Démonstration automatique de théorème

Nicolas Daire

Table des matières

1 Analyse

1.1 Présentation du problème

Sur un échiquier de dimensions $n \times n$, on veut placer n dames sans qu'elles ne se menacent mutuellement, en suivant les règles des échecs, càd sans qu'il n'y ait 2 dames sur la même ligne, la même colonne ou la même diagonale.

Tout d'abord, nous pouvons constater qu'il y a une unique solution pour $n = 1$, et aucune pour $n = 2$ et $n = 3$. Nous pouvons alors être tentés de nous demander si, $\forall n \geqslant 4$, il existe au moins une solution. Ensuite, nous pouvons nous intéresser au nombre de solutions correspondant à un n donné, et plus précisément au nombre de solutions distinctes, ne pouvant pas être obtenues par rotation et/ou symétrie les unes par rapport aux autres. Nous appellerons solution *essentielle* une solution retenue pour représenter une classe.

1.2 Types utilisés

Un échiquier de taille $n \times n$ étant analogue à un tableau bidimensionnel, nous utiliserons le type *list* en Python. Pour une plus grande commodité, plutôt que de garder les notations classiques correspondant aux échiquiers, nous utiliserons une numérotation de 0 à $n - 1$ à la fois sur les lignes et les colonnes, en partant du coin supérieur gauche.

Une solution sera représentée par une liste de longueur n, le i-ème élément j correspondant à une dame aux i-ème ligne, j-ème colonne.

Nous utiliserons également des listes, de booléens cette fois, pour indiquer si les colonnes et les diagonales (droite et gauche) contiennent déjà une dame. Ces listes seront notées respectivement C, Dd et Dg, et seront de taille n pour la première, $2n-1$ pour les deux autres, le 0 correspondant à la colonne de gauche, au coin inférieur gauche, et au coin inférieur droit. Une dame placée en (i, j) est donc sur la diagonale droite $n - i + j - 1$, et sur la diagonale gauche $2n - i - j - 2$.

Les autres variables et notations seront introduites au fur et à mesure.

1.3 Existence d'une solution pour $n \geqslant 4$

1.3.1 Démonstration

Cas où $n \not\equiv 2\,[6]$ et $n \not\equiv 3\,[6]$

Soient i et j la ligne et la colonne d'une dame. Nous allons montrer que la disposition décrite ci-dessous est toujours solution. Il s'agit d'une symétrie centrale, dans laquelle l'échiquier est parcouru par 2 *escaliers* formés par des déplacements du cavalier dans la même direction, le premier partant de la case $(1, 0)$.
Pour $n \in \mathbb{N}^*$ tq $n \equiv 0[6]$ ou $n \equiv 4[6]$,

$$i = \begin{cases} 2j + 1 & \text{pour } j < \frac{n}{2} \\ 2j - n & \text{pour } j \geqslant \frac{n}{2} \end{cases}$$

En remplaçant dans les expressions de Dd et Dg,

$$Dd = \begin{cases} n - j - 2 & \text{pour } j < \frac{n}{2} \\ 2n - j - 1 & \text{pour } j \geqslant \frac{n}{2} \end{cases}$$

$$Dg = \begin{cases} 2n - 3j - 3 & \text{pour } j < \frac{n}{2} \\ 3n - 3j - 2 & \text{pour } j \geqslant \frac{n}{2} \end{cases}$$

j étant différent pour tous les éléments d'une même moitié verticale de l'échiquier, les dames d'une même moitié ne peuvent se confronter.
Supposons par l'absurde que 2 dames de moitiés différentes se menacent.
Selon Dd :

$$n - j_1 - 2 = 2n - j_2 - 1$$

$$j_2 - j_1 = n + 1 > n - 1$$

Selon Dg :

$$2n - 3j_1 - 3 = 3n - 3j_2 - 2$$

$$j_2 - j_1 = \frac{n+1}{3} \notin \mathbb{N}$$

Il y a contradiction dans les 2 cas.
Cette disposition est donc bien une solution.

Pour $n \in \mathbb{N}^*$ tq $n \equiv 1[6]$ ou $n \equiv 5[6]$, montrons qu'il suffit de prendre la solution décrite précédemment pour $n-1$, et de rajouter une dame dans le coin supérieur droit. Cela revient à vérifier que Dg correspondant à $(0, n-1)$, soit $n-1$ est libre.
Supposons par l'absurde qu'une dame occupe cette diagonale.
Si $j < \frac{n}{2}$:

$$2n - 3j - 3 = n - 1$$
$$j = \frac{n-2}{3} \notin \mathbb{N}$$

Si $j \geqslant \frac{n}{2}$:

$$3n - 3j - 3 = n - 1$$
$$j = \frac{2n-1}{3} \notin \mathbb{N}$$

La diagonale est bien libre.

Cas où $n \equiv 2\,[6]$ ou $n \equiv 3\,[6]$

Pour $n = 6k+2$, $k \in \mathbb{N}^*$, une confrontation a lieu quand $j_2 - j_1 = 2k+1$ dans la disposition précédente. Il existe néanmoins des solutions systématiques, dont nous allons seulement donner la formule, mais qui se démontrent de manière analogue. Il s'agit encore d'une symétrie centrale, dans laquelle l'échiquier est divisé en 4 parties. Dans celle la plus à gauche, l'escalier part de la dernière case de la moitié supérieure de l'échiquier. Dans la suivante, l'escalier commence à la ligne 0 ou à la ligne 1, selon la parité.

Pour $n = 6 \times 2l + 2$, $l \in \mathbb{N}^*$,

$$i = \begin{cases} 2j + \frac{n-2}{2} & \text{pour } j < \frac{n+2}{4} \\ 2j - \frac{n+2}{2} & \text{pour } \frac{n+2}{4} \leqslant j < \frac{n-2}{2} \\ 2j - \frac{n-2}{2} + 1 & \text{pour } \frac{n-2}{2} \leqslant j < \frac{3n-4}{4} \\ 2j - \frac{n-2}{2} - n + 1 & \text{pour } j \geqslant \frac{3n-4}{4} \end{cases}$$

Pour $n = 6 \times (2l+1) + 2$, $l \in \mathbb{N}^*$,

$$i = \begin{cases} 2j + \frac{n-2}{2} & \text{pour } j < \frac{n+2}{4} \\ 2j - \frac{n+4}{2} + 1 & \text{pour } \frac{n+2}{4} \leqslant j < \frac{n-2}{2} \\ 2j + \frac{n+4}{2} - n & \text{pour } \frac{n-2}{2} \leqslant j < \frac{3n-4}{4} \\ 2j - \frac{n-2}{2} - n + 1 & \text{pour } j \geqslant \frac{3n-4}{4} \end{cases}$$

Pour $n \equiv 3\,[6]$, il suffit de rajouter une dame dans le coin supérieur droit de l'échiquier agrandi.

On peut donc proposer un algorithme qui explicite une solution pour un n donné.

```
def explicite(n):
        n_2 = n // 2
        etat = [0] * n
        r = n % 6
        if r != 2 and r != 3:
```

```
        for i in range(n_2):
                etat[i] = 1 + 2 * i
                etat[n_2 + i] = 2 * i
else:
        m = n if n % 2 == 0 else n - 1
        for i in range(n_2):
                j = (2 * i + n_2 - 1) % m
                etat[i] = j
                etat[m - 1 - i] = m - 1 - j
if r % 2 == 1:
        etat[-1] = n - 1
return etat
```

1.3.2 Vérification

Le programme suivant permet de vérifier en temps linéaire si une disposition est solution ou non. Pour chaque dame, les trois listes (colonne et 2 diagonales) sont mises à jour si la case n'était pas menacée. Dans le cas contraire, cela indique que 2 dames se menacent, donc la disposition n'est pas solution. Si toute la liste a pu être parcourue, la vérification est établie.

```
def estSolution(l):
    n=len(l)
    C=[False]*n
    Dd=[False]*(2*n-1)
    Dg=[False]*(2*n-1)
    for i,j in enumerate(l):
        if not C[j] and not Dd[n-i+j-1]\
          and not Dg[2*n-i-j-2]:
            C[j]=True
            Dd[n-i+j-1]=True
            Dg[2*n-i-j-2]=True
        else:
            return False
    return True
```

2 Résolution étape par étape

2.1 Recherche exhaustive des solutions

Dans un premier temps, nous allons rechercher l'ensemble des solutions pour un n donné en entrée. Pour cela, il est nécessaire de mettre à jour les statuts des colonnes et diagonales (libres ou occupées) au fur et à mesure que l'on parcourt l'échiquier.

2.1.1 Gestion de l'échiquier

ajoutReine modifie en temps constant le statut de la colonne et des diagonales liées à une case si cette dernière n'était pas menacée, et renvoie un booléen indiquant si l'opération a pu être effectuée. La modification de la liste R sera expliquée dans la prochaine sous-section.

retireReine modifie en temps constant le statut des directions liées à une case occupée, la rendant libre.

remplitLigne, en $O(n)$, ajoute la première reine qui peut être placée dans la ligne, et renvoie un booléen indiquant ci cela a été possible.

```
def ajoutReine(i,j):
    if not C[j] and not Dd[n-i+j-1] and not Dg[2*n-i-j-2]:
        C[j]=True
        Dd[n-i+j-1]=True
        Dg[2*n-i-j-2]=True
        R[i]=j
        return True
    return False

def enleveReine(i,j):
    C[j]=False
    Dd[n-i+j-1]=False
    Dg[2*n-i-j-2]=False

def remplitLigne(i,j):
    while j<n:
        if ajoutReine(i,j):
            return True
        j+=1
    return False
```

2.1.2 Parcours de l'échiquier

Le n souhaité est saisi, et les autres variables sont initialisées. c représente le nombre de solutions, S la liste des solutions, R une liste qui indique pour chaque ligne jusqu'à quel emplacement les solutions éventuelles ont été testées (-1 si l'exploration de la ligne n'a pas commencée pour la configuration en cours). Le parcours commence à la première ligne, et se poursuit en remplissant les lignes suivantes si possible, en reculant dans le cas contraire, puis en réavançant dès que possible. Si l'on a pu arriver jusqu'à la dernière ligne en rajoutant des reines, on ajoute la solution à la liste des solutions, puis on continue sur cette même ligne. La liste R permet de progresser dans une ligne, en testant les possibilités des lignes suivantes, puis de réinitialiser la ligne quand il n'est plus possible de poursuivre. Le parcours s'arrête quand la dernière possibilité

a été explorée, soit à l'extrémité de la dernière ligne de l'échiquier, soit après avoir reculé jusqu'au bout de la ligne 0. Ce parcours se fait hélas en $O(n!)$.

```
n=int(input())
c=0
S=[]
R=[-1]*n
C=[False]*n
Dd=[False]*(2*n-1)
Dg=[False]*(2*n-1)

i=0
while R[-1]<=n-1 and i>=0:
    if remplitLigne(i,R[i]+1):
        if i==n-1:
            c+=1
            S.append(R[:])
            enleveReine(n-1,R[n-1])
        else:
            i+=1
    else:
        R[i]=-1
        if i!=0 and R[i]!=n-1:
            enleveReine(i-1,R[i-1])
        i-=1
```

Le même algorithme peut être écrit en récursif, le code est fourni en complément.

2.2 Tri des solutions

Une fois les solutions obtenues, il est intéressant de les trier afin d'obtenir les solutions essentielles, qui nous permettront par ailleurs de générer les autres en cas de besoin.

2.2.1 Classe d'équivalence d'une solution essentielle

L'on peut définir la classe d'équivalence d'une solution comme l'ensemble des solutions pouvant être obtenues par rotation et/ou symétrie les unes des autres. Son cardinal ne peut dépasser 8. Les 3 fonctions ci-dessous sont en $O(n)$.

rotation réalise une rotation dans le sens horaire d'une disposition de dames.

symétrie réalise une symétrie axiale par rapport à l'axe vertical.

classe génère la classe d'équivalence d'une disposition sans doublons, par rotations successives, puis une symétrie, et enfin les rotations de cette symétrie.

```
def rotation(m):
    r=[0]*len(m)
    for i,j in enumerate(m):
        r[j]=len(m)-i-1
    return(r)

def symetrie(m):
```

```
    s=[len(m)-1-i for i in m]
    return s

def classe(m):
    C=[m[:]]
    r=rotation(m)
    for i in range(3):
        if not r in C :
            C.append(r)
        r=rotation(r)
    s=symetrie(m)
    for i in range(4):
        if not s in C:
            C.append(s)
        s=rotation(s)
    return(C)
```

2.2.2 Obtention des solutions essentielles et des représentants

tri permet d'obtenir une liste ne contenant qu'un seul représentant par classe d'équivalence, en comparant chaque solution avec les membres des classes d'équivalences des solutions précédentes, et en ajoutant finalement la solution à la liste des solutions essentielles si elle ne correspondait à aucune classe. Cette fonction est en $O(n \times (\text{nombre de solutions})^2)$

genere recrée, à partir de la liste des solutions essentielles, l'ensemble des solutions regroupées par classes, en $O(\text{nombre de classes} \times n)$

```
def tri(S):
    if S!=[]:
        R=[S[0][:]]
        for s in S:
            i=0
            while i<len(R):
                if s in classe(R[i]):
                    i+=len(R)+1
                else:
                    i+=1
            if i==len(R):
                R.append(s)
    else :
        R=[]
    return R

def genere(R):
    T=[classe(r) for r in R]
    return T
```

2.2.3 Calcul du nombre de solutions essentielles

S'il s'agit seulement de compter le nombre de solutions essentielles, et non de les renvoyer, on peut concevoir un algorithme plus performant.

Il faut remarquer pour cela qu'une solution n'est jamais invariante par symétrie axiale, car on pourra trouver des reines qui ne sont pas sur l'axe de symétrie et qui sont envoyées sur d'autres dames suivant une ligne ou une diagonale selon le type de symétrie, ce qui correspond à des attaques.

Pour déterminer le nombre de solutions essentielles, il suffit donc de compter le nombre de solutions trouvées comptées avec leur multiciplité, et de diviser le résultat par 8, soit le nombre de permutations considérées, la multiplicité d'une solution étant ici le nombre de permutations qui laissent cette solution invariante.

Pour déterminer la multiplicité d'une solution il faut générer ses 8 permutations en $O(n)$, puis tester l'égalité pour chacunes d'elles en $O(n)$ soit une complexité globale en $O(n)$. On effectue ce traitement sur chaque solution, la complexité de l'algorithme est donc en $O(n \times$ (nombre de solutions)).

```
def nombre_essentielles(solutions):
        if not solutions:
                return 0
        if solutions == [[0]]:
                return 1
        multiplicites = 0
        for etat in solutions:
                rot = etat[:]
                for _ in range(3):
                        rot = rotation(rot)
                        if etat == rot:
                                multiplicites += 1
        return (len(solutions) + multiplicites) // 8
```

2.2.4 Résultats

Les solutions ont pu être déterminées pour $n \leqslant 15$ avec les codes Python, et les résultats sont connus jusque $n = 27$.

n	solutions	essentielles
1	1	1
2	0	0
3	0	0
4	2	1
5	10	2
6	4	1
7	40	6
8	92	12
9	352	46
10	724	92
11	2680	341
12	14200	1787
13	73712	9233
14	365596	45752
15	2279184	285053
16	14772512	1846955
17	95815104	11977939
18	666090624	83263591
19	4968057848	621012754
20	39029188884	4878666808
21	314666222712	39333324973
22	2691008701644	336376244042
23	24233937684440	3029242658210
24	227514171973736	28439272956934
25	2207893435808352	275986683743434
26	22317699616364044	2789712466510289
27	234907967154122528	29363495934315694

2.3 Affichage

Les solutions étant désormais obtenues et réparties en classes, il est souhaitable de pouvoir les afficher. Cependant, comme le nombre de solutions devient rapidement très important quand n augmente, il peut être préférable de n'afficher que les solutions essentielles, voire une partie de ces solutions si elles sont trop nombreuses. Pour cette raison, nous rajoutons un second entier à saisir en entrée, z, indiquant le nombre maximal de solutions essentielles que l'on veut voir affichées. Si de plus la condition est satisfaite, et que le nombre total de solutions est inférieur à 100, nous choisirons d'afficher l'ensemble des solutions réparties selon les classes.

2.3.1 Représentation d'un échiquier

L'affichage de figures se rapprochant d'échiquiers a été réalisée à l'aide de divers modules, apparaissant ci-dessous. echiquier est une fonction auxiliaire en $O(n^2)$ pouvant être appelée pour créer un échiquier basique, sans dames, à l'aide de rectangles de taille adaptée.

```
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

def echiquier(n,ax):
```

```
    for k in range(n):
        r=n%2
        if (k%2-r)==0:
            for l in range(0,n,2):
                ax.add_artist(matplotlib.patches.Rectangle((k/n, l/n)
                    ,2/(2*n),2/(2*n),color = 'black',fill = True))
        else:
            for l in range(1,n,2):
                ax.add_artist(matplotlib.patches.Rectangle((k/n, l/n)
                    ,2/(2*n),2/(2*n),color = 'black',fill = True))
```

2.3.2 Représentation des solutions par classes

Dans une grille ayant pour nombre de lignes le nombre de classes à afficher et 8 colonnes, on fait appel pour chaque solution à la fonction permettant de créer l'échiquier basique, puis on y rajoute les dames sous forme de cercles. Cette fonction est en $O(n^2 \times$ nombre de solutions). Les premières lignes servent simplement à bien accorder les noms et les adjectifs de la légende.

```
def representation(n,m,nb,liste):
    fig = plt.figure()
    if n==2 or n==3:
        fig.suptitle("Pour␣n={},␣aucune␣solution".format(n))
    elif n==1:
        fig.suptitle("Pour␣n=1,␣1␣unique␣solution")
    elif nb==1:
        fig.suptitle("Pour␣n={},␣{}␣solutions,␣dont␣1␣seule␣essentielle".
            format(n,m))
    else:
        fig.suptitle("Pour␣n={},␣{}␣solutions,␣dont␣{}␣essentielles".
            format(n,m,nb))
    gs = GridSpec(nb, 8)
    for i,classe in enumerate(liste):
        for j,representant in enumerate(classe):
            ax=plt.subplot(gs[i,j])
            ax.tick_params(labeltop=False,labelbottom=False,labelleft=
                False,labelright=False)
            ax.xaxis.set_visible(False)
            ax.yaxis.set_visible(False)
            ax.add_artist(matplotlib.patches.Rectangle((0, 0),1,1, color
                = 'seashell',fill = True))
            echiquier(n,ax)
            for li,co in enumerate(representant):
                ax.add_artist(matplotlib.patches.Circle((1-(li+0.5)/n, (
                    co+0.5)/n), 0.9/(2*n), color = 'red', fill=True))
    plt.show()
```

2.3.3 Représentation de solutions essentielles seulement

Cette fois-ci, les solutions sont affichées au nombre de 10 par lignes, sans répartition particulière, dans la limite du nombre choisi. Les lignes de code n'ayant d'autre utilité que de respecter les accords de la la langue française et la cohérence de la légende (*aucune, unique* etc) ne sont pas recopiées ci-dessous.

```
def representationSimplifiee(n,m,nb,z,liste):
    fig = plt.figure()
    liste=liste[:min(nb,z)]
    gs = GridSpec((len(liste)+9)//10, 10)
    for i,solution in enumerate(liste):
        ax=plt.subplot(gs[i//10,i%10])
        ax.tick_params(labeltop=False,labelbottom=False,labelleft=False,
            labelright=False)
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        ax.add_artist(matplotlib.patches.Rectangle((0, 0),1,1, color = '
            seashell',fill = True))
        echiquier(n,ax)
        for li,co in enumerate(solution):
            ax.add_artist(matplotlib.patches.Circle((1-(li+0.5)/n, (co
                +0.5)/n), 0.9/(2*n), color = 'red', fill=True))
    plt.show()
```

3 Programme complet

3.1 Programme

Les différents éléments étudiés précédemment sont rassemblés dans le programme présenté dans les pages suivantes. Les premières lignes et les dernières lignes permettent d'avoir un nombre raisonnable d'échiquiers affichés.

3.2 Exemples

Ci-dessous les entrées et soties pour deux exemples :

3.3 Complexité

Les différentes fonctions présentes dans le programme ont pour la plupart des complexités raisonnables, excepté celles concernant l'affichage. En revanche, le parcours principal, en $O(n!)$ est très limitant quant aux valeurs de n que l'on peut se permettre de demander (difficultés autour de 15 en python), mais cette complexité ne peut pas être grandement améliorée si l'on veut une recherche exhaustive des solutions.

Etudions de plus près la complexité de l'algorithme de recherche exhaustive des solutions.

Remarquons en premier lieu que le nombre de solutions au problème ne peut excéder $n!$. En effet les dames doivent se situer sur des lignes distinctes, une solution au probème est donc une permutation des colonnes d'un échiquier sur lequel on a initialement positionné les dames sur des lignes distinctes.

Si l'on considère le programme écrit en récursif, on peut faire l'hypothèse que le temps passé dans la fonction est négligeable, et que les opérations d'ajout et de délétion sur le type set sont effectuées en temps constant. Dans le cas du programme itératif, on peut similairement faire l'hypothèse que le temps passé dans la boucle interne est négligeable. La complexité de l'algorithme dépend alors directement du nombre d'états finaux atteints (lorsque l'échiquier est complètement attaqué), ce qui mène donc à une complexité en $O(n!)$.

Cependant on se rend vite compte que le nombre réel de solutions est très inférieur à $n!$, l'arbre de recherche est donc fortement élagué et la complexité réelle de l'algorithme est moindre.

Le nombre exact de solutions est difficile à déterminer, mais on peut tout de même minorer la complexité de l'algorithme. Pour cela il suffit de remarquer que lorsqu'on pose la i-ème sur la i-ème ligne de l'échiquier, celle-ci attaque au maximum 3 colonnes distinctes sur chacunes des lignes subséquentes (la colonne et les deux diagonales issues de sa position). Le nombre de colonnes sur lequelles on peut poser les dames diminue donc de 3 avec chaque nouvelle dame posée, il faut donc poser au moins $\lceil \frac{n}{3} \rceil$ dames sur les premières lignes de l'échiquier, dans le cas idéal où les dames menacent toutes des positions distinctes, pour obtenir une ligne entièrement attaquée, et ainsi aboutir à un état final de l'échiquier dans l'arbre de recherche. Le nombre d'états finaux est donc minoré par $\sum_{i=0}^{\lfloor \frac{n}{3} \rfloor - 1}(n-3i)$, qui est lui-même minoré par $\lfloor \frac{n}{3} \rfloor !$ (pour avoir une expression plus concise).

La complexité de l'algorithme est donc comprise entre 2 formes factorielles, ce qui explique pourquoi le temps d'exécution croît très rapidement lorsque n augmente.

3.4 Difficultés rencontrées

L'écriture des programmes de résolution et de tri n'a pas posé de problème particulier, et n'a pas nécessité d'astuce recherchée. En revanche, l'affichage des solutions et des échiquiers a créé quelques difficultés, par manque de connaissance des modules utilisés. De même, l'affichage d'un échiquier créé sous LaTex a nécessité la création de diverses fonctions.

4 Compléments

4.1 Autres fonctions possibles

Code alternatif de la recherche exhaustive de solutions, écrit en récursif.

```
def recherche_exhaustive(n):
        etat = [0] * n
```

```
diagonales = [[0] * 2 * n, [0] * 2 * n]
solutions = []
libres = set(range(n))
def rec(dame):
        if dame == n:
                solutions.append(etat[:])
                return

        for position in libres:
                if diagonales[0][dame + position] == diagonales
                    [1][n + dame - position] == 0:
                        libres.remove(position)
                        etat[dame] = position
                        diagonales[0][dame + etat[dame]] += 1
                        diagonales[1][n + dame - etat[dame]] += 1
                        rec(dame + 1)
                        diagonales[0][dame + etat[dame]] -= 1
                        diagonales[1][n + dame - etat[dame]] -= 1
                        libres.add(position)

rec(0)

return solutions
```

4.2 Code de l'échiquier

4.3 Exemples de problèmes liés

5 A faire

Vérifier cohérence du 2, surtout les noms des variables (un R et un c à modifier...)
Améliorer paragraphe complexité
Parler des tentatives de recuit etc dans Difficultés rencontrées
Rajouter et expliquer les programmes dans Autre fonctions possibles
Rajouter et expliquer le code de l'échiquier sous LaTex
Remplir exemples de problèmes liés (je peux m'en charger)
Finir la partie 1 : vérifier les formules et terminer le programme explicite etc (idem, après la vérif de Nath)