Stratégies de Résolution de Problèmes BE1 : Cavalier et Backtracking

Taïga Gonçalves & Thomas Robbins

10 Septembre 2021

Alexandre Saïdi

Table des matières

Introduction

Ce rapport présente la résolution du problème du parcours du cavalier : étant donné un échiquier de taille $n \times n$ ($n > 4$) et un cavalier, nous souhaitons vérifier numériquement si en partant d'une case donnée, le cavalier peut parcourir toutes les cases de l'échiquier une et une seule fois. Pour ce faire, nous chercherons dans un premier temps à appliquer une méthode de retour en arrière (dite de *backtracking*) permettant d'obtenir une première solution quelle que soit la case de départ. Dans un deuxième temps, nous simplifierons le passage d'une case à la suivante en modifiant l'implémentation de l'échiquier. Enfin, nous chercherons à optimiser le programme en mettant en place une heuristique du type *Best First*.

1 Méthode de retour en arrière : *backtracking*

Dans cette partie, nous implémenterons une méthode de retour en arrière offrant une première solution au problème du cavalier pour une case donnée. Pour ce faire, nous implémenterons l'échiquier de la manière suivante : ce sera une matrice $n \times n$ dont les valeurs seront toutes initialisées à -1. La case choisie prendra la valeur n^2, et à chaque nouvelle case choisie dans le parcours, cette nouvelle case prendra la valeur suivante, jusqu'à ce que la matrice soit remplie des nombres 1 à n^2.

```
echiquier = np.matrix([-1 for i in range(taille**2)]).reshape(taille,
    taille)
```

Le choix de la case suivante dans le parcours se fait de manière simple : une liste contenant tous les déplacements possibles du cavalier est utilisée pour calculer plusieurs couples de coordonnées. Parmi ces cases, s'il y a une case dite "prometteuse" (c'est-à-dire si ses coordonnées sont bien comprises dans la matrice $n \times n$ et que sa valeur est encore -1), le parcours continue sur cette case.

```
def prometteur(newX, newY):
    return 0 <= newX < taille and 0<= newY < taille and echiquier[newX
    , newY] == -1
```

Le retour en arrière se fait dès que le cavalier est bloqué mais que tous les numéros n'ont pas encore été inscrits : on supprime alors la valeur de la case sur laquelle le cavalier se trouve (on "retourne en arrière"), de manière à regarder s'il n'existait pas d'autres cases prometteuses. S'il n'y en a pas d'autres, le cavalier retourne une fois de plus à la case précédente, et ainsi de suite.

```
def AES_parcours_cavalier_un_succes_suffit(derniere_case_traitee,
    prochain_numero):
    if prochain_numero > taille**2:
        return True # Toutes les cases (au nombre de taille**2) ont
    ete visitees
    else:
        x,y = derniere_case_traitee
        for i in range(8): # Etude de tous les deplacements possibles
            nextX = x + tabDeltaXY[i][0]
            nextY = y + tabDeltaXY[i][1]
            if prometteur(nextX,nextY):
                compteurs["compteurTentative"] += 1 # Tente de
    resoudre le probleme en deplacant le cavalier dans la case (nextX,
    nextY)
                echiquier[nextX, nextY] = prochain_numero
                if AES_parcours_cavalier_un_succes_suffit((nextX,
    nextY), prochain_numero+1):
```

```
                    return True # Recursivite pour verifier si toutes les cases non encore visitees sont accessibles en partant de (nextX,nextY)
                compteurs["compteurRetourArriere"] += 1 # Retour en arriere pour tenter un autre deplacement
                echiquier[nextX, nextY] = -1
        return False # Retourne False si aucun deplacement n'aboutit a la resolustion du probleme
```

Cette méthode a été implémentée à l'aide d'une méthode récursive. Pour suivre les actions réalisées dans cette boucle récursive, deux compteurs ont été crées : l'un compte le nombre de déplacements du cavalier (nombre de tentatives), et l'autre compte uniquement les déplacements réalisés lors d'un retour en arrière.

```
    compteurs = {"compteurTentative":0, "compteurRetourArriere":0}
```

Un premier test de ce programme a été réalisé dans la figure 1. Un solution est trouvée et la différence des deux compteurs donne 24, ce qui cohérent avec le nombre de déplacements sans retour que doit effectuer le cavalier. Les compteurs donnent également une estimation de la complexité de l'algorithme. Théoriquement, dans le pire des cas (c'est-à-dire en cas d'échec), les n^2 cases de l'échiquier possédant au plus 8 voisins, la complexité est en $O(n^{16}) \approx O(2^{16logN})$. Ce calcul est une sur-estimation. En pratique, si une solution existe pour des paramètres donnés, alors l'algorithme renvoie cette solution plus rapidement. Dans le cas de la figure 1, la complexité empirique est $O(5^7)$.

```
Donnez la taille de l'échiquier (>4): 5

Entrez x0 (Ligne de la case de départ): 0

Entrez y0 (Colonne de la case de départ): 0



[[ 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]]


{'compteurTentative': 74300, 'compteurRetourArriere': 74276}
Temps écoulé : 0.6193442344665527 s
```

FIGURE 1 – Résolution d'un échiquier de taille 5×5 par *backtracking*

2 Simplification de la recherche de la case suivante

Dans l'état actuel des choses, le code de la fonction *prometteur* réalise cinq tests, dont quatre pour s'assurer que la case testée est bien une case de la matrice. Nous cherchons ici à éliminer le cas de figure où la case testée n'est pas dans la matrice.

Afin d'y parvenir, nous allons modifier l'implémentation de l'échiquier : nous allons rajouter une bande de cases de largeur 2 autour de la matrice originelle (de taille $n \times n$)

```
1     echiquierBandes = np.matrix([-1 for i in range((taille+4)**2)]).
      reshape(taille+4,taille+4)
2       for i in range(taille+4) :
3           for j in range(taille+4) :
4               if i<2 or j<2 or i>=taille+2 or j>=taille+2 :
5                   echiquierBandes[i,j] = taille**2+1 # Affectation d'un
      numero >taille**2 dans la bande interdite pour rendre son acces
      impossible (grace a la fonction prometteur(case))
```

Ainsi, la fonction *prometteur* en est grandement simplifiée : puisque l'implémentation précédente assure de ne pas se trouver dans le cas où la case à tester est hors de la matrice, il ne reste plus qu'un test à réaliser, celui de savoir si la case a déjà été visitée.

```
1     def prometteur(newX, newY):
2     return echiquierBandes[newX, newY] == -1
```

La figure 2 correspond au résultat du même problème traité dans la figure 1, après ajout de la bande interdite. Le résultat obtenu est identique, mais l'exécution du code est légèrement plus rapide. En effet, le nombre de test réalisé dans chaque boucle a été divisé par 5, et donc la complexité a elle même été divisée par 5. Même si le code est plus rapide, cette amélioration reste négligeable lorsque la taille n de l'échiquier devient grand.

```
Donnez la taille de l'échiquier (>4): 5

Entrez x0 (Ligne de la case de départ): 0

Entrez y0 (Colonne de la case de départ): 0



[[ 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]]


{'compteurTentative': 74300, 'compteurRetourArriere': 74276}
Temps écoulé : 0.5196349620819092 s
```

FIGURE 2 – Résolution d'un échiquier de taille 5×5 par *backtracking*

3 Heuristique *Best First*

Nous allons maintenant modifier l'approche de manière à limiter le nombre de retours en arrière : nous allons utiliser une heuristique du type *Best First*. Plutôt que de choisir la première case prometteuse, il s'agit de trouver le meilleur choix à effectuer afin d'éviter de rencontrer des problèmes lors d'étapes suivantes. Ici, le meilleur choix sera la case à partir de laquelle il existe le moins de cases prometteuses ("voisins"). En effet, si nous sélectionnons les cases avec le plus petit nombre de voisins d'abord, il restera encore celles avec un plus grand nombre de voisins lorsqu'un nombre important d'étapes aura été effectué : il n'y aura pas de risque de tomber sur une case sans voisins.

Pour implémenter cette méthode, nous ferons donc usage d'une matrice de voisins dont chaque case contiendra le nombre de voisins encore disponibles. Ainsi, à chaque étape il sera nécessaire de mettre à jour cette matrice afin de corriger le nombre de voisins encore disponibles.

```
1     echiquierVoisins = np.matrix([9 for i in range((taille+4)**2)]).
    reshape(taille+4,taille+4) # Matrice de taille identique    la
    precedente, indiquant le nombre de voisins non visitees de chaque
    case. La valeur par defaut est 9 (soit plus que la maximum de
    voisins) pour distinguer les cases non accessibles (bande interdite
    )
2     init_voisin() # Calcul le nombre de voisins dans chaque cases de l
    'echiquier (ne modifie pas la bande interdite)
```

Le calcul des valeurs de la matrice des voisins est extrêmement aisé : il suffit de créer une boucle parcourant les voisins de la case en question est de vérifier s'ils sont prometteurs.

```
1 def init_voisin():
2     for i in range(2,taille+2): # Prend en compte le decalage du a la
    bande interdite
3         for j in range(2,taille+2):
4             nbVoisinsLibres = 0 # Nombre de voisins libres pour la
    case(i,j)
5             for k in range(8): # Verifie l'accessibilite de chaque
    case voisine de (i,j)
6                 NX = i + tabDeltaXY[k][0]
7                 NY = j + tabDeltaXY[k][1]
8                 if prometteur(NX,NY) :
9                     nbVoisinsLibres += 1 # la case (NX,NY) est
    accessible depuis (i,j) et prometteur (i.e. non visit e), donc c'
    est un voisin libre
10            echiquierVoisins[i,j] = nbVoisinsLibres # affecte le
    nombre de voisins libres de la case(i,j)
```

Cependant, le choix de la meilleure case nécessite un raisonnement plus long. On se donne une case c_0. Tout d'abord, il s'agit de parcourir les voisins de c_0 et de trouver le

plus petit nombre de voisins v (fonction *trouverLePremierVoisinDeDegreMinimalLibre*). Ensuite, il s'agit de reparcourir la liste des voisins et de stocker tous ceux qui ont un nombre de voisins v (fonction *trouverTousLesMeilleursVoisins*). En effet, rien ne garantit que le nombre minimal de voisins v est propre à une unique case, il peut y en avoir plusieurs. Ainsi, la sortie de la fonction totale *trouverMeilleurVoisinsLibres* est un vecteur de choix possibles.

```
def trouverMeilleurVoisinsLibres(case):
    vecteurMeilleursVoisins = [] # Liste contenant tous les voisins de la case etudiee, dans l'ordre croissant de leur nombre de voisin libre

    def trouverLePremierVoisinDeDegreMinimalLibre(case):
        """Cherche le plus petit nombre de voisins libres parmi les voisins de la case etudiee"""
        X,Y = case[0], case[1]
        min_k = 9 # Valeur renvoyee par la fonction par defaut plus grande que 8 (max voisins)
        for k in range(8): # Parcours tous les voisins de la case etudiee
            Nx,Ny = X+tabDeltaXY[k][0], Y+tabDeltaXY[k][1]
            if not prometteur(Nx, Ny) : continue

            if min_k > echiquierVoisins[Nx,Ny] :
                min_k = echiquierVoisins[Nx,Ny] # Actualise le nombre minimal de voisins libres
        return min_k

    def trouverTousLesMeilleursVoisins(case, min_k):
        """Renvoie la liste de tout les voisins ayant le minimum de voisin libre (minimum connu grace a la fonction trouverLePremierVoisinDeDegreMinimalLibre"""
        X,Y = case[0],case[1]
        for z in range(8): # Parcours tous les voisins de la case etudiee
            Nx, Ny = X+tabDeltaXY[z][0], Y+tabDeltaXY[z][1]
            if not prometteur(Nx,Ny) : continue
            if min_k == echiquierVoisins[Nx,Ny]:
                vecteurMeilleursVoisins.append((Nx,Ny)) # Ajoute le voisin si son nombre de voisins libres est minimal
        return vecteurMeilleursVoisins

    min_k = trouverLePremierVoisinDeDegreMinimalLibre(case)
    if min_k >= 0:
        vecteurMeilleursVoisins = trouverTousLesMeilleursVoisins(case, min_k)
    return vecteurMeilleursVoisins
```

Une fois créée la fonction de sélection des meilleurs voisins, nous pouvons réécrire la fonction du parcours du cavalier. La boucle qui s'appliquait précédemment à tous les

voisins de la case traitée ne s'applique plus qu'aux meilleurs voisins : on ne peut choisir que celles-ci.

```
def AES_parcours_cavalier_un_succes_suffit(derniere_case_traitee, prochain_numero):
    if prochain_numero > taille**2:
        return True # Toutes les cases (au nombre de taille**2) ont ete visitees
    else:
        x,y = derniere_case_traitee
        for caseChoisie in trouverMeilleurVoisinsLibres(derniere_case_traitee): # Parcours seulement les cases possedant le plus petit nombre de voisins
            miseAJourVoisins(caseChoisie) # La case choisie n'est plus consideree comme libre du point de vue de ses voisins
            nextX,nextY = caseChoisie[0],caseChoisie[1]
            compteurs["compteurTentative"] = compteurs["compteurTentative"]+1 # Tente de resoudre le probleme en deplacant le cavalier a la case (nextX,nextY)
            echiquierBandes[nextX, nextY] = prochain_numero
            if AES_parcours_cavalier_un_succes_suffit((nextX, nextY), prochain_numero+1):
                return True # Recursivite pour verifier si toutes les cases non encore visitees sont accessibles en partant de (nextX, nextY)
            compteurs["compteurRetourArriere"] = compteurs["compteurRetourArriere"]+1 # On retourne en arriere pour tenter un autre deplacement
            remiseAJourVoisins(caseChoisie) # La case qui etait choisie redevient libre du point de vue de ses voisins
            echiquierBandes[nextX, nextY] = -1
        return False # Retourne False si aucun deplacement n'aboutit a la resolustion du probleme
```

Dès lors qu'un choix a été effectué parmi les meilleurs voisins, il s'agit comme précisé auparavant de mettre à jour la matrice des voisins (fonction *miseAJourVoisins*). Il suffit de reparcourir les voisins de la case en enlevant les voisins parcourus, c'est-à-dire ceux auxquels ont été affectés un numéro.

Il faut également créer la fonction opposée, qui rajoute un voisin dans le case où un retour en arrière est effectué (fonction *remiseAJourVoisins*).

```
def miseAJourVoisins(case):
    """La case etudiee est choisie, ses voisins ne doivent donc plus la considerer libre"""
    X,Y = case[0], case[1]
    for z in range(8): # Parcours tous les voisins de la case etudiee
        Nx,Ny = X + tabDeltaXY[z][0], Y+tabDeltaXY[z][1]
        if not prometteur(Nx, Ny) : continue
        if echiquierVoisins[Nx,Ny] > 0 : echiquierVoisins[Nx,Ny] -= 1 # Enleve un voisin libre
```

```
8
9  def remiseAJourVoisins(case):
10     """La case etudiee n'est plus choisie (retour en arriere), ses
       voisins doivent donc la reconsiderer libre"""
11     X,Y = case[0],case[1]
12     for z in range(8): # Parcours tous les voisins de la case etudiee
13         Nx,Ny = X + tabDeltaXY[z][0], Y+tabDeltaXY[z][1]
14         if not prometteur(Nx, Ny) : continue
15         echiquierVoisins[Nx,Ny] += 1 # Rajoute un voisin libre
```

La figure 3 résout le même problème que les figures 1 et 2, mais en un temps minimal, puisqu'aucun retour en arrière n'est effectué.

```
Donnez la taille de l'échiquier (>4): 5

Entrez x0 (Ligne de la case de départ): 0

Entrez y0 (Colonne de la case de départ): 0



[[26 26 26 26 26 26 26 26 26]
 [26 26 26 26 26 26 26 26 26]
 [26 26  1 12 25 18  3 26 26]
 [26 26 22 17  2 13 24 26 26]
 [26 26 11  8 23  4 19 26 26]
 [26 26 16 21  6  9 14 26 26]
 [26 26  7 10 15 20  5 26 26]
 [26 26 26 26 26 26 26 26 26]
 [26 26 26 26 26 26 26 26 26]]



[[ 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]]


{'compteurTentative': 24, 'compteurRetourArriere': 0}
Temps écoulé : 0.0009949207305908203 s
```

FIGURE 3 – Résolution d'un échiquier de taille 5×5 par *backtracking* après utilisation de l'heuristique *Best First*

Dans la plupart des cas, lorsqu'une solution existe pour des paramètres donnés, le code n'effectue aucun retour en arrière. La figure 4 montre l'un des rares situations où le code doit réaliser un retour en arrière. Ainsi, lorsqu'une solution existe, il est possible d'espérer d'obtenir cette dernière avec une complexité en $O(n^2)$, ce qui permet de résoudre des problèmes pour des échiquiers de très grandes tailles (figure 5), alors qu'une méthode sans heuristique a beaucoup de difficultés à résoudre un problème pour échiquier de taille 7×7.

```
Donnez la taille de l'échiquier (>4): 7

Entrez x0 (Ligne de la case de départ): 4

Entrez y0 (Colonne de la case de départ): 2



[[13 36 29 32 11 48 21]
 [28 33 12 49 22 31 10]
 [37 14 35 30 41 20 47]
 [34 27 38 23 46  9 40]
 [15 24  1 42 39  6 19]
 [ 2 43 26 17  4 45  8]
 [25 16  3 44  7 18  5]]


{'compteurTentative': 70, 'compteurRetourArriere': 22}
Temps écoulé : 0.002991914749145508 s
```

FIGURE 4 – Résolution d'un échiquier de taille 7×7 par *backtracking* après utilisation de l'heuristique *Best First*

```
Donnez la taille de l'échiquier (>4): 50

Entrez x0 (Ligne de la case de départ): 0

Entrez y0 (Colonne de la case de départ): 0



[[2501 2501 2501 ... 2501 2501 2501]
 [2501 2501 2501 ... 2501 2501 2501]
 [2501 2501    1 ...   28 2501 2501]
 ...
 [2501 2501  462 ...  433 2501 2501]
 [2501 2501 2501 ... 2501 2501 2501]
 [2501 2501 2501 ... 2501 2501 2501]]



[[   1    4  105 ...  142  131   28]
 [ 104 2363    2 ...   27  134  143]
 [   3  100 2365 ...  148   29  132]
 ...
 [ 458  461  520 ... 1051  432  487]
 [ 513   76  463 ...  434  965   52]
 [ 462  457  514 ...   53  486  433]]


{'compteurTentative': 2499, 'compteurRetourArriere': 0}
Temps écoulé : 0.1177511215209961 s
```

FIGURE 5 – Résolution d'un échiquier de taille 50×50 par *backtracking* après utilisation de l'heuristique *Best First*

Cette méthode possède également une complexité très affaibli dans le pire des cas. En effet, le code n'essaye plus les 8 déplacements possibles du cavalier à chaque récursion (grâce à la fonction `trouverMeilleurVoisinsLibres`), ce qui réduit considérablement la nombre total de tentatives à réaliser. Ce nombre de tentatives peut être différent selon la configuration étudiée, comme le montre les figures 6 et 7. Pour rappel, la méthode non heuristique possède une complexité dans le pire des cas si grande qu'il n'est pas possible de résoudre un échiquier de taille 7×7, qu'il y ait ou non une solution.

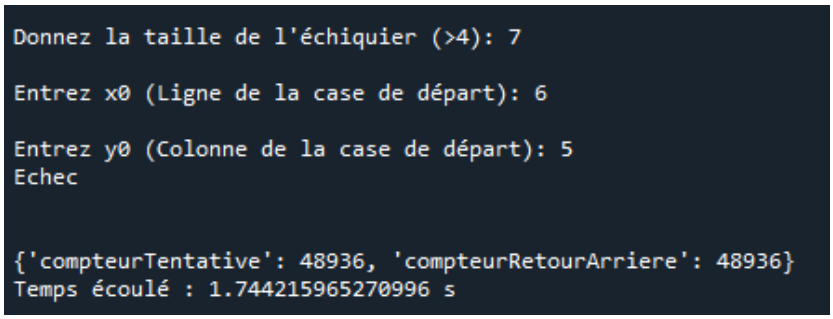

FIGURE 6 – Résolution d'un échiquier de taille 7×7 par *backtracking* après utilisation de l'heuristique *Best First*

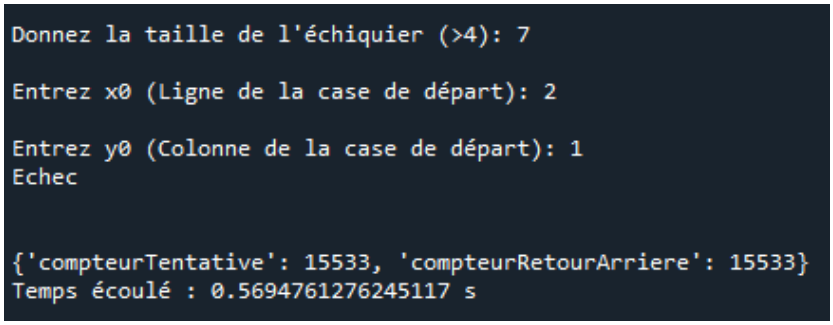

FIGURE 7 – Résolution d'un échiquier de taille 7×7 par *backtracking* après utilisation de l'heuristique *Best First*

4 Conclusion

Dans ce rapport, nous avons présenté notre démarche de résolution du problème du parcours du cavalier. Tout d'abord, nous avons adopté une méthode de backtracking : le cavalier choisit une case prometteuse (non encore parcourue et située dans la matrice implémentant l'échiquier) et revient en arrière pour changer de choix dès qu'il arrive sur une case à partir de laquelle il ne peut plus bouger (s'il n'a pas fini). Ensuite, nous avons cherché à optimiser cette méthode : nous avons implémenté deux bandes autour de la matrice originelle de manière à éliminer quatre tests de la fonction *prometteur*. Enfin, nous avons adopté une nouvelle approche, toujours avec l'optimisation précédente, qui

permet de réduire les retours en arrière à un nombre quasiment nul. C'est la méthode de *Best First*, où nous effectuons le choix le plus probable d'éviter les retours en arrière, ici la case prometteuse avec le moins de voisins encore disponibles. C'est de là que vient le nom *Best First* : nous optons pour le meilleur choix.

5 Complément : Résolution d'un labyrinthe

En complément, l'algorithme à essais successifs révocable a également été implémenté pour résoudre un problème de parcours de labyrinthe. Le problème a été résolu avec et sans heuristique. Ici, l'heuristique utilisé ne permet pas de réduire la complexité de l'algorithme, mais de trouver un des chemins les plus courts pour atteindre le labyrinthe.

Le code complet est donné dans le fichier *BonusLabyrinthe*