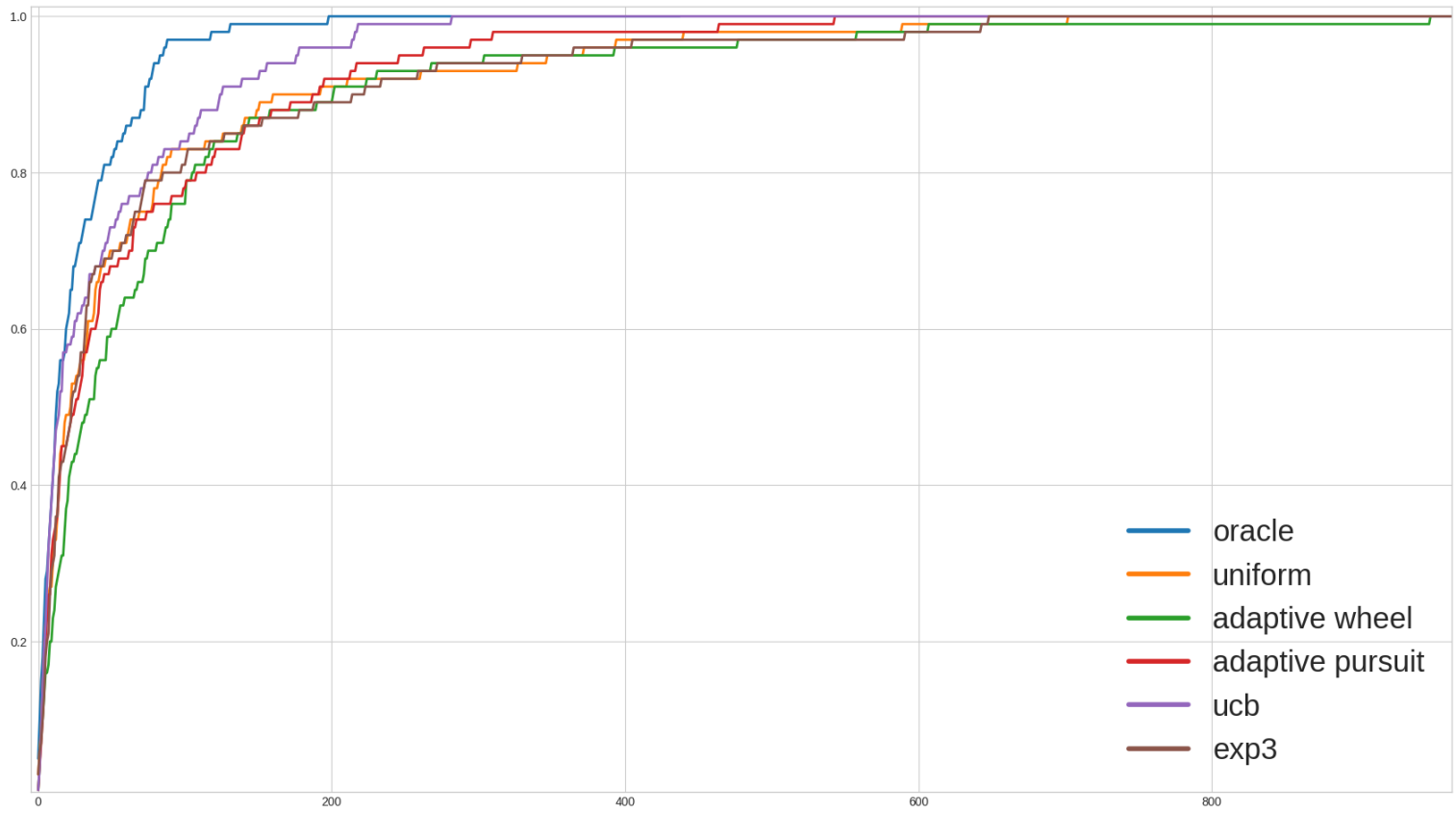

FIGURE 3.2 – Récapitulatif des scores obtenus pour 2 exécutions

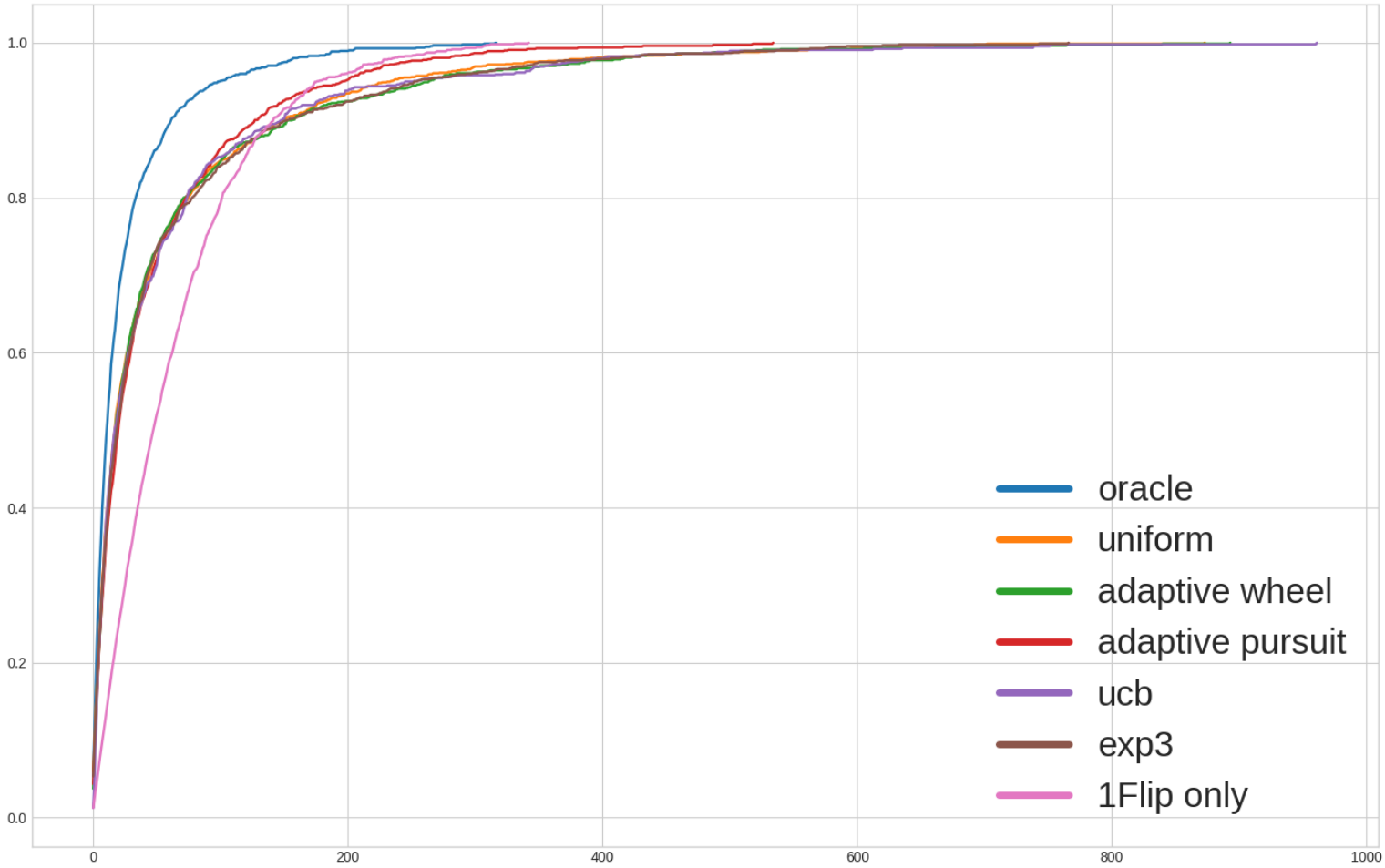

FIGURE 3.3 – Récapitulatif des scores obtenus pour 15 exécutions

3.4 ORACLE MYOPE

3.4.1 Présentation

L'oracle Myope est le sélecteur d'opérateurs qui nous servira de référence pour la suite. Il est censé donner une solution proche de l'optimale. L'oracle regarde/calcule la meilleure mutation (i.e. le meilleur opérateur à appliquer) pour la prochaine itération et l'applique. Bien que proche de l'optimal, il ne l'est pas car il ne prévoit que l'itération suivante, d'où son nom "Myope". En effet, si nous avions eu un opérateur 4-flips, ce cas aurait pu se présenter :

Solution 1 : 4 étapes

1. 00000 00000 00
2. Appliquer 2* l'opérateur 5 flips
3. 11111 11111 00
4. Appliquer 2* l'opérateur 1 flip
5. 11111 11111 11

Solution 2 : 3 étapes

1. 00000 00000 00
2. Appliquer 3* l'opérateur 4 flips
3. 11111 11111 11

3.4.2 Résultats

Mutations dégradantes autorisées | 1 individu de taille 100

Nous pouvons constater que le score progresse rapidement pour arriver à 0.75, au bout d'environ 50 itérations. Ceci est en corrélation avec l'utilisation de l'opérateur 5 flips qui progresse très vite dès le début. Il est ensuite remplacé par l'opérateur 1 flip.

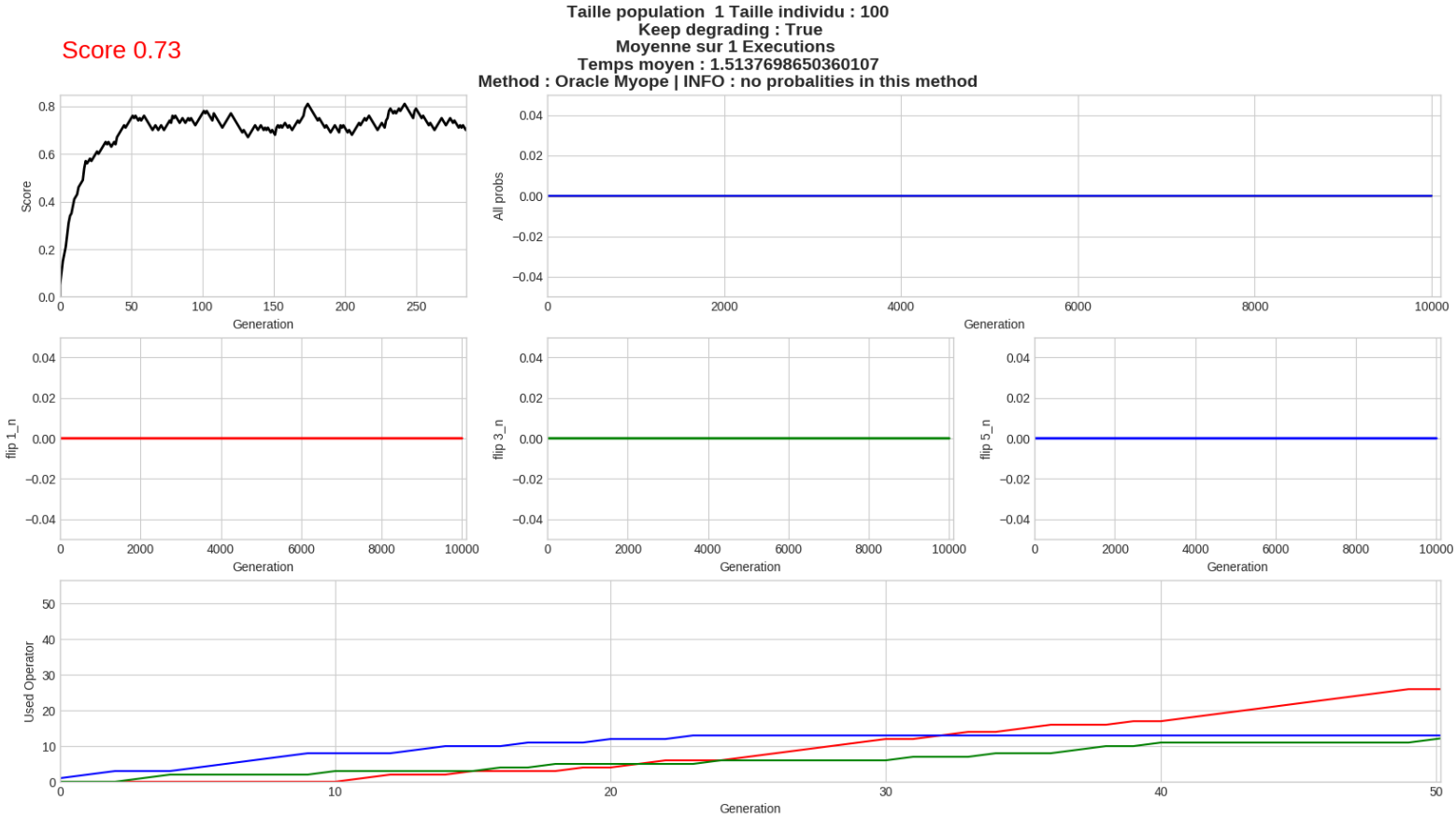

FIGURE 3.4 – En bleu : le nombre d'utilisation de l'opérateur 5 flips qui est remplacé par le 1 flip après 30 générations.

Du fait que l'on garde les mutations dégradantes, l'algorithme ne peut pas terminer et stagne à un score autour de 0.75, ce qui est normal. Note : les mêmes résultats sur le choix

des opérateurs ont étés obtenus pour un individu de taille 1000.

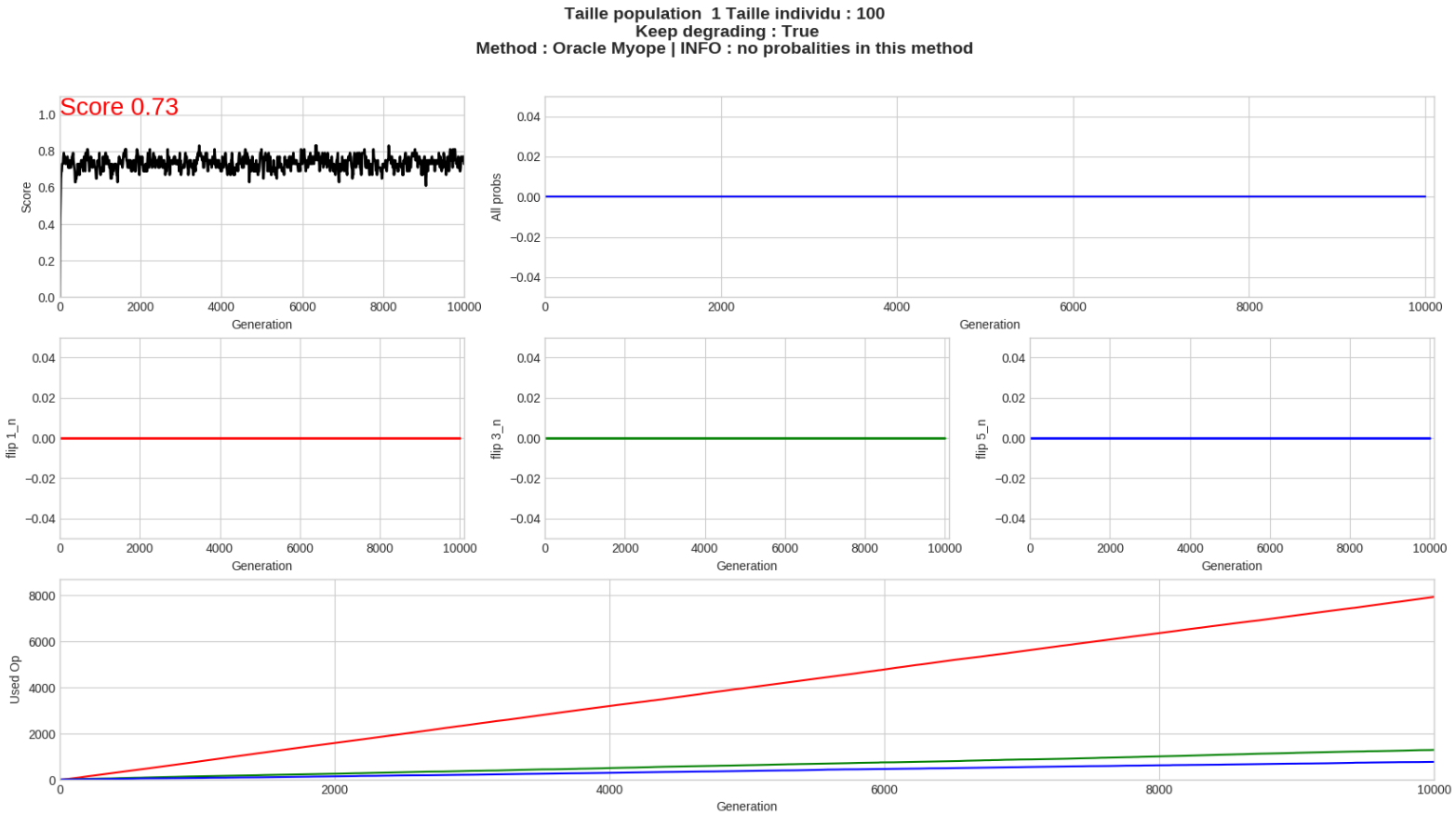

FIGURE 3.5 – L'algorithme ne peut pas terminer et stagne autour de 0.75

Pas de mutations dégradantes | 1 individu de taille 100 | Moyenne sur 20 exécutions

En refusant les mutations dégradantes nous arrivons à une solution proche de l'optimale sur le choix des opérateurs. Il faut environ 275 itérations pour arriver à la solution. Nous pouvons voir que l'opérateur 5 flips est utilisé en premier pendant 27 générations, puis l'opérateur 3 flip prend la relève pendant 14 générations. Enfin, l'opérateur 1 flip est utilisé jusqu'à la fin.

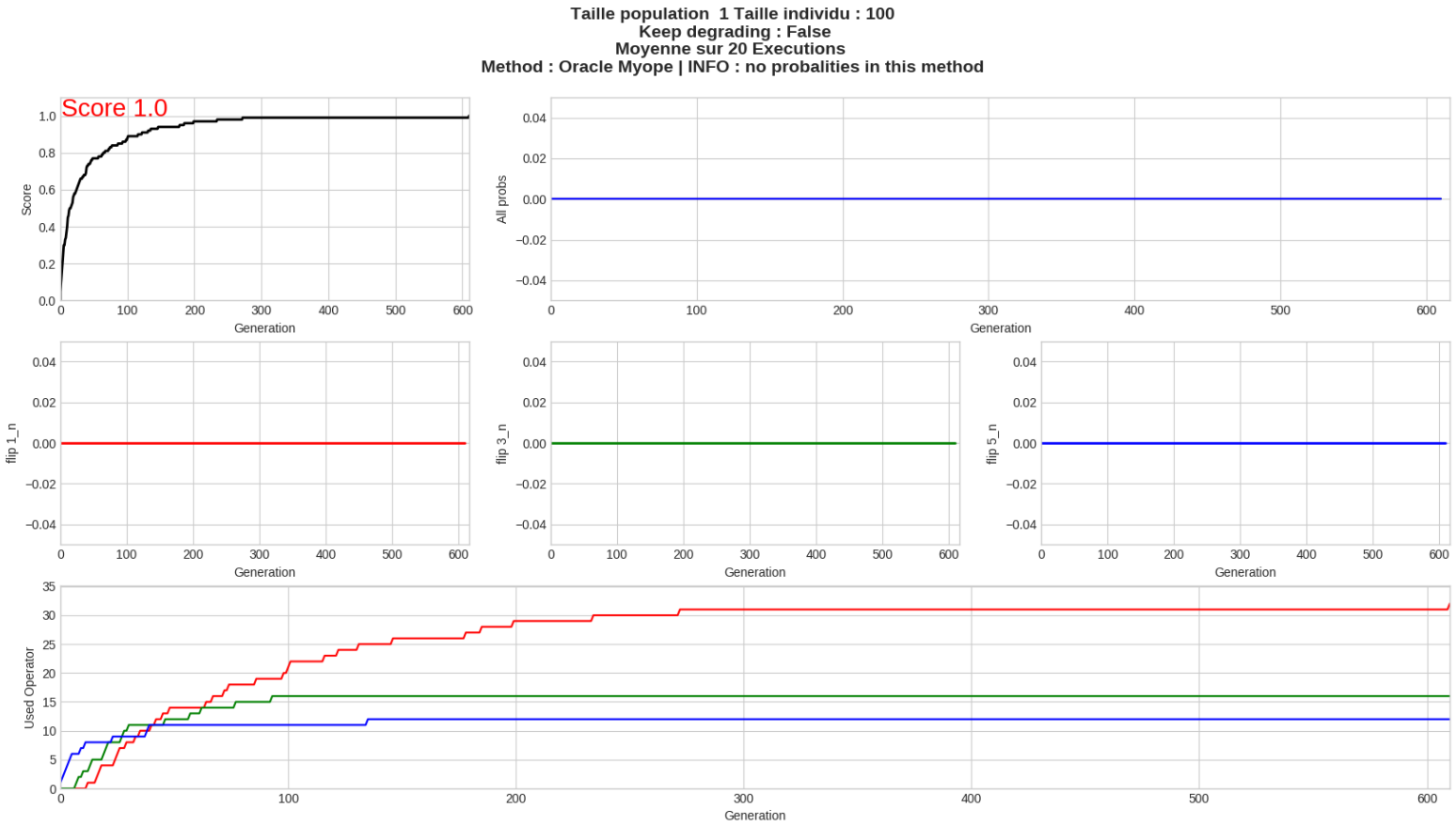

FIGURE 3.6 – Moyenne de l'oracle sur 20 exécutions. Le score est obtenu en environ 275 itérations

Pas de mutations dégradantes | 1 individu de taille 1000 | Moyenne sur 20 exécutions

Pour un individu de taille 1000, il faut un peu plus de 7000 itérations et un peu moins de 2 secondes.

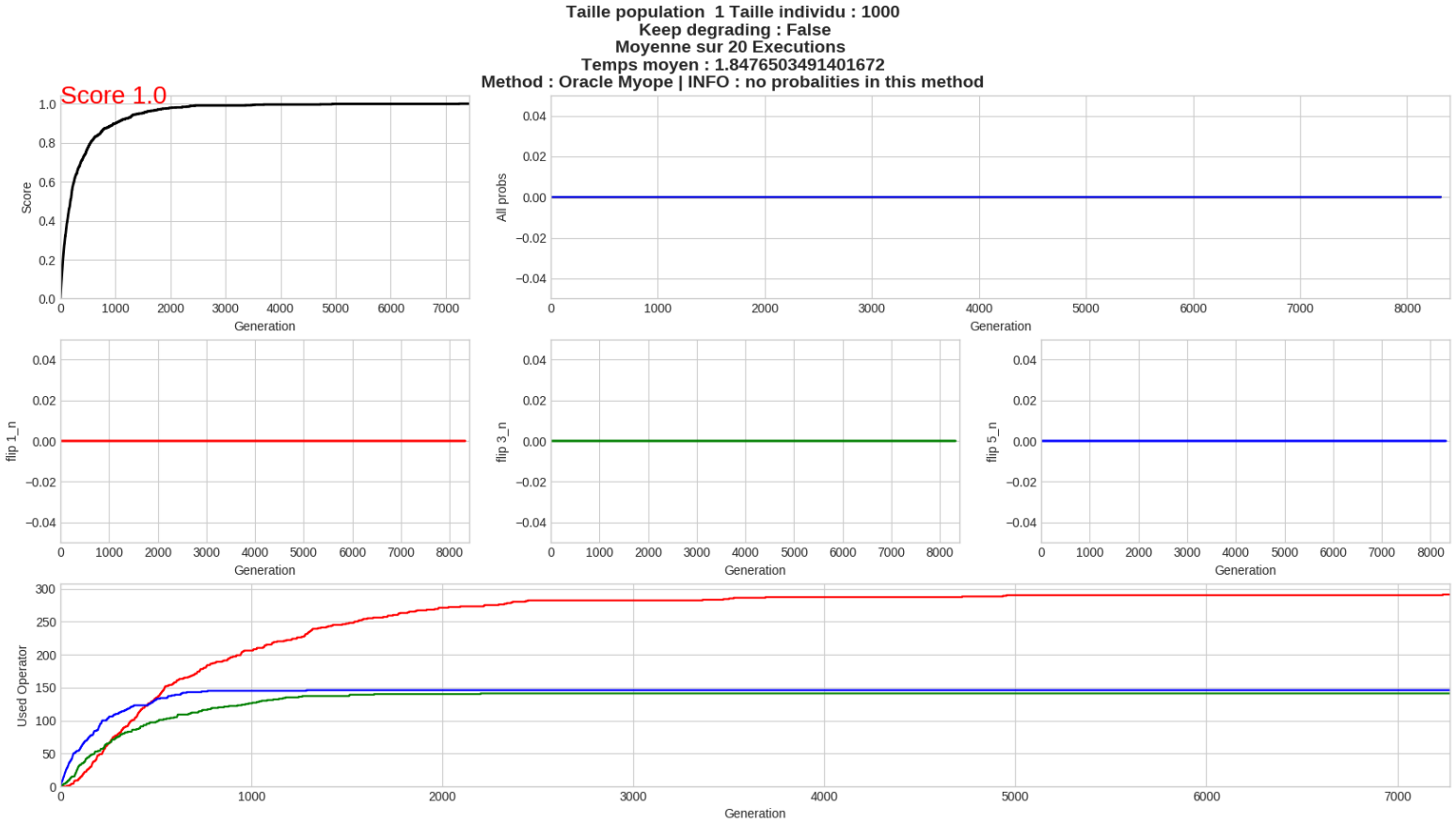

FIGURE 3.7 – Pour un individu de taille 1000, il faut un peu plus de 7000 itérations et un peu moins de 2 secondes.

Impact et rôle d'une population

Nous testerons ici le rôle d'une population de taille 20. Il n'y a pas de croisement et nous retenons les deux meilleurs individus pour les muter. Les autres paramètres liés à la population seront étudiés plus tard.

Mutations dégradantes | 20 individus de taille 1000 | Moyenne sur 20 exécutions

Grâce à la population nous réduisons le nombre d'itérations nécessaires à 3000 tandis que le temps moyen a augmenté de 0.5 seconde.

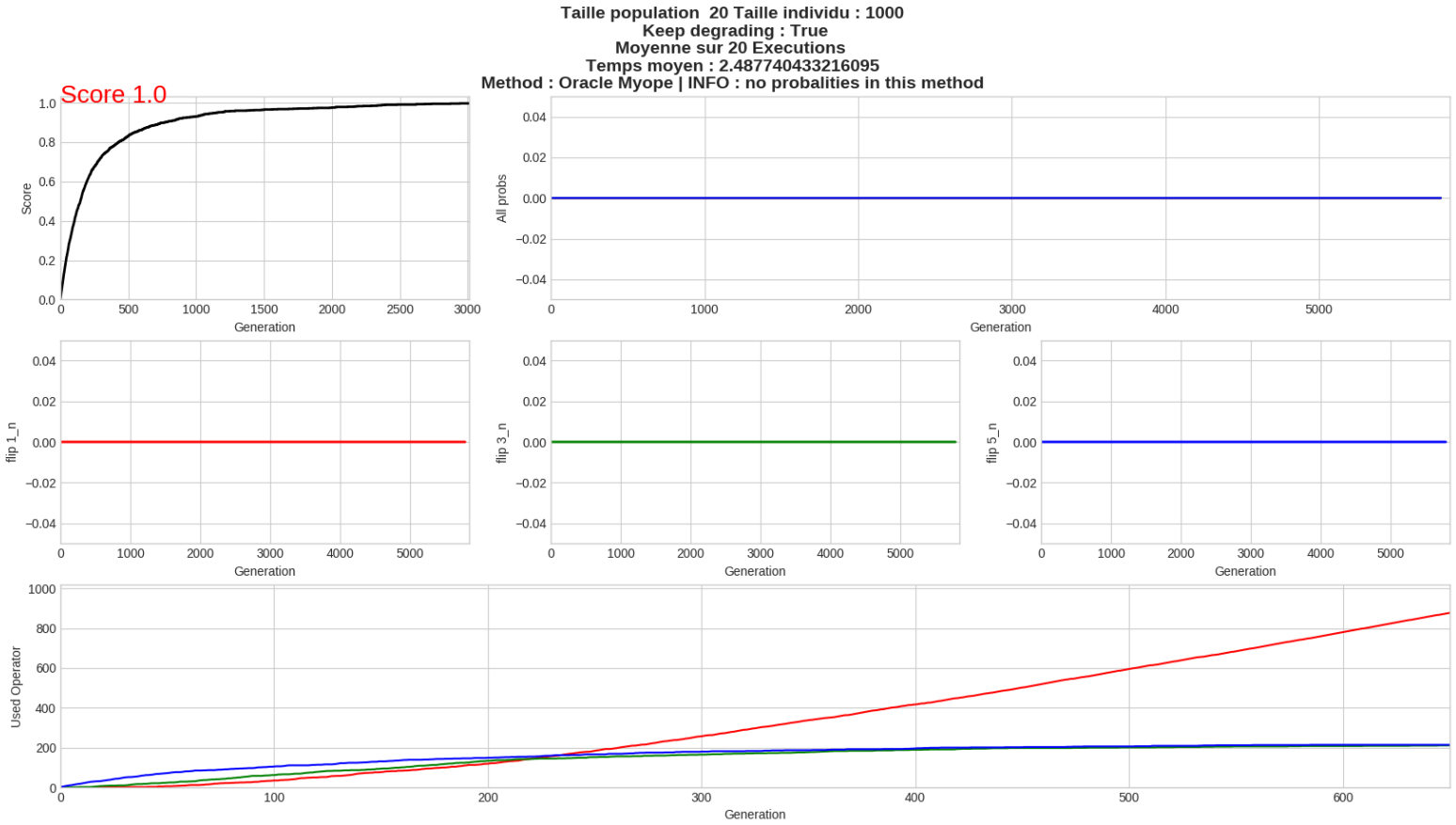

FIGURE 3.8 – Le fait d'utiliser une population divise le nombre d'itérations nécessaires par 2 en augmentant légèrement le temps

3.4.3 Conclusion

L'oracle Myope est un très bon indicateur sur l'efficacité des opérateurs en fonction des génération. Il nous servira de base pour la suite. Nous avons également montré qu'utiliser une population -sans croisement- a un très fort impact sur la vitesse de résolution du problème. Nous passons de 7000 itérations à 3000 pour des individus de taille 1000.

De plus, l'utilisation d'une population nous permet de ne pas courir dans un minimum local comme le fait notre oracle actuellement.

Cependant, pour une population plus large avec des individus plus grand, et suivant le nombre d'opérateurs cela ne serait pas viable, d'où le fait d'utiliser des sélecteurs d'opérateurs moins coûteux comme nous allons le voir dans les parties suivantes. Pour comparer l'efficacité de l'oracle

À noter que j'ai des réserves sur le temps : il me semble que la gestion des graphiques prennent plus de temps que la résolution du problème en elle-même. Il serait intéressant de voir le temps réel, mais cela n'est pas possible actuellement avec l'architecture du code et la gestion des résultats.

3.5 ROULETTE FIXE

3.5.1 Présentation

La roulette fixe est un opérateur de sélection très simple qui choisi un opérateur suivant une probabilité donnée par l'utilisateur et fixe.
Pour ce problème j'ai simplement laissé les probabilités à : $\frac{1}{nombreoperateurs}$

3.5.2 Conclusion

La roulette fixe est moins efficace que notre oracle (18000 contre 7000 itérations pour un individu), cependant elle favorise la diversité. Voir les résultats ci dessous.

3.5.3 Résultats

Mutations dégradantes | 1 individu de taille 1000 | Moyenne sur 20 exécutions

La première chose que nous pouvons remarquer est que le score stagne cette fois ci autour de 0.5 et non plus 0.75 pour l'oracle. Cela prouve que l'oracle est plus efficace. De plus, la linéarité du choix des opérateurs montre l'uniformité dans leur utilisation.

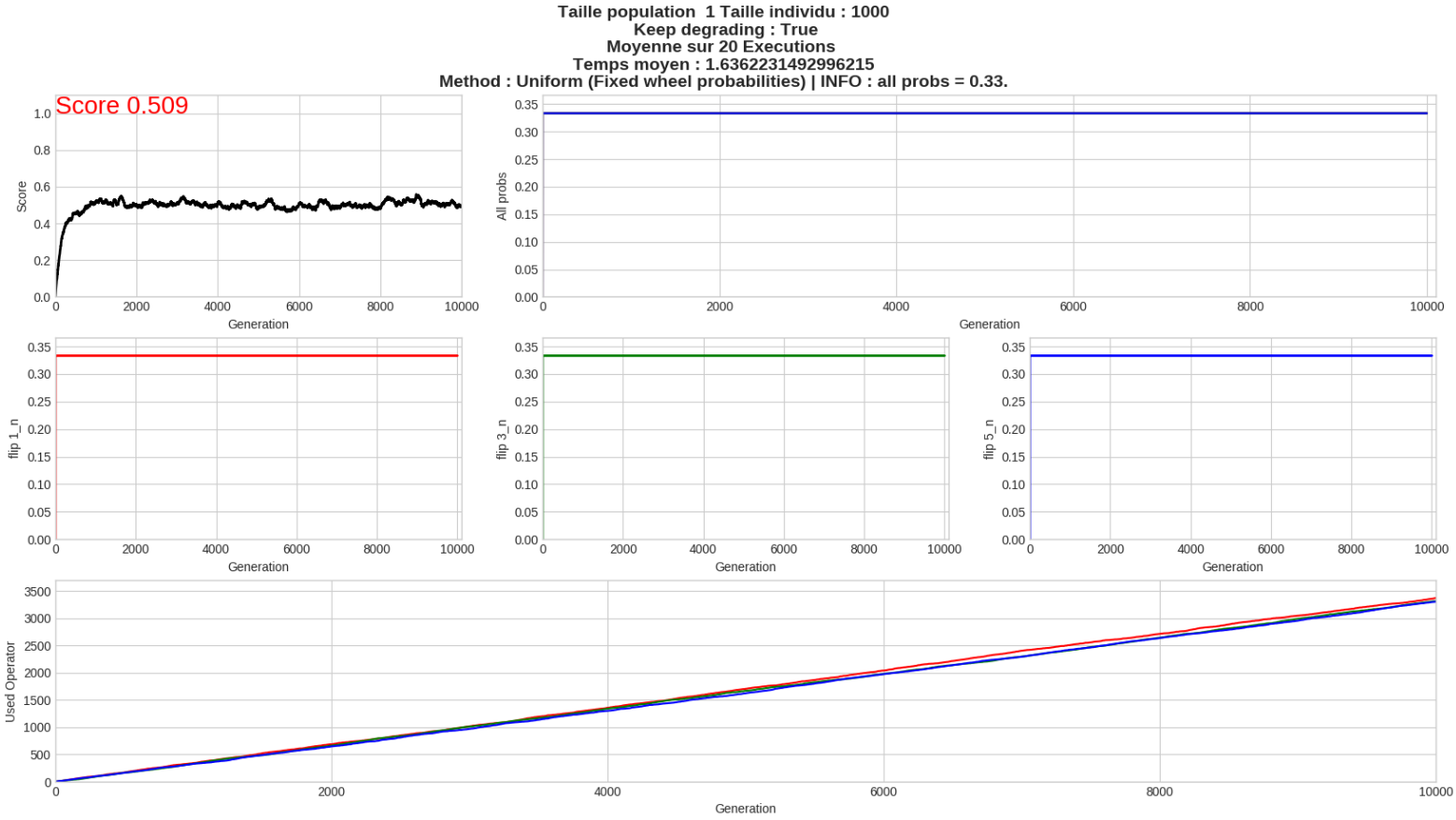

FIGURE 3.9 – En gardant les mutations dégradantes et sans population, le score de la roulette fixe est inférieur à celui de l'oracle

Pas de mutations dégradantes | 1 individu de taille 1000 | Moyenne sur 20 exécutions

En refusant les mutations dégradantes nous remarquons que la roulette fixe est beaucoup moins efficace que l'oracle : il lui faut environ 18000 itérations pour trouver une solution.

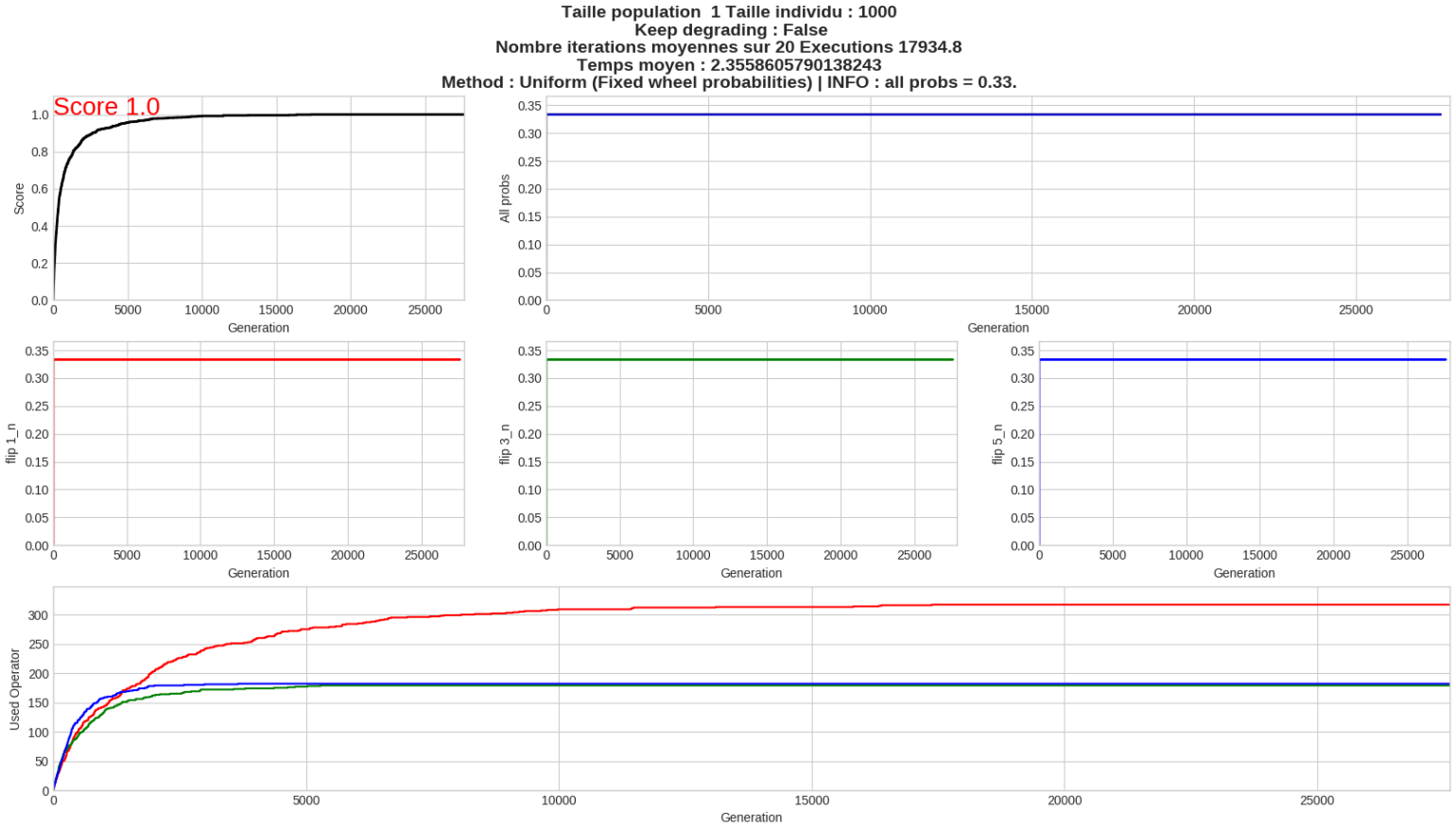

FIGURE 3.10

Impact et rôle d'une population

Pas de mutations dégradantes | 20 individu de taille 1000 | Moyenne sur 20 exécutions

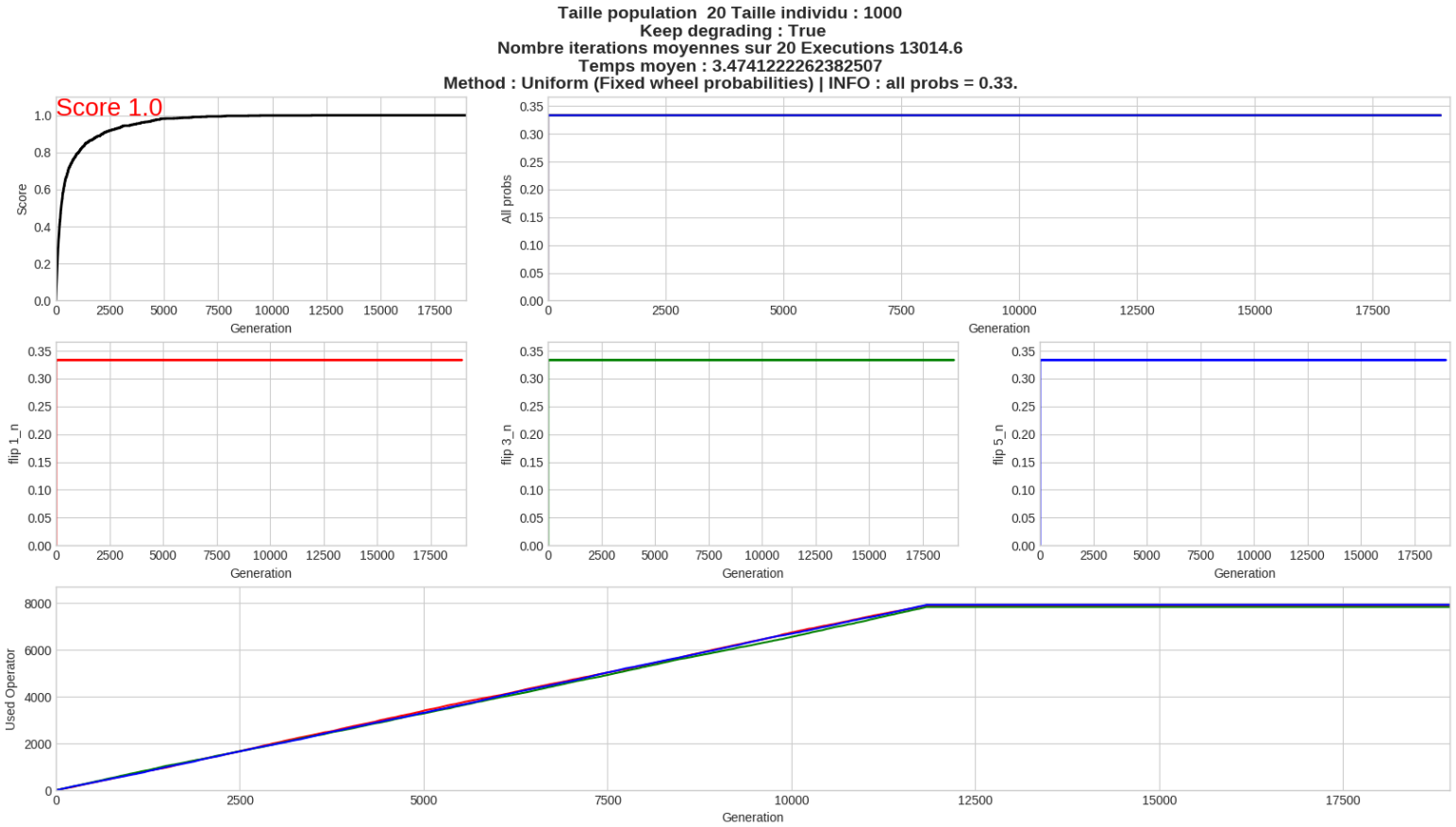

FIGURE 3.11 – L'utilisation d'une population réduit le nombre d'itérations à 13000, tout en augmentant le temps à 3.4 secondes.

3.6 ADAPTIVE WHEEL

3.6.1 Présentation

La roulette adaptive permet de faire varier la probabilité de l'utilisation d'un opérateur en fonction du temps. Elle est calculé grâce à la formule :

$$probabiliteoperateur = pmin + \left(1 - nombreOperateurs * pmin\right) * \frac{scoreoperateur}{sommescoreoperateurs}$$

Un tableau contient les probabilités de chaque opérateurs et permet de les déclencher. Pmin est la borne inférieure. Grâce à ce paramètre, tous les opérateurs ont une probabilité minimum de 20% d'être utilisé à chaque génération. Pmin favorise également la diversité de leur déclenchement en mettant à jour le score d'un opérateur s'il est utilisé. Une fois mis à jour, et s'il a obtenu un meilleur score, l'opérateur aura donc plus de chance d'être utilisé.

J'ai fixé pmin à 0.2. Note post-tests : en modifiant l'algorithme j'ai obtenu une courbe de probabilité proche de celle du score pour chaque opérateur.

3.6.2 Résultats

Mutations dégradantes | 1 individu de taille 100 | Moyenne sur 20 exécutions

Sur un individu le taille 100 la roulette adaptive semble fonctionner. En effet, on retrouve l'opérateur 5n en premier, puis le 1n prend le relais. Cependant, les probabilités semblent stagner autour de 1/3 à partir de 30 générations, en laissant l'opérateur 1n légèrement devant.

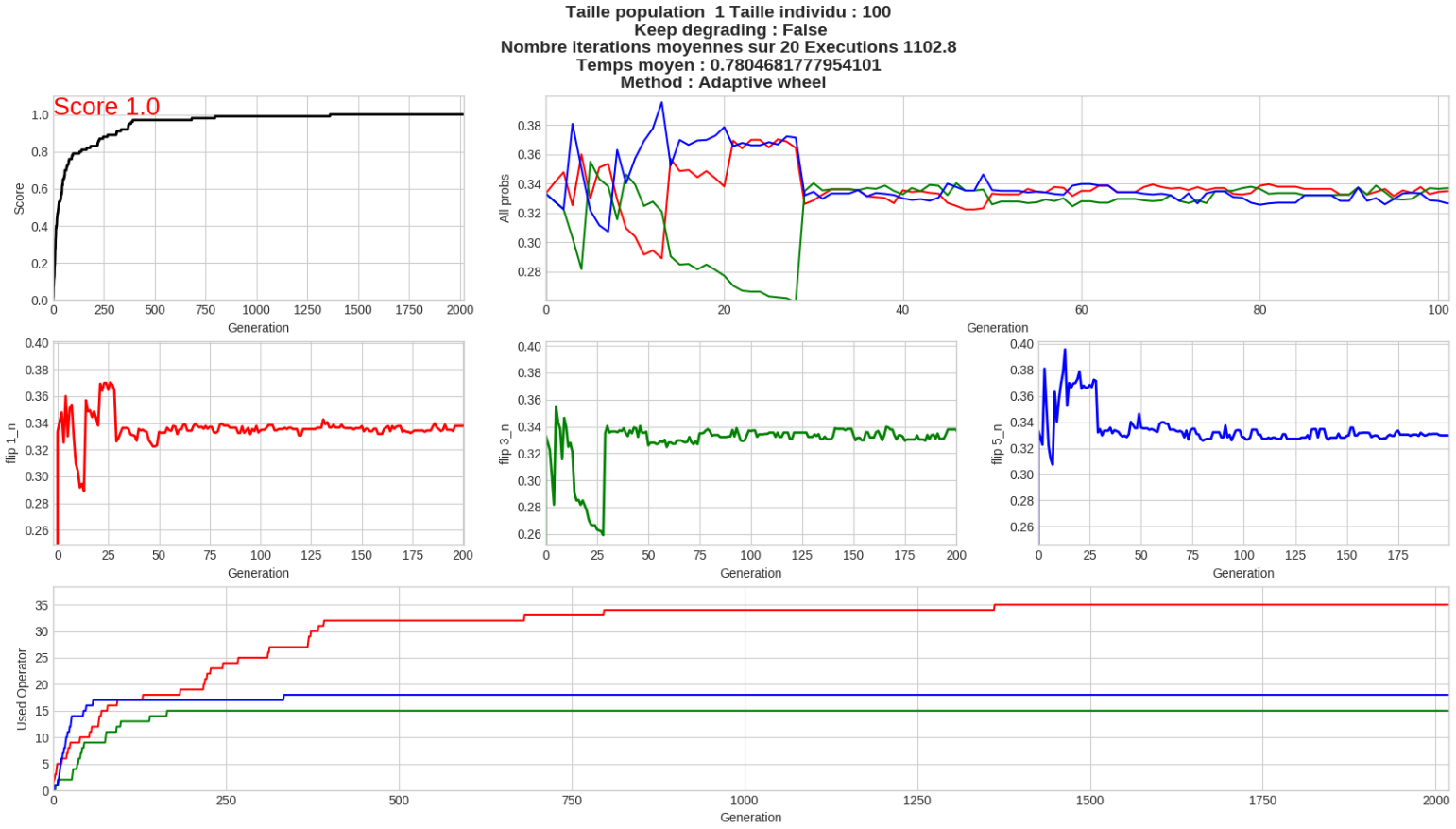

FIGURE 3.12

Rôle de l'adaptive wheel

Comme nous pouvons le voir sur cette image, l'adaptive wheel permet de relancer un opérateur et voir s'il est plus efficace afin d'augmenter sa probabilité. Ici le 1 flip reste devant car il reste le plus efficace.

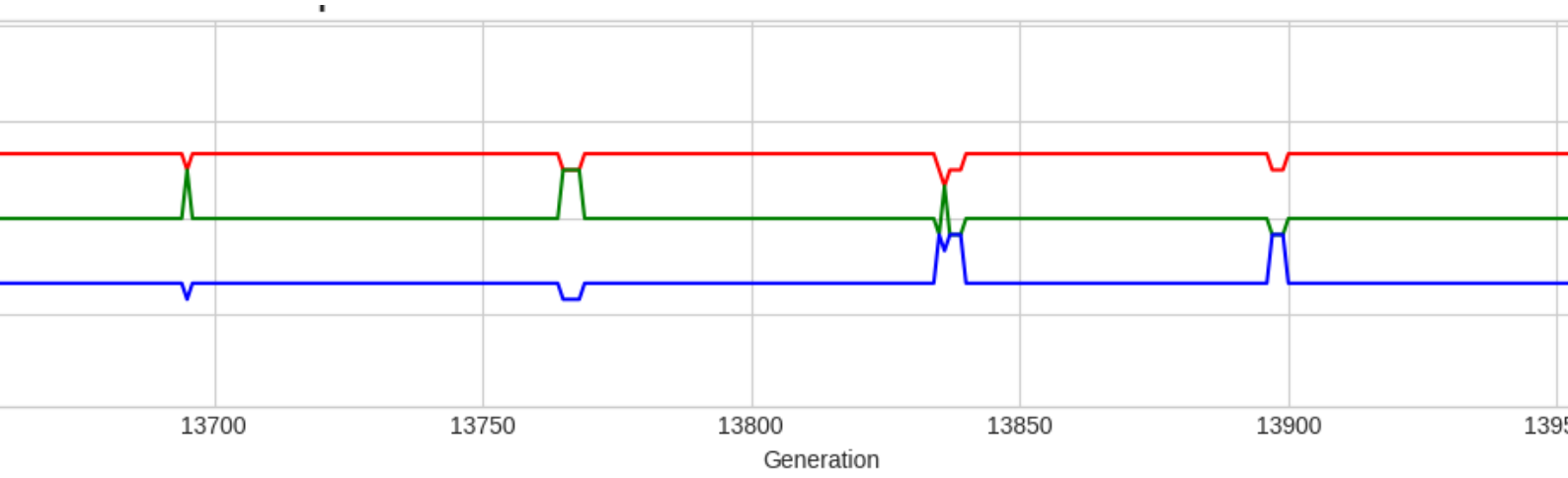

FIGURE 3.13 – L'adaptive wheel permet de remettre à jour les scores des opérateurs

3.6.3 Conclusion

Bien que sa gestion des opérateurs semblent correct, la roulette adaptive ne semble pas apporter un réel intérêt en terme de temps de résolution pour ce problème. Il serait intéressant de l'essayer avec d'autres paramètres tels que des individus plus grands et des opérateurs plus variés.

3.7 Adaptive Pursuit

3.7.1 Présentation

En terme de fonctionnement, l'adaptive pursuit est semblable à l'adaptive wheel. En plus de la borne inférieure, cet algorithme permet de définir une borne supérieure ainsi qu'un paramètre d'exploration. Comme pour l'adaptive wheel, les bornes supérieures et inférieures ont étés définies à 0.2.

Formules python

Meilleur opérateur :

best_operator.probability += self.beta * (self.pmax - best_operator.probability)

Autres opérateurs :

op.probability += self.beta * (self.pmin - op.probability)

Les résultats suivants ont été réalisé sur 20 exécutions et sur une population de 20 individus sans croisement.

Très grande exploration (b=0.1) | 20 individus de taille 100 | Moyenne sur 20 exécutions

Les courbes de probabilité de chaque opérateurs permettent de voir qu'avec une grande exploration l'algorithme contrôle régulièrement si un meilleur choix est possible. Le nombre d'itérations est de 475.

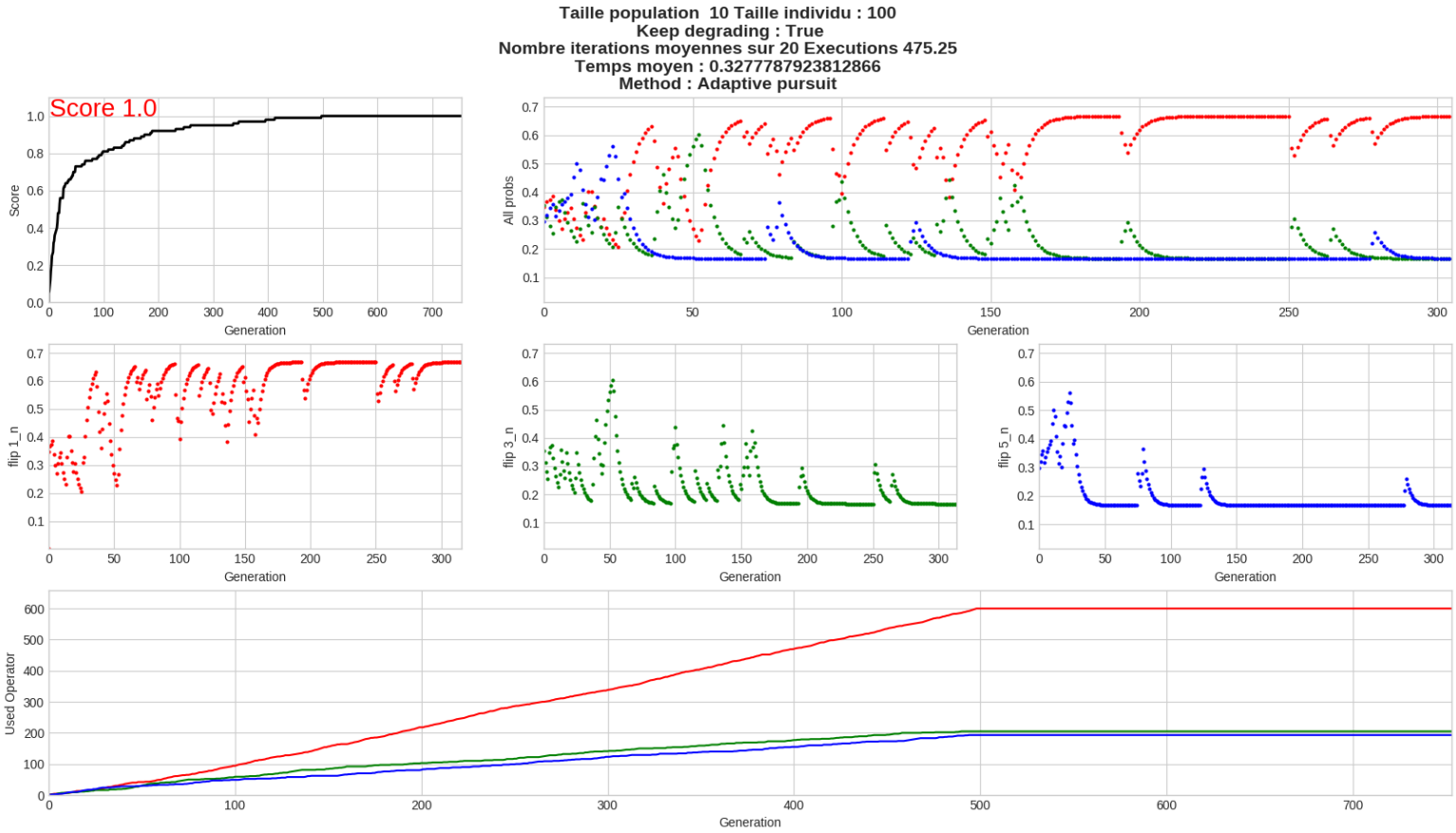

FIGURE 3.14 – Adaptive Pursuit avec une très grande exploration

Grande exploration (b=0.2) | 20 individus de taille 100 | Moyenne sur 20 exécutions

Les courbes de probabilité de chaque opérateurs permettent de voir qu'avec une grande exploration l'algorithme contrôle régulièrement si un meilleur choix est possible. Le nombre d'itérations est de 441.

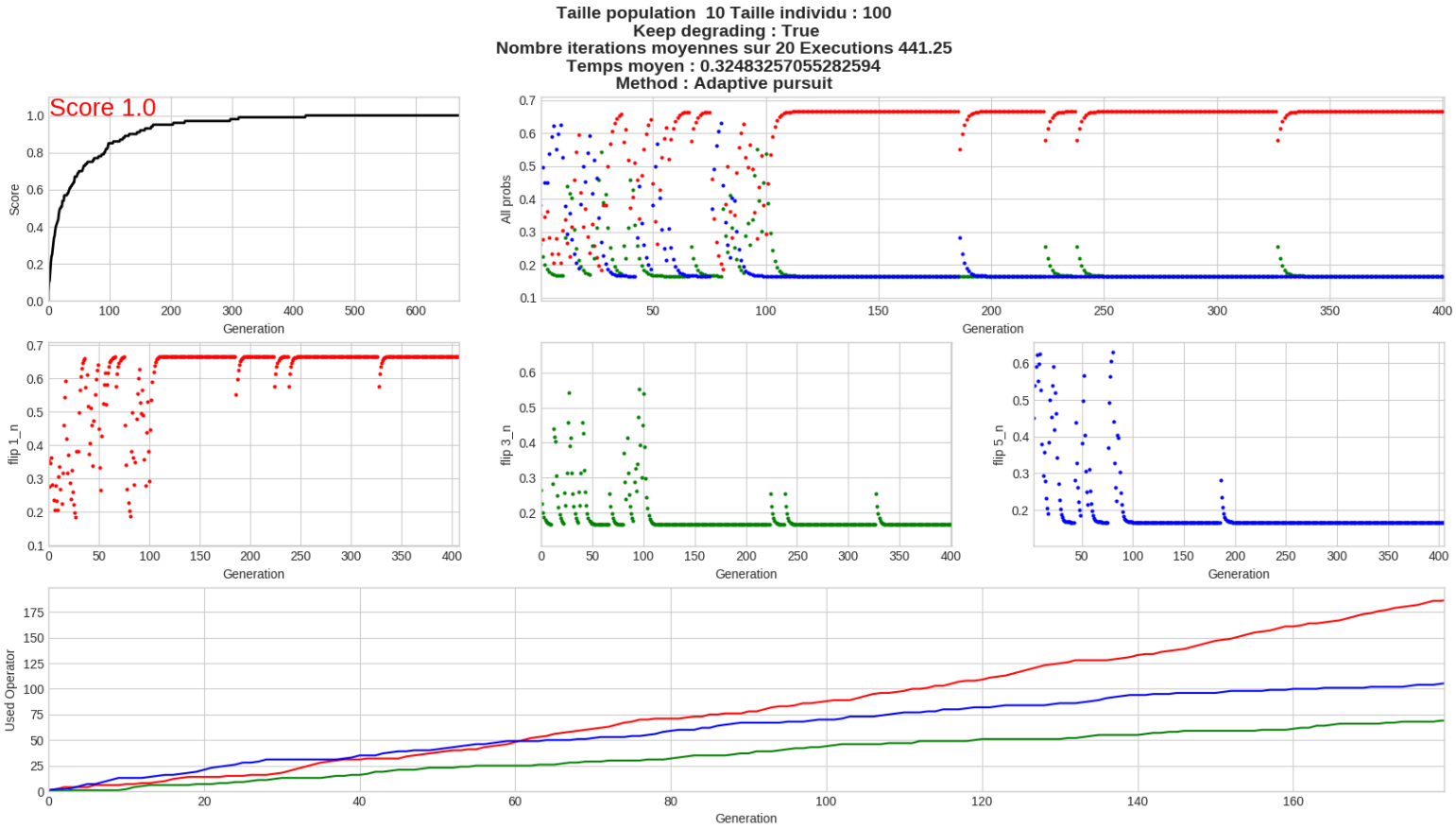

FIGURE 3.15 – Adaptive Pursuit avec une grande exploration

Faible exploration (b=0.9) | 20 individus de taille 100 | Moyenne sur 20 exécutions

Avec une faible exploration, l'algorithme fourni toujours une sélection en favorisant le 5n en premier puis le 3n et enfin le 1n. Cependant elle est de manière "On/Off". Sur 20 exécutions, c'est le choix d'exploration qui a donné les meilleurs résultats, bien que la différence soit faible avec une grande exploration. 20 exécutions ne sont pas assez pour en conclure quoi que ce soit, bien qu'il y ait une tendance à avoir de meilleurs résultats plus l'exploration est faible (475, 441 puis 435).

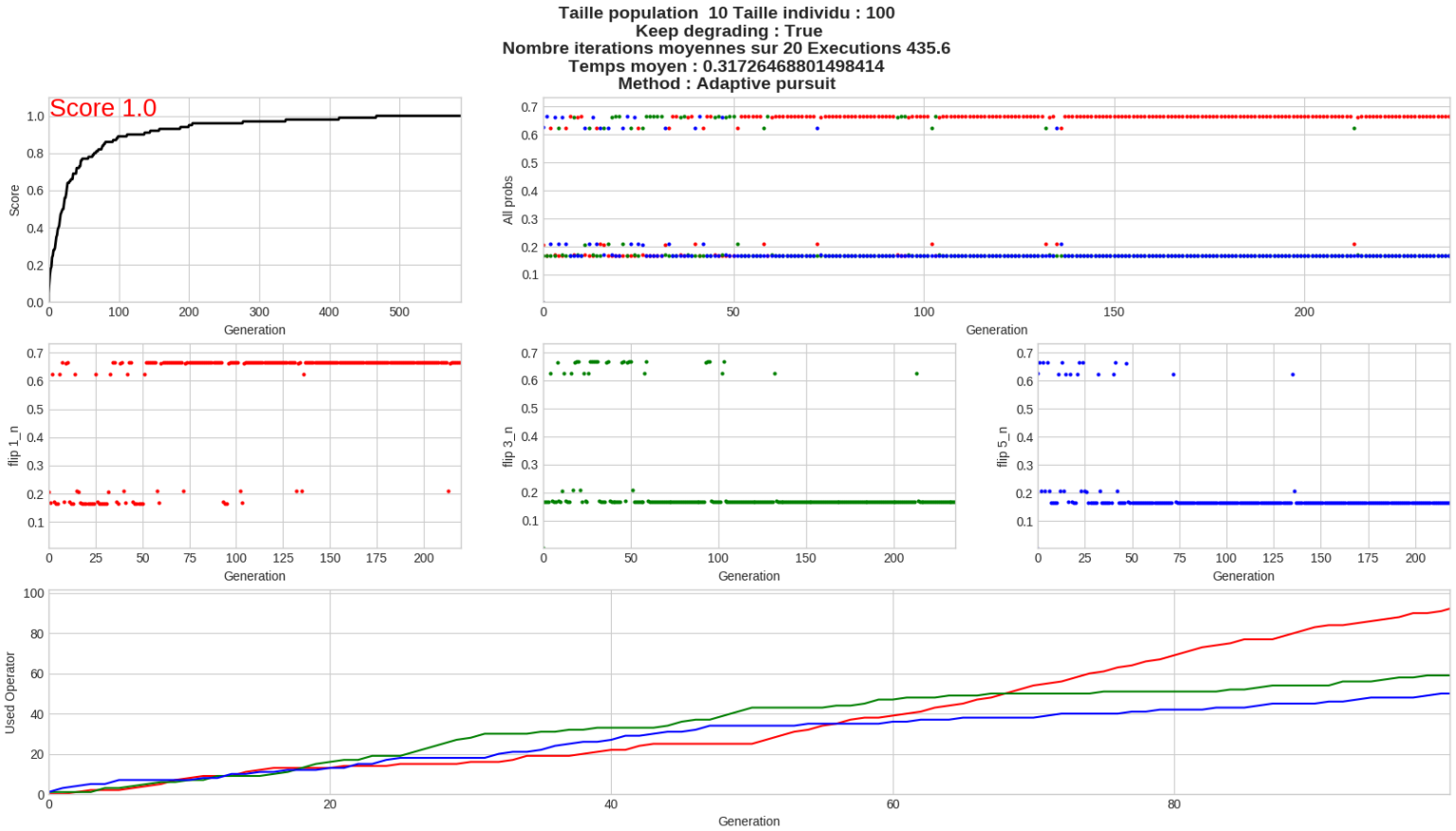

FIGURE 3.16 – Adaptive Pursuit avec une faible exploration

3.7.2 Conclusion

Cet algorithme fourni de bons résultats, qu'une exploration faible favoriserait tout en fournissant une bonne diversification.

3.8 Upper Confidence Bound

3.8.1 Présentation

L'algorithme UCB garde en mémoire la récompense moyenne des opérateurs, puis, à chaque itération calcule la récompense de chaque opérateur suivant une formule prenant en compte l'exploration. Si un opérateur est utilisé, sa récompense moyenne est mise à jour.

Ceci permet de ne plus utiliser un opérateur si sa récompense moyenne devient trop basse et permet de favoriser un autre. Les opérateurs ne sont plus déclenchés suivant une probabilité. En effet, c'est l'opérateur ayant obtenu le meilleur score ucb qui est automatiquement appliqué. UCB est très intéressant, cependant j'ai du réduire le nombre de valeurs moyennes à garder afin d'accélérer la recherche d'un nouvel opérateur plus efficace. En effet, un opérateur jugé efficace au début peut être très lent à redescendre et de ce fait obtenir un score inférieur à un nouvel opérateur. C'est par exemple le cas du 5n qui avait une moyenne très haute au début, et de ce fait sa moyenne était biaisée. En ne gardant que $taillemoyenne = \frac{tailleIndividu}{10}$, UCB s'adapte plus rapidement.

UCB | 20 individus de taille 100 | Pas de fenêtrage

Sans fenêtrage, les courbes d'ucb sont lisses et longues à être mise à jour, même avec une grande exploration.

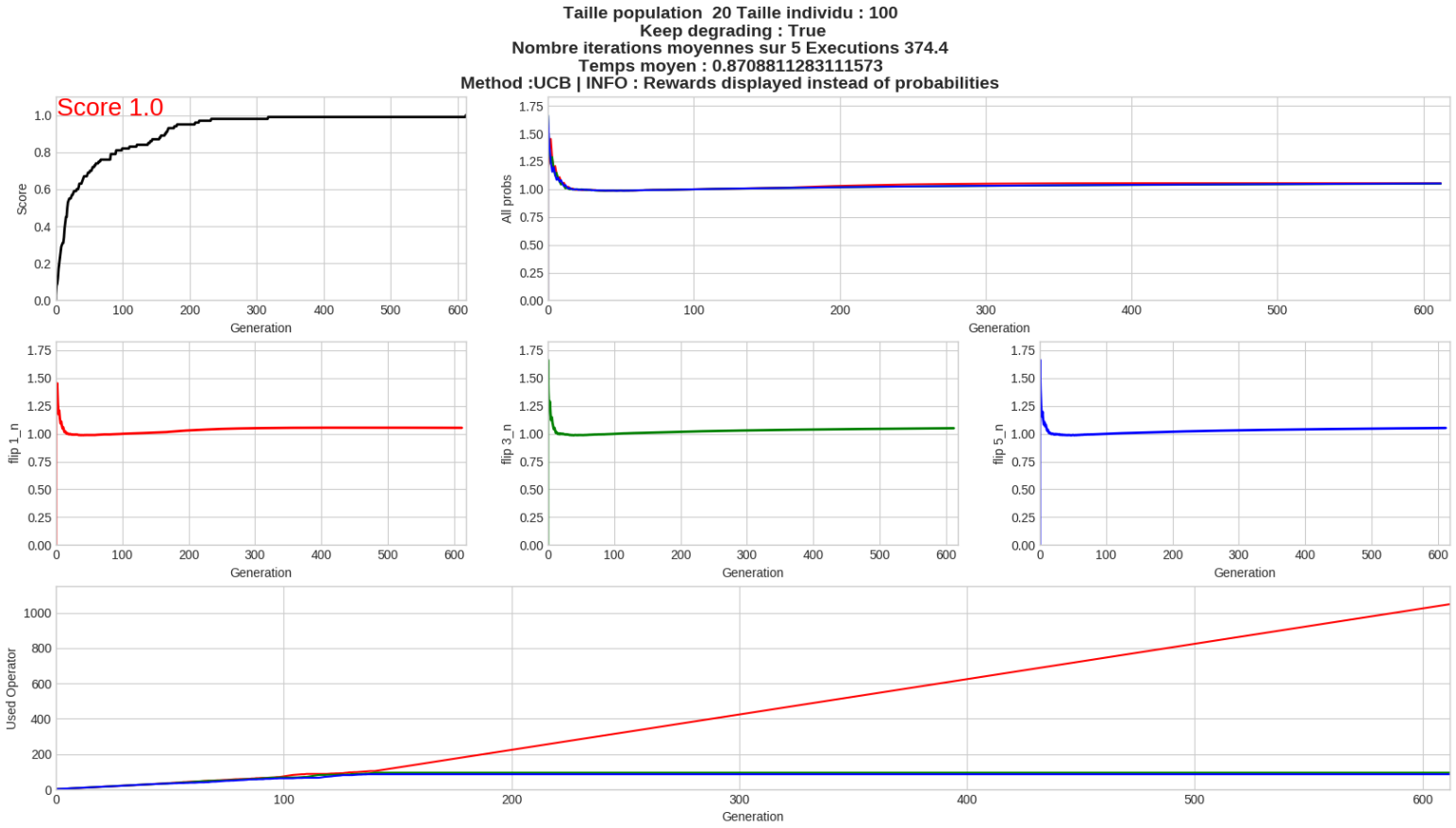

FIGURE 3.17

UCB | 20 individus de taille 100 | Fenêtrage de la moyenne sur 5 valeurs

Les petits à-coups permettent de re-contrôler l'efficacité d'un opérateur.

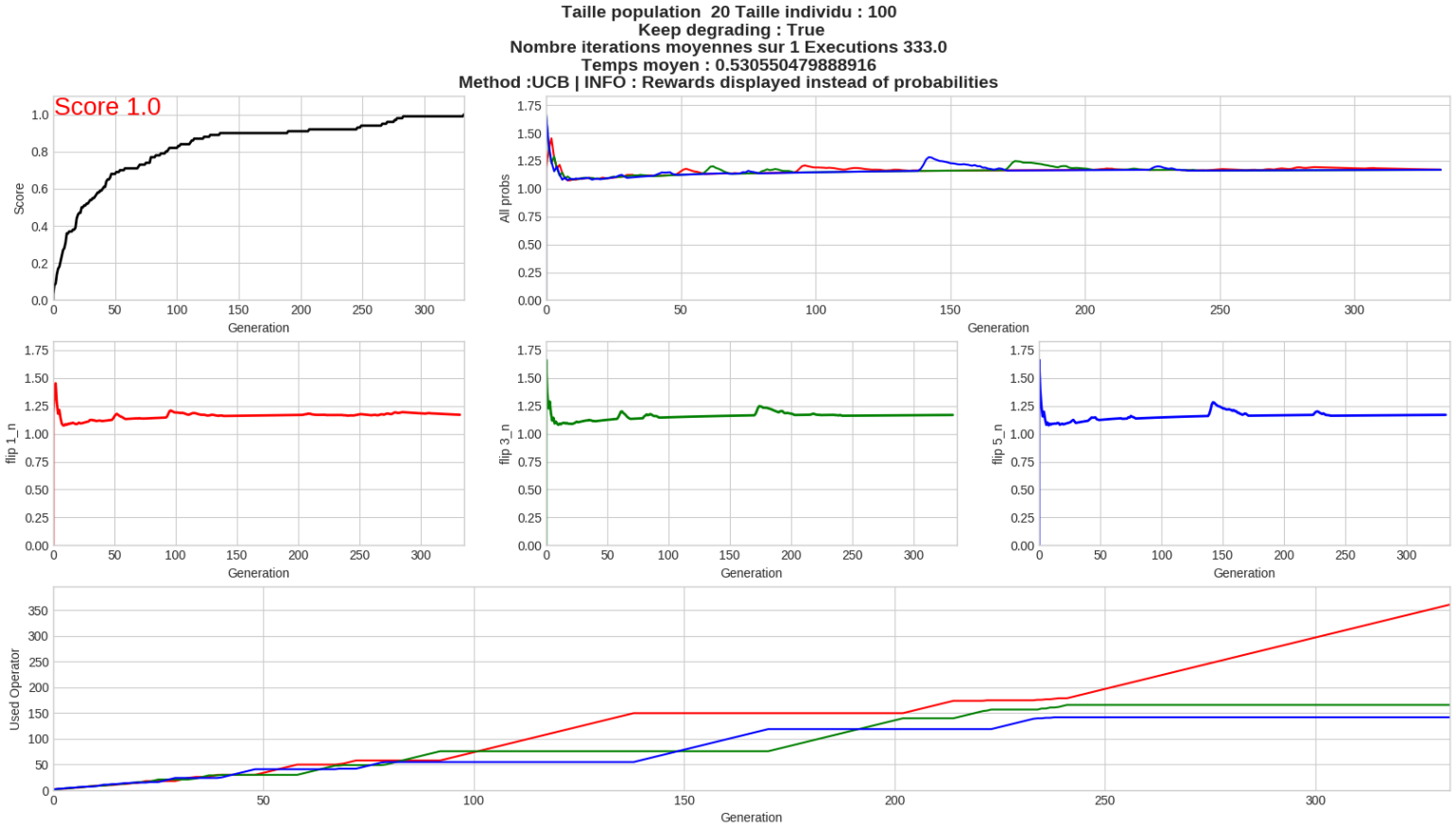

FIGURE 3.18

UCB | 20 individus de taille 100 | Pas de fenêtrage

Sans fenêtrage et sur des individus de grande taille il arrive parfois que la solution ne soit pas trouvé avant un nombre très très grand d'itérations (le temps que la moyenne redescende). Nous pouvons ainsi remarquer que la courbe descend, mais trop peu. Ajouter du fenêtrage permet de la faire redescendre beaucoup plus vite.

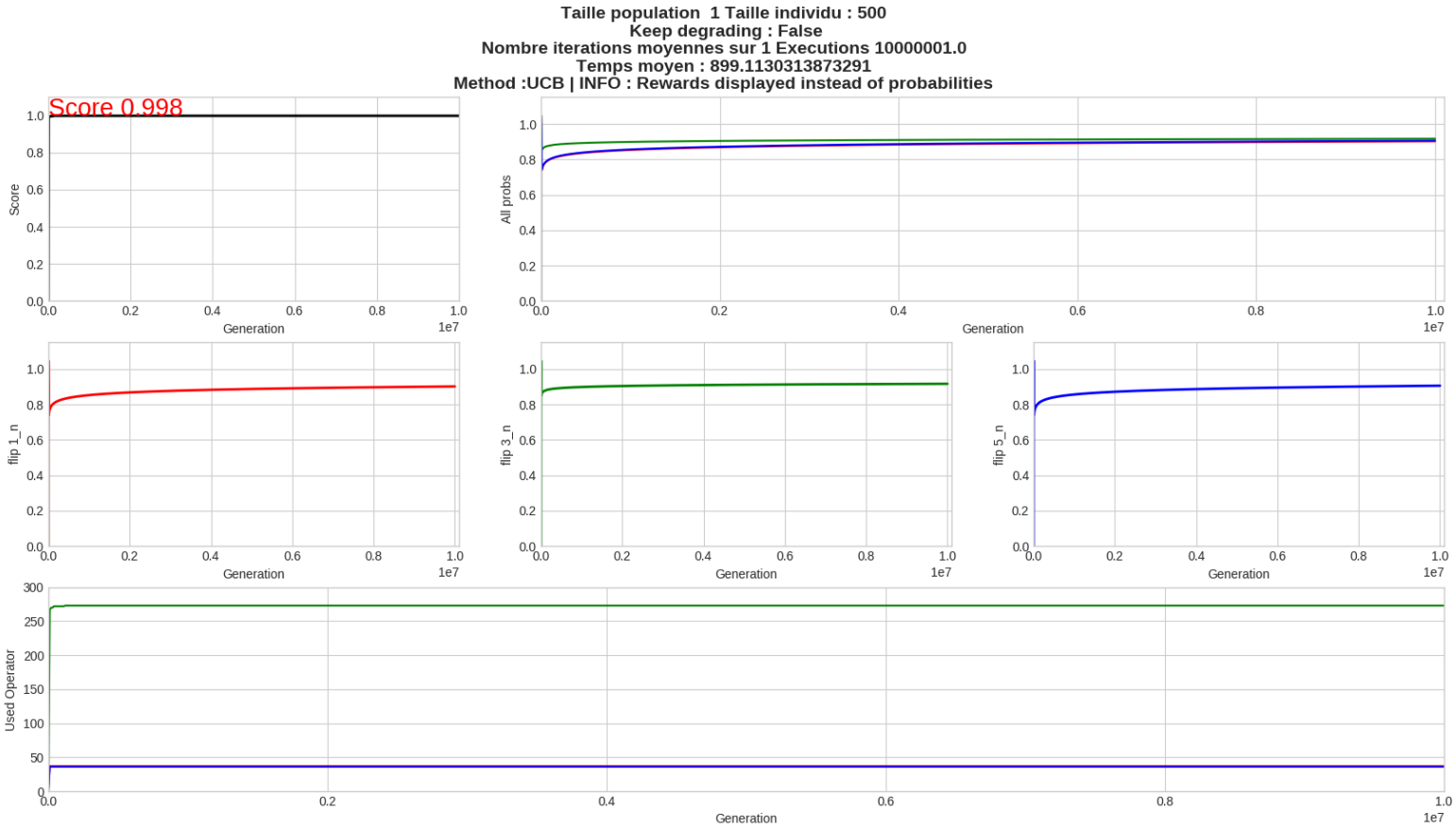

FIGURE 3.19

UCB | 20 individus de taille 1000 | Fenêtrage de la moyenne sur 50 valeurs

Ici le fenêtrage prouve son efficacité : une solution est trouvé. Le fenêtrage permet de faire remonter l'opérateur 1n vers la génération 200.

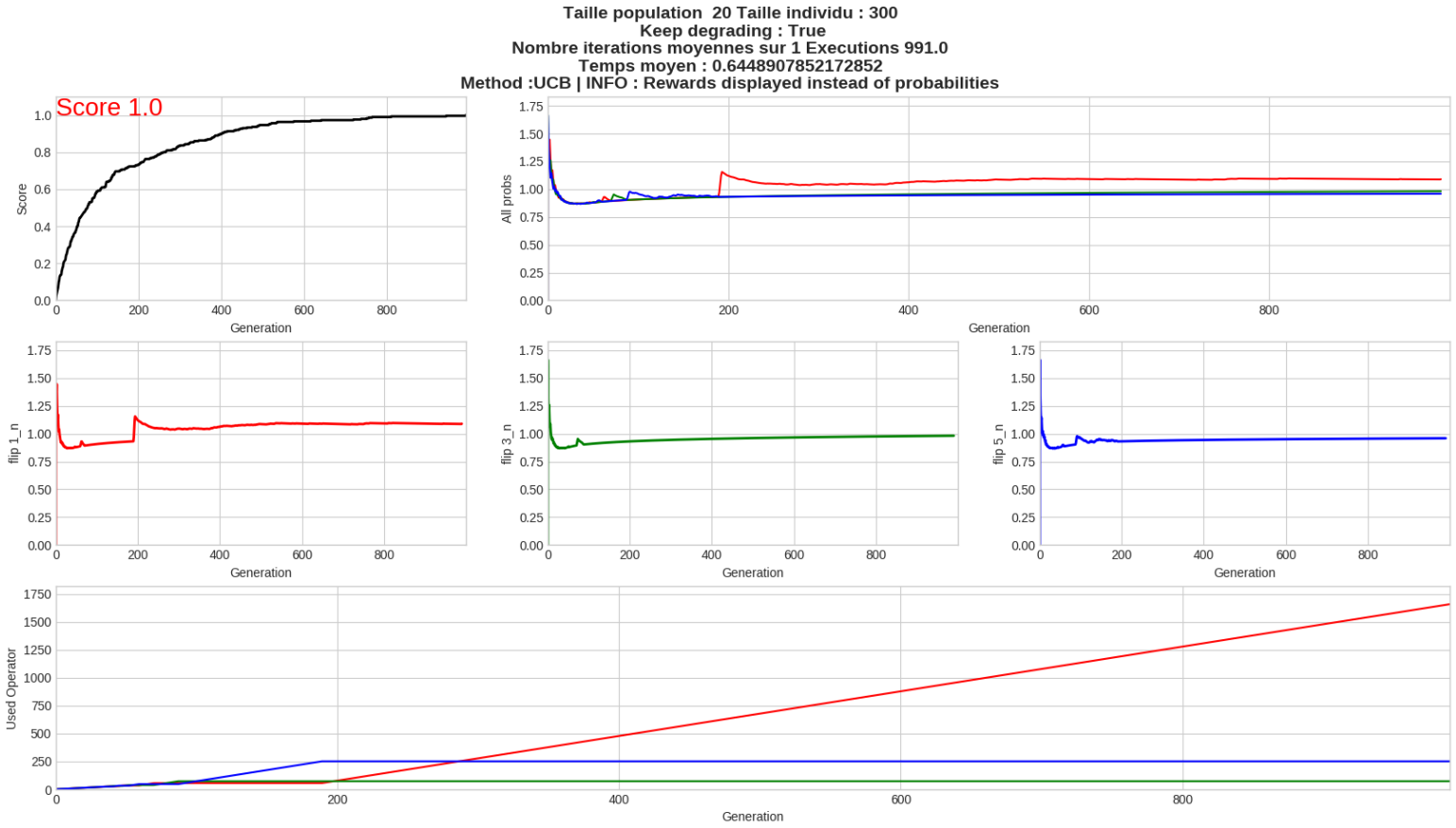

FIGURE 3.20

3.8.2 Tentative d'amélioration

En plus du fenêtrage, j'ai essayé d'augmenter l'exploration individuelle des opérateurs en comparant le score ucb à sa moyenne. Si son score n'est pas au dessus de sa moyenne depuis trop longtemps, alors sa moyenne est remise à zéro. Voici un résultat obtenu avec cette méthode :

UCB | 20 individus de taille 1000 | Tentative d'amélioration

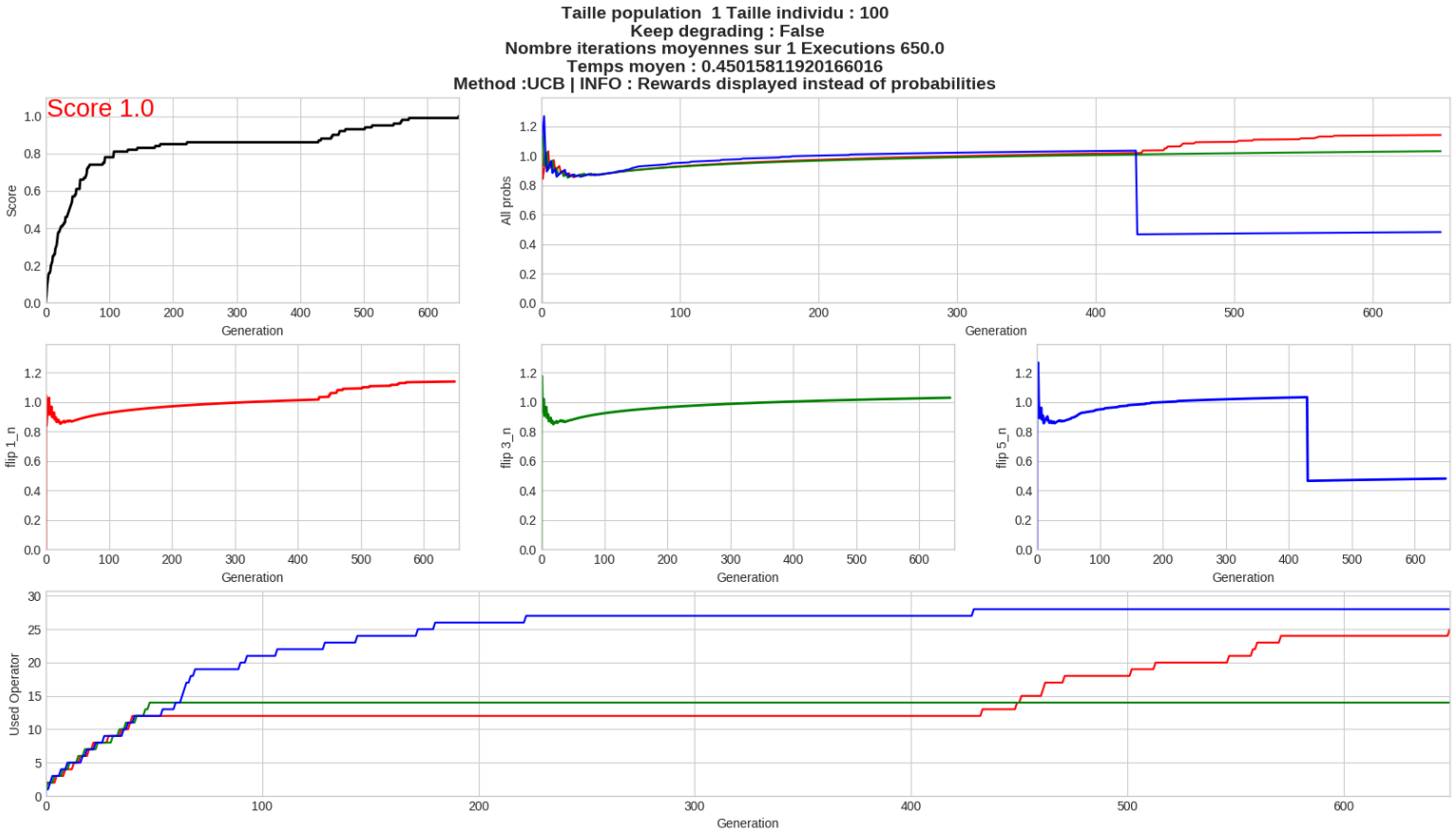

FIGURE 3.21 – L'opérateur 5n est remis à 0 pour laisser l'opérateur 1n

3.8.3 Conclusion

L'algorithme UCB est très intéressant bien qu'à priori il doive être adapté au contexte pour fonctionner.

3.9 EXP3

3.9.1 Présentation

Implémentation de l'algorithme trouvé sur :
`https://www.math.univ-toulouse.fr/~jlouedec/demoBandits.html`
Cet algorithme permet d'ajouter des poids aux probabilités.

3.9.2 Résultats

Grande Exploration

Étonnant, l'opérateur 3 flips est mis en avant comparé au 5 flips. Le 1 flip prend bien le relais.

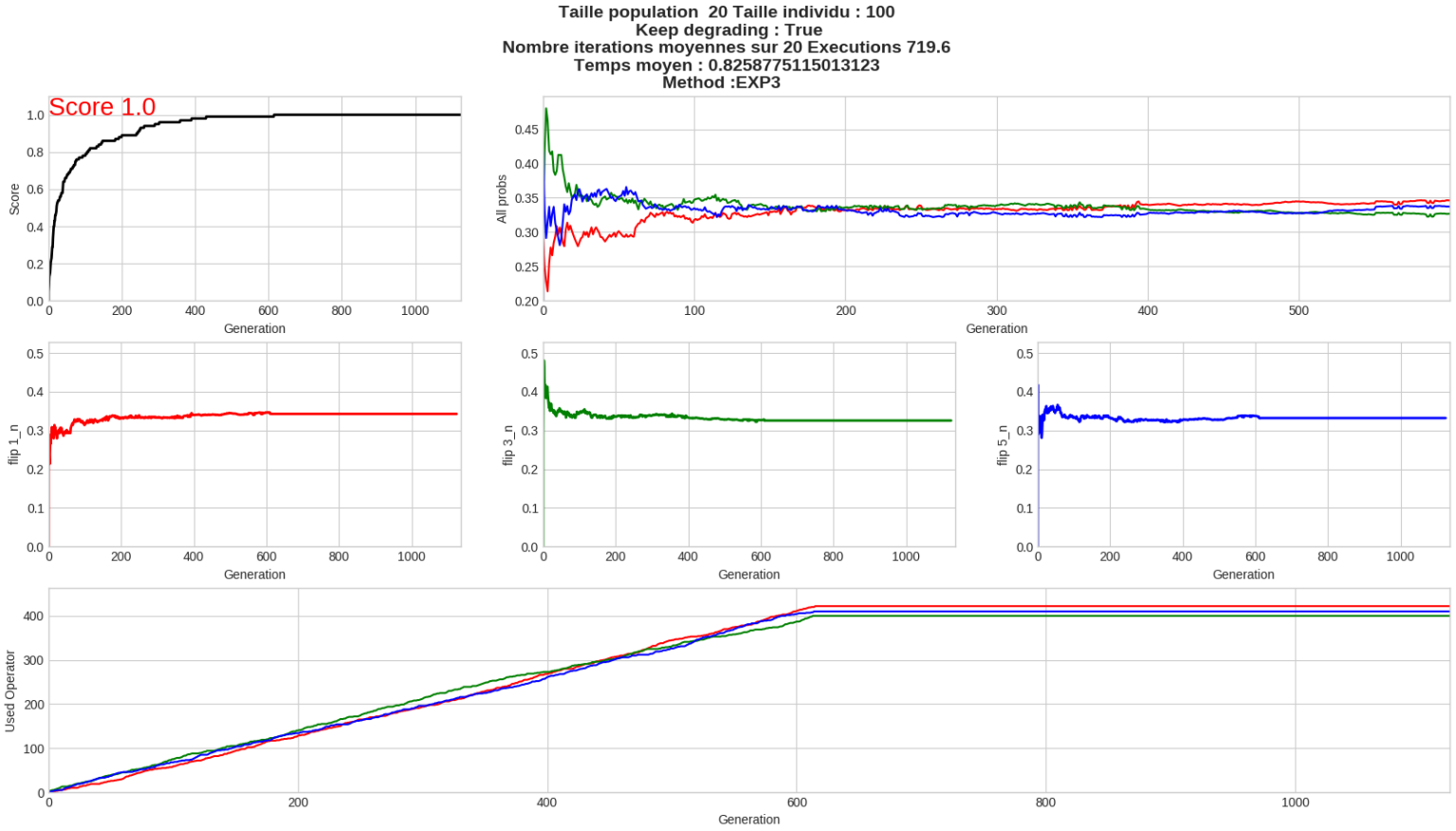

FIGURE 3.22

Faible Exploration

L'exploration étant nulle, les poids des probabilités le sont également. Les probabilités sont donc égales et l'Exp3 agit comme une roulette fixe.

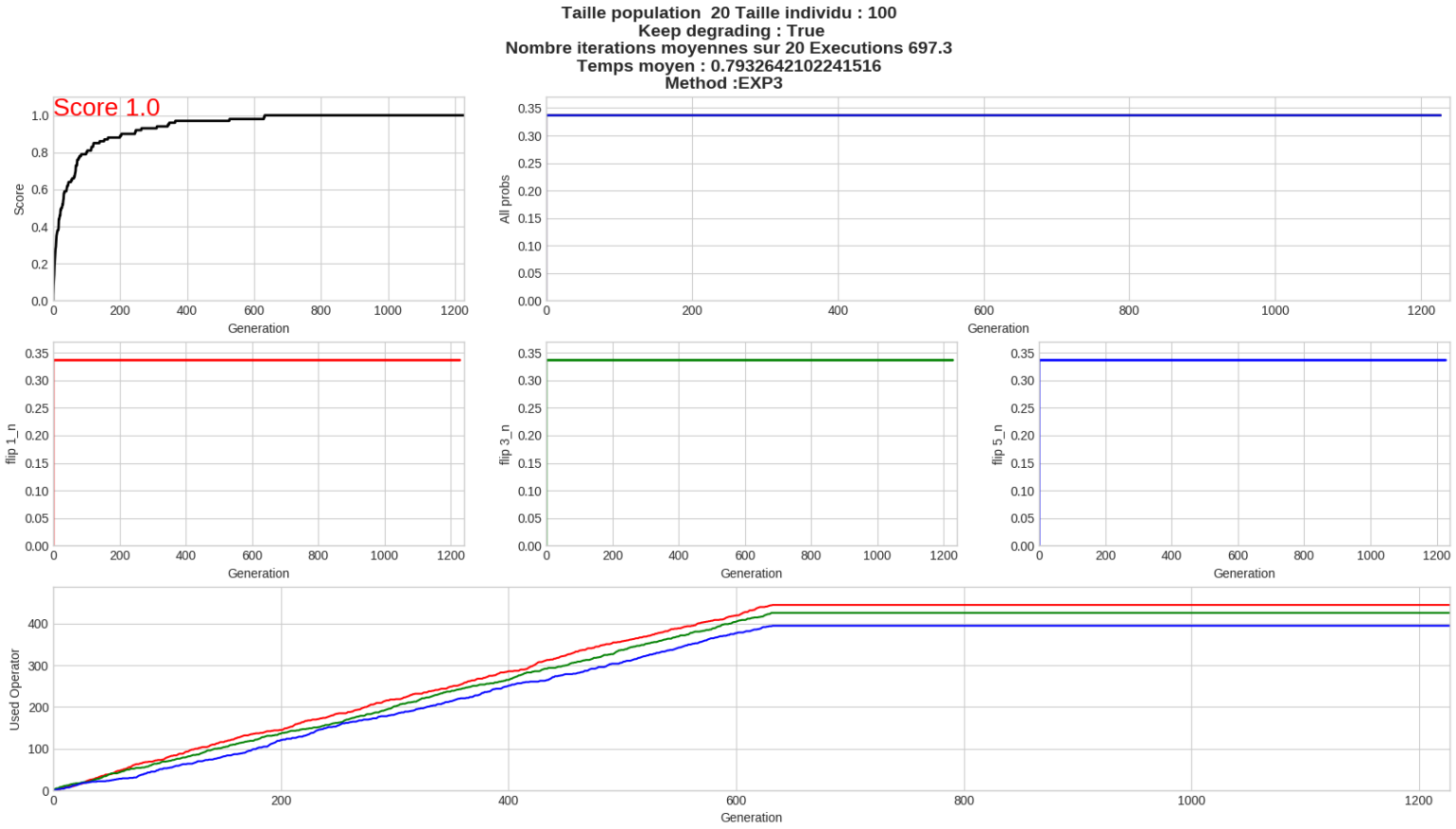

FIGURE 3.23

3.9.3 Conclusion

Plus de recherches devraient être réalisé pour conclure, mais Exp3 semble se rapprocher d'une roulette adaptive améliorée.

4 Analyse de la sélection des individus

4.1 Critère de score

Suppression des deux individus les moins bons | Oracle Myope

Les résultats sont identiques à ceux obtenus précédemment

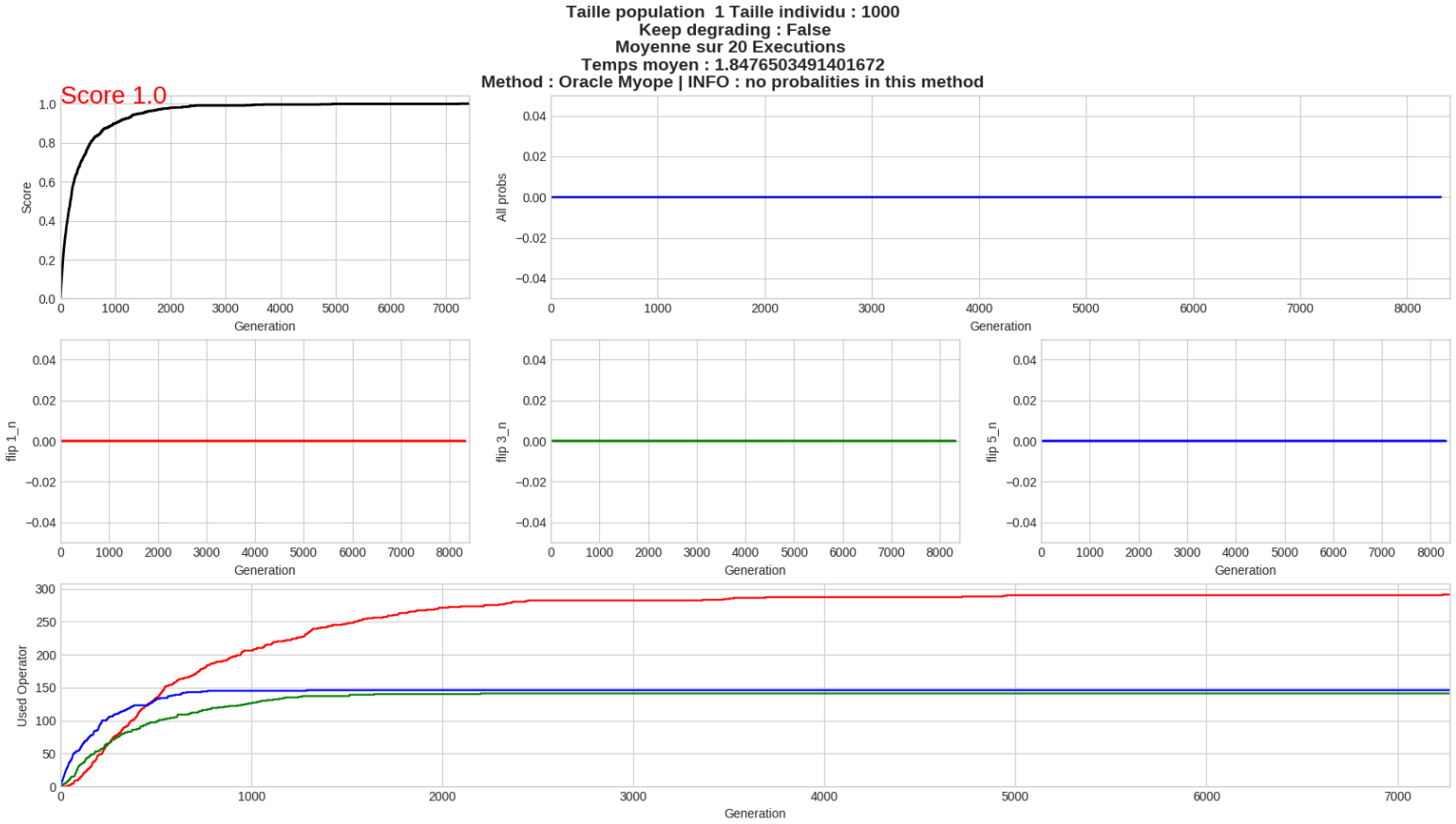

Figure 4.1

4.2 Critère d'âge

Suppression des deux individus les plus vieux | Oracle Myope

Il semblerait que le critère d'âge soit beaucoup moins efficace que celui du score. En effet un certain bruit vient pertuber l'avancement de la résolution.

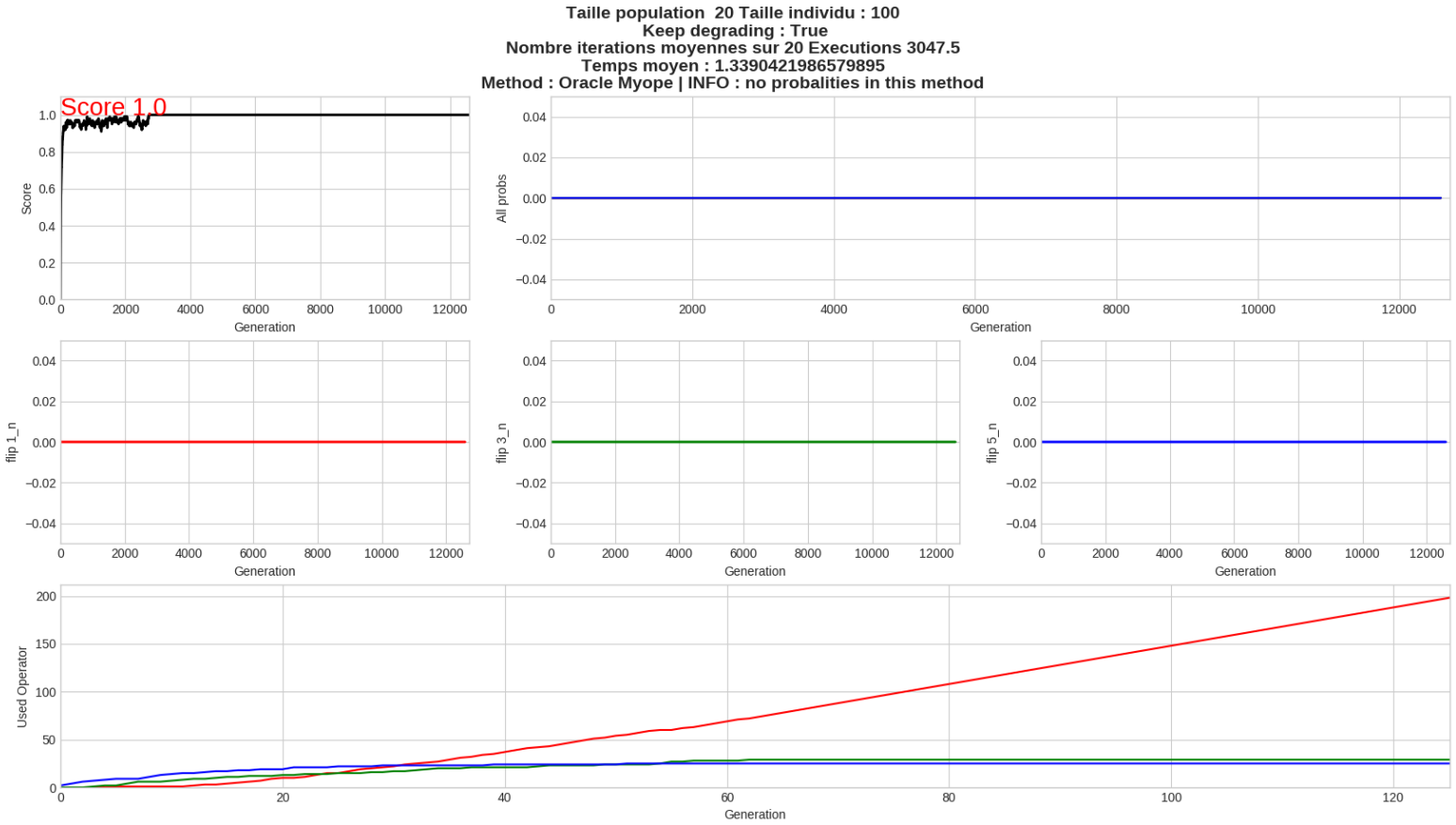

Figure 4.2

4.3 Conclusion

Bien que le critère de score semble plus efficace, j'émets tout de même des réserves dessus. Je n'ai pas eut le temps de contrôler son bon fonctionnement.

5 Conclusion

Cette analyse d'un algorithme génétique nous a permis de mettre en avant l'efficacité de certains critères, et notamment sur la sélection des opérateurs. L'oracle myope reste le plus efficace, mais il devient beaucoup trop coûteux lorsque le calcul du score se complexifie ou que le nombre/taille des individus augmente. L'utilisation d'un sélecteur d'opérateur nécessitant moins de calcul est donc primordial. L'algorithme UCB nous a donné de bons résultats, bien qu'il doive être adapté au problème.

Avec plus de temps il aurait été intéressant de poursuivre les recherches, et notamment sur la gestion de la population.

Enfin, il ne faut pas oublier que notre problème ne possède qu'un seul minimum local, qui est également le global, et que l'algorithme pourrait donner des résultats totalement différents appliqué à un autre problème.