Projet
Wordle Mind

April 18, 2022

Binome:
Xinyu Huang 3803966
Ruohui Hu 21102304

Professeur: M.Patrice Perny

Contents

1 Introduction

Dans ce projet, on s'intéresse à modéliser le jeu wordle mind en appliquant les méthodes de satisfaction de contraintes et un algorithme génétique:

1.1 Principe du jeu.

Le jeu consiste pour le joueur à deviner le mot(wordle).Pour obtenir l'information, chaque fois le joueur porpose un mot du dictionnqire et le programme lui indique combien de mot bien placés et mal placés.

Le but de ce projet et de chercher à découvrir le mot secret en utilisant le moins d'essais possibles.

1.2 Différents algorithmes appliqués.

- CSP
  - A1:retour arrière chrnologique
  - A2:retour arrière chrnologique avec arc cohérence
- Algorithme génétique
- Algorithmes proposés
  - Méthode CSP Probabiliste
  - Méthode CSP *triche*

1.3 Fonctions en commun.

- **readfile(file)** : à partir le fichier dicto.txt, générer une dictionnaire dont les indices sont n(nombre de lettre) et les valeurs sont la liste des mots de ce longueur.
- **check_correct(instance, word)** : comparer la différence entre le mot généré par nous-même(instance) et le mot secret(mot) en retournant le nombre de lettres bien placés et mal placé.

1.4 Question 1

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

Response :

- CSP : Le programme qui utlise les méthodes csp converge nécessairement vers la solution. Étant donnée une liste de mot, à chaque fois on retire un mot alératoirment depuis la liste et en appliquant csp et arc-consistant, on supprime certaines mots de la liste qui ne satisfait plus la domaine ou les constraintes. Donc en reproduisant une liste de taille plus en plus petit et les domaine et constraintes plus en plus resreint, on converge forcément vers la solution réelle.
- Génétique :

2 Partie 1 : Modélisation et résolution par CSP

2.1 Question 2

Présenter et expliquer vos procédures de recherche d'une solution compatible puis implanter ces procédures au coeur 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 moyens de résolution et du nombre moyen d'essais nécessaires lorsque n augmente.

2.2 Principe de l'algo CSP

- **Paramètres**
  - word : mot secret
  - list_mot : la dictionnaire(la liste de mots de même longueur que word)
  - list_domaine : pour vairable xi(lettre positionné à la position i du mot), généré son domaine depuis la dictionnaire.(Les lettres possibles pour xi)
- **Procédure**
  - Fixer la domaine à l'aide de **list_domain**
  - Générer alératoirement un mot **solution** de la **list_mot**
  - Compter le nombre de bien/mal placés par la fonction **check_correct**
  - Quand bien_place < n :
    - Si bien_place==0 alors supprime de la domaine de **xi** le valeur **solution[i]** et supprime de la liste les mots qui ne satisfont plus la domaine
    - (A2 Arc-consistant) Si bien_place + mal_place==0 alors supprime de la domaine de **xi** tous les lettres apparaisent dans le mot **solution** et supprime de la liste les mots qui ne satisfont plus la domaine
    - Générer alératoirement un mot et compet le nombre de bien/mal placés
  - retourne le nombre d'essais

2.3 A1:retour arrière chrnologique

Application de CSP: chaque fois si bien_placé égal à 0, ça veut dire tous les xi sont fausses donc on supprime les lettres de la domaine

```
def solver_a1(word, list_mot, list_domain):
    n=len(word)
    pb=Problem()
    print("Word to guess: ",word)

    for i in range(n):
        name_variable=str(i)
        pb.addVariables(name_variable,list_domain[i])

    solutions=deepcopy(list_mot)
    bien_place=1
    index=np.random.randint(solutions.shape[0])
    solution=solutions[index]
    bien_place, mal_place = check_correct(solution, word )
    solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=index
    ])
```

```
    counter=1
    true_count=0
    print(counter, solution, word, bien_place, mal_place )
    while bien_place<n:
        counter+=1
        if bien_place==0:
            for i in range(n):
                if solution[i] in pb._variables[str(i)]:
                    pb._variables[str(i)].remove(solution[i])
                    solutions=solutions[np.where(solutions[:,i]!=solution[i])]

        index=np.random.randint(len(solutions))
        solution=solutions[index]
        solutions=np.array([solutions[i] for i in range(len(solutions) ) if i!=index])
        bien_place, mal_place = check_correct(solution, word)
        print(counter, solution, word, bien_place, mal_place )
    return counter
```

Listing 1: A1:retour arrière chrnologique

2.4 A2:retour arrière chrnologique avec arc cohérence

Application de CSP: idem que A1

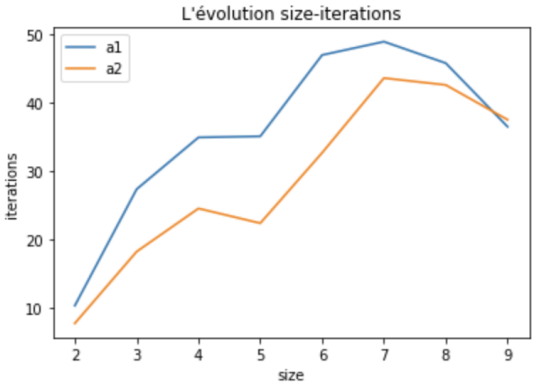
Application de arccohérence : Quand $bien\_place + mal\_place = 0$, alors pour tous les xi et xj on peut créer un arc tel que $real(x_i) \neq x_j$ et $real(x_j) \neq x_i$,qui revient à réduire la domaine.

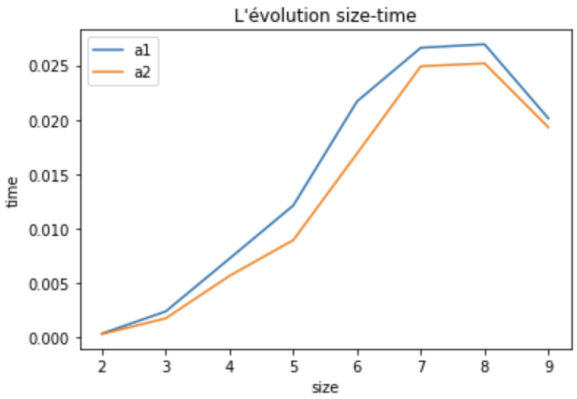
```
def solver_a2(word, list_mot, list_domain):
    ....
    ....
    line 20 du solver_a1
    while bien_place<n:
        counter+=1
        if bien_place==0:
            if mal_place==0:
                for i in range(n):
                    for j in range(n):
                        if solution[i] in pb._variables[str(j)]:
                            pb._variables[str(j)].remove(solution[i])
                            solutions=solutions[np.where(solutions[:,i]!=solution[i])]
            else:
                for i in range(n):
                    if solution[i] in pb._variables[str(i)]:
                        pb._variables[str(i)].remove(solution[i])
                        solutions=solutions[np.where(solutions[:,i]!=solution[i])]
    line 26 du solver_a1
    ...
    ...
    return counter
```

Listing 2: A2:retour arrière chrnologique avec arc cohérence

2.5 Expérimentation

On veut tester le temps moyens de détermination du mot secret aussi le nombre d'itération correspondante avec 200 exécutions de la recherche.





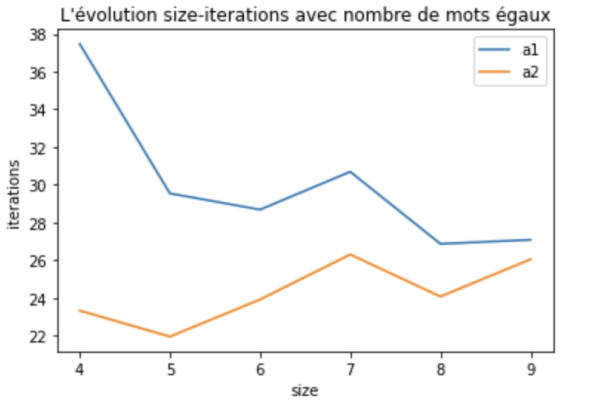
Response:

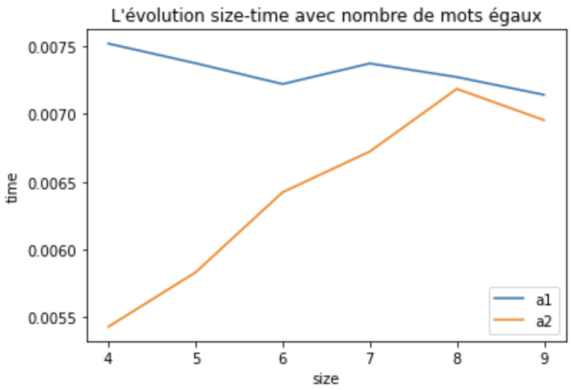
Remarque 1 :On a remarqué que l'algo a1 est bien mieux que a2 en nombre de l'itération est aussi en temps de calculs.

Il est logique car pour chaque mot chosit, a2 restreint mieux la domaine donc la taille de la liste des mots est plus petit que celle de a1. Donc théoriquement, a2 est une borne inférieure à a1 en term de nombre d'itération.

Et aussi il est logique que moins d'itérations l'algo nécessite, moins le temps de calcul nécessite pour cet algo.

Remarque 2 :On a aussi remarqué que l'évolution des deux algos n'est pas strictment croissante, donc par l'intuition, on le raisonne par le nombre de mots ne sont pas très équilibre entre les différentes taille, donc on a aussi tester sur une taille fixe qui est la taille minimale des n de 4 à 9:





Alors on peut bien remarqué la différence de ces deux figures par rapport aux celles qui ne fixent pas le nombre de mots. On peut donc en conclure que il n'existe pas de relation croissante entre n et le nombre d'itérations.

Remarque 3 :Une autre chose par l'intuition et que je puisse remarque est que pour n de valeur de plus en plus grand, les deux algos se convergent en itérations et en temps.

Cette propriété peut être expliqué par le fait que : plus n est grande, plus de lettre on va engendrer dans un mot donc il est plus probable que $nb\_malplace \neq 0$ donc les deux algos sont égaux donc cette situation.

3 Partie 2 : Modélisation et résolution par algorithme génétique

3.1 Question 3

Coder un algorithme génétique dans lequel le choix de la prochaine tentative se fera aléatoirement parmi l'ensemble E. Fixer les paramètre de probabilité de mutation, taille de E, taille d ela population et nombre de générations afin d'obtenir des résultats de bonne qualité en un temps acceptable. Etudier l'évolution du temps moyen de résolution et du nombre moyen d'essais nécessaires lorsque n augmente et comparer avec les résultats obtenus pour la partie 1. Représenter sous forme de graphique les résultats obtenus.

3.2 Algorithme génétique

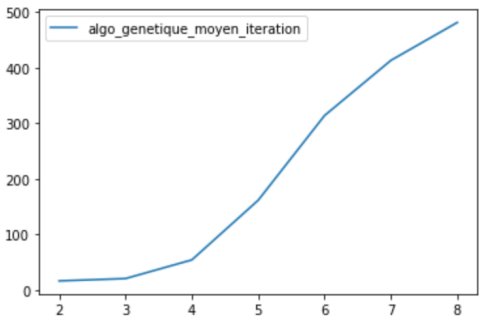
3.3 Principe de l'algo genetique

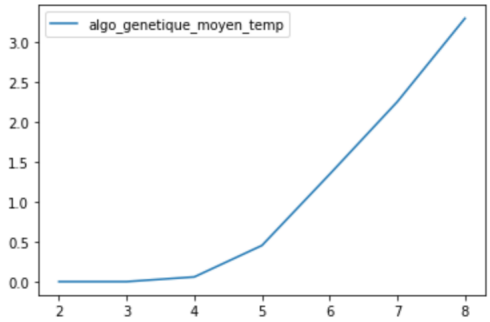
- **Paramètres**
  - word : mot secret
  - list_mot : la dictionnaire (la liste de mots de même longueur que word)
  - list_domaine : pour vairable x_i (lettre positionné à la position i du mot), généré son domaine depuis la dictionnaire.(Les lettres possibles pour x_i)
  - Probabilite de mutation : $p\_mutation$
  - Probabilite de mutation_aleatoire : $p\_mutation\_alea$
  - Probabilite de mutation_echange : $p\_mutation\_echange$
  - Probabilite de mutation_inverse : $p\_mutation\_inverse$
  - Taille de E : $\_mu$
  - Taille de la population : $\_lambda$
  - Nombre de generation : $nb\_generation$
  - Taille maximum de E : $MAXSIZE$
  - Taille maximum de generation : $MAXGEN$
  - Le temp maiximum pour exécuter : $MAXTEMP$
  - Une fonction d'evaluation : $fitness(str, str) \rightarrow float$
  - Un operateur de selection des parents : $selection(list, 1) \rightarrow list$
  - Un operateur de croisement: $croisement\_fond(str, str, int) \rightarrow list1, list2$
  - Trois operateurs de mutation:
    - $aleatoire(str) \rightarrow new\_str$: changement aléatoire d'un caractère,
    - $echange(str) \rightarrow new\_str$: échange entre deux caractère
    - $inverse(str) \rightarrow new\_str$: inversion de la séquence de caractères entre deux positions al éatoires
- **Procédure**
  - Fixer la domaine à l'aide de **list_domain**
  - **Initialisation de la population** :

    - Générer alératoirement un mot **solution** de la **list_mot**
    - Compter les fitness par la fonction **fitness**
    - If fitness = 0 : supprimer de la domaine de **Xi** de le valeur **solution[i]** Et refaire la generation.
    - : reprter **_mu** fois retourne l'ensemble parents E
- **Selection :** La probabilité d'un mot d'être selectionné comme parent sera proportionnelle à **Fitness**
- **Operation :**
  pour chaque generation, générer **_lambda fils f1** grâce aux opérations de croisement et de mutations:
    - if **f1** est dans parents, refaire l'evoluation.
    - if **fitness(f1,word)** = 0: supprimer de la domaine de **Xi** de le valeur **solution[i]** Et refaire la generation.
    - repeter jusqu'à **nb_generation**
- **Arrêter** Tant que treouver la mot secret OU timeout soit atteint OU maxsize pour la population OU maxgen pour la generation

3.4 Expérimentation

On veut tester le temps moyens de détermination du mot secret aussi le nombre d'itération correspondante avec 20 exécutions de la recherche.





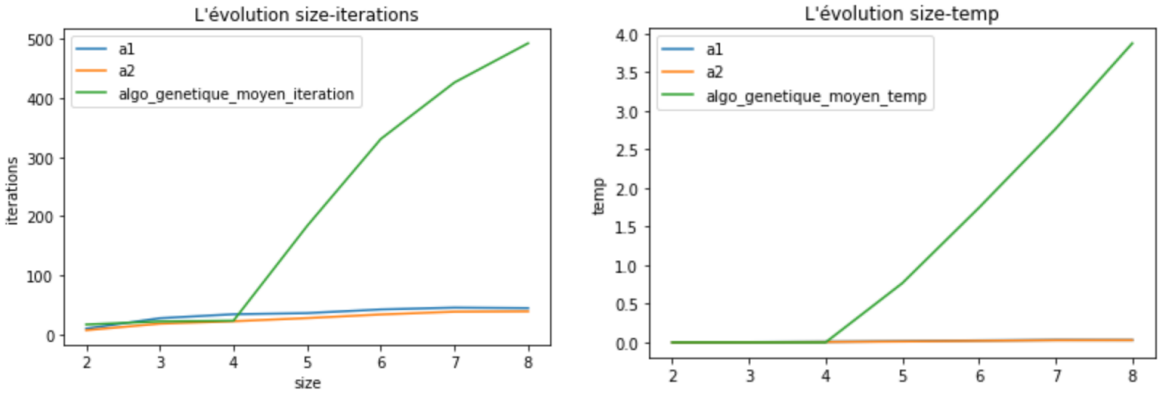
Response:

Remarque 1 :On a remarqué que l'évolution de cette algo est strictement croissante, le temp moyenne et nombre d'iteration augmente quand le taille de mot augmente

Il est logique car pour chaque mot chosit, comme la taille de mot augmente , il faut chercher plus de taille de la domaine donc il faut plus de temp et iteration.

3.5 Comparaison des algos CSP(a1_a2_gene)

On va afficher les figures de temp_moyen et nb_iter



Response:

Quand la taille de mot est petite, algorithme genetique est plus rapide que les algos CSP. Mais quand la taille augmente,l'algo genetique prends plus de temp et plus d'iteration.

4 Partie 3 : détermination et résolution de la meilleure tentative

Il peut être intéressant d'évaluer a priori la valeur informative d'une tentative(essai d'un mot) pour réduire efficacement a priori l'utilité d'une tentative donnée, à la suite de celles déjà éffectuées. Une telle évaluation peut être utilisée pour choisir la meilleure tentative dans une population de solutions compatibles engendrée par l'algorithme génétique. On peut aussi modifier l'algorithme de la partie 1 pour engendrer plusieurs solutions compatibles avec l'information disponible et choisir parmi elles la meilleure. Dans les deux cas, on cherchera à évaluer à quel point la sélection de la meilleure tentative peut accélérer la résolution du problème.

4.1 Méthode CSP Probabiliste

Pour pouvoir récuperer le mot qui porte plus d'information, on veut chosir chaque fois le meillere mot dans notre algorithme, donc même algo que a2 sauf la sélection du mot.

Principe :

- **Paramètres**
  - word : mot secret
  - list_mot : la dictionnaire (la liste de mots de même longueur que word)
  - list_domaine : pour vairable x_i (lettre positionné à la position i du mot), généré son domaine depuis la dictionnaire.(Les lettres possibles pour x_i)
- **Fonctions**
  - **generate_probabiliste_dict(list_mot, n)** qui génère une dictionnary qui retourne la valeur probabiliste pour un mot donné comme indice.
  - **find_index(list_mot,mot)** retourne l'indice pour un mot dans la liste

- **Procédure**
  - idem que algo2 sauf la partie sélection

4.2 Méthode CSP *triche*

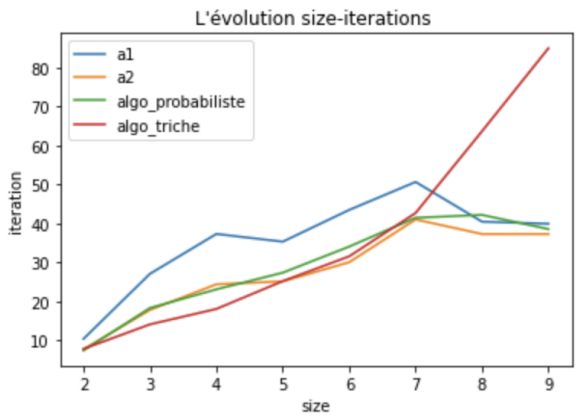
Cet algorithme est intérdit dans ce projet car il teste des combinaisons incompatibles, mais il est toujours intéréssant de comparer entre les différents algo.
Principe :

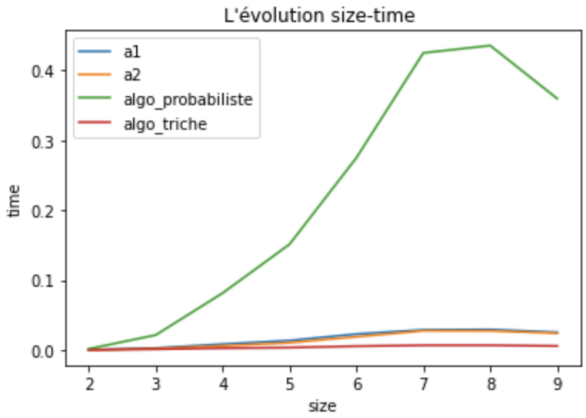
- **Paramètres**
  - word : mot secret
  - list_mot : la dictionnaire (la liste de mots de même longueur que word)
  - list_domaine : pour vairable x_i (lettre positionné à la position i du mot), généré son domaine depuis la dictionnaire.(Les lettres possibles pour x_i)
- **Fonctions**
  - **find_bien_place(solution, n, count_bien, list_index, word)** qui génère une dictionnary qui retourne la valeur probabiliste pour un mot donné comme indice.
  - **correct_mal_place(solution, n, count_bien, count_mal, list_index, word)** retourne l'indice pour un mot dans la liste
- **Procédure**
  - Fixer la domaine à l'aide de **list_domain**
  - Générer alératoirement un mot **solution** de la **list_mot**
  - Compter le nombre de bien/mal placés par la fonction **check_correct**
  - Quand bien_place < n :
    - Si bien_place > indice_fixe alors cherche l'index de la bonne lettre avec la fonction **find_bien_place**
    - Si mal_place > 0 alors corriger le mot pour obtenir le vrai index de la lettre.
    - Générer alératoirement un mot et compter le nombre de bien/mal placés
  - retourne le nombre d'essais

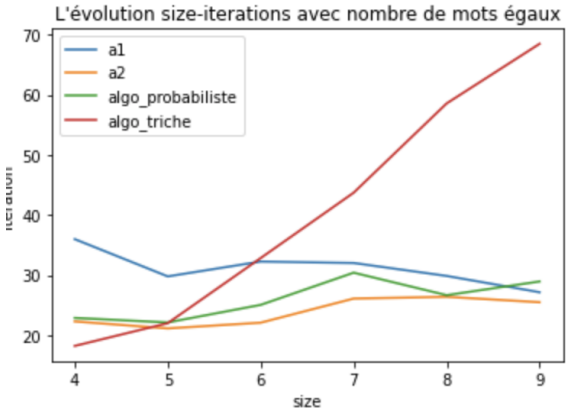
4.3 Comparaison des algos CSP

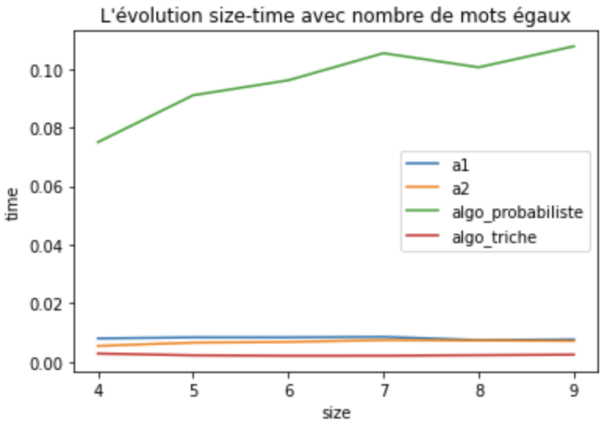
On va afficher les figures de deux cas:

- sans fixer le nombre de mots
- fixer le nombre de mots









Remarque

- On a remarqué que l'algo_triche est très bon pour n petit, mais quand n devient plus grand, il devient le plus mauvais algo.
- On a remarqué que le nombre d'itérations nécessite pour l'algo probabiliste est très proche de algo2, mieux que algo1. Quand n augmente, les trois algos convergent et l'algo_triche augmente et devient le plus mauvais.
- On a remarqué que l'algo probabiliste est le plus mauvais en term de temps consommé. Et l'algo triche est le meilleur.

Conclusion:

- Préférence de l'algo csp: algo2 > algo triche(n petie) > algo1 = algo probabiliste
- Dans notre cas, algo probabiliste n'améliore pas notre vitesse car plus la valeur de probabilité est élevé pour un mot, plus il y a de chance que le valeur bien_placé soit supérieur à 0, ce qui récompense l'information de ce algo nous apporte.

5 Annexe

5.1 Fonctions en commun

```
def readfile(file):
    lines=[]
    dic_file=dict()
    with open(file) as f:
        for line in f:
```

```
            word=[i for i in line.rstrip()]
            lg_word=len(word)
            if lg_word in dic_file:
                dic_file[lg_word].append(word)
            else:
                dic_file[lg_word]=[word]
    return dic_file
```

Listing 3: readfile

```
def check_correct(instance, word):
    bien_place=0
    mal_place=0
    list_not_match1=[]
    list_not_match2=[]
    for i in range(len(instance)):
        if instance[i]==word[i]:
            bien_place+=1
        else:
            list_not_match1.append(instance[i])
            list_not_match2.append(word[i])
    for letter in list_not_match1:
        if letter in list_not_match2:
            list_not_match2.remove(letter)
            mal_place+=1
    return bien_place, mal_place
```

Listing 4: check$_{c}orrect$

5.2 Fonctions de la partie 1

```
def solver_a1(word, list_mot, list_domain,render=0):
    n=len(word)
    pb=Problem()


    for i in range(n):
        name_variable=str(i)
        pb.addVariables(name_variable,list_domain[i])

    solutions=deepcopy(list_mot)
    bien_place=1
    index=np.random.randint(solutions.shape[0])
    solution=solutions[index]
    bien_place, mal_place = check_correct(solution, word )
    solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=index
    ])
    counter=1
    true_count=0
    if render==0:
        print("Word to guess: ",word)
        print(counter, solution, word, bien_place, mal_place )
    while bien_place<n:
        counter+=1
        if bien_place==0:
            for i in range(n):
                if solution[i] in pb._variables[str(i)]:
                    pb._variables[str(i)].remove(solution[i])
                    solutions=solutions[np.where(solutions[:,i]!=solution[i])]

        index=np.random.randint(len(solutions))
        solution=solutions[index]
        solutions=np.array([solutions[i] for i in range(len(solutions) ) if i!=index])
        bien_place, mal_place = check_correct(solution, word)
        if render==0:
            print(counter, solution, word, bien_place, mal_place )
    return counter
```

Listing 5: A1:retour arrière chrnologique

```python
def solver_a2(word, list_mot, list_domain,render=0):
    n=len(word)
    pb=Problem()

    for i in range(n):
        name_variable=str(i)
        pb.addVariables(name_variable,list_domain[i])

    solutions=deepcopy(list_mot)
    bien_place=1
    index=np.random.randint(solutions.shape[0])
    solution=solutions[index]

    bien_place, mal_place = check_correct(solution, word )
    solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=index
    ])
    counter=1
    if render==0:
        print("Word to guess: ",word)
        print(counter, solution, word, bien_place, mal_place )
    while bien_place<n:
        counter+=1
        if bien_place==0:
            if mal_place==0:
                for i in range(n):
                    for j in range(n):
                        if solution[i] in pb._variables[str(j)]:
                            pb._variables[str(j)].remove(solution[i])
                            solutions=solutions[np.where(solutions[:,j]!=solution[i])]
            else:
                for i in range(n):
                    if solution[i] in pb._variables[str(i)]:
                        pb._variables[str(i)].remove(solution[i])
                        solutions=solutions[np.where(solutions[:,i]!=solution[i])]

        index=np.random.randint(len(solutions))
        solution=solutions[index]
        solutions=np.array([solutions[i] for i in range(len(solutions) ) if i!=index])
        bien_place, mal_place = check_correct(solution, word)
        if render==0:
            print(counter, solution, word, bien_place, mal_place )

    return counter
```

Listing 6: A2:retour arrière chrnologique avec arc cohérence

5.3 Fonctions de la partie 2

5.4 Fonctions de l'algo probabiliste

```python
def generate_probabiliste_dict(list_mot, n):
    nb_dict=dict()
    pb_dict=dict()
    total=len(list_mot)
    for i in range(n):
        nb_dict[i]=dict()
    for mot in list_mot:
        for i in range(n):
            letter=mot[i]
            if letter in nb_dict[i]:
                nb=nb_dict[i][letter]
                nb_dict[i][letter]=nb+1
            else:
                nb_dict[i][letter]=1
    for index, mot in enumerate(list_mot):
        pb_dict[str(mot)]=np.sum([ np.log(nb_dict[i][mot[i]]/total) for i in range(n)
    ])
    return pb_dict

```

```python
def find_index(list_mot,mot):
    for index,m in enumerate(list_mot):
        if str(m)==str(mot):
            return index
```

Listing 7: Fonction generate et find_index

```python
def solver_csp_probabiliste(word, list_mot, list_domain, render=0):
    n=len(word)
    pb=Problem()
    for i in range(n):
        name_variable=str(i)
        pb.addVariables(name_variable,list_domain[i])

    solutions=deepcopy(list_mot)
    bien_place=1
    pb_dict=generate_probabiliste_dict(list_mot, n)
    index=max(pb_dict, key=pb_dict.get)
    del pb_dict[index]
    index=find_index(solutions,index)
    solution=solutions[index]

    bien_place, mal_place = check_correct(solution, word )
    solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=index
    ])
    counter=1
    if render==0:
        print("Word to guess: ",word)
        print(counter, solution, word, bien_place, mal_place )
    while bien_place<n:
        counter+=1
        change=0
        if bien_place==0:
            change=1
            if mal_place==0:
                for i in range(n):
                    for j in range(n):
                        if solution[i] in pb._variables[str(j)]:
                            pb._variables[str(j)].remove(solution[i])
                            solutions=solutions[np.where(solutions[:,j]!=solution[i])]
            else:
                for i in range(n):
                    if solution[i] in pb._variables[str(i)]:
                        pb._variables[str(i)].remove(solution[i])
                        solutions=solutions[np.where(solutions[:,i]!=solution[i])]
        if change==1:
            pb_dict=generate_probabiliste_dict(solutions, n)
        index=max(pb_dict, key=pb_dict.get)
        del pb_dict[index]
        index=find_index(solutions,index)
        solution=solutions[index]
        solutions=np.array([solutions[i] for i in range(len(solutions) ) if i!=index])
        bien_place, mal_place = check_correct(solution, word)
        if render==0:
            print(counter, solution, word, bien_place, mal_place )
    return counter
```

Listing 8: Algo probabiliste

5.5 Fonctions de l'algo triche

```python
  def solver_triche(word, list_mot, list_domain, render=0):
    def find_bien_place(solution, n, count_bien, list_index, word):
        counter=0
        for i in range(n):
            if i not in list_index:
                avant=solution[i]
                solution[i]="_"
                bien_place_new,_ =check_correct(solution, word)
```

```
                counter+=1
                if bien_place_new<count_bien:

                    solution[i]=avant
                    return i,counter
        print("Erreur : func_find_bien_place")

    def correct_mal_place(solution, n, count_bien, count_mal, list_index, word):
        counter=0
        for i in range(n):
            if i not in list_index:
                for j in range(n):
                    if j not in list_index:
                        temp=solution[j]
                        solution[j]=solution[i]
                        counter+=1
                        bien_place_new, mal_place_new =check_correct(solution, word)
                        if bien_place_new>count_bien :#and <count_mal:
                            return j,bien_place_new,mal_place_new,counter
                        solution[j]=temp
    n=len(word)
    pb=Problem()
    solutions=deepcopy(list_mot)
    for i in range(n):
        name_variable=str(i)
        pb.addVariables(name_variable,list_domain)

    index=np.random.randint(len(solutions))
    solution=solutions[index]
    solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=index
    ])
    counter=1
    bien_fix=0
    list_index=[]

    bien_place=0
    while bien_place<n:

        bien_place, mal_place = check_correct(solution, word)
        if bien_place==0:
            if mal_place==0:
                for i in range(n):
                    for j in range(n):
                        if solution[i] in pb._variables[str(j)]:
                            pb._variables[str(j)].remove(solution[i])
                            solutions=solutions[np.where(solutions[:,j]!=solution[i])]
            else:
                for i in range(n):
                    if solution[i] in pb._variables[str(i)]:
                        pb._variables[str(i)].remove(solution[i])
                        solutions=solutions[np.where(solutions[:,i]!=solution[i])]
        if bien_place==n:
            print(counter, solution, word, bien_place, mal_place )
            break

        while bien_place>bien_fix:
            ind,count=find_bien_place(solution, n, bien_place, list_index, word)
            list_index.append(ind)
            solutions=solutions[np.where(solutions[:,ind]==solution[ind])]
            counter+=count
            bien_fix+=1
        bien_place, mal_place = check_correct(solution, word)
        if bien_place==n:
            print(counter, solution, word, bien_place, mal_place )
            break
        while mal_place>0:
            j,bien_place,mal_place_new,count=correct_mal_place(solution, n, bien_place
    , mal_place,list_index, word)
            list_index.append(j)
            bien_fix+=1
            solutions=solutions[np.where(solutions[:,j]==solution[j])]
```

```
            counter+=count
            mal_place=mal_place_new
        index=np.random.randint(len(solutions))
        solution=solutions[index]
        solutions=np.array([ solutions[i] for i in range(solutions.shape[0] ) if i!=
    index])
        bien_place, mal_place = check_correct(solution, word)
        counter+=1
        if render==0:
            print(counter, solution, word, bien_place, mal_place )
    return counter
```

Listing 9: algo triche