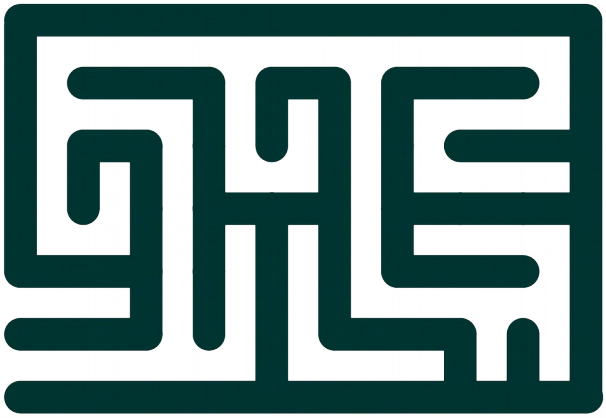
Ismatou Souaré
Thomas Citharel
Valérian Ynden

Le 22 Janvier 2016

DOSSIER LABYRINTHE



UNIVERSITE D'ORLEANS

ORLÉANS iut INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

Table des matières

I – Présentation du sujet

L'objectif de ce projet en groupe était de programmer le jeu de plateau très connu « Labyrinthe ».

Il s'agit d'un jeu de société permettant de deux à quatre joueurs de se confronter dans une chasse aux trésors dont le terrain est un labyrinthe dont la configuration change à chaque tour.

Chaque case du labyrinthe est représenté par une carte à jouer. Les trésors contenus par ces cartes sont représentés par des icônes. Les joueurs sont représentés par des pions.

Certaines cartes sont fixes tandis que d'autres sont amovibles et leur déplacement permettra de changer la configuration du labyrinthe.

Une carte supplémentaire permet ce déplacement et est différente d'un tour sur l'autre, car elle se fait insérer sur certaines lignes ou colonnes, en sortant une autre à sa place.

Le joueur qui trouve tous les trésors qui lui ont été assignés le premier gagne la partie.

II – Répartition du travail

Ismatou s'est chargée de réaliser l'implémentation des matrices tant dans la version classique que dans la version objet ainsi que les fonctions permettant de réaliser les décalages sur celles-ci.

Thomas s'est chargé d'implémenter les cartes en version classique et en objet, le labyrinthe en version classique et en objet mais aussi les modifications nécessaires dans les fichiers du labyrinthe texte et graphique tant en version classique qu'en version objet.

Valérian s'est chargé d'implémenter les joueurs en version classique et en version objet ainsi que l'algorithme de recherche de chemin dans le labyrinthe.

Les phases de débogage ont pour la plupart été réalisées ensemble.

III – Structures de données

La structure de données choisie pour chaque élément de notre labyrinthe est le dictionnaire.

En Python le dictionnaire est plus versatile que les listes, notamment par le fait que ses clés sont nommées et que le dictionnaire peut contenir des données de type différents. L'utilisation d'un dictionnaire permet d'avoir une extensibilité du code.

1 . Matrice

Structure de données :

```
Matrice = {
      'Ligne' : (int),
      'Colonne' : (int),
      'Valeurs' : list(T)
}
```

La clé « Ligne » maintient le nombre de lignes de la matrice, la clé « Colonne » maintient le nombre de colonnes de la matrice.

La clé « Valeurs » maintient une liste de valeurs, le type de données contenues dans cette liste dépend de l'utilisateur. De la manière dont nous allons utiliser les matrices par la suite, une valeur correspondra à une carte.

2 . Joueur

Structure de données :

```
Joueur = {
      'idjoueur' : int,
      'trésors' : list(int),
      'position' : (int, int)
}
```

La clé « idJoueur » permet de maintenir l'identifiant d'un joueur, l'identifiant du joueur correspond à son index dans la liste des joueurs.

La clé « trésors » correspond à la liste des trésors que le joueur doit trouver au cours de sa partie. Celle-ci est différente pour chaque joueur et est initialisée en début de partie.

La clé « position » maintient la position de départ du joueur. En objet, la propriété « position » du joueur est actualisée à chaque tour.

3 . Carte

Structure de données :

```
Carte = {
      'direction' : list(bool),
      'pions': list(int),
      'tresor' : int
}
```

La clé « direction » maintient une liste de booléens, la longueur de cette liste est fixe (4 éléments), chaque booléen représente la présence d'un mur sur la carte. La règle spécifiant qu'une carte ne peut contenir que 1 ou 2 murs est gérée dans l'initialisation des cartes.

La clé « pions » maintient une liste d'entiers représentant la présence de pions sur la carte, les éléments dans cette liste sont les identifiants des joueurs.

La clé « trésor » représente la présence ou non d'un trésor sur la carte. Si l'entier positif est différent de 0 alors un trésor est présent, et l'entier est l'identifiant de ce trésor.

4 . Labyrinthe

Structure de données :

```
Labyrinthe = {
      'joueurs' : list(joueur),
      'nbJoueurs' : int,
      'plateau' : matrice,
      'trésors' : list(int),
      'nbTrésors' : int,
      'joueurCourant' : int,
      'phaseCourante' : int,
      'carteAmovible' : carte,
      'directionInterdite' : str,
      'rangéeInterdite' : int
}
```

La clé « joueurs » maintient la liste des joueurs de la partie, les éléments de cette liste sont les dictionnaires représentant les joueurs.

La clé « nbJoueurs » représente le nombre de joueurs de la liste.

La clé « plateau » permet d'obtenir la matrice représentant le plateau de jeu.

La clé «trésors » maintient la liste des trésors présents sur le plateau, celle-ci est actualisée lorsque des trésors sont trouvés. Le nombre de trésors restant sur le plateau est consultable à travers la clé « nbTrésors ».

La clé « joueurCourant » permet d'obtenir l'identifiant du joueur dont c'est le tour.

La clé « phaseCourante » permet d'obtenir le numéro de la phase en cours (1 ou 2)

La clé « carteAmovible » maintient la carte à jouer.

Les clés « directionInterdite » et « rangéeInterdite » permettent de gérer la règle des coups interdits.

IV – Algorithmes implémentés

1 . Décalage dans la matrice

Afin de décaler les cases sur une ligne ou une colonne puis insérer une nouvelle valeur (mais aussi récupérer la valeur éjectée) dans la matrice nous avons utilisé une approche itérative.

Nous commençons par récupérer la valeur qui sera éjectée par le décalage à l'indice voulu.

Pour chaque élément concerné on récupère sa valeur puis on l'affecte à la position voulue dans la matrice, puis nous insérons la nouvelle valeur à la position devenue libre.

Enfin nous retournerons la valeur éjectée.

```
def decalageColonneEnBas(matrice, numCol, nouvelleValeur=0):
    valeurRecup=getVal(matrice, getNbLignes(matrice) - 1,numCol)

    for j in range(getNbLignes(matrice) - 1, 0, -1):
        nouvelleVal = getVal(matrice, j - 1, numCol)
        setVal(matrice, j, numCol, nouvelleVal)

    setVal(matrice, 0, numCol, nouvelleValeur)
    return valeurRecup
```

2 . Initialisation des trésors du joueurs

Les trésors sont initialisés aléatoirement à partir du nombre total de trésors et éventuellement à partir du nombre de trésors maximum par joueurs. Pour assurer que deux joueurs ne reçoivent pas le même trésors à trouver on stocke dans un ensemble les identifiants des trésors déjà distribués.

```
def initTresor(joueurs, nbTresors, nbTresorMax):
      listeTresorsDejaDistribues = set()
      for i in range(1,len(joueurs)+1):
            n = nbTresorMax if nbTresorMax != 0 else (nbTresors //
len(joueurs))
            j = 0
            while (j < n):
                  idTresorADistribuer = random.randint(1, nbTresors-1)
                  if (idTresorADistribuer not in
listeTresorsDejaDistribues):

      listeTresorsDejaDistribues.add(idTresorADistribuer)
                              joueurs[i-1]
["trésors"].append(idTresorADistribuer)
                              j += 1
```

3 . Codage des murs

Les murs d'une carte sont représentés par une liste de 4 booléens, qui sont transformés en bits de valeur 0 ou 1 afin de coder un nombre permettant de représenter la carte à afficher. Le nombre est sous la forme bObSbEbN.

```
def coderMurs(c):
    return int(str(int(c['direction'][3]))+str(int(c['direction'][2]))+
str(int(c['direction'][1]))+str(int(c['direction'][0])),2)

def decoderMurs(c,code):
    for i in range(len(code)):
        c['direction'][i] = bool(int(code[i]))
```

4 . Initialisation du labyrinthe

Les joueurs sont créés en fonction des paramètres du labyrinthe.

Une matrice (7x7) représentant le plateau est créée pour recevoir les cartes.

```
if 1 < len(joueurs) and len(joueurs) < 5:
      joueurs[0]['position'] = (0,0)
      joueurs[1]['position'] = (0,6)
      if len(joueurs) >= 3:
            joueurs[2]['position'] = (6,0)
      if len(joueurs) == 4:
            joueurs[3]['position'] = (6,6)

tresors = [i for i in range(nbTresors)]
tresorsPos = {}
```

La position des joueurs est initialisée dans les coins du labyrinthe, ainsi que la liste des trésors.

```
setVal(plateau,0,0,Carte(True,False,False,True,0,[1]))
setVal(plateau,0,2,Carte(True,False,False,False,1,[]))
setVal(plateau,0,4,Carte(True,False,False,False,2,[]))
setVal(plateau,0,6,Carte(True,True,False,False,0,[2]))
...
```

On initialise les cartes fixes sur le plateau avec les pions joueurs s'il y a lieu.

```
listeCartesAmovibles,carteAmovible2 = creerCartesAmovibles(13,nbTresors)
      k = 0
      for i in range(7):
            for j in range(7):
                  if i % 2 == 1:
                        setVal(plateau,i,j,listeCartesAmovibles[k])
                        tresorsPos[listeCartesAmovibles[k]['tresor']] = (i,j)
                        k+=1
                  elif i % 2 == 0 and j % 2 == 1:
                        setVal(plateau,i,j,listeCartesAmovibles[k])
                        tresorsPos[listeCartesAmovibles[k]['tresor']] = (i,j)
                        k+=1
      carteAmovible = Carte(bool(carteAmovible2[0]), bool(carteAmovible2[1]),
bool(carteAmovible2[2]), bool(carteAmovible2[3]),0,[])
```

On récupère les cartes amovibles générées aléatoirement par la fonction *creerCartesAmovibles()*, et on les attribue dans l'ordre des positions des cartes restant à être initialisées.

```
def creerCartesAmovibles(tresorDebut,nbTresors):
      nbCartes = 34
      listeCartesAmovibles = []
      carteAmovible = None

      for i in range(nbCartes):
            r = random.randint(1, 12)
            while r == 7 or r == 11: # 7 = 0111b, 11 = 1011b
                  r = random.randint(1, 12)

            murs_s = "{0:04b}".format(r)
            murs = [int(x) for x in murs_s]
            listeCartesAmovibles.append(Carte(bool(murs[0]), bool(murs[1]),
bool(murs[2]), bool(murs[3]), (i + tresorDebut) if (i + tresorDebut) <
nbTresors else 0, []))

      carteAmovible = [int(x) for x in "{0:04b}".format(random.randint(1,
3))]

      def rand(item):
            return random.randint(0, nbCartes - 1)

      return (sorted(listeCartesAmovibles, key = rand), carteAmovible)
```

La fonction *creerCartesAmovibles()* initialise des cartes dont les murs sont choisis de manière aléatoire, met un des trésors restants dessus tant qu'il en reste et génère la carte amovible avec des murs aléatoires. Enfin, l'ordre des cartes amovibles est mélangé afin qu'aucune partie ne se ressemble.

Enfin, on initialise certains paramètres (joueur courant, phase courante).

Les 12 cartes non amovibles reçoivent toutes un trésor.

5 . Algorithme A* (A Star)

Nous avons choisis d'utiliser l'algorithme A* pour la recherche de chemin car il s'agit d'un algorithme robuste pour la recherche de chemin dans les graphes.

Notre implémentation utilise une structure de base appelée Nœud afin de représenter les cases à traiter du labyrinthe.

```
Noeud = {
      'x' : int,
      'y' : int,
      'poids' : int,
      'heuristique' : int
}
```

Les clés « x » et « y » représente la position sur la matrice du nœud, la clé « poids » représente le coût pour atteindre cette case.

A* est une extension de l'algorithme de Dijkstra utilisant une valeur heuristique afin d'orienter ses recherches.

L'heuristique est défini par la formule :

```
h = abs(noeud1.x - noeud2.x) + abs(noeud1.y - noeud2.y)
```

Notre implémentation intègre une pile prioritaire afin de choisir le nœud à traiter à chaque itération de notre algorithme.

La pile est prioritaire car le nœud est inséré dans cette pile en fonction de sa valeur heuristique.

De ce fait, le nœud à traiter en priorité (heuristique la plus faible) est toujours au sommet de la pile.

```
def InsererNoeudDansPile(pile, noeud):
      i = 0
      while i < len(pile) and noeud["heuristique"] < pile[i]
["heuristique"]:
            i += 1

      pile.insert(i, noeud)
```

L'algorithme utilise un ensemble (appelé « visites ») afin de répertorier l'ensemble des positions du plateau déjà visitées ainsi qu'un dictionnaire (appelé « traces ») afin de garder la trace des connections entre les différentes cases du labyrinthe pour la reconstruction du chemin s'il existe.

L'algorithme permet de s'arrêter une fois la destination atteinte car par sa conception il permet d'obtenir un des plus courts chemins (s'il en existe plusieurs).

Dans le cas où la destination n'a pas pu être atteinte et que la pile de nœuds à traiter est vide l'algorithme s'arrête car aucun chemin permettant d'atteindre la destination existe.

Pour chaque nœud on va vérifier ses connections avec les cases voisines en obtenant un vecteur depuis une liste de vecteurs représentant les 4 directions Nord (-1, 0), Est (0, 1), Sud (1, 0) et Ouest (0, -1).

```
def GetDir(i):
      dirs = [(-1, 0), (0, 1), (1, 0), (0, -1)]
      return dirs[i]

...
for i in range(4):
      d = GetDir(i)
...
```

On s'assurera que les coordonnées sont bien valides (pas de coordonnées hors des bornes de la matrice).

Une fois la case voisine voulue sélectionnée on va vérifier que cette position n'a pas déjà été traitée, pour cela on va vérifier que la case n'est pas présente dans l'ensemble des positions déjà visitée et que le poids de celle-ci est supérieur. Dans le cas contraire on doit la traiter à nouveau car un meilleur chemin est disponible.

```
if pos_v in visites and poids_v >= n["poids"]:
      continue
else:
      ...
```

Afin de traiter un nœud (représentant une case du labyrinthe) on doit récupérer la case représentée par ce nœud puis vérifier les conditions de passage d'un case à l'autre.

```
if (d == (-1, 0)):
      if (passageNord(c_courante, c_voisin)):
            passage = True
elif (d == (0, 1)):
      if (passageEst(c_courante, c_voisin)):
            passage = True
...
```

Si

Un passage entre les deux cartes existe, on va ajouter le nœud correspondant à la case voisine dans notre pile de nœud à traiter

```
if passage:
      if  poids_v < n["poids"] or pos_v not in [(k["x"], k["y"]) for k in
pile]:
            traces[pos_v] = pos_courante
            n2 = Noeud(pos_v[0], pos_v[1], poids_v, poids_v +
Heuristique(pos_v, arrivee))
            InsererNoeudDansPile(pile, n2)
```

Si en fin d'algorithme un chemin est disponible, il va falloir reconstruire celui-ci à partir des traces enregistrées.

On va reconstruire le chemin en partant de la position de la destination en remontant les traces jusqu'à la position de départ. La liste contient à cet instant le chemin de la destination jusqu'à l'origine, il faudra inverser cette liste puis la retourner.

```
def ReconstruireChemin(traces, depart, arrivee):
      courant = arrivee
      chemin_ = [courant]

      while courant !=  depart:
            courant = traces[courant]
            chemin_.append(courant)

      return list(reversed(chemin_))
```

V – Méthode de tests

Nous avons procédé de deux façons pour tester notre code source.

Nous avons intégré des tests unitaires à la fin de chaque fichier.

Ensuite des tests en conditions réelles ont été réalisés (lancement du jeu et détection des bugs en vue de corrections)

VI – Difficultés rencontrées

Au vu des choix effectués sur nos structures de données, nous avons été amenés à écrire des modifications dans notre API labyrinthe, en effet les indices ne concordaient pas avec l'utilisation faite des indices dans labyrinthe*.py.

Pour réaliser ces modifications, nous avons dû nous y reprendre à plusieurs fois afin d'obtenir une stabilité dans notre code.

Des difficultés ont été rencontrées notamment sur les paramètres par défaut dans l'environnement Python. En effet ces derniers sont évalués au moment de la création de la fonction dans l'environnement et non à chaque appel de ces fonctions.

Ces arguments pointent donc toujours vers la même variable, ce qui devient problématique dans le cas ou ceux-ci sont des types au fonctionnement particulier comme les listes.

VII – Bugs restants

L’assainissement des entrées par l'utilisateur en mode texte n'était pas implémenté au moment de la soutenance, depuis ceci a été effectué.

VIII – Extensions apportées

L'algorithme de recherche de chemin n'est pas celui de l'inondation (voir IV.5).

IX – Annexes

1 . Journal de bord Ismatou

15 Janvier :

- Implémentation du fichier matrice.py

18 Janvier :

- Écriture des tests unitaires pour le fichier matrice.py en version classique

19 Janvier :

- Implémentation des matrices en objet

20 Janvier :

- Relecture du code écrit par les autres membres du groupe et débogage du code en mode texte.

21 Janvier :

- Débogage du code en mode texte objet.

22 Janvier :

- Rédaction du dossier.

Difficultés:

Au cours de l'écriture du fichier matrice.py, j'ai eu à réaliser un décalage des valeurs de telles sortes que la valeur entrée se répétait tout long de la colonne ou la ligne selon le décalage effectué mais la valeur récupérée était la bonne restait à régler le problème de la répétition de la nouvelle valeur introduite.

2 . Journal de bord Thomas

17 Décembre :

- Définition d'une structure pour *Carte.py* et *Joueur.py* avec le groupe.

18 Décembre :

- Implémentation de la structure *Carte.py* et fonctions associées.

11 Janvier :

- Démarrage du travail sur *labyrinthe.py* (compréhension,fonctions de base)

15 Janvier :

- Initialisation des cartes (trésors, pions) sur *labyrinthe.py*. Premieres tentatives d'affichage de *labyrintheTexte.py*.

18 Janvier :

- Travail sur le labyrinthe impératif :
  - Attribution de la position des joueurs
  - Attribution des trésors
  - Attribution des cartes amovibles
  - Analyse de l'implémentation de l'algorithme du chemin par inondation
  - Test accessible avec *astar.py*
  - Implémentation des inputs textes dans *labyrintheTexte.py*

19 Janvier :

- Correction de bugs sur le labyrinthe classique en texte, ajout de la fonction *finirtour*
- Travail sur labyrinthe graphique et démarrage de l'objet

20 Janvier :

- Multiples corrections sur les fichiers *Carte.py* et *labyrinthe.py*. Passage en version objet : Implémentation de *Carte.py* et *labyrintheOO.py*.
- Corrections sur la version impérative de *labyrinthe.py*. Implémentation de la règle pour ne pas avoir de pions sur la carte amovible en objet.

21 Janvier :

- Résolution de problèmes liés aux indices des trésors. Ajout de commentaires, nettoyage du code.

22 Janvier :

- Amélioration de la gestion des entrées clavier. Rédaction du dossier.

3 . Journal de bord Valérian

18 Décembre :

- Définition avec le groupe des structures et répartition du travail

19 Décembre :

- Implémentation de joueur en version classique

15 Janvier :

- Début programmation *labyrinthe.py*

18 Janvier :

- Initialisation des cartes amovibles en version classique

19 Janvier :

- Implémentation de l'algorithme A* (A Star) pour le pathfinding en version classique

20 Janvier :

- Finalisation de l'algorithme A* en version classique
- Correction d'un bug d'initialisation de variable par défaut dans la fonction *Carte()* quand le paramètre par défaut est une liste
- Implémentation des fonctions *getCoordonneesJoueurCourant()* et *getCoordonneesTresorCourant()*

21 Janvier :

- Implémentation de la règle pour que les pions ne soient pas sur la carte amovible en cas d'éjection d'une carte contant un pion
- Implémentation de l'algorithme A* en objet (classe *AStar*, classe *Noeud*, classe *Pile*)
- Implémentation de Joueur en objet (classe *Joueurs*, classe *Joueur*)
- Modification de l'algorithme A* pour utiliser correctement les nœuds et une pile prioritaire (insertion à la position voulue en fonction de ses attributs)

22 Janvier :

- Dernières corrections de bugs
- Commentaires et nettoyage des fichiers de code source