IFT – 3335 : **TP1- Sodoku** **Hiver-2018**

Nom : Boumediene Boukharouba

Nom : Ahmed Mohammed

RAPPORT

Question1 :

Formulation du problème de Norvig :

Notion d'état : une configuration donnée du jeu (une grille de 9x9 séparée en 9 carrés de 9 cases)

État de départ : une configuration où la grille est initialisé avec certaines valeur (de 1 à 9)

État but : une configuration du jeu tel que toutes les cases sont rempli avec un nombre de 1 à 9 et qu'il n y a pas 2 chiffres identiques dans un même carré (3X3) , une ligne ou une colonne.

État Successeur : Une configuration de jeu donnée. Une assignation d'un nombre de 1 à 9 à un carrée de la grille qui était vide dans la configuration (état) précède.

Coût d'étape : Non nécessaire

Question 2 :

Description de notre implantation :

Nous avons d'abord initialisé un compteur *counter* à zéro et nous l'incrémenterons à chaque assignation d'une valeur, même si cette dernière n'est pas la bonne, ce qui met en évidence le nombre de nœud exploré. Nous avons préféré d'incrémenter le *counter* pour tous les nœuds exploré et non que pour les bons nœuds afin de afin d'illustré le temps en pire cas. (le nœud correspond à un état successeur) (ex : counter ++ lorsqu'on trouve un conflit)

```python
def assign(values, s, d, count=0):
    """Eliminate all the other values (except d) from values[s] and propagate.
    Return values, except return False if a contradiction is detected."""
    global counter
    counter += 1
    other_values = values[s].replace(d, '')
    if all(eliminate(values, s, d2) for d2 in other_values):
        return values
    else:
        return False
```

Aussi on a changé l'implémentation de la fonction solve (dans la solution du Norvig) et on a ajouté des fonctionnalités pour mettre le compteur à 0 avec chaque nouvelle grille passé et aussi on a ajouté des fonctionnalités pour imprimer la grille résultante et le nombre des nœuds explorés.

```python
def solve(grid):
    global counter
    values = search(parse_grid(grid))
    print("\n")
    if values != False:
        display(values)
    print("\nNoeuds Explorés : ", counter, "\n----------------------------------\n")
    counter = 0

    return values
```

Question 3 :

L'implémentation de l'algorithme Hill-Climbing pour ce problème ne fut pas Difficile du moment où nous avions, dans la description du cours, une description de l'implémentation. De ce fait, nous avons défini une fonction *Calculate_hill* qui prend en paramètre values, conflicts, tuples, counter , qui va choisir une valeur aléatoire pour chaque cases en tenant compte des valeurs possibles et cela via la méthode *initialize_randomly* qui prend en paramètre les values et les square_peers, square_peers étant une liste de List qui met en évidence les carrée 3x3.

Square_peers :

```
[['A1', 'A2', 'A3', 'B1', 'B2', 'B3', 'C1', 'C2', 'C3'], ['A4', 'A5', 'A6', 'B4', 'B5', 'B6', 'C4', 'C5', 'C6'], ['A7', 'A8', 'A9
', 'B7', 'B8', 'B9', 'C7', 'C8', 'C9'], ['D1', 'D2', 'D3', 'E1', 'E2', 'E3', 'F1', 'F2', 'F3'], ['D4', 'D5', 'D6', 'E4', 'E5', 'E
6', 'F4', 'F5', 'F6'], ['D7', 'D8', 'D9', 'E7', 'E8', 'E9', 'F7', 'F8', 'F9'], ['G1', 'G2', 'G3', 'H1', 'H2', 'H3', 'I1', 'I2', '
I3'], ['G4', 'G5', 'G6', 'H4', 'H5', 'H6', 'I4', 'I5', 'I6'], ['G7', 'G8', 'G9', 'H7', 'H8', 'H9', 'I7', 'I8', 'I9']]
```

Initialize_randomly fut facilement implémenté, car cela ne nécessité que de faire quelques vérification on parcourant la grille en utilisant deux boucles for (voir code)

Ensuite nous avons, tel que veut la description de l'algorithme, calculer le nombre de conflit après chaque swap (swap, recalcule, deswap) et nous avons choisis celui qui a le moins de swap. Enfin, via une simple vérification, on choisit si le prochain état est meilleur que l'état courant ou bien si nous sommes en présence d'un maximum local.

```
    if min_conflict < conflicts:
        values[swap[0]], values[swap[1]] = values[swap[1]], values[swap[0]]
    return values, min_conflict, swap, counter
```

Si le nombre minimal de conflits est inférieur è celui qu'on a calculé en avant on applique le swap

```
    if sum > conflicts:
        display(values)
        print("Minimum Local\nConflits : ", conflicts, "\nNoeuds Explorés : ", counter,
              "\n---------------------------------\n")
        return False
```

Sinon on est dans le cas d'un maximum local et on retourne False après faire l'affichage de nombre des nœuds explorés et le nombre de conflits

QUESTION 4 : expliqué comment on a implémenté le recuit simulé

L'implémentation de l'algorithme du recuit simulé consiste à prendre une valeur T qui représente la température initialisée avec 0.5. Aussi, on prend 2 valeurs aléatoire (non fixés) d'un des square peers de la grille (choisits aléatoirement) après l'initialisation de la grille et calculer le score de la grille (nombre des valeurs unique dans chaque rangée et colonne de la grille « current_score »). Et on applique un swap entre ces 2 valeurs puis on recalcule le score pour la grille après le changement "candidate_score".

```
def search_simulated_annealing(values):
    square_peers = [cross(rs, cs) for rs in ('ABC', 'DEF', 'GHI') for cs in ('123', '456', '789')]
    fixed_values = [k for k, v in values.items() if len(v) == 1]

    values = initialize_randomly(values, square_peers)
    current_score = calculate_score(values)
    conflicts = calculate_conflicts(values)

    T = 0.5
    counter = 0

    while conflicts != 0 and counter < 100000:
        new_values = values.copy()
        peer = random.choice(square_peers)
        peer = list(set(peer) - set(fixed_values))

        if len(peer) > 0:
            ind_1 = random.choice(peer)
            peer_copy = peer.copy()
            peer_copy.remove(ind_1)
            ind_2 = random.choice(peer_copy)

            new_values[ind_1], new_values[ind_2] = new_values[ind_2], new_values[ind_1]
            candidate_score = calculate_score(new_values)
```

Ensuite, on calcule la différence entre le candidate_score et le current score et on l'utilise pour calculer la température par rapport au current_score

```
if math.exp(-diff / T) - random.random() > 0:
    values = new_values
    current_score = candidate_score
```

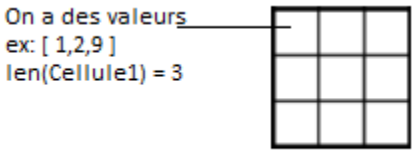
Si la valeur est supérieure à 0 alors on a un meilleur résultat avec un meilleur score. Alors, on fixe le swap qu'on a fait et on met le candidate_score comme current_score. Ensuite, on réduit la température T légèrement après la multiplier par 0.9999

On a aussi implémenté un compteur pour calculer le nombre des étapes fait et si jamais on dépasse 100000 on sort de la boucle et retoure False après l'affichage de nombre de conflits.

Question 5 :

L'Heuristique utilisé tend à améliorer le temps d'exécution de l'algorithme ainsi que le résultat. Tout d'abord l'heuristique implémenté consiste à :

- On a que, pour chaque grille 3X3 (unit), 9 cellules où chacune d'elle contient des valeurs possibles.



On va calculer l'ensemble des length de chaque cellule et les additionner ensemble et cela pour chaque unit.

$$\text{Ex : Unit\_1} = \Sigma \text{ len(cellule(i))} \quad \text{tel que } 0< i < 10 \quad i \in \mathbb{N},$$

On fait cela pour les neuf units puis on crée une list afin de trier ces derniers de façon ascendant .

On initialise des valeurs aléatoires pour chaque cellule.

Nous utilisons la liste trié pour traiter la prochaine Unit (celle qui a le moins de valeurs possible dans l'ensemble de ses cellules) (car le nombre de possibilité pour assigner une valeur à une cellule sera le plus petit ainsi accélérer le temps d'exécution)

On boucle sur les Unit

- On génère tous les swaps possibles sur la Unit et aussi,
- Pour chaque swap on calcule le score

- Lorsque c'est fait, on assigne le swap dont le score est le plus petit (cela met en évidence la configuration qui a le plus de valeur unique en tenant compte de la colonne et et rangé en question, cela règlera le problème de conflit plus rapidement)

Aussi, si nous sommes en présence d'un minimum on choisit une autre cellule/swap randomly

QUESTION 6 :

Comparaison entre les algorithmes après les fonctionner sur 100 configurations aléatoires de départ :

Algorithme / Comparaison	Norvig's depth first search	Hill Climbing	Simulated Annealing	Heuristique
Pourcentage	Entre 98 et 100 %	0 et 1%	Entre 92 et 95%	100%
Nombre moyen des nœuds explorés total	37652	136880	1341322	828819
Nombre total des nœuds explorés	380	1383	13549	8372
Temps moyen d'exécution (sec)	0.01	2.47	3.86	1.57
Temps maximal d'exécution (sec)	0.02	3.28	29.59	11.68

L'algorithme de Norvig semble supérieur aux autres algorithmes car il est moins compliqué, beaucoup plus rapide, explore beaucoup moins de nœuds et donne un résultat presque optimal dans la majorité des cas. Mais, Il existe des cas ou l'algorithme de Norvig deviant beaucoup moins rapide que les algo (simulated annealing et heuristique) comme le cas de la grille hard1. Dans ce cas les résultats sont les suivants :

Algorithme / Comparaison	Norvig's depth first search	Hill Climbing	Simulated Annealing	Heuristique
Nombre total des nœuds explorés	7282622	11895	7407	2664
Temps d'exécution (sec)	94.7	2.62	2.17	1.05

L’algo du Hill climbing est le seul qui n’a pas trouvé le bon résultat mais les 2 autres ont surperformés l’algo de Norvig.