Projet RP

HUYNH Tan Khiem

Muyang Shi

Introduction

Question 1:

- Après chaque mots proposé, on réduit le domaine des mots compatibles. Donc on va forcement converger vers le mot caché.

Partie 1

Question 2:

- Pour commencer, on définie une fonction `get_words`, qui va prendre en paramètre le fichier `dico.txt` et retourner une liste contenant tous les mots possibles dans ce fichier.

- En suite, on a besoin une fonction `compare_string` qui prendre en paramètres un mot proposé et le mot caché et retourne le nombre de caractères bien placées ainsi que le nom de caractères mal placées.

In [5]:
```
from constraint_prog import *
compare_string("tarte", "dette")
```

Out[5]:
```
(2, 1)
```

- La fonction `generate_constraint` va générer des contraintes possibles, à partir d'un mot et le nombre de caractères bien et mal placées dans ce mot

In [6]:
```
generate_constraint("tarte", 2, 1)
```

Out[6]:
```
[{'right': [('t', 0), ('a', 1)], 'wrong': [('t', 3)], 'not_in': ['r', 'e']},
 {'right': [('t', 0), ('r', 2)], 'wrong': [('t', 3)], 'not_in': ['a', 'e']},
 {'right': [('t', 0), ('t', 3)], 'wrong': [('a', 1)], 'not_in': ['r', 'e']},
 {'right': [('t', 0), ('t', 3)], 'wrong': [('r', 2)], 'not_in': ['a', 'e']},
 {'right': [('t', 0), ('t', 3)], 'wrong': [('e', 4)], 'not_in': ['a', 'r']},
 {'right': [('t', 0), ('e', 4)], 'wrong': [('t', 3)], 'not_in': ['a', 'r']},
 {'right': [('a', 1), ('r', 2)], 'wrong': [('e', 4)], 'not_in': ['t']},
 {'right': [('a', 1), ('t', 3)], 'wrong': [('t', 0)], 'not_in': ['r', 'e']},
 {'right': [('a', 1), ('e', 4)], 'wrong': [('r', 2)], 'not_in': ['t']},
 {'right': [('r', 2), ('t', 3)], 'wrong': [('t', 0)], 'not_in': ['a', 'e']},
 {'right': [('r', 2), ('e', 4)], 'wrong': [('a', 1)], 'not_in': ['t']},
 {'right': [('t', 3), ('e', 4)], 'wrong': [('t', 0)], 'not_in': ['a', 'r']}]
```

- On peut constater que grâce à cette fonction, une constraint qui n'est pas possbile comme {'right': [('a', 1), ('r', 2)], 'wrong': [('t', 0)], 'not_in': ['t', 'e']} ne va pas être générée

- La fonction `check_compatible` va tester si une instanciation est consistante ou pas, grâce à Forward Checking. A partir d'une instanciation, on va d'abord trouver tous les mots contenant cette instanciation dans la liste de mots qu'on a crée. Puis on va itérer sur tous les mots déjà proposé et on va éliminer les mots qui ne sont pas compatibles, grâce à la fonction `generate_constraint`. En fin, si la liste de mots compatibles n'est pas vide, on réduit les domaines de variables qui ne sont pas encore instanciées et on retoure True (pour dire que l'instanciation courante est consistante) et les nouveaux domaines

- La fonction `backtracking` implante l'algorithme de Retour Arrière Chronologique avec Forward Checking. On va instancier les caractères de gauche à droit séquentiellement, après chaque instanciation on élimine les domaines de caractères pas instanciées, jusqu'à trouver un mot compatible avec tous les mots déjà proposés.

- En fin, la fonction `solve_wordle` prendre en arguments un mot secret, une list de mots possibles et va itérativement proposer des mots compatibles en faisant appel à `backtracking` pour trouver le mot caché. Elle commence avec une liste de mots déjà proposés vide, après chaque essai le mot proposé va être ajouté à cette liste et l'essai après `backtracking` va être appelée avec cette liste pour prendre en compte ce nouveau mot.

Expérimentation:


n_csp

t_csp

Partie 2

Question 3:

- Opérateur de mutation: échanger deux caractères aléatoirement avec un mutation rate égale 0.001
- Opérateur de croisement: avec deux parent x et y, choisir aléatoirement un point π, puis à définir élément fils z comme étant $z_i = x_i$ pour $i \leq \pi$ et $z_i = y_i$ pour $i \geq \pi$
- Si ces deux opérateur produisent les mots interdits, on choisira le mot le plus proche dans le dictionnaire au sens d'une distance étant définie comme le nombre de différentes caractères entre deux mots
- La "fitness" est définie comme l'inverse de nombres d'incompatibilités avec les essais précédents. On va pondérer la population par la fitness de chaque individu divise par la fitness totale et la sélection les parents se fait proportionnelle à cette pondération

- La fonction `GA` implante l'algorithme génétique avec une taille de population fixée à 100, un nombre de génération fixé à 100, la probabilité de mutation égale 0.001, la taille de l'ensemble E fixée à 20. On itère jusqu'à atteindre le nombre maximum de génération ou la taille maximum de E.
- La fonction `solve_wordle_GA` est comme `solve_wordle`, la seule différence est que à chaque essai, le mot compatible va être trouvé avec une évolution stratégie, au lieu d'un algorithme de satisfaction de constraintes

Expérimentation:


n_ga

t_ga


n_csp_vs_ga

t_csp_vs_ga

Partie 3:

On prose quelques méthodes pour évaluer à priori l'utilité d'une tentative donnée comme suivant:

Idée 1:

- On évalue chaque mot par la somme des fréquences de chaque caractère individuel dans ce mot. Par exemple: $score[\text{" adieu "}] = freq[\text{" a "}] + freq[\text{" d "}] + \ldots$
- Avec cette évaluation, notre stratégie sera essayer de proposer les mots avec les caractère les plus fréquentes. Comme ça, à chaque essai, on a plus de chance d'approcher le mot caché.

Idée 2:

- L'un des inconvénients de l'approche ci-dessus est qu'elle va favoriser les mots qui répètent des lettres à haute fréquence. Par exemple, "alala" a un score plus haut parce qu'il a 3 "a". Mais c'est pas ce qu'on veut parce que ça manque de couverture.
- Le vrai problème est qu'on évalue des lettres dans toutes les positions de manière égale où nous savons que la position compte. Au lieu de cela, on peut calculer des fréquences de caractères par position au lieu de fréquences globales. Ensuite, on peut évaluer les mots comme suit: $score[\text{" adieu "}] = freq[(\text{" a "}, 0)] + freq[(\text{" d "}, 1)] + \ldots$
- Autrement dit, le score pour "adieu" est le nombre de fois que on a vu "a" en 0-ème position plus le nombre de fois que l'on a vu "d" en 1-ère position, etc.
- Cette idée a encore un problème. Suppose qu'on trouve que "a" a la fréquence la plus élevée. On essaie un mot avec "a" et on constate qu'il existe bien dans le mot caché. Mais on ne peux pas faire grande chose avec cette nouvelle information parce que "a" est une lettre à haute fréquence et on n'est pas en mesure d'éliminer beaucoup d'autres mots. Pour résoudre ce problème, on va essayer de choisir le mot qui contient plus "d'information"

Idée 3:

- Pour résoudre le problème au dessus, on va en fait utiliser la notion d'entropy.
- Pour calculer d'entropy, d'abord on va normaliser toutes les fréquences en divisant par la longeur de notre dictionnaire: $p[(\text{" a "}, 0)] = freq[(\text{" a "}, 0)]/len(dict)$
- Puis on calcule l'entropy as: $entropy[x] = p[x] * (1 - p[x])$
- Cette valeur is plus élevée quand $p[x] \approx 0.5$. C'est à dire on essaie de trouver un mot "au milieu", qui va nous aider à mieux couvrir la distribution des mots.
- Donc le score d'un mot sera: $score[\text{" adieu "}] = entropy[(\text{" a "}, 0)] + entropy[(\text{" d "}, 1)] + \ldots$

In [ ]:
```
```