

DS n° 1 d'informatique

Vendredi 16 octobre 2026

T. Lemanissier

PC lycée Carnot



— Les calculatrices ne sont pas autorisées (parce que inutile pour cette épreuve).
— Une réponse doit être justifiée pour être valable.
— La rédaction, la présentation (résultats à encadrer), le soin sera pris en compte.

I Base de données d'un serveur de jeu en ligne

Le gestionnaire d'un site en ligne permettant de jouer à différents jeux a mis au point une base de données correspondant à ses besoins. Elle est constituée de 4 tables :

— La table `Clients` contient des informations sur les utilisateurs du site. Elle est constituée des attributs suivants :
  — `id_client` : identifiant unique du client, nombre entier ;
  — `nom` : nom du client, chaîne de caractères ;
  — `prenom` : prénom du client, chaîne de caractères ;
  — `email` : email du client, chaîne de caractères ;
  — `date_inscription` : date d'inscription du client au format 'JJ-MM-AAAA' ;
  — `anniversaire` : jour anniversaire du client au format 'JJ-MM'.
— la table `Jeux` contient des informations sur les jeux disponibles. Elle est constituée des attributs suivants :
  — `id_jeu` : identifiant unique du jeu, nombre entier ;
  — `nom_jeu` : nom du jeu, chaîne de caractères ;
  — `description` : brève description du jeu, chaîne de caractères ;
  — `prix` : prix du jeu, nombre flottant.
— La table `Factures` enregistre les transactions des achats de jeux. Elle est constituée des attributs suivants :
  — `id_facture` : identifiant unique de la facture, nombre entier ;
  — `id_client` : référence vers la table `Clients`, nombre entier ;
  — `id_jeu` : référence vers la table `Jeux`, nombre entier ;
  — `date_achat` : date d'achat du jeu au format 'JJ-MM-AAAA'.
— la table `Records` qui enregistre les records ou scores des différents utilisateurs dans les jeux qu'ils ont achetés. Elle est constituée des attributs suivants :
  — `id_record` : identifiant unique du record, nombre entier ;
  — `id_client` : référence vers la table `Clients`, entier ;
  — `id_jeu` : référence vers la table `Jeux`, entier ;
  — `score` : score du joueur pour ce jeu, nombre entier ;
  — `date_record` : date du record au format 'JJ-MM-AAAA'.

1. Donner une clé primaire et les clés étrangaires de la table facture.
2. Le gestionnaire veut souhaiter leur anniversaire à ses clients et leur proporser un bon d'achat àcette occasion. Écrire une requête `SQL` permettant de récupérer le nom, le prénom et l'email des clients ayant anniversaire le 3 juillet.
3. Le gestionnaire souhaite faire le récapitulatif des achats d'un client particulier. Écrire une requête `SQL` affichant le nom, le prénom et les dates des achats effectués par le lient ayant pour email 'myriam.sarr@ccinp.edu'.
4. Écrire une requête SQL qui donne les noms des jeux dont Martin Dupont a établi le record au cours de l'année 2020.

5. Le gestionnaire du site veut mettre à l'honneur les meilleurs scores pour chaque jeu, ceci sur la page d'accueil du site. Écrire un requête `SQL` affichant chaque nom de jeu avec le meilleur score pour ce jeu.
6. Le gestionnaire du site veut pouvoir identifier ses meilleurs clients. Écrire une requête `SQL` affichant le nom, l'email et le montant total dépensé pour chaque client du site qui ont dépensé plus de 200 € rangé par total dépensé décroissant.

   Attention, un client peut avoir fait plusieurs achats, il faudra donc faire la somme de tous ses achats.
7. Écrire une requête `SQL` affichant le nom, le prénom du joueur, le nom du jeu, le meilleur score du joueur à ce jeu ainsi que le record de tout les joueur à ce jeu.

II Des algorithmes pour résoudre le problème de n reines

Les jeux de société (échecs, dames, reversi, awale, etc.) sont une grande source d'inspiration pour tester l'efficacité de différents algorithmes.

Nous nous proposons ici d'en étudier plusieurs autour d'un défi issu du jeu d'échecs.

Considérons une grille de taille $n \times n$. Une reine est une pièce du jeu qui peut se déplacer du nombre de cases que l'on souhaite le long d'une ligne, d'une colonne ou d'une diagonale montante ou descendante (figure 1).

On cherche à placer `n` reines dans cette grille de telle façon qu'il n'y ait pas plus d'une reine sur chaque ligne, sur chaque colonne et sur chaque diagonale montante ou descendante[1] (figure 2).

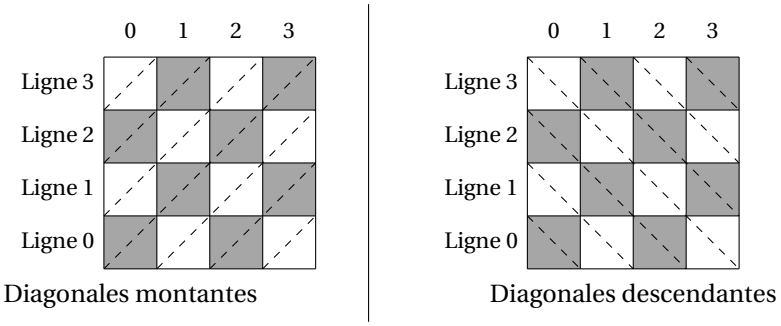


FIGURE 1 – Diagonales dans le cas $n = 4$

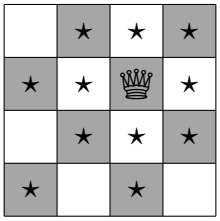


FIGURE 2 – Une reine a été placée : les cases qu'elle contrôle et qui ne peuvent pas être occupées par d'autres reines sont figurées par le symbole ⋆

II.A - Introduction

Pour $n = 3$ et donc une grille 3×3, le problème n'a pas de de solution. Deux reines suffisent à contrôler toutes les case. Il est donc impossible de placer une troisième reine tout en respectant la consigne. La figure 3 illustre cette situation dans un cas, les autres s'en déduisant par symétrie ou rotation.

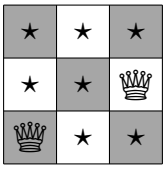


FIGURE 3 – Cas $n = 3$: deux reines contrôlent toutes les cases

Pour une grille 4×4 (figure 4) ainsi que pour des grilles de taille supérieure, il y a toujours une solution au problème étudié.

1. Ce problème a été en 1848 par Max Bezzel pour un échiquier ordinaire, partiellement résolu par Carl Friedrich Gauss, puis complètement résolu par Franz Nauck en 1850.

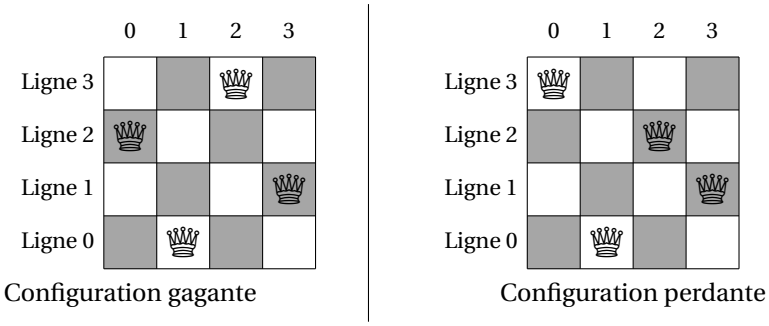


FIGURE 4 – Deux exemples de configurations pour $n = 4$

8. Expliquer pourquoi la configuration perdante de la figure 4 ne résout pas le problème.
9. On décide de coder une configuration par une liste `L` telle que `L[i]` soit égale au numéro de colonne où se trouve la reine de la ligne `i`. Par exemple, la configuration de la figure 5 sera codée par la liste `L = [0, 3, 1, 4, 2]`.

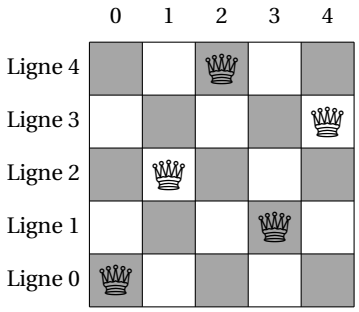


FIGURE 5 – Un exemple de configurations gagnante pour $n = 5$

Donner les listes décrivant les configurations de la figure 4.

II.B - Mise en place d'un affichage

Les solutions que nous obtiendrons dans les sous-parties suivantes seront des listes de `n` positions pour les reines (comme décrit à la sous-partie II.A). Pour aider l'utilisateur, nous souhaitons afficher la grille solution trouvée et pour cela nous implémentons quelques fonctions basiques sur les listes Python.

10. Écrire une fonction `ligne` prenant pour paramètre un entier naturel `n` et qui renvoie une liste de taille `n` constituée uniquement de 0.

    Par exemple, pour `n = 5`, l'appel `ligne(5)` renverra `[0, 0, 0, 0, 0]`.

    Remarque : l'utilisation de structures de données du module `numpy` comme `array` n'est pas autorisée.
11. Écrire une fonction `grille` prenant pour paramètre un entier `n` et qui renvoie une liste de `n` listes distinctes, chacune contenant `n` fois l'entier `0`.

    Par exemple, pour `n = 3`, l'appel `grille(3)` renverra `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`.

On considère ainsi que pour une grille `G`, représentée par une liste de listes, `G[i][j]` correspond à la case de la i-ème ligne et j-ème colonne.

12. Écrire une fonction `remplir` prenant pour paramètre une liste `L` de taille `n` représentant une configuration. Cette fonction cée une grille de taille $n \times n$ ne contenant que des zéros. Puis, elle remplace `G[i][j]` par la valeur 1 lorsque `L[i]=j`. Enfin, elle renvoie la grille ainsi obtenue.

    Remarque : une fonction ayant une complexité temporelle en `O(len(L))` (c'est-à-dire linéaire en la taille de la liste) sera valorisée.

    Par exemple, pour `n=4` :

```
>>>> G = remplir([1,3,0,2])
>>>> G
[[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0]]
```

13. La représentation des grilles produite par l'interpréteur n'est pas visuellement satisfaisante (pour l'utilisateur). Écrire une fonction `affiche` qui prend pour paramètre une grille `G`, qui ne renvoie rien mais affiche la grille `G` ligne par ligne.

    Par exemple :

```
>>>> G = remplir([1,3,0,2])
>>>> G
[[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0]]
>>>> affiche(G)
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
```

II.C - Tester si une configuration est valide

Considérons une liste `L` ne contenant que des entiers entre `0` et `n-1` inclus. On admet que cette liste de positions pour les reines résout le problème étudié pour une grille de taille $n \times n$ si, et seulement si, :

(a) tous les `L[i]` sont distincts ;

(b) tous les `L[i]+i` sont distincts ;

(c) tous les `L[i]-i` sont distincts ;

(d) cette liste `L` est de taille `n`.

14. Expliquer ce que traduit d'une part le respect de (a) et d'autre part le respect de (b) et (c) pour le problème considéré.

15. Écrire une fonction `tous_distincts` prenant en paramètre une liste `L` de positions et renvoyant `True` si tous ses éléments sont distincts, `False` sinon. Cette focntion procède comme suit :
    - on initialise un dictionnaire `presents` vide ;
    - on parcourt la liste `L`. Si on rencontre un élément `x` non encore présent parmi les clés du dictionnaire, on insère la clé `x` dans le dictionnaire avec comme valeur le booléen `True` (par exemple). Si par contre cet élément est déjà présent, on quitte la fonction en renvoyant le booléen `False` ;
    - si le parcours complet de la liste n'a pas repéré de doublon, on renvoie `True`.

Un étudiant propose le code alternatif suivant qui comporte plusieurs erreurs :

```
def tous_distincts2(L) :
    n = len(L)
    for i in range(n) :
        for j in range(i, n) :
            if L[i] == L[j] :
                return True
    return False
```

16. Écrire une version corrigée de la fonction `tous_distincts2`.

    Comparer la complexité temporelle des codes `tous_distincts` et `tous_distincts2` (corrigé) dans le pire des cas, ceci pour une liste `L` de taille `n` donnée en paramètre.

17. Écrire une fonction `est_possible` prenant en paramètre une liste `L` de positions et qui renvoie un booléen indiquant si `L` satisfait simultanément les trois conditions (a), (b) et (c), ou pas.

    On pourra utiliser plusieurs appels à la fonction `tous_distincts`.

    Par exemple, `est_possible([2, 4, 1, 7, 0, 6, 3, 5])` renvoie `True` et `est_possible([3, 4, 1, 7, 0, 6, 2, 5])` renvoie `False`.

18. Écrire une fonction `est_solution` prenant en paramètres une liste `L` de positions, un entier `n` représentant la taille de la grille et renvoyant un booléen indiquant si `L` satisfait simultanément les quatre conditions (a), (b), (c) et (d), ou pas.

    On part du principe que `L` contient uniquement des entiers de `{0,1,...,n-1}`, il est donc inutile de le vérifier.

Nous diposons maintenant d'une fonction permettant de tester si une liste de positions permet de résoudre le problème pour une grille de taille $n \times n$.

II.D - Une première approche par recherche aléatoire

Considérons une liste `L` d'éléments distincts. On appelle permutation de cette liste, toute liste de même taille contenant les éléments de `L` dans un ordre éventuellement différent.

Par exemple, si `L = [0, 1, 2]`, les permutations de `L` sont les 6 listes :

`[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0].`

Une première méthode pour trouver une configuration solution du problème des `n` reines consiste à générer aléatoirement des permutations de `[0, 1, ..., n-1]`. Chaque permutation représente une configuration possible. Il suffit alors de tester si elle est valide. Si le nombre de permutations testées est grand et que `n` ne l'est pas trop, on peut espérer trouver une solution.

19. La fonction `sample` du module `random` peut renvoyer une permutation aléatoire d'une liste `L`. Il y a plusieurs manières d'importer cette fonction. recopier et compléter les codes pour utiliser `sample` comme indiqué.

    Méthode 1 :

```
.....
rd.sample(L, len(L))
```

    Méthode 2 :

```
.....
sample(L, len(L))
```

    Méthode 3 :

```
.....
random.sample(L, len(L))
```

Par la suite, on considèrera avoir importé `sample` avec la méthode 2. Pour générer une permutation aléatoire d'une liste `L`, on saisira alors `sample(L, len(L))`.

Par exemple, si `L = [0, 1, 2, 3, 4]`, `sample(L, 5)` pourra renvoyer `[1, 0, 3, 4, 2]` ou toute autre liste permutation de `L`.

20. Écrire une fonction `recherche_aleatoire` prenant en paramètres deux entiers naturels non nuls `n` et `N`.

    Cette fonction crée d'abord la liste `L = [0, 1, ..., n-1]`. Ensuite, elle crée au maximum `N` permutation de `L` et teste si elles sont solutions du problème des `n` reines.

    Dès qu'une liste solution est trouvée, elle est renvoyée. En cas d'échec après `N` essais, la liste vide `[]` est renvoyée.

En prenant `N = 100 000`, on trouve ainsi rapidement des configurations gagnantes jusqu'à `n = 12`.

À titre documentaire, on signale qu'il est possible d'accélérer cette recherche aléatoire en utilisant le fait que lorsqu'une configuration est fausse, l'erreur vient souvent du mauvais placement des premières reines. La parti II.G exploitera partiellement cette remarque.

On se propose dans la sous-artie suivante de procéder à une recherche exhaustive des configurations solutions, quitte à se restreindre à des entiers `n` plutôt petits.

II.E - Déterminer toutes les permutations de `L = [0, 1, ..., n-1]`

21. Écrire une fonction `insertion` prenant pour paramètres un entier `x`, une liste `L`, un indice `i` et renvoyant une nouvelle liste obtenue en :
    - recopiant les termes de la liste `L` d'indices `0` à `i-1` inclus ;
    - puis en ajoutant `x` à la position `i` ;
    - enfin, en recopiant les termes de la liste `L` depuis l'indice `i` jusqu'à l'indice `len(L)-1` inclus.

    On renvoie donc une liste avec un élément de plus que al liste `L` : l'élément `x` inséré à l'indice `i`.

    Par exemple, l'appel `insertion(5, [2, 3, 6, 4], 1)` renvoie la liste `[2, 5, 3, 6, 4]` et l'appel `insertion(5, [2, 3, 6, 4], 4)` renvoie la liste `[2, 3, 6, 4, 5]`.

    Attention : l'usage d'une quelconque fonction Python faisant cette insertion est bien sûr interdit ici.

On considère maintenant le code suivant :

```
L = [[0, 1], [1, 0]]
v = 2
P = []
for l1 in L :
    for i in range(v+1) :
        p1 = insertion(v, l1, i)
        P.append(p1)
```

22. Quel est le contenu de `P` à l'issue du premier passage dans la première boucle ?
    Donner le contenu de `P` après l'exécution complète du code ci-dessus.
    (On demande à chaque fois de respecter l'ordre des différents éléments de `P`).
    Expliquez exactement ce que fait ce code.
23. En vous inspirant du code précédent écrire une fonction `permutations` prenant en paramètre un entier `n` supérieur ou égal à 2 et renvoyant la liste de toutes les permutations possibles des entiers de `0` à `n-1`.
    (c'est-à-dire toutes les listes possible contenant les entiers de `0` à `n-1` une fois et une seule).
    Par exemple, `permutations(3)` renverra une liste de sous-listes de la forme suivante (les sous-listes n'étant pas forcément dans cet ordre) :

    `[[0, 1, 2], [0, 2, 1], [2, 0, 1], [2, 1, 0], [1, 0, 2],[1, 2, 0]].`

24. En déduire une fonction `configurations` prenant en paramètre un entier `n` et renvoyant la liste de toutes les configurations satisfaisant les conditions imposées par le jeu.

Avec cette fonction, on peut vérifier qu'il y a 10 configurations possible pour `n = 5` et 92 configurations possible pour `n = 8`.

Cependant, on peu constater que certaines configurations sont similaires : elles peuvent se déduire les unes des autres par symétrie ou rotation. la sous-partie suivante détermine les configurations similaires.

II.F - Classer les configurations similaires

Deux configurations sont dites similaires si l'on peu passer de l'une à l'autre par rotations successives d'un quart de tour, par symétrie axiale, ou par une composition des transformations précédentes.

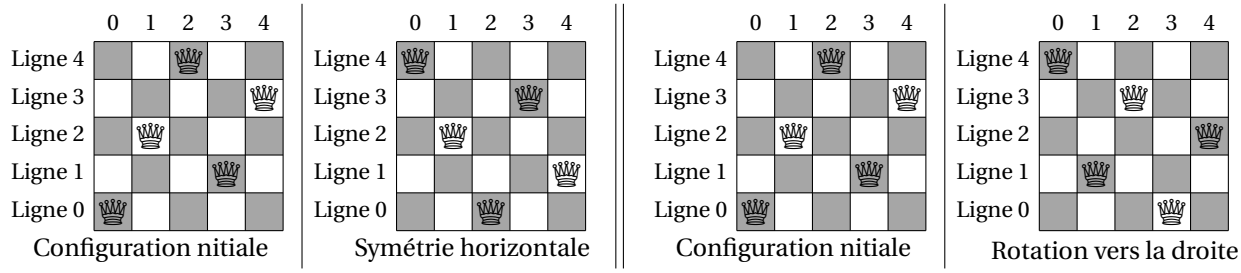


FIGURE 6 – Un exemple de symétrie horizontale et un exemple de rotation vers la droite

25. On se place dans une grille de taille 5×5 et on considère la configuration initiale représentée par la liste `L = [0, 3, 1, 4, 2]` (figure 6).
    Donner, sous forme d'une liste d'entiers de 0 à 4, la liste `L1` correspondant à une symétrie horizontale de la grille et la liste `L2` correspondant à une rotation d'un quart de tour vers la droite de la grille.
    De façon générale, comment obtient-on la liste `L1` correpondant à une symétrie horizontale à partir d'une liste `L` initiale ?
26. Écrire une fonction `symetrie` prenant en paramètre une liste `L` représentant une configuration et renvoyant la liste correspondant à une symétrie horizontale de la grille.
    Par exemple, cette fonction appliquée à `[1, 0, 2]` renvoie `[2, 0, 1]`.
    Attention : l'usage de la méthode `reverse`, de la focntion `reversed` ou du slicing `L[::-1]` est interdit ici.
27. Écrire une fonction `rotation` prenant en paramètre une liste `L` représentant une configuration et renvoyant la liste correspondant à la configuration obtenue par rotation d'un quart de tour vers la droite : chaque reine à la position ligne `i`/colonne `L[i]` se retrouve, après rotation, à la position ligne `L[i]`/colonne `n-1-i`.
    Par exemple, cette fonction appliquée à `[2, 5, 3, 1, 7, 4, 6, 0]` renvoie `[0, 4, 7, 5, 2, 6, 1, 3]`.

On suppose que l'on dispose d'une fonction similaire prenant en paramètres deux listes `L1` et `L2` de taille `n` représentant des configurations. Cette fonction renvoie `True` si `L1` et `L2` sont des configurations similaires, `False` sinon. Cette fonction est basée sur les fonctions symétrie et rotation mais il n'est pas demandé de l'implémenter.

On donne alors le code d'une fonction `partition` qui prend en paramètre un entier `n`.

```
1  def partition(n) :
2      C = configurations(n)
3      P = []
4      for c in C :
5          test = True
6          for elt in P :
7              if similaires(elt[0], c) :
8                  elt.append(c)
9                  test = False
10         if test :
11             P.append([c])
12     return P
```

28. Que fait la ligne 2 de la fonction `partition`?
    Quel est le type de la variable renvoyée par l'appel `partition(n)` ? Décrire complétement la structure de cette variable.
    Expliquer en quelques phrases comment fonctionne cette fonction `partition`.

II.G - Des graphes pour aller plus vitre et plus loin

Le test de toutes les permutations possibles fonctionne bien jusqu'à `n=10`. Au delà, le nombre de permutations à tester est trop important; on propose ici une autre approche basée sur le parcours du graphe implicite représentant les différents étas possibles d'une grille.

La figure 7 illustre ce point de vue.

La démarche consiste à parcourir (partiellement) ce graphe depuis la situation initiale (une grille vide). On place une reine, on regarde si la configuration obtenue est valide. Si c'est le cas, on place une nouvelle rein jusqu'à obtention éventuelle d'une solution.

En cas d'échec, on revient à l'étape précédente (c'est comme si on enlevait la dernière reine jouée) et on continue avec les autres coups à tester.

Le code mis en œuvre s'inspire ainsi d'un parcours de graphe calssique (et récursif).

On note que :

— dès que l'on trouve une solution au problème, on arrête le parcours et on renvoie cette solution;

— on revient en arrière (c'est-à-dire à l'étape précédente) quand on obtient une grille invalide;

— en cas de parcours complet du graphe sans avoir trouvé de solution on renvoie la valeur -1;

— on ne visite *a priori* pas la totalité des sommets du graphe, ce qui est un facteur de rapidité.

29. Un fonction `resout_graphe` prend en paramètre un entier $n \geqslant 4$ correspondant à la taille de la grille et renvoie une liste gagnante de position ou -1 si aucune solution n'a été trouvée. Elle utilise une fonction `explore`.
    Compléter les zones en pointillés dans la fonction `explore` pour que la fonction `resout_graphe` effectue le travail demandé.

```
1  def explore(L, n) :
2      if not est_possible(L) :
3          return ..........
4      if len(L) ==..........: # si fin de recherche victorieuse
5          return L
6      for i in range(n) :
7          res = explore (L + [i], n)
8          if res != -1: # si une solution a été trouv ée
9              return res
10     return ..........
11 def resout_graphe(n):
12     rep = explore ([] , n)
13     return rep
```

Le code précédent renvoie une solution pour `n=26` en moins d'une minute sur un ordinateur base de gamme.

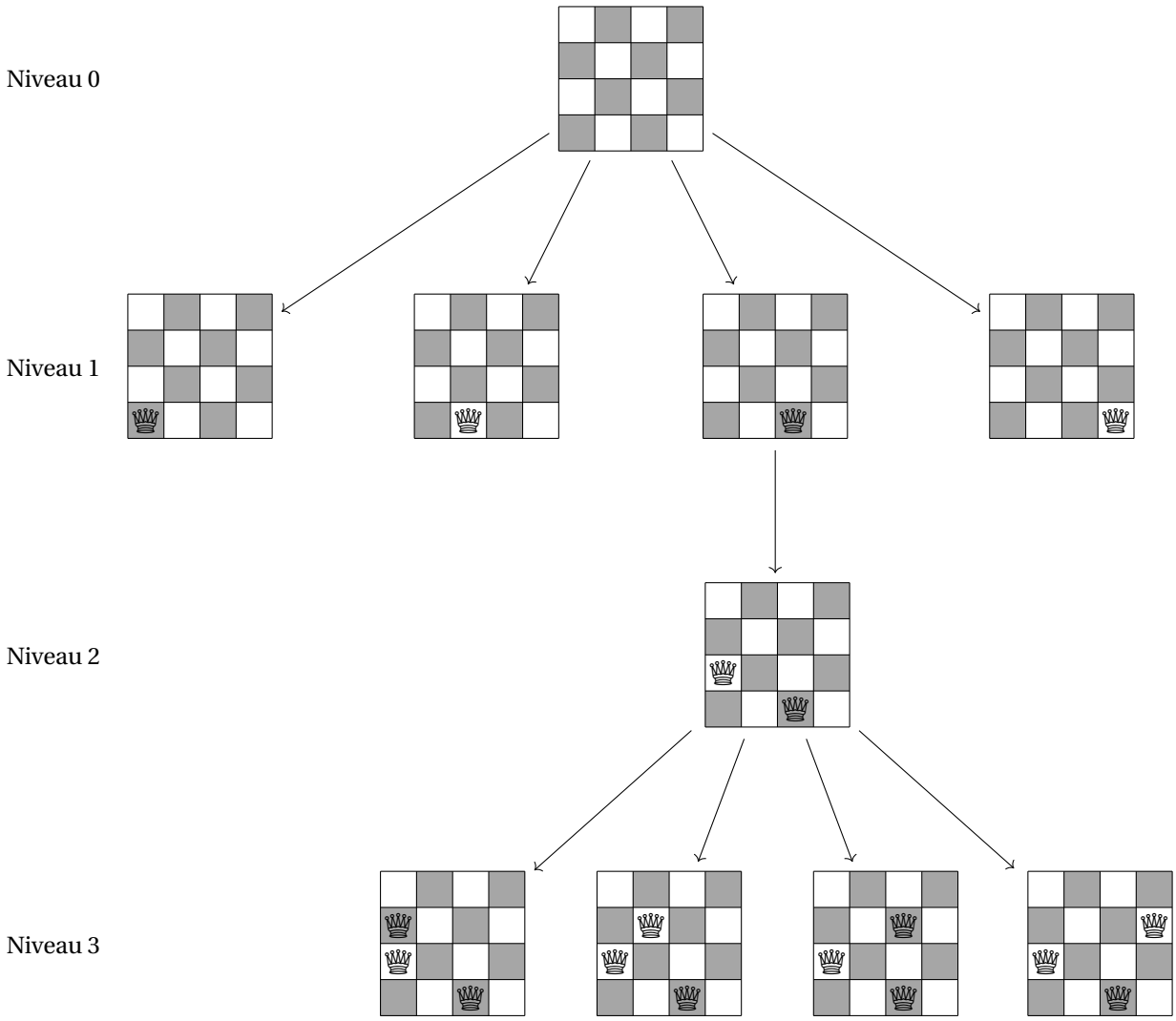


FIGURE 7 – Un extrait du graphe des coups pour une grille 4 × 4

30. L'exploration du graphe de la question 29 s'arrête dès qu'une solution est trouvée. On peut décider de continuer l'exploration du graphe pour déterminer les autres solutions au problème.

    On se propose ici de seulement compter le nombre de solutions grâce à une fonction `compte` prenant en paramètre un entier $n \geqslant 4$ correspondant à la taille de la grille.

    Compléter les zones en pointillés dans la fonction `explore2` pour que la fonction `compte` effectue le travail demandé.

```
def explore2(L, n):
    if not est_possible(L):
        return ..............
    if ..............:
        return 1
    nb_sol = 0
    for i in range (n):
        nb_sol = nb_sol +..............
    return ..............
def compte(n):
    rep = explore2 ([] , n)
    return rep
```