Stratégies de résolution de Problèmes

ECL - S7- 2A

BE1

Méthodes de retours arrières (BackTrack)

Avec introduction aux méthodes Heuristiques

2022 - 2023

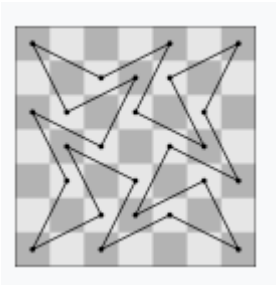
Plan :

1 Introduction

Sujet du BE : réalisation du parcours d'un cavalier sur un échiquier N x N par des essais successifs.

L'algorithme à Essais Successif est une classe d'algorithmes utilisés dans de nombreuses méthodes de résolution.

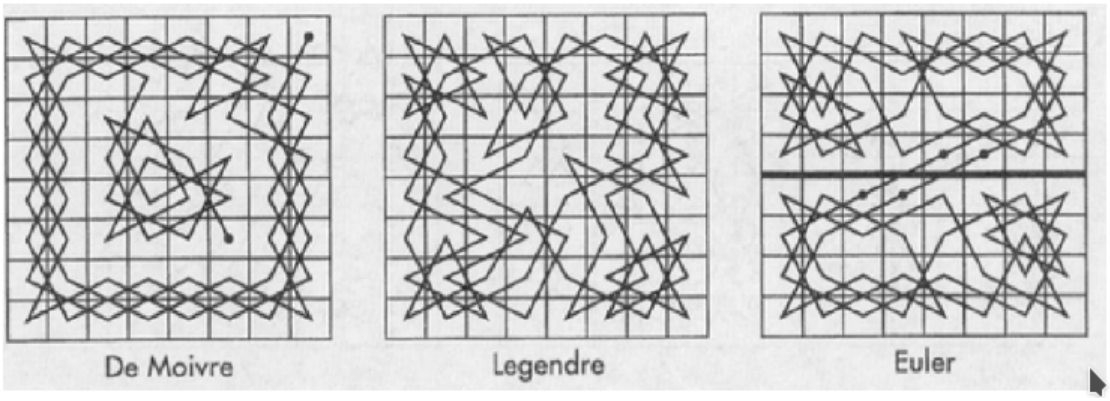
Exemples de solutions :



Parcours sans croisement maximal sur un échiquier de taille 7 × 7 (total de 24 sauts)

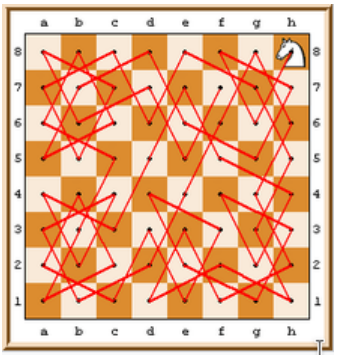
Parcours sans croisement maximal sur un échiquier de taille 7 × 7 (total de 35 sauts)

D'autres exemples de réalisation :



2 Parcours du cavalier

La recherche d'un tour du cavalier est un cas particulier des graphes **hamiltoniens** dans la théorie des graphes.



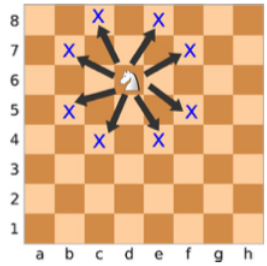
De plus comme un cavalier passe toujours d'une case noire à une case blanche et vice-versa, il s'ensuit que le graphe des mouvements du cavalier est un graphe biparti (les 2 couleurs : voir cours)

Dans le cas d'une recherche d'un parcours sans circuit (boucle), il existe une solution pour tout échiquier de taille ≥ 5 (selon la case de départ).

Le parcours du cavalier ou **le cavalier d'Euler** est connu au moins depuis l'an 840 (où le théoricien Arabe *Rumi* qui était aussi un joueur d'échecs avait proposé une solution).

3 Données du problème

On se donne un échiquier N x N , $N > 4$ et un cavalier du jeu d'échecs.



La figure ci-contre montre les mouvements possible de ce cavalier.

En partant d'une case de départ, le but est de visiter **toutes** les cases de l'échiquier par le cavalier en ne visitant chaque case qu'une seule fois.

☞ Notons qu'il n'y a pas de solution pour $N \leq 4$.

3.1 Rappels et position du problème

Le problème présenté ici relève d'une classe de problèmes d'affectation sous contraintes (réalisé par un parcours de graphe sous contraintes) où l'on atteint chaque étape depuis l'étape précédente en respectant certaines contraintes.

Ce problème s'inscrit dans le cadre général des stratégies de recherche **révocables** (Algorithmes à Essais Successifs = AES) mettant en place une méthode basique de tentatives avec des retours arrières.

Pour utiliser un exemple réel, lorsque nous cherchons une sortie dans un labyrinthe, nous appliquons cet algorithme et pour ne pas tourner en rond, nous "marquons" les couloirs déjà empruntés.

Plus formellement, soit $X = \{x_1, .., x_n\}$ l'ensemble de variables du problème et $D = \{d_1, .., d_n\}$ l'ensemble de domaines de valeurs de X où x_i prendra une valeur dans d_i. Soit également un ensemble de contrainte C qui décrira les couples de valeurs compatibles. A l'étape k où on cherche à *affecter* une valeur à la variable x_k, on choisira une valeur $v \in d_k$ telle qu'aucune des contraintes (dans C) posées sur $x_1 ... x_k$ ne soit violée. Si telle n'est pas le cas, on choisira *successivement* une autre valeur dans d_k. Si une telle possibilité n'existe pas (i.e. d_k épuisé), on remettra en cause la valeur donnée à x_{k-1}. Cette action est un *retour arrière* (backtrack).

Dans ce BE, après l'application d'une version basique, on étudiera une méthode **Heuristique** permettant d'aboutir plus rapidement à une solution.

3.2 Aspects pratiques du "Backtracking"

On détaille ici les aspects pratiques de la stratégies "retours arrières" (BackTracking) dont l'algorithme *à essais successifs* est une illustration.

La stratégie générale **à essais successifs** permet, au besoin, de retrouver toutes les solutions à un problème donné, ceci en couvrant la totalité de l'espace des solutions. Notez que ce parcours est un **parcours en profondeur** du graphe d'états.

Un exemple typique de cette stratégie est la recherche d'une sortie dans un labyrinthe.

4 L'algorithme "à essais successifs" (AES) général

Algorithme de principe AES_Basique :

Données : état_courant,
fonction_de_transition (pour faire un mouvement) $\sigma : etat \times action \rightarrow etat$
Résultat : {Succès, Echec}

Si état_courant == état_final
Alors renvoyer Succès
Sinon
  Pour tout **état_successeur** $= \sigma(etat\_courant, une\_action\_possible)$:
    Si AES_Basique(état_successeur) == succès
    Alors renvoyer Succès
renvoyer Echec

4.1 Algorithme Basique appliqué à un grahe d'états

Fonction AES_le_premier_succes_suffit;
Données : G : un graphe d'états ;
$Noeud\_courant$: le noeud (une variable = un état) courant que l'on traite dans cet appel
Résultat : Succès ou Echec (un booléen)
début
  si $Noeud\_courant$ *est un état final* **alors**
    renvoyer Succès
  sinon
    pour tous les *Noeud_suivant* **successeurs de** $Noeud\_courant$ *dans* G **faire**
      si *Prometteur(G, Noeud_suivant)* **alors**
        si *AES_le_premier_succes_suffit(G, Noeud_suivant)* = *Succès* **alors**
          renvoyer Succès
        fin
      fin
    fin
    renvoyer Échec
  fin
fin

Le résultat de cet algorithme sera soit un succès, soit un échec si aucun succès n'est signalé.

La fonction ***Prometteur(G, Noeud)*** vérifie si *Noeud_suivant* (i.e. si la valeur choisie pour la prochaine variable) permet d'avancer en respectant les contraintes du problème.

Exemple : Adaptation à un labyrinthe

☞ Pour mieux comprendre, déclinons cet algorithme pour l'adapter à notre balade dans un labyrinthe.

Avant de voir l'algorithme, notons que dans le cas d'un labyrinthe, la fonction *Prometteur(.,.)* nous dira par exemple que la direction **envisagée** (8 direction possibles)

- correspond à un couloir qui est dans le plan (on peut décider d'aller au Nord-Est qui n'existe pas à ce point !)
- n'est pas déjà emprunté (on a des haricots pour marquer notre parcours)
- ne mène pas à un trou / falaise ! (nous sortirais du labyrinthe)
- mène à un couloir empruntable (n'est pas inondé, n'est pas habité par des zombies) Adaptation à un labyrinthe
- ...

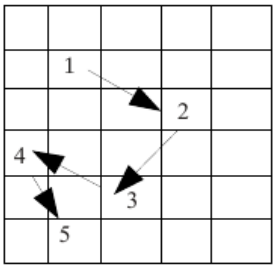
```
Fonction AES_le_premier_succes_suffit_adapte_au_labyrinthe ;
Données : G : un graphe d'états = le plan du labyrinthe ;
          Noeud_courant : le dernier couloir emprunté qui a mené à un choix / carrefour
Résultat : Succès ou Echec ( on s'en sort ou pas )
début
    si Noeud_courant est un état final = la sortie en vue alors
        renvoyer Succès
    sinon
        pour tous les Noeud_suivant = une des directions possibles à partir du Noeud_courant dans G faire
            si Prometteur(G, Noeud_suivant) = on peut effectivement emprunter la direction choisie alors
                si AES_le_premier_succes_suffit_adapte_au_labyrinthet(G, Noeud_suivant) = Succès alors
                    renvoyer Succès
                fin
            fin
        fin
        renvoyer Échec
    fin
fin
```

Revenons à l'échiquier :

Dans le cas du parcours du cavalier, le graphe sera l'espace des étapes et la fonction **Prometteur**(. , .) doit valider (ou non) la prochaine case choisie.

A noter qu'une simplification facile est immédiatement possible : on donne à toute case le numéro de la dernière case visité +1. De cette manière, on construit un trajet : si on réussit, il suffira de suivre les numéros de 1 à N^2 pour le connaître.

Exemple partiel pour le parcours du cavalier :
- On inscrite un nombre $1..N^2$ dans chaque case empruntée. De cette façon,
- Les chiffres représentent les étapes (voir ci-dessous),
- On opère les déplacements sous forme d'un L (comme en Échecs),
- On inscrit dans les cases 1, puis 2, ..., N^2,
- Les contraintes de chaque mouvement sont :
  + se déplacer comme se déplace un cavalier du jeu d'échécs ;
  + rester à l'intérieur de l'échiquier ;
  + ne pas repasser par une case déjà numéroté (c-à-d. "marquée").




```
bool prometteur( Echiquier, Taille, (New_X, New_Y)) :
    (New_X, New_Y) est une case valide dans Echiquier si les tests suivants réussissent :
        New_X ≥ 0; New_X < Taille
        New_Y ≥ 0; New_Y < Taille
        Echiquier[New_X][New_Y] == -1          // case non encore visitée = nœud du graphe non exploité
    Renvoyer le résultat de ces tests (vrai ou faux)
```

Si on reconsidère la (petite) formalisation ci-dessus :

- $X = \{x_i | i \in 1..N^2\}$ (chaque case est un variable)
- $D = \{i \in 1..N^2\}$ (toute case peut prendre un numéro $i \in 1..N^2$, et tous les domaines sont identiques)
- $C = \{(x_i, x_{i+1}), i \in 1..N^2-1 | x_{i+1}$ est un des 8 voisins possibles et libre de x_i à l'intérieur de l'échiquier$\}$

La tâche de vérifier **le respect de ces contraintes** est confiée à la fonction *Prometteur*.

Nous allons maintenant préciser notre algorithme et l'adapter aux détails de l'échiquier.

5 Adaptation au parcours du Cavalier

Dans le cas du parcours du cavalier, chaque case de l'échiquier représente un nœud du graphe du parcours et les successeurs possibles sont celles qui respectent les contraintes ci-dessus. Si à une étape donnée, on ne trouve pas une case "successeur" (*Noeud_suivant* dans l'Algorithme), ON REMET EN CAUSE le choix précédent.

L'algorithme ci-dessous est la version adaptée au problème du cavalier. Dans cette adaptation :

- On désigne une case de départ et on y inscrit le nombre 1 (la case départ), le *prochain_numero* à inscrire sera 2
- On appelle la fonction ci-dessous et lui passant l'échiquier (le Graphe d'états), la taille de l'échiquier, le *prochain_numero*, la *case_depart* (pour trouver ses successeurs),
- La fonction appelée reçoit donc un état courant (*case_actuelle* = la dernière case occupée), envisage chacun de ses successeurs, teste si ce successeur est *prometteur*, y inscrit le prochain numéro d'étape à inscrire, ce successeur là devient la *case_actuelle* du prochain appel,
- Et ... on recommence jusqu'à inscrire N^2 (= état final de succès)
- Si aucun des successeurs d'une case n'est prometteur, on revient en arrière.

La fonction **Prometteur** vérifie les contraintes citées ci-dessus.

Avant d'appeler l'algorithme ci-dessous (voir aussi la section 13 page 16 pour le code Python) :
l'échiquier aura été crée avec toutes ses cases initialisée à -1
Une case de départ est désignée et reçoit le nombre 1 : case départ.
On appelle la fonction suivante avec Num_etape = 2 pour la prochaine case (un choix alternatif est possible).

```
Fonction AES_parcours_cavalier_un_succès_suffit;
Données : G : un graphe sous forme de matrice ;
          N : taille échiquier
          Case_actuelle : (X, Y) : les coordonnées de la case qu'on visite dans cet appel,
          Num_etape : entier numéro de la prochaine case,
Résultat : Succès ou Echec (un booléen)        // en cas de succès, G contient le résultat

début
    si Num_etape > N^2 alors
        renvoyer Succès        // N^2 déjà inscrit, on a réussi ! On arrête sur le 1er succès
    sinon
        pour tous les Case_suivante successeur de Case_actuelle dans G faire
            si Prometteur(G, N, Case_suivante) alors
                Inscrire Num_etape dans Case_suivante
                si AES_parcours_cavalier_un_succès_suffit(G, N, Case_suivante, Num_etape + 1) =
                  Succès alors
                    renvoyer Succès
                fin
                Ici, il faut effacer Num_etape inscrit dans Case_suivante. VOIR PLUS LOIN
            fin
        fin
        renvoyer Echec
    fin
fin
```

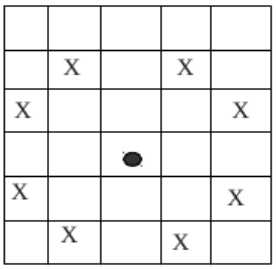
Le graphe des états est ici caractérisé par l'échiquier dont les cases porteront les numéros des étapes (de $1..N^2$). Pour la procédure ci-dessus, un échec est conclu si aucun succès n'est signalé.

Rappel : pour une matrice N x N d'entiers représentant l'échiquier, on numérotera les cases visitées dans l'ordre de 1 a N^2. Ce qui permet de *connaître le trajet* du cavalier en suivant les numéros des cases depuis 1 à N^2.
Le parcours est terminé (avec succès) si N^2 a été inscrit dans la dernière case visitée.

5.1 Calcul pratique des voisins d'une case

- Le graphe (l'échiquier) est représenté par une matrice $N \times N$.
- Soit les cases de la matrice $M[N][N]$ initialisée par -1 (voir plus loin pour un exemple de création de cette matrice).
- Soit la case actuelle (X, Y), calculons ses voisins.
- On sait qu'une case possède au plus 8 voisins (pour un saut de cavalier).
- Pour la case (X, Y), les successeurs possibles sont les cases
  $(X \pm 1, Y \pm 2)$
  $(X \pm 2, Y \pm 1)$

Ces valeurs reflètent le mouvement en L du cavalier.



La combinaison de ces valeurs donne jusqu'à 8 voisins possibles.
Bien entendu, toutes les cases n'ont pas 8 voisins possibles. Par exemple, la case (1, 1) n'a que 2 voisins, la case (1,2) ne possède que 3 voisins, ...

Chacun de ces successeurs est vérifié par la fonction **Prometteur** :
➞ Dans la boucle "Pour" de l'algorithme ci-dessus, on trouve une des 8 cases possibles.
➞ La fonction **Prometteur** se charge de savoir si le voisin proposé respecte les contraintes (case à l'intérieur de l'échiquier et non encore visitée).

Calcul pratique des voisins de (X,Y) (voir aussi la section 13 page 16) :
Pour faciliterle calcul des voisins, on utilisera 1 tableau (liste) de couples d'entiers Tab_delta_X_Y_Y

☞ Ce tableau doit être une donnée GLOBALE. Ne le déclarez pas dans la fonction AES car à chaque appel, vous allez le recréer !

Ce tableau est initialisé par les coordonnées relatives des voisins (les déplacements) :

Tab_delta_X_Y_Y=[[1,2], [1,-2], [-1,2], [-1,-2], [2,1], [2,-1], [-2,1], [-2,-1]]

où **Tab_delta_X_Y[i][0]** représente le déplacement par rapport à X et pour le même indice i ;
et **Tab_delta_X_Y[i][1]** donne le déplacement par rapport à Y [1].

Ainsi, le premier voisin de (X, Y) sera $(X + Tab\_delta\_X\_Y[0][0]\ ,\ Y + Tab\_delta\_X\_Y[0][1])$

Un autre exemple : le 4ème voisin de la case (X, Y) sera calculé par :
X + Tab_delta_X_Y[3][0] *et* *Y + Tab_delta_X_Y[3][1]*

D'une façon générale, pour trouver (Nx, Ny) = le ième voisin ($i = 0..7$) de la case (X,Y), on a :

Pour i=0..7
Nx = X + Tab_delta_X_Y[i][0]
Ny = Y + Tab_delta_X_Y[i][1]

Rappel : après avoir trouvé la case (Nx, Ny) un voisin de (X, Y), ce couple est vérifié par la fonction **Prometteur** pour s'assurer que :

1- (Nx, Ny) est à l'intérieur de l'échiquier,
2- la case (Nx, Ny) n'est pas déjà visitée.

Déjà visitée : qu'est-ce à dire ? :
A l'initialisation, toute case de la matrice est initialisée à -1 (libres = non encore visitée) et les cases visitées comporteront un numéro de l'étape $1..N^2$.
→ Donc une case non visitée est une case dont la valeur est = -1.

1. Vous pouvez bien entendu créer deux listes séparées : une liste pour les déplacements de X et une pour Y.

5.2 Initialisation de la matrice en Python

On peut utiliser les listes de Python et initialiser l'échiquier par (voir aussi la section 13 page 16) :

```
Taille = 5    # Taille > 4  dans tous les cas.
Echiquier=[[-1 for x in range(Taille)] for y in range(Taille)]
```

➛ Placer ces deux lignes dans *main* et déclarer les comme *globales*. Ce sera plus propre !

Notez que le type *array* est plus économique et qu'avec *numpy*, on a la notation matricielle M[i,j].

L'algorithme du parcours adapté à ces indications devient alors ...

5.3 Pseudo-algorithmes de parcours de cavalier

La partie principale (__main__ de Python) crée et initialise la matrice puis appelle l'algorithme à essais successifs. Pour des raisons pratiques, le pseudo-code de l'algorithme **AES_parcours_cavalier_un_succes_suffit** devient une fonction booléenne (un prédicat).

```
if __name__ == "__main__" :
    # Les variables déclarées dans __main__ sont globales en lecture.
    Tab_delta_X_Y_Y=[[1,2], [1,-2], [-1,2], [-1,-2], [2,1], [2,-1], [-2,1], [-2,-1]]

    Taille = 5    # Taille > 4  dans tous les cas.
    Echiquier=[[-1 for x in range(Taille)] for y in range(Taille)]
    x0, y0 = lire_caase-depart()
    Echiqiuer[x0, y0) = 1   # La première case du parcours
    prochain_numero = 2 # le No qu'il faudra donner à l aprochaine case
    if AES_parcours_cavalier_un_succes_suffit(Echiquier, Taille, (X0, Y0), prochaine_numéro) :
        afficher(Echiquier)
    else :
        print("Echec")'
```

☞ Attention : les indices commencent à 0.

bool AES_parcours_cavalier_un_succes_suffit(Echiquier, Taille,
Dernière_Case_Traitée, Prochain_Num_etape) :
➛ // la Dernière_Case_Traitée est un couple (X, Y)

Résultat : Succès ou Echec (un booléen) // en cas de succès, $Echiquier$ contient le résultat

début
  si $Num\_etape > Taille * Taille$ **alors**
    renvoyer Vrai // N^2 déjà inscrit, on a réussi ! On arrête sur le 1er succès
  sinon
    Soit (X,Y) = Dernière_Case_Traitée
    pour tous les $i = 0..7$ **faire**
      Next_X = X + Tab_delta_X_Y[i][0] ;
      Next_Y = Y + Tab_delta_X_Y[i][1] ;
      si *Prometteur(Echiquier , Taille, (Next_X, Next_Y))* **alors**
        Echiquier[Next_X][Next_Y] = Prochain_Num_etape
        si *AES_parcours_cavalier_un_succès_suffit(Echiquier, Taille,*
        (New_X, New_Y), *Prochain_Num_etape+1)* **alors**
          renvoyer Vrai
        fin
        Echiquier[Next_X][Next_Y] = -1 // Ca n'a pas marché, on efface nos traces !
      fin
    fin
    renvoyer Faux // Aucun des successeurs n'a abouti.
  fin
fin

La **fonction prometteur** (certifie ou non le prochain candidat (New_X, New_Y))

```
def prometteur(Echiquier, Taille, (New_X, New_Y)) -> bool :
    # (New_X, New_Y) est une case valide dans Echiquier si les tests suivants réussissent.
    # Une case non encore visitée vaut -1 (un n½ud du graphe non exploité)

    return 0 <= New_X < Taille and 0 <= New_Y < Taille and Echiquer[New_X][New_Y] == -1
```

Remarquez qu'il y a 5 tests dans la fonction prometteur.

Quelques résultats d'exécution :
- Pour N=4, il n'y a pas de solution
- Pour N=5 il y a plusieurs solutions dont certaines symétriques (indice à partir de 0) :
  Exemples de cases de départ possibles (elles mènent à un succès) :
  (0,0), (0,2), (0,4), (1,1), (2,2), (3,1), (4,0), (4,2), (4,4).

Par ex, pour départ = (0,0), on aura la matrice (de taille 5) de la solution (de 1 à 25) :

1	14	19	8	25
6	9	2	13	18
15	20	7	24	3
10	5	22	17	12
21	16	11	4	23

N.B. : décider d'une convention "propre" pour la gestion des indices : un tableau / liste commence à l'indice 0 mais adaptez l'algorithme si vous appelez la première case (1,1) (au lieu de (0,0).

N.B. : pour une case de départ (X,Y) donnée, il peut exister plusieurs solutions (selon le choix des voisins).
Remarquez aussi la symétrie des solutions pour les départs ((1,1) et (5,5), (1,5) et (5,1), ...) pour un taille d'échiquier = 5.

6 Notes sur la complexité (voir après cours 1)

On suppose ici un coût unitaire pour un essai (un appel récursif $\simeq$ un appel à *prometteur*). N est la taille de l'échiquier.

• Pour $N < 5$: pas de solution
• Pour $N > 4$ mais pas très grand : N^2 cases avec environ (8N-16) cases ayant entre 2 et 6 ; le reste aura 8 voisins.
• Pour **N grand** : toutes les N^2 cases sont considérées pouvant avoir 8 voisins possibles et donc on aura une complexité $O(N^{16}) \simeq O(2^{16 \log N})$.
  ➙ Cette valeur est une sur-estimation (voir cours) et se constate dans un cas d'échec (aucune solution pour les données N et (X,Y) de départ).

☞ Pour constater le nombre de tentatives, ajouter dans votre code :
→ un compteur global dans la fonction *Prometteur*.
→ et un autre pour compter les retours-arrières là où vous remettez -1 dans une case.

Indications sur l'équation de récurrence
La formulation de l'équation de récurrence (voir cours) de l'algorithme AES dans sa forme approximative est (pour $k = N^2$ = le nombre de cases) :

T(k)=8 * T(k-1) + C Hypothèse : toutes les cases ont 8 voisins
➙ $T(k) = 8^k = 2^{3k} = 2^{(3N^2)}$

Essais : à titre d'indication, pour N=5, la fonction de complexité (empiriquement mesurée pour un cas d'échec) est de l'ordre de $\Omega(2^{21})$.

Pour N=6, la même fonction est de l'ordre de $\Omega(2^{28})$.
Pour N=7 (k=49), on devrait atteindre plus de $2^{30} \simeq 10^9$).

☞ La complexité $8^{(N^2)}$ est une borne (trop) supérieure et on aimerait plus de précision. Mais le calcul formel et précis de cette complexité est très difficile (voir cours 1) et cette complexité doit être empiriquement proche de $\Omega(2^{N^2-2^{N-3}})$, N = taille de l'échiquier.

7 Travail à rendre pour cette étape

☞ Voir aussi la section 13 page 16 pour une aide.

• Réaliser cette version de base
• Ajouter un compteur pour compter le nombre de tentatives pour N et la case départ (X, Y) données. Il vous donnera une idée de la complexité de votre algorithme.
☞ Porter ces valeurs dans votre rapport final.
• Noté sur 12 (sur 20)

8 Obtenir toutes les solutions possibles

Un point crucial signalé dans les algorithmes AES précédents est l'expression des conditions d'arrêt des appels récursifs : dans les algorithmes précédents, en cas de succès, on arrête tout.

On peut décider que malgré un succès, on continuerait les appels pour savoir si une case (X,Y) ayant donné une solution ne pourrait-elle pas mener un autre succès. La recherche de toutes les solutions dans un AES est courante, p. ex. pour en choisir "la meilleure".

L'algorithme général AES devient une version exhaustive (les modifications sont en rouge) :

```
Fonction AES_toutes_solutions;
Données : Echiquier : un graphe ;
          Noeud_courant : le noeud courant que l'on traite
Résultat : Succès ou Echec (un booléen)

début
    si Noeud_courant est un état final alors
        Afficher ou enregistrer la solution que l'on vient de trouver
        Renvoyer Echec : Simuler un échec pour forcer à trouver les autres solutions
    sinon
        pour tous les Noeud_suivant successeur de Noeud_courant dans Echiquier faire
            si Prometteur(Echiquier , Noeud_suivant) alors
                si AES_toutes_solutions(Echiquier , Noeud_suivant) = Succès alors
                    renvoyer Succès
                fin
            fin
        fin
        renvoyer Echec
    fin
fin
```

Bien remarquer que même en cas de succès, on laisse les autres tentatives avoir lieu.
Dans ce cas, il est naturel que le résultat final soit un échec (on l'a provoqué) mais avant cet échec final, on aura eu les succès (si succès il y a dans le graphe de recherche).

8.1 Algorithme de toutes les solutions du cavalier

On adapte l'algorithme précédent et on obtient la fonction *AES_Cavalier_toutes_solutions* version exhaustive du parcours du cavalier (les modifications sont en rouge) :

```
bool AES_parcours_cavalier_toutes_solutions(Echiquier, Taille,
                                            Dernière_Case_Traitée, Num_etape) :
Résultat : Succès ou Echec (un booléen)        // en cas de succès, Echiquier contient le résultat
début
    si Num_etape > N^2 alors
        Proposer la solution = afficher la matrice
        Demander si l'on souhaite poursuivre ?
        Si poursuite demandée alors (simuler) renvoyer échec
        Sinon renvoyer Succès
    sinon
        // Le reste de l'algorithme ne change pas
        Soit (X,Y) = Dernière_Case_Traitée
        pour tous les i = 0..7 faire
            Next_X = X + Tab_delta_X_Y[i][0] ;
            Next_Y = Y + Tab_delta_X_Y[i][1] ;
            si Prometteur(Echiquier , Taille, Next_X, Next_Y) alors
                Echiquier[Next_X][Next_Y] = Num_etape
                si AES_parcours_cavalier_toutes_solutions(Echiquier, Taille,
                                                          (New_X, New_Y), Num_etape+1 ) alors
                    renvoyer Vrai
                fin
                Echiquier[Next_X][Next_Y] = -1        // Ca n'a pas marché, on efface nos traces !
            fin
        fin
        renvoyer Faux        // Aucun des successeurs n'a abouti.
    fin
fin
```

La réalisation de cette version de l'algorithme n'est pas demandée. Mais n'hésitez pas à la tester.
Pour ce faire, adapter l'algorithme de la section 5.3 de la page 7 aux modifications ci-dessus.

8.2 Remarques sur les stratégies

Les algorithmes à essais successifs ci-dessus opèrent un parcours de graphe en profondeur (**Depth-First**).
On discutera de la version en largeur (**Breath-First**) en cours.
Une autre stratégie couramment employée est la stratégie **Best-First** (ci-dessous).
Voir également en cours la stratégie révocable (sauvegarde du contexte par une pile gérée par soi-même ou celle des appels récursifs).

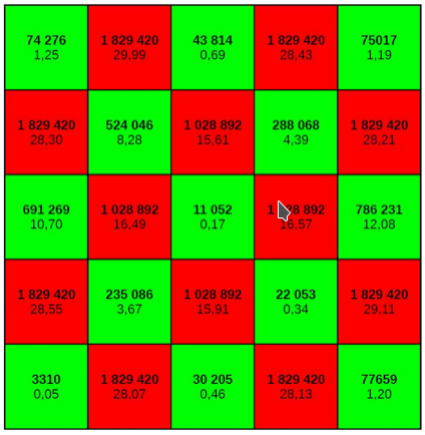
Remarques :

1- Pour N=5, les résultats pour N=5 de la version basique sont donnés ci-contre.

• En vert : cette case de départ a donné un succès.
Le 1er nombre est le nombre des retours arrières suivi d'une approximation du temps de l'exécution sur une machine basique (valeur très indicative pour comparer aux autres).

• En rouge : case de départ ayant donné un échec.
➛ Les nombres ont le même sens.

2- Pour la version avec Bande :



9 Amélioration

Le nombre de combinaisons est donc très grand et augmente de façon exponentielle avec la taille N de l'échiquier. Le nombre d'appels à la fonction *Prometteur* est du même ordre que ci-dessus.

Deux améliorations peuvent être proposées.

9.1 Bande périphérique

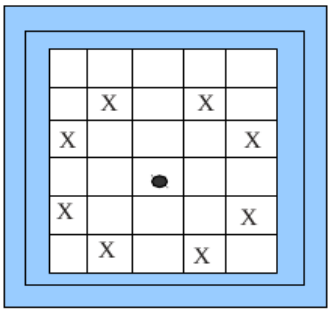
Dans l'algorithme AES, un facteur (certes constate) de la complexité vient du nombre de tests dans la fonction **Prometteur** : 5 tests par case voisine ; ce qui finit par couter 5 fois le nombre des retours arrières (voir la complexité ci-dessus).

On peut réduire ces 5 tests à 1 seul de la façon suivante :

• On entoure la matrice (l'échiquier) d'une bande d'épaisseur 2. C'est a dire, au lieu de travailler avec une matrice N x N, on utilise une matrice (N+4 x N+4).

Pour être considérée comme "déjà visitée", cette bande est initialisée par une valeur différente de -1 (par exemple zéro ou N^2+1 car -1 veut dire case libre).

Ainsi, puisque les cases de cette bandes sont marquées visitées, on ne les choisira jamais comme successeur d'une case et donc on ne risque plus d'aller en dehors de la matrice ; ce qui supprime les 4 premiers tests de la fonction "Prometteur".



10 Travail à rendre 2

Tester votre code avec cette modification et constater l'amélioration.

☞ Si vous le souhaitez, vous avez la possibilité de ne pas réaliser cette version si vous compter rendre la version optimale (voir ci-après).

• Noté sur 2 (sur 20) (+ les 12 points de la version de base)

N.B. : on peut s'amuser à d'abord occuper une moitié de l'échiquier avant de traiter l'autre moitié. D'autres "formes" de parcours ont été réalisés par certains (voir les 2 exemples réalisés par *Euler* lui-même en Introduction).

11 Optimisation : Heuristique

Une heuristique (ou euristique) est un "savoir faire" permettant d'améliorer une performance. Un exemple d'heuristique de la vie courante est le chemin par un sentier (éventuellement absent de toute carte) que vous connaissez pour atteindre plus rapidement un lieu.

Dans notre problème, il est empiriquement démontré (conjecture) que lors du choix d'un successeur, le choix de la case la plus contrainte conduit a une solution **sans retour arrière** [2] (si une solution existe)**.** Dans ce cas, le degré de contraintes d'une case est le nombre de ses cases voisines occupées.

Ainsi, lors du choix d'un voisin, parmi les 8 voisins possibles, on choisira celui dont le nombre de voisins libres est le plus petit. C'est à dire :
soit (X,Y) la case actuelle et (Next_X_i, Next_Y_i) un des 8 voisins non-visité possibles. Chacune des (Next_X_i,Next_Y_i) possède à son tour jusqu'à 8 voisins libres (non visités) possibles. Pour la prochaine étape de l'algorithme, choisir la case **libre** (Next_X_i, Next_Y_i) avec le plus petit nombre de voisins libres.

Pour cela, nous avons besoin d'une seconde matrice qui contiendra pour chaque case de la matrice principale, le nombre des voisins déjà visités. Cette matrice sera régulièrement mise à jour.

Avec cette amélioration (choix du "meilleur voisin"), l'algorithme peut devenir **irrévocable** : **vous ne reviendrez plus en arrière** sauf dans la condition suivante :
Condition : S'il y a plusieurs "meilleurs voisins", il faut prévoir des retours arrières.

A titre d'exemple, le départ (0,0) donnerait la solution (*avec 0 retours arrières*) :

1	**12**	**25**	**18**	**3**
22	**17**	**2**	**13**	**24**
11	**8**	**23**	**4**	**19**
16	**21**	**6**	**9**	**14**
7	**10**	**15**	**20**	**5**

On remarque que cette solution est différente de la solution sans heuristique. C'est normal : la suite de voisins considérée n'est pas la même.

☞ Vous disposez d'une aide Python sur cette heuristique : voir la section 13 page 16.

11.1 Algorithme avec heuristique quasi irrévocable

Dans la version Heuristique, on a besoin de maintenir une seconde matrice où l'on comptabilise le nombre de voisins libres de chaque case. Il faudra en permanence tenir à jour ces nombres car ils représente le nombre de voisins libres de chaque case.

Cette matrice des voisins est initialisée comme suit (pour N=5) :

2	3	4	3	2
3	4	6	4	3
4	6	8	6	4
3	4	6	4	3
2	3	4	3	2

puis le long des étapes, on met à jour le nombre de voisins libres de chacune des cases.
Voir *https ://fr.wikipedia.org/wiki/Problème_du_cavalier* pour les cases possibles pour N=8.

2. Sous condition ! Voir "condition" un peu plus bas

```
bool AES_cavalier_heuristique_un_succes_suffit(Echiquier, Matrice_Nb_Voisins_libres ,
                                               Taille, Dernière_Case_Traitée, Prochain_Num_etape) :
    ➙ // la Dernière_Case_Traitée est un couple (X, Y)

Résultat : Succès ou Echec (un booléen)        // en cas de succès, Echiquier contient le résultat

début
    si Num_etape > Taille * Taille alors
        renvoyer Vrai        // N² déjà inscrit, on a réussi ! On arrête sur le 1er succès
    sinon
        // On va chercher les meilleurs voisins (plusieurs possibles avec le même degré de liberté )
        Vecteur_des_meilleurs_Voisins = Trouver_Meilleurs_Voisins_Libres (
                    Echiquier, Matrice_Nb_Voisins_libres , Taille, Dernière_Case_Traitée);
        pour tous les couples (Next_X, Next_Y) de Vecteur_des_meilleurs_Voisins faire
            Echiquier[Next_X][Next_Y] = Prochain_Num_etape
            Mise_A_Jour_Voisins(Matrice_Nb_Voisins_libres, (Next_X, Next_Y))
            si AES_cavalier_heuristique_un_succes_suffit(Echiquier, Matrice_Nb_Voisins_libres , Taille,
                                                         (New_X, New_Y), Prochain_Num_etape+1 ) alors
                renvoyer Vrai
            fin
            Echiquier[Next_X][Next_Y] = -1        // Ca n'a pas marché, on efface nos traces !
            Remettre_A_Jour(Matrice_Nb_Voisins_libres, (Next_X, Next_Y))
        fin
        renvoyer Faux        // Aucun des successeurs n'a abouti.
    fin
fin
```

L'algorithme de principe (et pratique) du calcul du tableau des meilleurs voisins :
➙ Une aide est fournie dans la section 13 page 16 (pour la partie Python).

◦ Soit **Echiquier** (la matrice de parcours) et **Matrice_Nb_Voisins_libres** la matrice des Voisins
Et (X, Y) la case pour laquelle on cherche les meilleurs voisins

◦ Chercher une (première) case libre et prometteuse (Nx, Ny) de l'**Echiquier** dont le nombre de voisins (soit k) dans **Matrice_Nb_Voisins_libres** est minimal (mais $k \geq 0$))

→ Dans la pratique, parmi les voisins prometteurs de (X, Y), trouver la case (Nx_k, Ny_k) dont la valeur k dans Matrice_Nb_Voisins_libres est minimal. Il en existe peut-être plus d'un.

◦ Si un tel (Nx_k, Ny_k) existe Alors construire le vecteur $Vecteur\_des\_meilleurs\_Voisins$ ne contenant que les (Nx_k, Ny_k) ayant K voisins

Une méthode alternative (2e solution) est de :
◦ Créer simplement V le vecteur des voisins prometteurs de (X, Y); chacun des éléments de V (soit les (Nx, Ny)) est accompagné de $k' \geq 0$ le nombre de ses voisins,

◦ Trier le vecteur V selon le nombre de voisins (un tri externe); ceux dont la valeur k' de voisins est minimal seront en tête.

◦ Ne conserver que les premières cases dont le nombre de voisins est minimal ($k = min(\forall k')$).

☞ L'avantage de la 1ère solution est d'ignorer les voisins qui ne nous intéressent pas et d'éviter un tri.

Remarque : dans l'algorithme AES ci-dessus, on remarque les fonctions **Mise_A_Jour_Voisins()** et **Remettre_A_Jour()**. Si on veut éviter la fonction *Remettre_A_Jour()*, on copie-sauvegarde *Matrice_Nb_Voisins_libres* juste avant l'appel de **Mise_A_Jour_Voisins()**. Si l'appel récusif de AES n'aboutit pas, on restaure cette copie au lieu d'appeler Remettre_A_Jour().
➙ Notons que ces copies nous coûteront à chaque fois une matrice (env. $Taille^2$ copies nécessaires).

Remarques sur la stratégie : l'algorithme AES ci-dessus est, par nature, un algorithme **irrévocable** dans la mesure où si on trouve un seul *meilleurs_voisin* pour (X, Y), on avancera tête baissée sans révoquer notre choix ; il nous mènera à une solution (si une solution existe et que nos choix précédents ont été bons).

Par contre, s'il y a plus d'un *meilleurs_voisin*, on est alors obligé d'en tenir compte : non pas tous ces *meilleurs_voisins* nous conduiront à un succès (si succès il y a). On ne sait pas d'avance lequel nous conduira à ce succès !

Test :
par exemple, la version heuristique pour N=5 et le départ = (2,4) donne la solution suivante (après retours arrières) :

23	2	7	12	25
8	17	24	1	6
3	22	13	18	11
16	9	20	5	14
21	4	15	10	19

Et pour un départ en $(4, 2)$, la solution sera :

23	10	13	4	21
12	5	22	9	14
17	24	11	20	3
6	1	18	15	8
25	16	7	2	19

N.B. : La stratégie employée ici est appelée "meilleur d'abord" (**Best First**) au sein d'une stratégie d'affectation **First-Fail** (voir cours).
Notons également que dans cette version, on se contente du premier succès (c'est courant lorsqu'on utilise une heuristique).
N.B. : Placer un compteur pour comptabiliser le nombre de retours arrières sur les meilleurs voisins.

12 Travail à rendre de cette étape et le barème

☞ Voir la section 13 page 16 ci-après pour une aide Python.

Fournir le code (+ rapport + vos réflexions) contenant cette version heuristique avec la valeur du compteur des retours arrières .

Rappel : si vous avez réalisé la version avec bande, ce 3e travail à rendre est un bonus.
Important : réfléchissez au gain en complexité temporelle pour cette version.

Barème global :

- Version de base : notée sur 12
- Ajout de la bande : notée sur 14
- Version heuristique (sans bande) : notée sur 18.
- Version heuristique + bande : notée sur 20.

Rappel : utilisez le code fourni ci-après pour vous faciliter la réalisation.

13 Aides et compléments à la version Heuristique

Nous travaillons ici sans utiliser sans utiliser l'optimisation "ajout d'une bande d'épaisseur 2".

• **Initialisations de l'échiquier** (sans bande) :

```
Echiquier=[[-1 for x in range(Taille)] for y in range(Taille)]
Tab_delta_X_Y=[[1,2],[1,-2],[-1,2],[-1,-2],[2,1], [2,-1], [-2,1],[-2,-1]]
```

☞ **Si vous préférez numpy :**

```
import numpy as np
mat=np.matrix([-1 for i in range(25)]).reshape((5,5))
mat
"""     matrix([ [-1, -1, -1, -1, -1],
                 [-1, -1, -1, -1, -1],
                 [-1, -1, -1, -1, -1],
                 [-1, -1, -1, -1, -1],
                 [-1, -1, -1, -1, -1]]) """
mat[2,3] = 12
mat
```

• **Lire la taille de l'échiquier et la case de départ :**

```
print("Donner la taille de l'échiquer (> 4): ")
Taille=int(input('? '))

# Lecture de la case de départ (sur la mm ligne) en commençant à 0 (avec controle)
while (True) :
    print("La case de départ : donner x puis y (sur la mm ligne, en commençant à ze
    try :
        x,y = input().split() # ne peut lire que des chaines
        x,y=int(x),int(y)       # On convertie
    except : continue
    break
```

☞ **Pour arrêter un programme, il y a les solutions suivantes :**

```
raise SystemExit(0)

# OU
import sys;  # Puis
sys.exit(0) # Ou
sys.exit("Gotcha")

#OU un simple
quit()
```

14 Compléments à la version Heuristique

Initialisation de la matrice du nombre des voisins.

• Attention : version SANS la bande d'épaisseur 2 cases autour. Voir les deux commentaires ("+2") si bande.

```
# Calcul de la matrice du nbr des Voisins : version échiquier sans bande
def init_voisin() :
    global Echiquier
    global Tab_delta_X_Y
    global Taille
    global Matrice_Nb_Voisins

    for i in range(Taille) :
        for j in range(Taille) :
            nb_voisins_libres=0
            for k in range(8) :
                NX = i+Tab_delta_X_Y[k][0]  # ajouter +2 si bande
                NY = j+Tab_delta_X_Y[k][1]  # ajouter +2 si bande
                if prometteur(NX, NY): nb_voisins_libres+=1
            Matrice_Nb_Voisins[i][j]= nb_voisins_libres

"""TEST :
Pour Taille = 5, on obtient
  2  3  4  3  2
  3  4  5  4  3
  4  5  8  6  4
  3  4  6  4  3
  2  3  4  3  2

Pour Taille = 6, on obtient
  2  3  4  4  3  2
  3  4  5  6  4  3
  4  5  8  8  6  4
  4  6  8  8  6  4
  3  4  6  6  4  3
  2  3  4  4  3  2

Pour 7 :
  2  3  4  4  4  3  2
  3  4  5  6  6  4  3
  4  5  8  8  8  6  4
  4  6  8  8  8  6  4
  4  6  8  8  8  6  4
  3  4  6  6  6  4  3
  2  3  4  4  4  3  2
```

• **Code de recherche des meilleurs voisins :**
➙ La fonction suivante enferme deux autres fonctions locales.

```
def trouver_meilleurs_voisins_libres(Case) :
    global Tab_delta_X_Y
    global Taille
    global Matrice_Nb_Voisins
    Vecteur_des_meilleurs_Voisins = [] # Le compte rendu de cette fonction
    #-------------------------------------------
    def trouver_le_premier_voisin_de_degre_minimal_libre(Case) :
    # Chercher la case avec un nbr de voisin minimal, si elle existe
        X,Y = Case[0], Case[1]
        min_k=9 # Une valeur plus grande que 8 (max voisins)
        Nx_k, Ny_k = -1, -1
        for k in range(8) :
            Nx, Ny = X+Tab_delta_X_Y[k][0], Y+Tab_delta_X_Y[k][1]
            if not prometteur(Nx, Ny) : continue

            if min_k > Matrice_Nb_Voisins[Nx][Ny] :
                min_k = Matrice_Nb_Voisins[Nx][Ny]
                Nx_k, Ny_k = Nx, Ny

        return (min_k)

    #-------------------------------------------
    # On  a déjà une Case de degré k, on trouve les autres du même degré
    def trouver_tous_les_best_voisins(Case, min_k) :
        X,Y = Case[0], Case[1]
        # La premeier meilleur voisins déjà trouvé sera insérée dans
        # le vecteur ci-dessous
        for z in range(8) :
            Nx, Ny = X+Tab_delta_X_Y[z][0], Y+Tab_delta_X_Y[z][1]
            if not prometteur(Nx, Ny) : continue
            if min_k == Matrice_Nb_Voisins[Nx][Ny] :
                Vecteur_des_meilleurs_Voisins.append((Nx, Ny))
        return Vecteur_des_meilleurs_Voisins
    # ----------------------------------
    # Le corps de la fonction
    min_k = trouver_le_premier_voisin_de_degre_minimal_libre(Case)
    if min_k >= 0 : # on a trouvé au moins une case
        Vecteur_des_meilleurs_Voisins = trouver_tous_les_best_voisins (Case, min_k)

    return Vecteur_des_meilleurs_Voisins
```

• Le code de la mise à jour et de la remise à jour de la matrice des voisins.

```
#-------------------------------------------------------------
# La mise à jour de la matrice des voisins après le choix du meilleur voisin
# Case=(X,Y) est le meilleur voisin. Elle sera utilisée mais avant, il
# faut M.A.J SES voisins
# On fait -1 sur le nb_voisins de chacun des voisins de Case
def Mise_A_Jour_Voisins(Case) :
    global Tab_delta_X_Y
    global Matrice_Nb_Voisins

    X,Y = Case[0], Case[1]
    for z in range(8) :
        Nx, Ny = X+Tab_delta_X_Y[z][0], Y+Tab_delta_X_Y[z][1]
        if not prometteur(Nx, Ny) : continue
        if Matrice_Nb_Voisins[Nx][Ny] > 0 : Matrice_Nb_Voisins[Nx][Ny] -= 1

#-------------------------------------------------------------
def ReMise_A_Jour_Voisins(Case) :
    global Tab_delta_X_Y
    global Matrice_Nb_Voisins

    X,Y = Case[0], Case[1]
    for z in range(8) :
        Nx, Ny = X+Tab_delta_X_Y[z][0], Y+Tab_delta_X_Y[z][1]
        if not prometteur(Nx, Ny) : continue
        Matrice_Nb_Voisins[Nx][Ny] += 1 # On avait fait -1 dans Mise_A_Jour_Voisins
```