Compte Rendu

Résolution de Problèmes

Satisfaction de contraintes pour le Wordle Mind

Binôme :

Jules Cassan

Encadrant :

ThibautLust

Github : `https://github.com/White-On/Wordle.git`

Sommaire

1 Introduction

L'objet de ce projet est de développer et tester des méthodes de satisfaction de contraintes et un algorithme génétique pour la résolution d'un problème de wordle mind. Nous considérons une version de ce jeu dans laquelle on doit découvrir un mot caché du dictionnaire ("appelé mot secret") qui se compose de n lettres. Le jeu consiste pour le joueur 'a deviner le mot. Pour obtenir de l'information le joueur peut proposer un mot du dictionnaire et le programme lui indique combien de caractères du mot proposé sont corrects et bien placés et d'autre part combien de caractères sont corrects mais mal placés (par exemple si le décodeur propose le mot "tarte" alors que le mot secret est "dette" on aura 2 bien placés et 1 mal placé (attention, contrairement à la version originale de "wordle" la réponse n'indique pas quelle lettre est bien placée et quelle lettre est mal placée). Le joueur peut alors tenter un nouvel essai et ainsi de suite jusqu'à ce qu'il tombe sur le mot secret qui engendrera n bien placés et la partie s'arrête. Le but est bien sûr de cherchera découvrir le mot secret en utilisant le moins d'essais possibles.

Dans ce projet, on s'intéresse à réaliser un programme qui, à chaque étape du jeu, est capable de proposer un nouveau mot à essayer qui soit compatible avec toutes les informations accumulées lors des essais précédents (on s'interdit de tester des combinaisons incompatibles avec l'information disponible, même si elle sont informatives).

2 Convergence vers une solution

Le but du programme est qu'a chaque nouvelle itération, un mot compatible lui sois proposer. Mais si on est cohérent avec les mots proposée avant on ne dois pas proposer 2 fois le même mot. Notre programme dois donc toujours faire en sorte do proposer un mot qui n'a pas déjà été proposé et des mots possible/ compatible avec l'ensemble des mots dans le dictionnaire. Dans le pire des cas on aura proposé l'ensemble du dictionnaire et le mot à devinée fait partie du dictionnaire donc on convergera toujours à notre solution.

3 Outils standard du Projet

Pour ce projet, Il était utile de commencer par implémenter quelques outils utiles pour tout les algorithmes qui suivront. Tout d'abords, la récupération de notre dictionnaire est réalisée par la fonction *parse()* présente dans le fichier *Parse_ Wordle.py*. Cette fonction prend en paramètre le chemin du fichier (de base la récupération est celle du fichier *dico.txt*) et retourne un dictionnaire de mots sous la forme d'un dictionnaire python avec comme clé le nombre de lettre du mot et comme valeur un liste de mot de même taille. Cela permet de gagner du temps car il est inutile de chercher des mots qui ne sont pas de la même taille que le mot que l'on cherche. Une fonction (*give_ random_ word()*) sélectionne aussi un mot aléatoire dans le dictionnaire à partir de la taille du mot que l'on cherche. D'autres fonctions outils sont présente dans le fichier *Tools_ Wordle.py* pour la gestion de mots, leur construction et leur compatibilité.

Mais une des plus importante est la fonction *check_ correct2()* (il existe également *check_ correct() mais elle ne prenais pas en compte certaines contraintes mais elle est resté dans le projet comme trace* . C'est elle qui fait office de juge pour un mot proposé par l'algorithme et renvoie une liste contenant le nombre de lettre correctement placé, le nombre de lettre présente dans le mot mais mal placé, le nombre de lettre mal placé. Cette fonction est très importante car c'est elle qui nous donne les informations des essaies de nos algorithmes.

Le fonctionnement des algorithmes est également précisé dans le code et leur utilisation est précisé dans le fichier README.md.

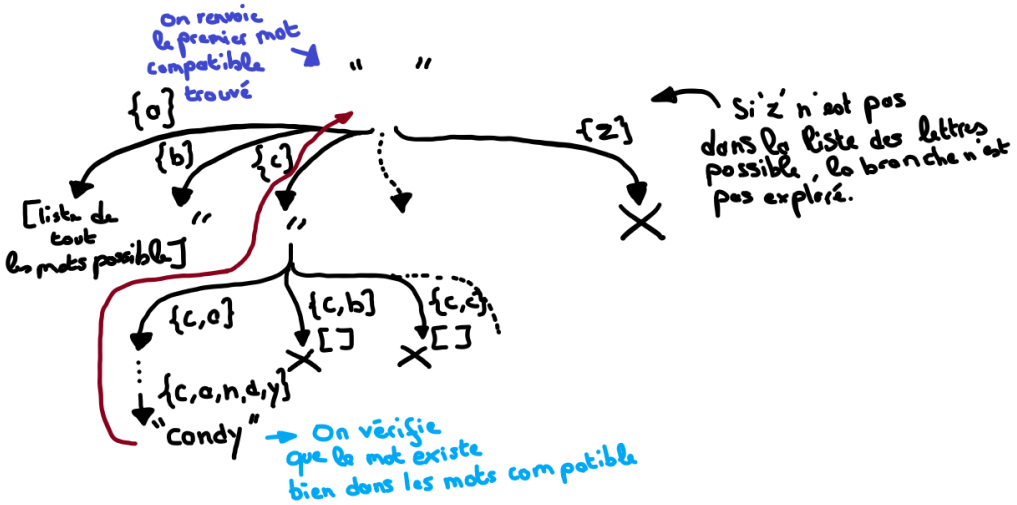
4 Modélisation et résolution par CSP

Dans la partie CSP , on vas devoir présenter à chaque itération un mot compatible (ou plus tard une liste de mots compatibles) avec nos tentatives précédentes et prendre le mots compatible le plus susceptible d'être celui que l'on cherche. La différence entre nos deux algorithmes est la manière de générer nos mots compatible sois avec un simple retour arrière chronologique, sois en le combinant avec du forward checking pour une recherche normalement plus rapide. Dans nos deux cas, On vas chercher a retirer des mots de notre champs des possibles en regardant le résultat des vérifications faites auprès de *check_correct*2().

On peux déjà distingué 4 cas possibles :
- Toutes les lettres sont correctement placé dans le mot, donc le mot que l'on viens de proposée est le mot cherché c'est gagné !
- Toutes les lettres ne sont pas dans le mot, donc on peux retirer toutes les lettres comprise dans le mot que l'on viens d'essayer des lettre disponible pour la génération de nouveaux mots
- Toutes les lettres sont dans le mot mais ne sont pas bien placé ou correctement placé, alors les seuls lettres avec les quels on peux généré les prochains mots doivent être uniquement composée de ces lettres
- Le résultat est ambiguë avec des lettres bien/mal/pas dans le mot, ce qui ne nous donne que peu d'information.

4.1 CSP : retour arrière chronologique

La génération d'une liste de mot compatible avec retour arrière chronologique se présente comme ceci : On pars d'une liste vide de mot compatible que l'on vas remplir au cours du temps. Pour chaque lettres disponible (au début toutes les lettres de l'alphabet) on vas lancer une recherche en profondeur des mots que l'on peu crée. Au fur et a mesure, on ajoute des lettres a notre mot pour le construire. On peux se rendre compte que le mot que l'on construit sur une de no recherche ne donnera probablement aucun résultats car aucun mot du dictionnaire ne peux être construit avec cet agencement de lettre, on ferme donc la recherche sur cette branche. Si la liste de mot possible est réduite a une seul mot, alors on sais que l'on ne pourra construire qu'un seul mot donc il n'est pas nécessaire de continuer la recherche plus loin et donc la branche revoie le mot trouvé. On se retrouve donc avec une liste de mot compatible avec le dictionnaire et avec les essaie précédent car aucun mot n'a été construit à partir de lettres qui ne figure pas dans le mot. La génération de mot pour le CSP avec retour arrière chronologique simple est contrôler par la fonction *gener_ compatible()*



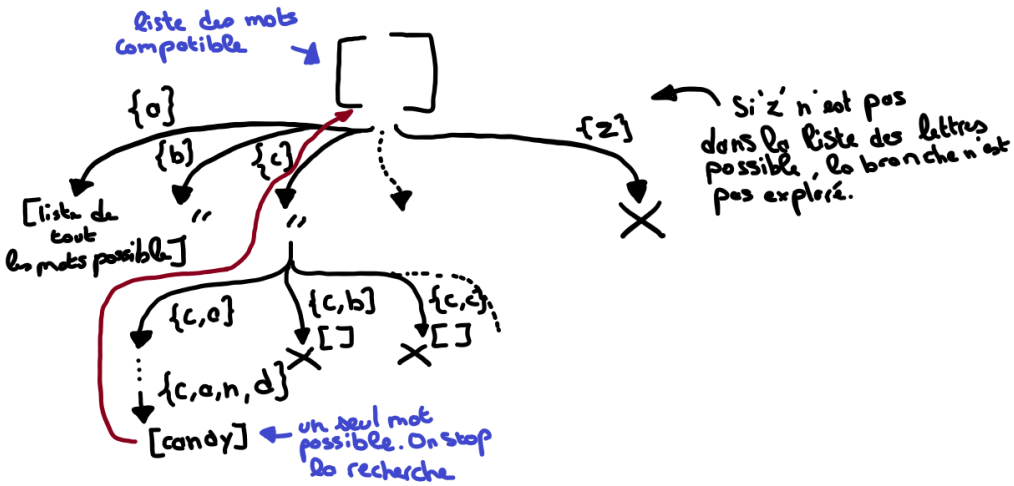
Pour la recherche plus "basique" que l'on retrouve normalement dans la partie 1 du projet (noté A1) on vas plus simplement juste cherche à construire un mot à partir de la même méthode récursive mais juste prendre la première solution que l'on trouve et la tester.

4.2 CSP : retour arrière chronologique avec arc cohérence

Pour la génération de mot compatible avec forward checking, on part avec la même base que pour la résolution précédente. On vas cependant essayer de voir plus loin pour réduire le champ des mots possible plus rapidement. En effet, on dois pouvoir retirer de la liste des mot possible tout les mots qui se construisent avec des lettres qui ne figure pas dans la liste des caractère disponible. cela permet d'éviter de crée des embranchement de recherche trop large et devrais pouvoir réduire le temps d'exploration.

4.3 Amélioration par recherche du meilleurs mot

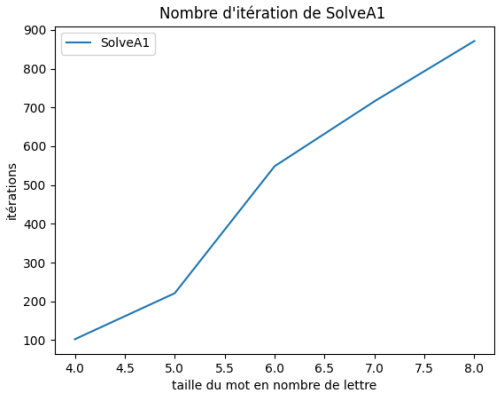
Le dernier cas correspond à la majorité des cas. On vas donc essayer d'exploité ce cas en sélectionnant le mot en fonction de poids/score sur chacune des lettres que l'on attribue en fonction du résultat de $check\_correct2()$. Donc, si on plus de la moitié des lettres qui sont correctement placée ou au moins dans le mot, alors on donne un score positif à chaque lettre. Dans le cas contraire, on donne une pénalité de score sur toutes les lettres. On se servira ensuite des score de chaque lettres pour choisir parmi la liste des mots générés lequel est le mot le plus susceptible d'être correct.



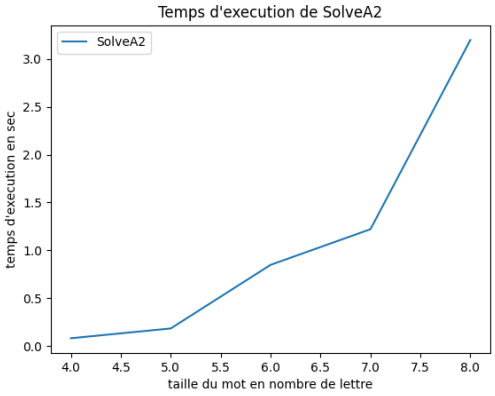
4.4 CSP : Résultats et discussion des performances

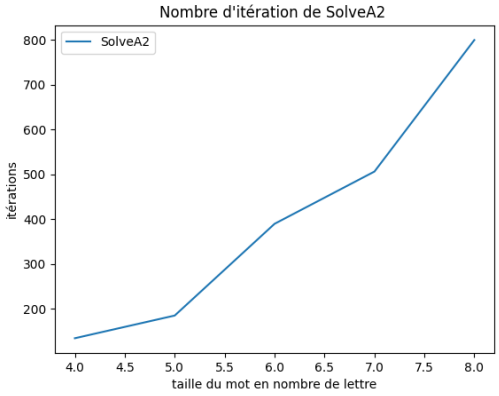
Pour les résultats on vas comparer les temps d'exécutions des différentes versions de la résolution par CSP. Les meilleurs résultats sont ceux avec une faible temps d'exécution moyen pour un nombre de lettre dans le mot que l'on cherche a résoudre ainsi que le nombre de tentative par rapport au nombre de mot dans notre dictionnaire pour le nombre de lettre. Les tests suivants sont tous les temps moyens de détermination du mot secret sur 20 instances de taille n = 4 jusqu'à n = 8.

4.4.1 CSP simple A1

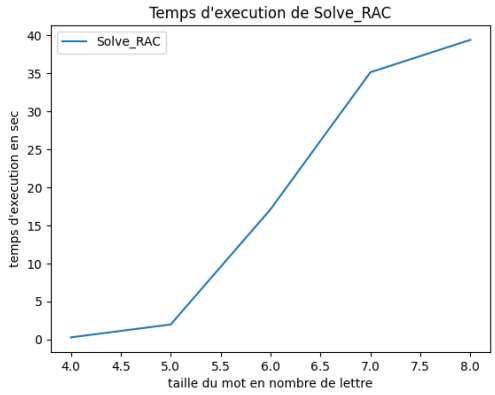


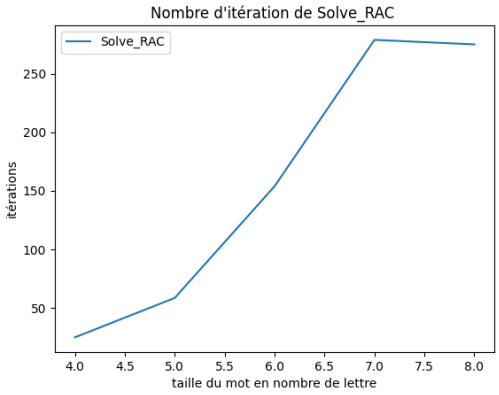
4.4.2 CSP simple A2



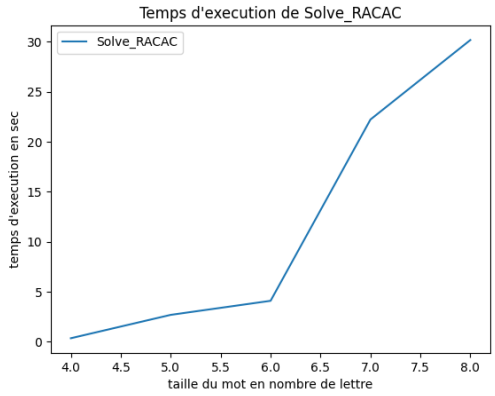


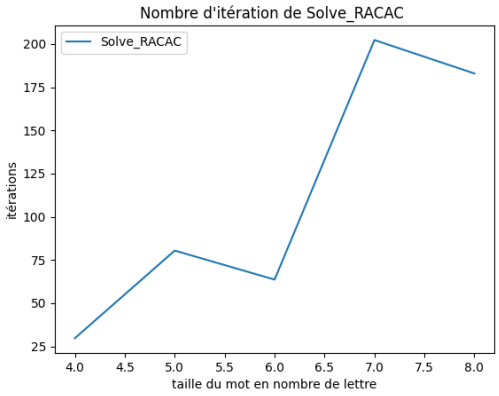
4.4.3 CSP Retour arrière chronologique avec sélection de mot (RAC)



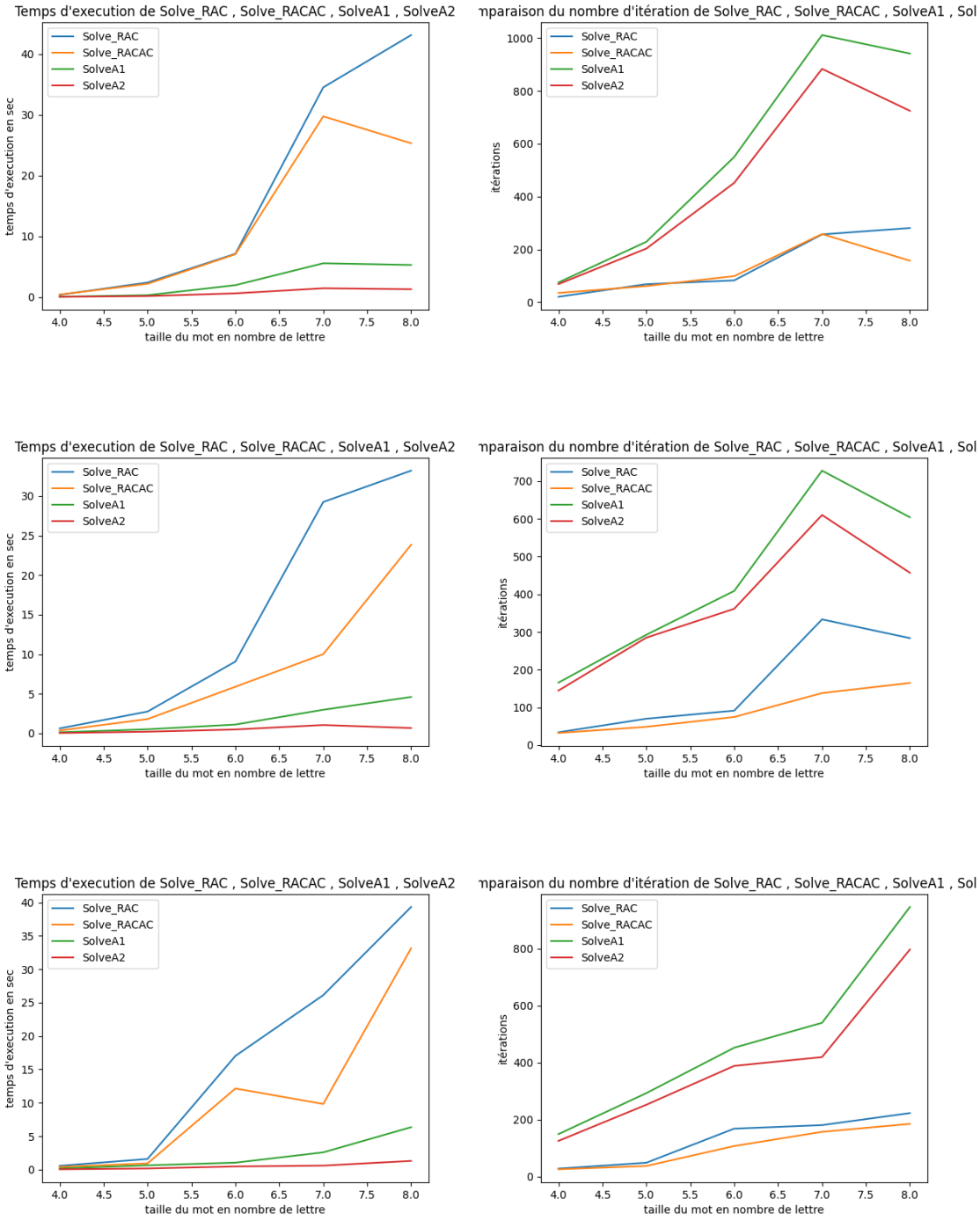


4.4.4 CSP Retour arrière chronologique avec sélection de mot (RACAC)





4.4.5 Comparaison



la série de tests qui est liée aux deux graphiques précédent est la recherche par nos 4 algorithmes du même mot pour chaque instance (il y a 20 instance au total) .

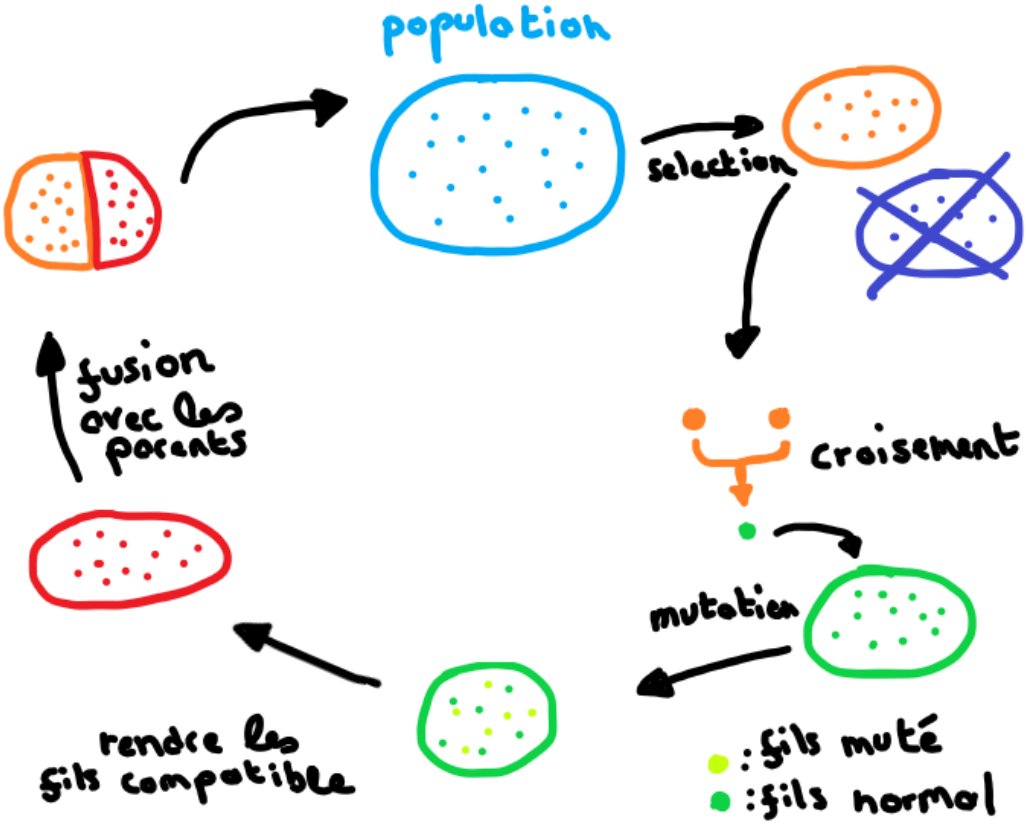
On peux voir que les deux premiers algorithmes, A1 et A2, ont des résultats similaires. A2 semble un peux plus efficace mais sont tout les deux bine plus performant en terme de temps d'exécution par rapport à la version où l'on cherche le meilleur mot à proposé. Mais a contrario, ceux sont ces même algorithme qui se démarquent par une nombre de tentative bien moins important que A1 et A2. Il faut donc faire un compromis entre vitesse de résolution ou nombre de tentative effectuée pour retrouvée le même mot.

4.5 Amélioration possible

Un paramètre important à prendre déjà en compte est qu'un facteur aléatoire reste présent dans la recherche. En effet, avec de la chance sur nos première itérations, on peux avoir plusieurs mots ne contenant que des lettres qui n'appartiennent pas à notre mot et donc réduire très vite les possibilités de mots compatibles. C'est cette remarque qui nous amène a une proposition d'amélioration : lors des premières itérations, il serais peu être plus judicieux de proposer les mot contenant des caractères en plusieurs exemplaires et contenant des caractères peux utilisée dans notre dictionnaire.

5 Modélisation et résolution par algorithme génétique

Pour la modélisation avec l'algorithme génétique, on cherche encore une fois a générer une liste de mot compatibles. On pars au début d'une liste de mot compatible avec le dictionnaire prise au hasard. Puis on range tout ces mot dans l'ordre du meilleur au pire avec comme paramètre de comparaison leur valeur adaptative (ou fitness). On prend la meilleur moitié de population (que nous allons décrire comme les parents) et on réalise des croisements entre les parents pour jusqu'à avoir une autre moitié de population pour fusionner les anciens bon parents et les nouveaux enfants. Cette nouvelle génération subit ensuite des mutations aléatoires pour élargir les solutions et on évalue de nouveaux leur valeur adaptative et on ajoute les mots compatibles à la liste E (notre liste de résultats). On réitère ce processus jusqu'à ce que la liste E atteigne sa capacité maximal ou que l'on atteigne le nombre de génération maximal. Ce processus de création de la liste E est gérée par la fonction *algo_genetique()* puis utilisé par *Solve_Genetic()*. Pour la sélection d'un mot, on fait comme pour la résolution CSP pour exploiter nos essaie précédents. On retrouve donc le même système de score. Par la suite ce système de score est également utilisé pour le calcul de la valeur adaptative.



5.1 Valeur adaptative ou fitness

La valeur adaptative avais pour consigne de dépendre du nombre d'incompatibilité des essais précédents. En effet, à travers le score qui cette fois ne peux que augmenter (dans la résolution CSP, le score ou poids de chaque lettre pouvais diminuer), on considère que la valeur adaptative d'un mot compatible est la somme des valeurs de toutes ses lettres. Le calcul de la fitness se trouve dans la classe *Genetic_ Word* dans la méthode *evaluation()*.

5.2 Création de solution fils

La création de solution fils ou d'individu fils s'effectue par crossover. En effet, on part de deux mots parents, on prendre le début du premier parent et on complète avec le reste du 2eme parent et on obtient un mot fils. le croisement de deux mot parents est réalisée par la méthode *breed()*.

5.3 Mutation

La mutation d'un mot est effectué après le croisement de deux mots pour en faire une solution fils et chaque fils a une chance de subir une mutation (20%). Le mot peux alors subir 3 mutations différentes avec pour chaque une probabilité de 1/3. La mutation d'un mot est gérée par la méthode *mutation()* Les différentes mutations sont :

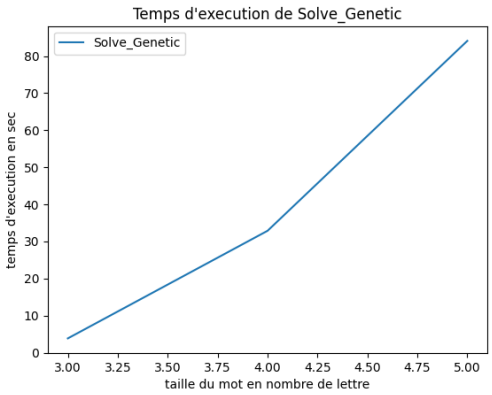
- On modifie un des caractère au hasard dans notre mot (la méthode *randomCaracterChange()*)

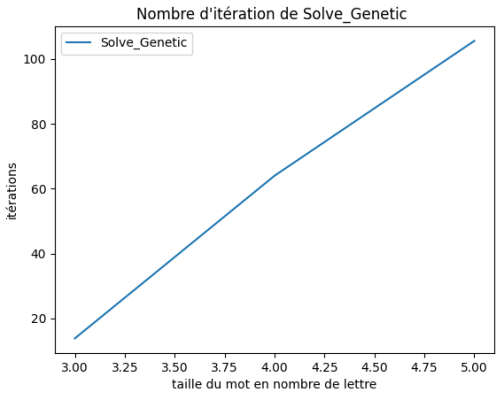
- On échange deux caractère aléatoire dans notre mot (la méthode *exchangeCaracter()*)

- On refait un crossover avec un autre mot de la population (la méthode *crossover()*)

Après avoir fait muter notre mot, il est très probable qu'il ne sois pas compatible avec la liste de mots de notre dictionnaire. C'est pourquoi on vas prendre le mot qui a la distance d'édition la plus faible de notre mot (ceci est fait par la fonction *closest_ word()* dans *Tools_ Wordle.py*.

5.4 Algorithme génétique : Résultats et discussion des performances





Tout d'abords, l'algorithme génétique implémenté semble avoir un problème de performance important du à la recherche du mot le plus proche lorsque le mot construit après croisement ou une mutation n'est pas compatible avec les mots du dictionnaire.

5.5 Améliorations possibles

Comme pour le CSP, retirer des lettres de notre champs des possible sembles une très bonne idée encore ici. Mais une autre approche différente de celle que nous avons effectué est possible. Au lieu de prendre la meilleur moitié de la population pour crée des solutions fils pour recrée une population, on pourrais prendre le meilleur individu et recrée toute une population en partant de cette base. Dans cette esprit, on pourrais donc tester le meilleur mots à la fin de notre génération de mot et puis continuer la recherche génétique a partir de ce dernier.

Il serais aussi possible de changer ou d'ajouter une mutation supplémentaire de recherche local pour crée un algorithme mémétique.

6 Références et inspirations pour aller plus loin

Pour réaliser ce projet, je me suis basé entièrement sur le cours de Résolution de Problème de 2022. Mais pendant mes recherche sur le projet, je suis tombé sur une autre manière d'aborder le problème que je tenais a partager qui je pense pourrais s'adapter convenablement à nos contraintes d'implémentassions du problème. L'idée est de valué l'information donnée par une solution que l'on propose ce qui nous permet de savoir en fonction du dictionnaire entier quel mot serais le plus avantageux a tester. Vidéo : `https://youtu.be/v68zYyaEmEA`