BE Labyrinthe

Septembre 2022

Version Elèves (AS)

2022-23

I Introduction

• Trouver la sortie dans un Labyrinthe (espace 2D discrétisé).
• On suivra le principe d'un parcours de graphe à la recherche d'une solution dans un espace d'états. Ce sujet a été traité depuis très longtemps. L'algorithme utilisé est un **algorithme à essais successifs révocable** (**AES**) (avec retours arrières, ...)
• Dans un premier temps, on étudie l'algorithme AES puis on ajoutera la calcul du trajet avant de s'intéresser à une heuristique.

I-1 Parcours générique avec retours arrières

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

```
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)
1  début
2      si Noeud_courant est un état final alors
3          renvoyer Succès
4      sinon
5          pour tous les Noeud_suivant successeurs de Noeud_courant dans G faire
6              si Prometteur(G, Noeud_suivant) alors
7                  si AES_le_premier_succes_suffit(G, Noeud_suivant) = Succès alors
8                      renvoyer Succès
9                  fin
10             fin
11         fin
12         renvoyer Échec
13     fin
14 fin
```

• Explications : voir BE1 (S7 2020-21).
• Note : en ligne 5 de l'algorithme ci-dessus, on pourrait d'abord constituer un ensemble de candidats (soit $candidat\_set$) puis traiter chaque élément de cet ensemble.

```
....
candidat_set ← tous les successeurs de Noeud_courant dans G
pour tous les Noeud_suivant ∈ candidat_set faire
    si Prometteur(G, Noeud_suivant) alors
        si ... alors
        fin
    fin
fin
```

☞ Il faudra cependant faire bien attention aux situations où un des éléments de $candidat\_set$ se modifie par un des appels récursifs à AES suite à quoi cet élément là ne sera plus un véritable candidat (mais il l'avait été avant sa modification, lorsqu'on a constitué l'ensemble des candidats).
Ce rôle de contrôle est donné à la fonction **prometteur**. Ainsi, si le k^{em} successeur est affecté lors du traitement d'un précédent candidat, et si (p. ex.) cette modification implique que l'on écarte le traitement du k^{em} successeur, la fonction *prometteur* devra écarter ce successeur qui n'en est plus un !

I-2 Adaptation au parcours du labyrinthe

• A noter

◦ Toute case a potentiellement 4 voisins possibles. On évite les déplacements en diagonale.

Il suffit donc d'utiliser une liste de 4 couples qui donne les deltas $(\Delta x, \Delta y)$ pour se déplacer (comme dans le cas du cavalier).

◦ Plus tard : la version itérative (ainsi que le parcours en largeur !)

```
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
```

• A noter

◦ Convention de numérotation des cases (lire ci-dessous) :

— libre : -1

— murs : on note ces cases '1'

— départ : 2

— arrivée : 3

— occupé / marqué : 4

Cette numérotation est en rapport direct avec le calcul du trajet.

Concernant le trajet emprunté en cas de succès, on pourra, comme pour le cavalier, inscrire dans les cases le numéro de l'étape. Ce qui nous donnera le trajet en cas de succès. Mais dans ce cas, la convention de numérotation ci-dessus devrait être modifiée de sorte que l'intersection des numéros des étapes et la convention ci-dessus soit vide.

Par contre, si le trajet est construit contenant la suite des cases empruntées, nou spouvon sconserver cette convention.Voir la version avec Trajet.

☞ Marquage des cases empruntées :

➛ Si on souhaite un seul chemin, on peut marquer les cases pour ne pas y revenir. Le fait d'inscrire les numéros des étapes dans les cases permet de

II Remarque sur la complexité de la recherche

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 grand** : toutes les N^2 cases sont considérées pouvant avoir 4 voisins possibles et donc on aura une complexité $O(N^4) = O(2^{4\log N})$.

➞ Cette valeur est une sur-estimation (*pessimiste*) et se constate dans un cas d'échec (aucune solution pour N et (X,Y) de départ donnés).

III Heuristique du PCC

◦ On commence par la case de départ et dans les 4 direction, on note "1" dans cases successeurs qui auront été filtrés par la fonction prometteur. On ne prends donc pas en compte les murs et les cases hors échiquier.
◦ De manière récursive (ce sera plus simple), on inscrit la valeur $min(actuelle\_valeur, V+1)$ dans les cases successeurs de chaque case dont la valeur est V. $actuelle\_valeur$ est la valeur éventuelle et actuelle de la case qui recevra une nouvelle valeur.
• Cela permet le calcul des PCCs.

III-1 Algorithme de numérotation

Cet algorithme est également à base du schéma AES. Voir la version alternative ensuite.

Fonction numerotation_des_cases ;
Données : $Echiquer$: le labyrinthe ;
$Mat\_des\_numeros$: une matrice de la taille de $Echiquer$ qui reçoit les numéros
$Noeud\_courant = case\_dont\_on\_numrote\_les\_voisins$,
$numero\_de\_cette\_case$
Résultat : $Mat\_des\_numeros$: la matrice contenant les numéros (circulaires) des cases
début
  si $Noeud\_courant$ est un état final = la sortie en vue **alors**
    retourne
  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 Noeud_suivant correspond à la case départ (2) / Arrivée (3) **alors**
          continue
        fin
        si $Mat\_des\_numeros[Noeud\_suivant]$ contient déjà un numéro $< numero\_de\_cette\_case$ **alors**
          continue # cela évite de tourner en rond sans améliorer les numéros déjà affectés
        fin
        $Mat\_des\_numeros[Noeud\_suivant]$ =
          min($Mat\_des\_numeros[Noeud\_suivant]$, $numero\_de\_cette\_case$ + 1) ;
        numerotation_des_cases($Echiquer$,$Mat\_des\_numeros$, $Noeud\_suivant$, $Mat\_des\_numeros[Noeud\_suivant]$)
      fin
    fin
  fin
fin

• Initialisation avant d'appeler l'algorithme ci-dessus :

Initialiser $Mat\_des\_numeros$: inscrire ∞ dans les cases ne contanant pas les valeurs mur/départ/arrivée (1/2/3)
$Mat\_des\_numeros$ = initialiser_matrice_des_numeros($Echiquer$)
numerotation_des_cases($Echiquer$,$Mat\_des\_numeros$, $Case\_dpart$, $offset$)

☞ A propos de offset : la numérotation commencera à 101 pour les voisins de départ, pour ne pas interférer avec la convention de numérotation.

Fonction prometteur :

bool prometteur(Echiquier, Taille, New_X, New_Y) :
  <New_X, New_Y> est une case valide dans *Echiquier* si les tests suivants réussissent :
    $New\_X \geq 0; New\_X < Taille$
    $New\_Y \geq 0; New\_Y < Taille$
    Echiquier[New_X][New_Y] != 1 // mur
  Renvoyer le résultat de ces tests (vrai ou faux)

III-2 Exemple de numérotation

Exemple de numérotation :
Pour un labyrinthe tel que :

```
1   0   2   0   0
0   1   0   0   1
0   0   0   1   0
0   1   0   0   0
1   0   0   1   3
```

La matrice de numérotation sera :

```
  1   100   2     100   101
104    1    100   101    1
103   102   101    1     105
104    1    102   103   104
  1   104   103    1     3
```

III-3 Les chemins les plus courts (PCCs)

Une fois la matrice des numéros disponible, il suffira de partir de la case départ, choisir un des voisins qui contient la valeur $offset$ puis, de voisin en voisin, trouver une case qui contient le numéro de la case actuelle+1 ... jusqu'à la cas arrivée (sortie).
☞ Pour trouver l'ensemble des PCCs, une fois de plus, l'alorithme AES nous servira.

Ci-dessous, l'algorithme qui donnera un de ces PCCs. Adapter pour obtenir tous les PCCs.

```
Fonction Un_des_PCCs ;
Données : Echiquer : le labyrinthe ;
          Mat_des_numeros : une matrice de la taille de Echiquer qui reçoit les numéros
          Noeud_courant
Résultat : une liste des cases qui constitue le trajet
début
    si Noeud_courant est un état final = la sortie en vue alors
        retourne [Noeud_courant] # dernière case du trajet
    fin
    pour tous les Noeud_suivant = une des directions possibles à partir du Noeud_courant dans G faire
        si Mat_des_numeros[Noeud_suivant] = Mat_des_numeros[Noeud_courant] + 1 alors
            # On va construire le restant du trajet
            Trajet_restant=Un_des_PCCs(Echiquer, Mat_des_numeros, Noeud_suivant)
            si Trajet_restant != [] alors
                retourne [Noeud_suivant] + Trajet_restant # trajet
            fin
        fin
    fin
    renvoyer []
fin
```

☞ Si trajet=[], on n'aura pas de solution pour le labyrinthe testé.
☞ Le trajet sera complet quand on aura ajouté la case de départ au trajet (non vide) obtenu.
☞ Comme pour le cas du cavalier, on peut extraire tous les trajets PCCS (qui auront la même longueurs).

IV Bonus+ : Version probabiliste

A l'image des processus markoviens (automate probabiliste), faire un parcours probabiliste dans ce labyrinth. Initialement,

- les cases de départ recoivent une probabilité nulle (pour ne pas y aller) ;
- les cases d'arrivées recoivent une probabilité maximale (1) ;
- les murs et obstacles : nulle ;
- les autres cases : dans les 4 directions cardinales possibles, chaque cases voisine aura initialement une probabilité identique aux autres ;
- lorsqu'une cases est empruntée, on abaisse sa probabilité légèrement (pour ne pas y aller une 2e fois) sans toute fois se priver totalement de cette possibilité.
- lorsque l'on revient en arrière sur une case (donc échec), il faudra traduire l'échec observé en empruntant cette case par une forte baisse de sa probabilité.
- Ainsi, on choisit toujours la case voisine la plus probable.....pour effectuer un parcours (sans forcement obtenir le chemin le plus court) ;
- Dans un 2e temps, on essaiera d'obtenir le chemin le plus court : le chemin dont les cases ont les meilleures probabilités.

☞ Il serait souhaitable (mais lourd en calcul) pour que si la probabilité d'une case diminue, de modifier la probabilité des ces cases voisines pour avoir une somme de probabilités = 1. ET ce à propager dans tout l'échiquier.
En général, ces calculs (propagation de probabilités) sont ignorés pour ne pas alourdir l'exécution de code.

☞ Une version graphique (tkinter , QT, ?) vous permettra d'observer l'évolution des probabilités avec des couleurs différentes : rouge : probabilité basse, verte : probabilité haute, orange : probabilité intermédiaire, ...
➛ Noter que ces couleurs changent au fur et à mesure que l'on avance dans le labyrinthe.

V Annexes : aide et code divers

V-1 Remplissage aléatoire d'un labyrinthe

• Ci-dessus, le début de la classe Labyrinthe et le code du remplissage (si on n'a pas fourni les paramètres départ et arrivé)+ un exemple du main.
➛ On évitera que le départ est l'arrivée soient trop proches. Pour cela, on exige que Départ et Arrivée soient sur les bords et on évitera que ces deux cases soient aléatoirement placées sur le même bord (sauf dans les coins opposés).

```
class Labyrinthe:
    voisins = [(−1, 0), (0, 1), (1, 0), (0, −1)]

    def __init__(self, taille, debut=None, fin=None):
        self.__taille = taille
        if debut != None:
            L,C = debut
            assert(0 <= L < self.__taille and 0 <= C < self.__taille)
             #assert (debut >= 0 and debut < taille * taille)
        if fin != None:
            L, C = debut
            assert (0 <= L < self.__taille and 0 <= C < self.__taille)

        self.__debut = debut  # peut être None. De la forme (L,C)
        self.__fin = fin
        self.__matrice = []

        # On décide que la case debut et fin doivent avoir une distance de (taille/2) en ligne et en col
        self.__distnce_entre_debut_et_fin_not_used=math.ceil(taille/2) # arrondi par exèse de taille/2
        print('labyrinth créé')


    def remplir(self, nb_murs):
        # Fonction utilitaire pour définir debu / fin (ce sont des couples (L,C)
        def fixer_case_debut_fin() : # On vient ici car debut et fin = None. Il faut les fixer

            # On décide que la case debut et fin doivent avoir une distance de DIST (eg. 3) en ligne et en col
            # Fixer les cases début et fin (peuvent être données)
```

```
        N = self.__taille  # Pour simplifier les écritures
        num_case_debut = -1
        num_case_fin =  -1

        # Pour les tirages random j'ai besoin des ces num case plutot que des tuples (pour debut et fin)
        if num_case_debut not in range(N * N) and num_case_fin not in range(N * N):
            # Si non définis, mettre début (2) et cible (3) sur le contour extérieur
            Top_bande = list(range(N))  # [0.. taille]
            Bottom_bande = list(range(N * N - N, N * N))
            Left_bande = list(range(0, N * N, N))
            Right_bande = list(range(N - 1, N * N, N))

            Les_4_listes = [Top_bande, Bottom_bande, Left_bande, Right_bande]
            # Tirage du début : on choisit l'un de ces 4 liste de contours pour y mettre le début
            quelle_liste_debut = random.randint(0, 3)  # On a 4 listes de contours 0..3
            # On sait que chaucun des 4 listes est de taille N  : len(Les_4_listes[quelle_liste])=N
            num_case_debut = Les_4_listes[quelle_liste_debut][random.randint(0, N - 1)]
            # Tirage de fin != debut
            quelle_liste_fin = random.randint(0, 3)

            # Debut est fixé. On ne le touch eplus
            self.__debut = num_case_debut // N, num_case_debut % N

            while True:
                while quelle_liste_debut == quelle_liste_fin:
                    quelle_liste_fin = random.randint(0, 3)
                num_case_fin = Les_4_listes[quelle_liste_fin][random.randint(0, N - 1)]
                if num_case_fin != num_case_debut: break

            self.__fin = num_case_fin // N, num_case_fin % N

    # On met des 0 partout d'abord
    for i in range(0, self.__taille):
        lig = [0 for i in range(self.__taille)]
        self.__matrice.append(lig)

    if self.__debut == None and self.__fin == None :
        fixer_case_debut_fin()
        print("debut / fin = ", self.__debut, self.__fin)
        self.__matrice[self.__debut[0]][self.__debut[1]] = 2
        self.__matrice[self.__fin[0]][self.__fin[1]] = 3
    # Else : l'utilisateur les a déjà fourni

    # Tirage aléatoire de k  '1' (mur)
    assert (nb_murs > 0 and nb_murs < self.__taille * self.__taille)

    # on a traité d'abord case debut et fin (que l'on cenvertit en un num_case
    num_case_debut = self.__debut[0] * self.__taille + self.__debut[1]
    num_case_fin = self.__fin[0] * self.__taille + self.__fin[1]

    # On met les murs après avoir traité case debut et fin
    for i in range(nb_murs):
        case_mur = random.randint(0, self.__taille * self.__taille - 1)
        if case_mur == num_case_debut or case_mur == num_case_fin : # recommencer
            continue

        ligne = case_mur // self.__taille
        col = case_mur % self.__taille
        self.__matrice[ligne][col] = 1



if __name__ == "__main__":
    taille=7
    lab = Labyrinthe(taille) # sans donner debut/fin. seront générés aléatoirement
    lab.remplir(taille*taille//3)
    lab.afficher()
    ....
```