Compte rendu SAE S2.02 : Exploration Algorithmique

1)Analyse du problème

1.1) Sujet

Il s'agit d'écrire un programme qui permet de déterminer toutes les solutions pour le problème des 8-dames avec un algorithme de type "backtracing". Ce problème consiste à placer 8 dames sur un échiquier sans que chacune se menacent. On commencera le problème avec un échiquier vide. L'utilisateur pourra tout de même saisir la taille de l'échiquier supérieur à 3 puisque le problème n'est pas possible pour un échiquier de taille trois par trois mais aussi pour avoir la possibilité de résoudre le problème pour n dames.

1.2) Analyse Descendante du problème

La résolution du problème général que représente le problème des huit dames peut être effectué sous la décomposition suivante :

- Réflexion pour aborder le sujet
- Résolution dans un premiers temps avec un algorithme naïf
- Analyse des problèmes rencontré avec ce programme
- Optimisation de la vérification de la position des dames
- Résolution du problème par la mise en place de l'algorithme de "backtracing"
- Affichage du nombre de solutions pour un échiquier de taille n par n et la possibilité d'afficher tous les échiquiers si l'utilisateur le souhaite

2.1) Analyse Fonctionnelle

2.1.1) Analyse

On souhaite résoudre le problème des 8 dames mais aussi pour n dames avec n un entier supérieur à 3.

Les points critiques :

- Alerte l'utilisateur s'il saisit une taille inférieure ou égale à 3
- Tester le temps de résolution pout un n grand
- Optimiser la vérification des positions des dames
- Optimiser le "Backtracing" à l'aide du récursif pour éviter un temps de résolution trop grand

C'est pour mettre en place ce Backtracing qu'on peut utiliser le principe du récursif qui utilise ce "Retour en arrière".

2.2) Présentation des Algorithmes

- Les Algorithmes présent ci-dessous sont le fruit d'une recherche continu qui a abouti à l'algorithme souhaité dans l'analyse descendante du problème.

2.2.1 : (Ethan Collin) Algorithme naïf :

```
import time
def solve_n_queens(n):
    # Initialiser la liste des combinaisons possible
    liste_solutions = []
    # Initialiser la liste de placements possibles pour
    placements = [[] for i in range(n)]
    for i in range(n):
        for j in range(n):
            # Ajouter toutes les positions possibles po
            placements[i].append((i, j))

    # Trouver toutes les combinaisons possibles de plac
    all_combinations = cartesian_product(placements)

    # Trouver la première combinaison valide de placeme
    for combination in all_combinations:
        if is_valid(combination):
            liste_solutions.append(combination)
    return liste_solutions
```

Cet algorithme va résoudre le problème des n dames en testant toutes les solutions c'est à dire qu'il va tester toutes les combinaisons de n dames pour trouver celles qui sont valide avec la fonction "is_valid()". Cette fonction de validation va vérifier toutes les paires de dame avec la fonction "conflit". Il y a aussi l'utilisation de la fonction "cartesian_product" qui va trouver toutes les combinaisons possibles de placement de chaque reine et renvoie une liste de tuple, chaque tuple représentant une combinaison de placement de reines. Cependant il est clair que "solve_n_queens" va donc posséder un temps de résolution assez conséquent lorsque le paramètre n est grand.

2.2.2 : (Desperrois Lucas) :

```
def placer_reine(lig,echequier,n,l):
                                                    #boucl
    for col in range(n):

        echequier[lig][col]=1;
        if(valide_position(echequier,lig,col)):
                                                    #si on
                                                    #on ap
            if(lig<n-1):

                placer_reine(lig+1,echequier,n,l);
            else:
                                                    # lors
                                                    # au c
                l[0]+=1;



        echequier[lig][col]=0;                      #si la
    return l[0];                                    # le c
```

Explication du récursif :

- Cette fonction placer_reine est chargé de placer les reines sur chaque ligne. Elle boucle donc sur les n colonnes de l'échiquier et placer alors la dame sur la première colonne possible. Puis elle s'appelle elle-même en récursif(placer_reine) pour la ligne L+1. Et dès que la 8 -ème dame est placée c'est bon on ajoute un au compteur du nombre de solution(à l'indice 0 du tableau l). Mais on ne s'arrête pas là car la 8 -ème instance continuer à évaluer les 8 colonnes possibles. Pus ça remonté à la 7-ème qui est toujours dans la boucle et passe à la colonne C+1 de la ligne 7. Si c'est bon ça réappelle la 8-ème instance pour tester cette position et on réitère la chose tant qu'il y a des solutions.

Explication valide_position:

```
def valide_position(grille,lig,col):
    i=lig;
    j=col;
    ok=True;
    n=len(grille);



    #vérification est

    while(j+1<n and ok==True):
        j=j+1;
        if(grille[i][j]==1):
            ok=False;

    j=col;
    #verification sud
    while(i+1<n and ok==True):
        i=i+1;
        if(grille[i][j]==1):
            ok=False;
    i=lig;
```

- Valide_position s'occupe de tester si la position de la dame qu'on souhaite poser est valide dans l'échiquier (tableaux deux dimensions). Elle utilise des boucle "while" pour vérifier dans chaque direction par rapport à la position testée. Il y a donc 8 vérifications (8 directions).

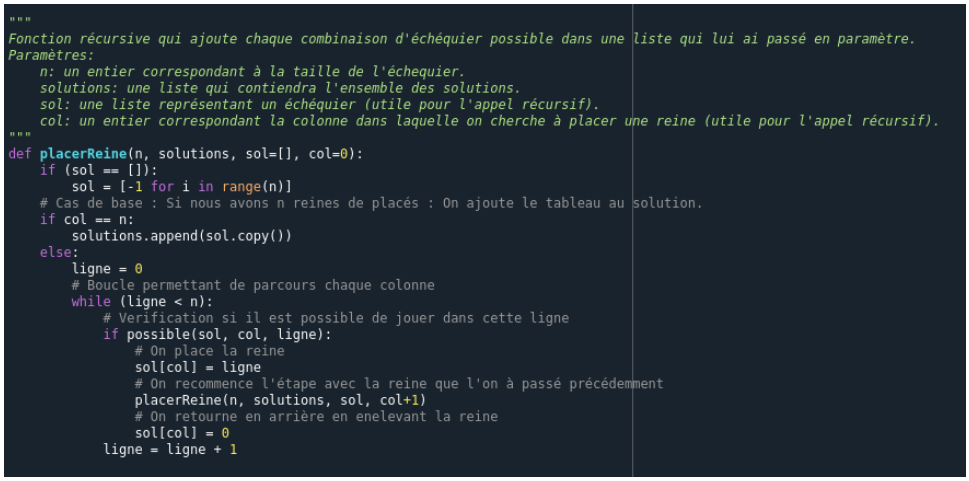
2.2.3 : (Levacher Ethan) :

Explication de la fonction : possible :

```
"""
Fonction boolean renvoyant True si il est possible de jouer la reine sans qu'elle sois en danger et false dans le cas contraire.
Paramètres:
    s: le tableau contenant les reines actuellement placé/
    ligne: un entier qui correspond à la ligne où l'on souhaite jouer notre reine.
    colonne: un entier qui correspond à la colonne où l'on souhaite jouer notre reine.
"""
def possible(s, ligne, colonne):
    jouable = True
    l = 0
    # On verifie pour chaque reine tant qu'il reste des reines et que c'est toujours jouable.
    while (l < ligne and jouable):
        # on récupère la colonne de la reine l
        c = s[l]
        # Verification le l est sur la même diagonale que la reine que l'on souhaite joué.
        # Et également si leurs colonne est différentes.
        jouable = (abs(ligne-l) != abs(colonne-c) and colonne != c)
        l = l + 1
    return jouable
```

- la fonction possible prend en paramètre une liste de solutions actuelle ainsi que la ligne et la colonne dans laquelle on souhaite placer une nouvelle reine et renvoie un booléen, true si la reine est jouable, false sinon, puis pour chaque colonne, on vérifie si la nouvelle reine est en danger. Pour cela on vérifie si elle se trouve sur la même diagonale, où la même colonne.

Explication de la fonction : placerReine :



```
"""
Fonction récursive qui ajoute chaque combinaison d'échéquier possible dans une liste qui lui ai passé en paramètre.
Paramètres:
    n: un entier correspondant à la taille de l'échequier.
    solutions: une liste qui contiendra l'ensemble des solutions.
    sol: une liste représentant un échéquier (utile pour l'appel récursif).
    col: un entier correspondant la colonne dans laquelle on cherche à placer une reine (utile pour l'appel récursif).
"""
def placerReine(n, solutions, sol=[], col=0):
    if (sol == []):
        sol = [-1 for i in range(n)]
    # Cas de base : Si nous avons n reines de placés : On ajoute le tableau au solution.
    if col == n:
        solutions.append(sol.copy())
    else:
        ligne = 0
        # Boucle permettant de parcours chaque colonne
        while (ligne < n):
            # Verification si il est possible de jouer dans cette ligne
            if possible(sol, col, ligne):
                # On place la reine
                sol[col] = ligne
                # On recommence l'étape avec la reine que l'on à passé précédemment
                placerReine(n, solutions, sol, col+1)
                # On retourne en arrière en enelevant la reine
                sol[col] = 0
            ligne = ligne + 1
```

- la fonction placerReine est une fonction récursive, elle prend en paramètre n, qui est le nombre de reine qui doit être placé, solutions, la liste dans laquelle elle stockera l'ensemble des solutions, sol, la liste des reines déjà placé, ce paramètre est utilisé lors de la récursivité, si aucune liste est donnée par défaut la liste sera vide et col, un entier correspondant à la colonne dans laquelle on souhaite placer la reine actuellement, par défaut il s'agit de la colonne 0.

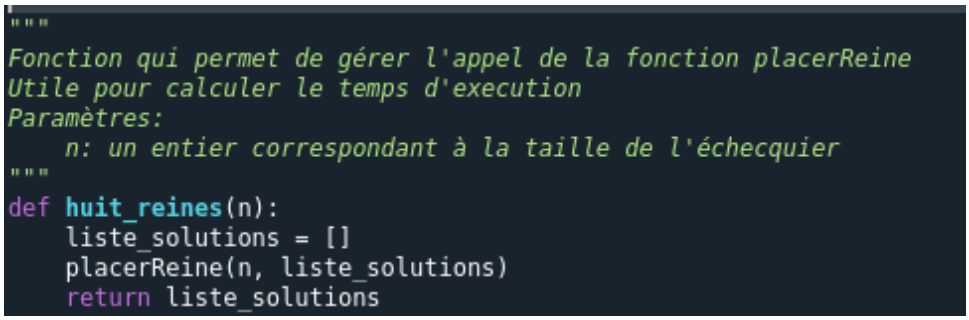
Si le paramètre sol est vide alors on initialise une liste de "-1" permettant de savoir qu'une reine n'est pas placé car –1 ne correspond à aucune case.

Le case de base de cette récursivité est lorsque la colonne (col) où l'on souhaite placer notre reine correspond à la taille du plateau, il n'y a donc plus de reine à placé, on ajoute donc la liste actuelle à l'ensemble des solutions.

Sinon on vérifie s'il est possible de placer une reine dans chaque ligne, si c'est le cas, on place la reine et on rappelle récursivement cette même fonction en passant en paramètre la liste avec la reine en plus ainsi qu'en incrémenter la colonne dans laquelle on souhaite placer une nouvelle reine.

Puis on réinitialise la liste d'avant pour l'analyse des lignes suivante. (Il s'agit du backtracing).

Explication de la fonction : huit_reines :



```
"""
Fonction qui permet de gérer l'appel de la fonction placerReine
Utile pour calculer le temps d'execution
Paramètres:
    n: un entier correspondant à la taille de l'échecquier
"""
def huit_reines(n):
    liste_solutions = []
    placerReine(n, liste_solutions)
    return liste_solutions
```

- Cette fonction permet de gérer l'appel de la fonction placerReine plus facilement, et de renvoyer la liste des solutions de façons plus simples.

2.3 Comparaison des Algorithmes

Avant même d'avoir une analyse on peut déjà comparer la complexité (le nombre d'opérations) des deux types d'algo que nous avons utilisé pour résoudre le problème. La complexité de l'algorithme de résolution brut (algorithme d'Ethan Collin, algorithme naïf) possède une complexité de $n^2/(n!(n^2-n)n!)$ ce qui donne pour 8 dames 4 426 165 368 opérations. Alors pour l'algorithme de GAUSS (algorithme de Lucas Desperrois et Ethan Levacher) la complexité est n! qui permet d'avoir 40 320

opérations pour résoudre le problème avec 8 dames. On observe donc déjà une grande différence considérable au niveau du nombre d'opérations effectués.

Maintenant comparons les temps d'exécutions pour le problème avec 8 dames :

Pour le programme d'Ethan Collin, qui teste l'entièreté des possibilités de damiers possible :

```
92 solutions on été trouvé.
Temps d'execution de la fonction solve_n_queens pour 8 dames :
31.124531745910645
```

On remarque que le programme est très long, 31s, c'est énorme pour un algorithme plutôt basic.

Pour le programme de Lucas Desperrois, qui utilise le backtracing mais avec une vérification plus couteuse :

```
92 solutions on été trouvé.
Temps d'execution de la fonction placer_reine pour 8 dames : 0.02932453155517578
```

Enfin pour le programme de Ethan Levacher, qui utilise le backtracing avec une vérification moins couteuse :

```
92 solutions on été trouvé.
Temps d'execution de la fonction huit_reines : 0.008275508880615234
```

3) Les difficultés rencontrés

1)Analyse du problème :

- Nous avons rencontré tout d'abord des problèmes pour comprendre les enjeux du sujet. On a rencontré des difficultés à mettre en placer un algorithme optimal qui utilise le récursif. C'est pour cela que nous avons choisi de tous d'abords écrire un algorithme "naïf" pour mieux saisir le problème des n-dames. Qui nous a permis de rédiger un code bien plus optimal à la suite de cela.

2) Utilisation du Backtracing :

- Pour comprendre le principe du "Backtracing" il a fallu que notre groupe effectue des recherches pour s'approprier la méthode pour le problème que nous devions résoudre. C'est notamment à l'aide de vidéos et de schémas que nous avons pu saisir toute la complexité du retour en arrière. Le récursif a été donc été un choix optimal pour mettre en place le Backtracing puisque c'était quelque chose qu'on maitrisait assez bien.

3) La comparaison des algorithmes :

- Il a été compliqué de trouver par quel procédé nous pouvions comparer nos algorithmes. Mais avec de la recherche finalement nous avons pu tout d'abords trouver des informations essentielles sur la complexité de nos travaux. Puis aussi nous avons découvert des bibliothèques python qui permettaient notamment de comparer nos temps d'exécution et qui est un caractère primordial de comparaison d'algorithmes.