Université Sorbonne
Campus Pierre et Marie Curie Jussieu

RP

RP Project Wordle

Élèves :
Hakim Amraoui, Ruizheng Xu

Enseignant
Thibaut Lust

18 avril 2022

SORBONNE UNIVERSITÉ

LIP6

Table des matières

Table des figures

1 Introduction

Question 1 : Expliquer pourquoi l'utilisation d'un tel programme permet de créer une séquence d'essais qui converge nécessairement vers la solution (le mot caché).

Au départ de cet algorithme, nous avons aucun mot proposé donc aucune information. Après avoir essayé un premier mot, on connaît maintenant le score de ce mot, c'est-à-dire combien de lettres sont correctes et bien placées, ainsi que combien de lettres sont correctes mais mal placées. En stockant les mots essayés précédemment et leurs scores, nous réduisons drastiquement le nombre de candidats respectant ces dites contraintes. Nous convergeons donc vers un ensemble de mots consistants.

Supposons maintenant que le mot secret ne respecte pas ces contraintes et est alors exclus des candidats possibles.

Soit la fonction `evalue(w)` telle que : `evalue(w)` = (BP, MP) qui évalue le mot w par rapport au mot secret avec BP étant le nombre de lettres correctes et bien placées et MP le nombre de lettres correctes mais mal placées. Si le mot secret n'est pas dans les candidats respectant ces contraintes, cela veut dire que le mot secret n'a pas toutes les lettres en commun avec lui-même, ce qui est contradictoire.

Ainsi, l'algorithme converge forcement vers le mot secret.

2 Partie 1 : Modélisation et résolution par CSP

Question 2 : Présenter et expliquer vos procédures de recherche d'une solution compatible puis implanter ces procédures au cœur d'un algorithme itératif de détermination du mot secret. Donner les temps moyens de détermination du mot secret sur 20 instances de taille n = 4. Etudier ensuite l'évolution du temps moyen de résolution et du nombre moyen d'essais nécessaires lorsque n augmente (on ira au moins jusqu''a n = 8).

2.1 A1 : retour arrière chronologique

L'algorithme de retour arrière chronologique suivant est celui vu dans le cours de M.Perny.

Algorithm 1 Back Tracking

```
1:  function BACKTRACK(i, V, D)
2:                                                  ▷ i instance courante
3:                                  ▷ V liste des variables non instanciées dans i
4:                                                  ▷ D domaine de chaque variables
5:                                                  ▷ C contraintes
6:
7:      if V = ∅ then
8:          return i
9:      else
10:         Choose x_k ∈ V
11:         for all v ∈ D_k do
12:             if  i ∪ (x_k → v) local consistant then
13:                 backTrack(i ∪ (x_k → v), V\x_k, D)
14:
```

Lors de notre procédure de retour arrière chronologique, l'idée est d'instancier chacune des lettres du mot avec une variable appartenant à son domaine. A chaque étape, on instancie la variable x_i avec la première lettre de son domaine D_{x_i}, si cette instance est consistante, alors on passe à la lettre x_{i+1}, sinon, on revient à la lettre x_i qu'on instancie avec la lettre suivante de son domaine.

Pour vérifier la consistance d'une instance, on vérifie qu'il existe un mot dans le dictionnaire commençant par les mêmes lettres.

Une fois que l'on a une instance d'un mot de même longueur que le mot secret, on vérifie si c'est le bon mot, si c'est bien le bon mot, BINGO, sinon on retourne sur la dernière variable instanciée.

2.2 A2 : retour arrière chronologique avec arc cohérence

Algorithm 2 Forward Checking

```
1: function FORWARD-CHECKING(i, V, D, C)
2:                                                  ▷ i current instance
3:                                  ▷ V list of uninitiated variable in i
4:                                                  ▷ D variable's domain
5:                                                  ▷ C constraints
6:
7:     if V = ∅ then
8:         return i
9:     else
10:        Choose x_k ∈ V
11:        for all v ∈ D_k do
12:            Save ( V\x_k
13:            if check − forward(x_k, v, V) then
14:                Forward-Checking(i ∪ (x_k → v), V\x_k, D, C)
15:            end if
16:            Restore V\x_k
17:
```

Algorithm 3 Check Forward

```
1: function CHECK-FORWARD(x_k, v, V, D)
2:
3:     consistant ← True
4:     for all v' ∈ D_j do
5:         if {x_k → v, x_j → v'} non consistant then
6:             D_j ← D_j\{v'}
7:         end if
8:         if D_j = ∅ then
9:             consistant ← False
10:        end if
11:        return consistant
12:
```

Pour l'algorithme de Forward Checking, après chaque instanciation, on réalise une arc-cohérence complète sur les variables non-instanciées.

2.3 Résultats

Voici les temps observés pour l'exécution de nos 2 algorithmes sur 20 instances de taille $n = 4$.

Mot secret	Temps en sec pour le Back Tracking	Temps en sec pour le Forward Checking
apes	1.49	0.50
coat	4.79	1.84
wilt	31.16	11.99
nuts	20.12	7.86
mire	19.03	7.07
rial	23.83	9.43
rush	24.91	9.58
doer	6.68	2.70
tiny	28.44	11.10
chip	4.52	1.65
pupa	22.93	8.98
slug	25.85	10.43
sago	25.01	9.94
tick	28.08	10.98
drop	7.01	2.54
fact	8.87	3.99
swam	27.36	9.76
wily	31.51	11.95
onus	20.57	8.17
mary	17.87	7.30

Ainsi, pour le Back Tracking, nous obtenus un temps moyen de 19.00 s, et 7.39 s pour le Forward Checking.

Nous avons exécuté ses deux algorithmes en faisant varier la taille des mots n de $n = 2$ à $n = 7$.

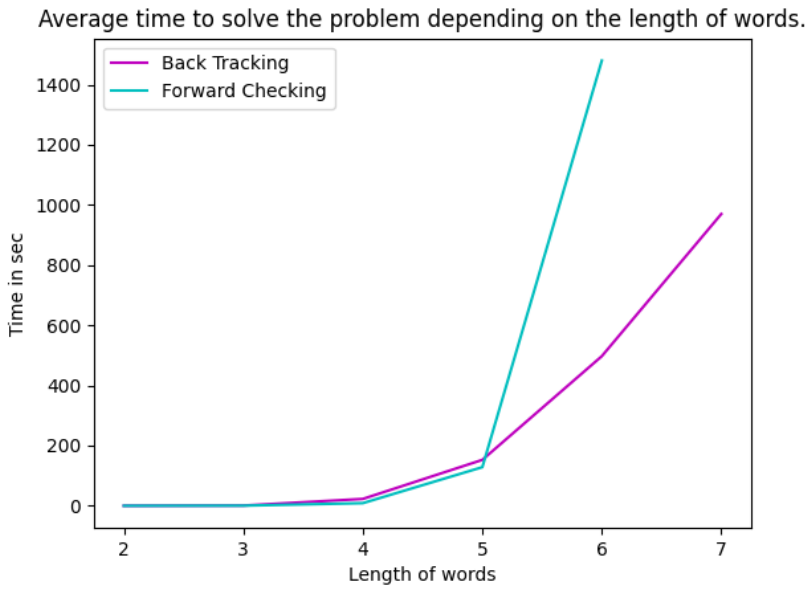

FIGURE 1 – Evolution du temps d'exécution en fonction de la longueur des mots

	column 1	column 2	column 3	column 4
1	Word	Length	Time in sec for BackTrack	Time in sec for Forward Checking
2	mr	2	0.006505250	0.007506608
3	no	2	0.005504369	0.008508205
4	ah	2	0.002501726	0.003502368
5	if	2	0.005003690	0.006005764
6	wag	3	0.8642	0.5364
7	wit	3	0.8582	0.5234
8	rat	3	0.6450	0.4028
9	mac	3	0.4674	0.2652
10	gold	4	11.4528	4.5879
11	whip	4	31.7998	11.968
12	lads	4	15.9832	6.4045
13	whir	4	31.9409	11.8797
14	surer	5	203.4109	168.9232
15	ogled	5	151.9376	130.2575
16	vanes	5	232.7511	193.1376
17	bilge	5	22.4162	21.2497
18	decked	6	243.7486	688.9940
19	sodden	6	1054.5024	3143.0837
20	chapel	6	194.3976	484.0027
21	kneads	6	496.8993	1604.2327
22	dredges	7	804.12	+9999
23	galleon	7	1091.26	+9999
24	formant	7	1015.14	+9999

FIGURE 2 – Evolution du temps d'exécution en fonction de la longueur des mots

Nous avons stoppé l'exécution du Forward Checking pour les mots de taille $n = 7$ car le temps a explosé (plusieurs heures).

On peut remarquer que le FC consomme moins de temps que BT pour $n < 5$. A partir de $n = 6$, FC prend environs 3 fois plus de temps. Ces algorithmes ne sont pas en temps polynomial.

3 Partie 2 : Modélisation et résolution par algorithme génétique

Pour cette partie, nous devons implémenter l'algorithme génétique pour trouver le mot secret.

L'algorithme génétique est un algorithme qui consiste à premièrement choisir une population d'individus aléatoires comme la population de base, puis pour un certain nombre de génération ou jusqu'à la satisfaction des contraintes, on génère de nouveaux individus qui constitueront une nouvelle population théoriquement meilleure que la population précédente, en utilisant des différents politiques de sélection, de croisement et de mutation (expliquer par la suite). Enfin, l'algorithme nous retournera le meilleur individu trouvé parmi toutes les générations.

Le pseudo-code de notre algorithme génétique est le suivant :

Algorithm 4 Genetic Algorithm

function GA($popSize, maxSize, maxGen, CXPB, MUTPB$)
▷ $popSize$: size of initial population
▷ $maxSize$: size limit of population
▷ $maxGen$: limit number of generation
▷ $CXPB$: crossover probability
▷ $MUTPB$: mutation probability

    initialize a population E with $popSize$ individuals
    initialize F as a list of corresponding fitness for E ▷ *evaluation*
    $gen \leftarrow 0$
    $bestInd \leftarrow 0$

    while *timeOut* not reached **do**
        $gen \leftarrow g + 1$
        $newPopulation \leftarrow []$

        $child1, child2 \leftarrow selection(G)$
        $child1, child2 \leftarrow crossover(child1, child2)$ with $CXPB$ probability
        $child1, child2 \leftarrow mutation(child1, child2)$ with $MUTPB$ probability
        append $child1$ and $child2$ to $newPopulation$ if they are compatible

        $G = G \cup newPopulation$
        update F with new G ▷ *evaluation*
        update $bestInd$
    return $gen, bestInd$

Avant de commencer l'itération, l'algorithme initialise une population d'individus (E) en piochant aléatoirement des mots dans le dictionnaire, ensuite, il évalue l'ensemble d'individus et mémorise leur score dans une liste correspondante. Puis, tant que le timeout n'est pas dépasser (5 minutes comme défini dans le sujet), il va générer des fils avec des parents choisis dans la population courante, ensuite, il applique les différents traitements

sur les fils, et les **concatène** dans la population courante si les nouveaux fils sont **compatibles** et réévalue la liste des scores de fitness, et ainsi de suite.

Donc à la fin de l'itération, nous aurons une liste d'individus (E) qui sont des mots compatibles et l'algorithme retournera celui qui a la meilleur valeur de fitness.

Pour pouvoir implémenter notre algorithme, nous devons définir d'abord les quatres fonctions les plus importantes qui sont la sélection, le croisement, la mutation et l'évaluation.

3.1 Compatibilité

L'ajout des nouveaux individus dans notre ensemble exige que ces derniers soient compatible avec notre ensemble. C'est-à-dire que l'évaluation de ces derniers doit être au supérieur ou égal à tous les individus déjà présents dans la population courante.

Donc pour tester la compatibilité, on évalue d'abord la valeur de fitness du mot, puis on parcourt la population, dès que le fitness du nouveau est inférieur à un des individus de la population courante, il n'est pas compatible.

3.2 Évaluation

La fonction d'évaluation est basée sur le nombre de lettres bien placées et mal placées. Ces informations seront stockées dans une liste de deux éléments [nbBienPlacee, nbMalPlacee]. Par exemple, si le mot secret est "agora", alors l'évaluation du mot "aggro" renverait [3, 1].

3.3 Sélection

Le choix des parents est basé sur la liste des valeurs de fitness F. Cette liste F contient pour chaque individu de la population, le nombre de lettres bien et mal placées, puis, et pour choisir le meilleur, nous avons décidé de calculer un autre score qui sera un entier, avec la formule suivante : **nbBienPlacee * 2 + nbMalPlacee**.

Par exemple, si le mot secret est "agora", alors l'évaluation du mot "aggro" est [3, 1], et pour comparer sa valeur de fitness avec les autres, on utilise la formule et on obtient une valeur de 3 * 2 + 1 = 7 (car le résultat de la comparaison entre des listes est moins expressive). Et ainsi, on pourra sélectionner les deux meilleurs individus de la population selon ce score qui donne un poids plus important pour les mots qui ont des lettres bien placées.

3.4 Croisement

En ce qui concerne le croisement, nous avons choisi deux différentes méthodes qui seront appliquées aléatoirement à probabilité égale :

— **Le croisement à 2-points (Two Points Crossover)**
On choisit aléatoirement deux positions p1, p2 différentes, et les fils hériteront les mêmes caractères des parents mais avec la partie entre p1 et p2 inversée.

— **Le croisement uniforme (Uniform Crossover)**
À chaque itération, on choisit aléatoirement l'attribution de chaque caractère des parents à chacun des fils avec une probabilité uniforme.

Une fois que le croisement est appliqué, on obtiendra deux fils, qui n'appartiennent pas forcément aux dictionnaires. Dans ce cas, on applique notre fonction *getClosestWord* qui cherche le mot le plus proche dans le dictionnaire du mot concerné.

3.5 Mutation

Enfin, pour la mutation, nous avons également implémenté deux différentes méthodes, mais sont utilisées dans des cas différents :

— **Remplacement des lettres (Replace)**
On tire un caractère du mot aléatoirement et on le remplace par une lettre tirée aléatoirement aussi.

— **Échanger de deux lettres (Swap)**
On tire aléatoire deux caractères du mot et on échange leur position.

Par ailleurs, comme indiquer au-dessus, le choix de la méthode à utiliser ne suit pas une probabilité, mais de l'évaluation du mot à muter. En effet, si l'évaluation du mot nous montrera deux cas différents :

— Le nombre de lettres bien placées est supérieur ou égal au nombre de lettres mal placées. Alors dans ce cas, on préfère appliquer la méthode de remplacement, car en effet, plus on a de lettres bien placées, plus on est proche du mot secret, ainsi, il serait plus intéressant de faire des remplacements des caractères pour tente de converger vers le mot secret.

— Le nombre de lettres bien placées est inférieur ou égal au nombre de lettres mal placées. Dans ce cas, on privilégiera l'échange des caractères, car l'objectif dans cette situation serait plutôt de trouver les bonnes positions pour les lettres mal placées.

Et ces opérations sont répétées plusieurs fois pour que la mutation puisse nous renvoyer un mot assez différent des parents et de la population courante pour agrandir au mieux notre ensemble. Et ensuite, on applique la fonction *getClosestWord* pour obtenir un mot valide.

3.6 Tests

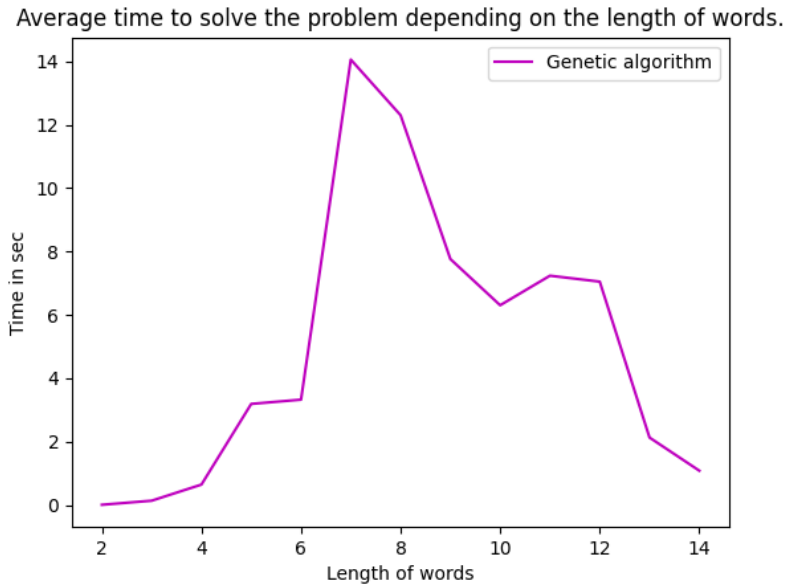

FIGURE 3 – Evolution du temps d'exécution en fonction de la longueur des mots pour l'algorithme génétique

Ce graphique montre le temps moyen d'exécution de l'algorithme génétique en fonction de la longueur d'un mot. On peut noter que contrairement aux algorithmes vu en **Partie 1** l'exécution reste en temps polynomial. Le temps croit jusqu'en $n \approx 8$ et décroît par la suite. Cela peut être expliqué par la fréquence moins importante de mot de plus de 8 lettres.

4 Partie 3 : Détermination de la meilleur tentative

Dans cette partie, on s'intéresse à la sélection du prochain mot à tester pour réduire efficacement le domaine des solutions admissibles, suite à un ensemble de tentatives déjà effectuées.

Une méthode que l'on propose serait basée sur la **fréquence** des lettres dans le dictionnaire. En effet, la fréquence d'apparition de chaque lettre n'est pas uniforme, par exemple les lettres "e", "a" sont plus fréquentes que les lettres "z", "w".

Donc cette méthode s'appuiera principalement sur la probabilité d'apparition des lettres. En parcourant la liste des mots compatibles, il cherche en calculant le mot qui a le plus de lettres avec une forte probabilité d'apparition, et le teste. Ainsi, si le mot testé est loin de la solution, alors on pourra éliminer d'avantages de domaines dans notre ensemble de solutions potentielles puisque le mot testé contient des lettres plus "fréquentes", qui apparaît plus souvent chez les autres solutions potentielles.