Projet – Labyrinthe

Rappel : Ce projet est à faire en binôme et à envoyer par email avant le dimanche 10 janvier 2020 minuit.

Présentation

Le but de ce projet est de trouver un chemin le plus cours possible menant d'un point de départ à une des sorties d'un labyrinthe. Pour ce faire nous allons utiliser l'algorithme A^*, qui est un des algorithmes les plus simples de l'intelligence artificielle.

La figure 1 illustre ce que nous souhaitons obtenir.

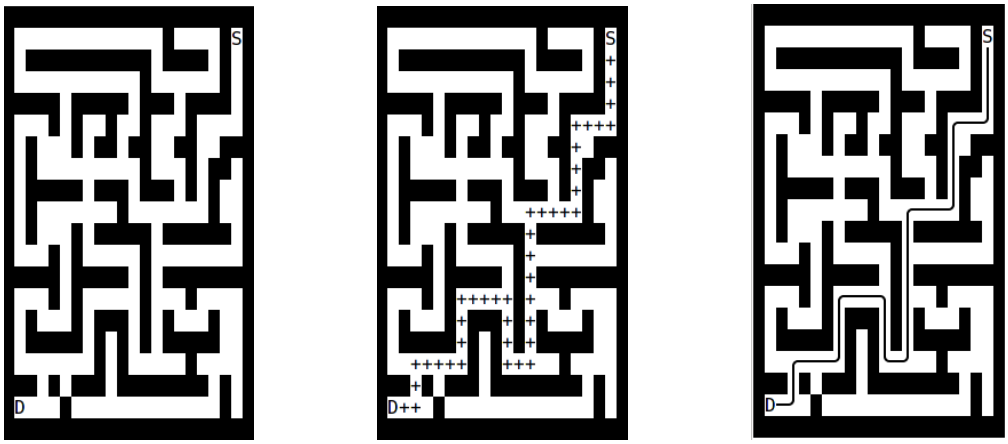


FIGURE 1 – À gauche, nous avons un labyrinthe avec son point de départ `D` et une sortie `S`. Sur les figures de droite nous avons le même labyrinthe mais avec le tracé d'un chemin de plus petite longueur reliant `D` à `S`.

Pour commencer

Pour ce projet vous aurez besoin d'un environnement `Python` fonctionnel ainsi que du module `NumPy`. L'archive `labyrinthe.zip` disponible à l'adresse

`http://www.lmpa.univ-littoral.fr/~fromentin/index.php?page=teaching/l2math`

contient les fichiers suivants :

— `labyrinthe.py`, qui est le squelette de votre projet. Le code `Python` que vous devrez produire sera à placer directement dans ce fichier.

— `lab1.txt`, ... qui sont des fichiers correspondant aux différents labyrinthes que vous rencontrerez dans ce sujet.

Ce qu'il faut rendre

Avant la date butoir vous m'enverrez un email à l'adresse :

`jean.fromentin@univ-littoral.fr`

Votre message devra contenir les *noms, prénoms et adresse email des deux membres du binôme*. Vous ajouterez le fichier `labyrinthe.py` dûment complété en pièce jointe de votre message. Si j'ai bien reçu votre message, j'en accuserai réception dans les heures qui suivent.

Organisation du projet

Ce projet est décomposé en quatre parties. La première partie est consacrée aux tas, la seconde a pour but de charger un labyrinthe et ainsi définir le monde dans lequel va travailler l'algorithme A^*. La résolution du labyrinthe sera l'objet de la troisième partie. Dans la dernière partie nous allons enrichir notre labyrinthe avec des pièges, ...

1 Tas

Cette partie a pour but d'implémenter en `Python` une structure de tas. Comme nous l'avons vu en cours, un tas est un arbre binaire presque complet pouvant être facilement implémenter à l'aide d'un tableau. Puisque, en `Python`, les tableaux ne sont pas des objets de bases, nous allons avoir recours au module `NumPy`.

Le fichier `labyrinthe.py` contient une structure `Tableau`, codant des tableaux d'entiers, qui sera utilisée tout au long de ce projet. Voici les différentes commandes associées à cette structure :

- `T=Tableau(capacite)` qui crée un tableau vide `T` de taille maximal `capacite` ;
- `len(T)` qui retourne la taille du tableau `T` ;
- `T.remplis(v)` qui remplit le tableau `T` au maximum de sa capacité avec l'entier `v`, la taille de `T` est alors égal à sa capacité ;
- `T[i]` qui permet d'accéder en lecture ou écriture à l'élément d'indice `i` du tableau `T` ;
- `T.echange(i,j)` qui échange les éléments d'indice `i` et `j` du tableau `T` ;
- `T.ajoute_fin(v)` qui ajoute, si possible, l'élément `v` à la fin du tableau `T`. La taille de `T` est alors augmentée de 1. Dans le cas où la taille du tableau est égale à sa capacité, l'ajout est impossible et un message d'erreur est retourné ;
- `T.supprime_fin()` qui supprime l'élément se trouvant à la fin du tableau. La taille de `T` est alors diminuée de 1. Si le tableau est vide, la suppression est alors impossible et un message d'erreur est retourné.

Afin de s'assurer que tout va bien et notamment que le module `NumPy` est bien installé, éxecuter le script `labyrinthe.py`. Les commandes se trouvant entre les lignes `87` et `101` devraient alors produire différents affichages que vous devez comparer à ceux contenus dans le fichier (en commentaire après les différents appels à `print`).

Important : *Si vous n'arrivez pas à passer cette étape de test, merci de contacter Guillaume Pagel via discord ou à l'adresse mail* `guillaume.pagel@univ-littoral.fr` *pour qu'il puisse vous aider.*

Exercice 1. Complétez la fonction `creation_tas_cours` afin qu'elle retourne un tableau de capacité 20 et initialisé à `[23, 15, 7, 12, 5, 6, 1, 4, 8, 2]`. Ce tas sera désigné par `tas_cours` dans le reste de cette partie.

La notion de tas nécessite d'avoir une fonction poids définie sur chacun des nœuds du tas. Comme nous allons utiliser différentes fonctions de poids, nous allons passer cette fonction en paramètre lorsque nous en avons besoin. La fonction `poids_max` présente dans le fichier est la plus simple que l'on puisse imaginer. Elle retourne simplement la valeur du sommet.

Exercice 2. Complétez la fonction `tas_ajoute(T,v,poids)` qui ajoute la valeur `v` au tas `T` muni de la fonction poids `poids`. Pour cela vous aurez besoin de calculer l'indice du père du nœud d'indice i.

Exemple : La commande `tas_ajoute(tas_cours,21,poids_max)` ajoute 21 au tas de l'exercice 1 et le transorme en `[23, 21, 7, 12, 15, 6, 1, 4, 8, 2, 5]` (décommenter la commande correspondante dans le fichier squelette).

Exercice 3. Complétez les fonctions `tas_est_feuille(T,i)` et `tas_possede_fils_droit(T,i)` qui testent si l'élément d'indice `i` du tas `T` est une feuille ou possède un fils droit respectivement.

Exercice 4. Complétez la fonction `tas_supprime(T,poids)` qui supprime et retourne la valeur du nœud de poids maximal du tas `T` muni de la fonction poids `poids`.

Exemple : La suppression de l'élément maximal du tas de l'exercice 1 retourne 23 et transforme le tas en `[15, 12, 7, 8, 5, 6, 1, 4, 2]`.

Exercice 5. Déterminez la complexité des algorithmes `tas_ajoute(T,v)` et `tas_supprime(T)` en fonction de la taille du tas `T`.

2 Labyrinthe

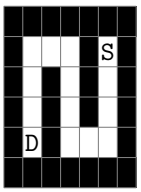
Les labyrinthes que nous considérerons dans ce projet seront toujours dessinés sur une grille rectangulaire et seront naturellement représentés en `Python` par des entiers. Nous utiliserons la convention suivante :

— 0 désigne un mur,

— 1 désigne une case vide,

— 2 désigne la case de départ,

— 3 désigne une sortie.



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

FIGURE 2 – Exemple de codage d'un labyrinthe à l'aide d'une grille d'entiers.

De plus nous supposerons toujours que les labyrinthes considérés sont encadré par un mur. Bien qu'une grille d'entiers puisse naturellement être représentée par un tableau de tableau, d'un point de vue algorithmique il est beaucoup plus avantageux de la représenter par un simple tableau. Une grille de largeur ℓ et de hauteur h sera alors donnée par un tableau de taille $\ell \times h$.

Pour aller plus loin nous devons préciser comment sont indexés les cases de la grille. Les indices commencent à 0. La case d'indices $(0, 0)$ est la case en bas à gauche tandis que celle en haut à doite est celle d'indices $(\ell - 1, h - 1)$. Ceci est compatible avec les coordonnées utilisées en géométrie.

Concernant le tableau repésentant la grille, les indices iront de 0 à $\ell \times h - 1$. Le tableau est obtenu en disposant les valeurs de la ligne en bas de la grille, puis celle d'après, etc. Par exemple la grille [1]

4	0	1	2
2	5	7	0
1	3	4	6

est donnée par le tableau `[1, 3, 4, 6, 2, 5, 7, 0, 4, 0, 1, 2]`.

Exercice 6. Complétez la fonction `indice(x,y,largeur,hauteur)` qui étant donnée une grille de largeur `largeur` et hauteur `hauteur` retourne l'indice dans le tableau de la case de coordonnée `(x,y)`. Dans l'exemple précédent, le chiffre `7` est aux coordonnées `(2,1)`, ce qui correspond à la case d'indice 6 dans le tableau. L'appel `indice(2,1,4,3)` retournera donc 6.

La fonction `charger_labyrinthe(nom_fichier)` charge le labyrinthe contenu dans le fichier `nom_fichier` et retourne un triplet `(donnees, largeur, hauteur)` où `donnees` est le tableau codant la grille du labyrinthe, `largeur` sa largeur et `hauteur` sa hauteur. Dans la suite un labyrinthe sera toujours donné par un tel triplet.

Comme vous pourrez le constater, le fichier d'un labyrinthe est structuré de la façon suivante :

— la première ligne informe sur la largeur du labyrinthe,

— la seconde ligne informe sur la hauteur du labyrinthe,

— les autre lignes codent la grille du labyrinthe avec les conventions :

  `#` pour un mur,

  `D` pour la case départ,

  `S` pour une sortie,

  ␣ un espace pour les cases vides.

L'archive que vous avez téléchargée contient différents labyrinthes :

— `lab1.txt` correspond au labyrinthe de la figure 1,

— `lab2.txt` correspond au labyrinthe de la figure 2,

— `lab3.txt` est un labyrinthe avec plusieurs sorties.

— `lab4.txt` et `lab5.txt` sont de grands labyrinthes de test.

1. Ce n'est pas une grille de labyrinthe, mais c'est pour que l'exemple soit compréhensible.

Exercice 7. Complétez la fonction `afficher_labyrinthe(L)` qui affiche le labyrinthe donné par le triplet `L`. On pourra reprendre dans un premier temps les mêmes caractères que ceux utilisés dans le fichier pour symboliser les murs, couloirs, points de départ et de sorties.

Exercice 8. Complétez la fonction `depart_labyrinthe(L)` qui retourne les coordonnées de la case de départ du labyrinthe donné par le triplet `L`.

Exercice 9. Complétez la fonction `sorties_labyrinthe(L)` qui retourne les coordonnées des cases de sorties du labyrinthe donné par le triplet `L`.

3 Résolution

Nous allons maitenant exploiter l'algorithme A^* afin de résoudre nos labyrinthes. Vous trouverez des détails concernant cet algorithme sur la page Wikipédia :

`https://fr.wikipedia.org/wiki/Algorithme_A*`

Commençons par introduire la notion de *position pondérée* qui est un quadruplet `(x,y,c,h)` où

- `(x,y)` sont les coordonnées d'une case de la grille du labyrinthe,
- `c` est le coût (en nombre de pas) pour arriver à cette position à partir de la case de départ,
- `h` est le coût heuristique correspondant au coût estimé (en nombre de pas) pour atteindre une des sorties du labyrinthe.

La résolution du labyrinthe se fera de la façon suivante :

1. Créer un tas de positions pondérés.
2. Ajouter la position de départ au tas.
3. Tant qu'une des sorties n'a pas été atteinte, retirer la position de coût minimal du tas puis ajouter au tas les positions voisines non déjà visitées.

Les cases voisines d'une case de la grille sont celles se trouvant au dessus, en dessous, à gauche et à droite de la case considérée.

Pour le point 1 nous allons utiliser les algorithmes de tas introduis à la partie 1. Nous devons donc convertir une position pondérée en entier.

Exercice 10. On pose $N = 2^{16}$ et $P = \{0, \dots, N-1\}^4$. On suppose de plus que toute position pondéré est un élément de P.

Justifiez que l'application

$$\begin{array}{rccc} \varphi : & P & \to & \{0, \dots, 2^{64}-1\} \\ & (x,y,c,h) & \mapsto & x + y \cdot N + c \cdot N^2 + h \cdot N^3 \end{array}$$

est une bijection.

Exercice 11. Complétez la fonction `chiffre_position_ponderee(x,y,c,h)` pour qu'elle retourne $\varphi(\mathtt{x}, \mathtt{y}, \mathtt{c}, \mathtt{h})$. Vous pourrez vérifier votre fonction avec $\varphi(1,2,3,4) = 1125912791875585$.

Exercice 12. Complétez la fonction `dechiffre_position_ponderee(n)` qui étant donné $n \in \{0, \dots, 2^{64}-1\}$ retourne le quadruplet $\varphi^{-1}(n)$.

Exercice 13. Le poids d'une position pondérée (x, y, c, h) est $-(c+h)$. Pourquoi avoir ajouter le signe $-$?

Complétez la fonction `poids_position_ponderee(n)` qui étant donné $n \in \{0, \dots, 2^{64}-1\}$ retourne le poids de la position pondérée $\varphi^{-1}(n)$

La structure de tas définie à la première partie munis de la fonction de poids définie à l'exercice précédent nous permet de définir des files de priorités de positions pondérées.

Concentrons nous maintenant sur le coût heuristique. La distance de Manathan entre $(x_1, y_1) \in \mathbb{Z}^2$ et $(x_2, y_2) \in \mathbb{Z}^2$ est l'entier $|x_1 - x_2| + |y_1 - y_2|$.

Exercice 14. Complétez la fonction `distance(x1,y1,x2,y2)` pour qu'elle retourne la distance de Manathan entre les couples `(x1,y1)` et `(x2,y2)`.

Le cout heuristique d'une position (x, y) de la grille est alors la plus petite distance Manathan entre (x, y) est les différentes sorties.

Exercice 15. Complétez la fonction `cout_heuristique(x,y,sorties)` qui étant donnés des entiers `x` et `y` et la liste `S` des coordonnées des sorties retourne le coût heuristique de `(x,y)`.

Pour l'algorithme A^* nous avons besoin de savoir si une case de la grille a déjà été visitée ou non. De plus nous avons besoin de savoir quelle case à permis de visiter cette case. Pour cela nous allons utiliser une autre grille d'entiers `visite`. Une case de cette nouvelle grille contiendra la valeur -1 si elle n'a pas été visitée et la valeur de $\varphi(x, y, c, h)$ si elle a été visitée à partir de la case pondérée (x, y, c, h). La grille `visite` aura la même forme que celle du labyrinthe et sera aussi donnée par un unique tableau.

Exercice 16. Complétez la fonction `positions_admissibles(x,y,L,visite)` qui retourne la liste des positions admissibles (non mur, non déjà visitées) accessibles à partir de la position (x, y) dans le labyrinthe donné par le triplet `L`. Le tableau `visite` est la grille des sommets déjà visités. Il pourra être utiles d'avoir recours à une ou des fonctions annexes bien choisies.

Exercice 17. Complétez la fonction `resoud(L)` resolvant le labyrinthe donné par le triplet `L`. La fonction retournera la liste des coordonnées du trajet solution. Le tas codant la file à priorité de l'algorithme A^* à une taille d'au plus le nombre de cases de `L`. Vous pouvez faire autant de fonctions annexes qu'il vous semble nécessaire.

Exercice 18. Complétez la fonction `afficher_labyrinthe_solution(L, solution)` qui permettra d'afficher dans le labyrinthe la solution donnée avec des caractères `+`.

4 (Bonus) Améliorations possibles

Si vous lisez cette section, c'est que votre programme est fonctionnel. Avant d'aller plus loin, nous demandons à ce que cette partie soit traité sur une copie séparée de votre programme. Ceci permettra deux choses :

— La version originale non améliorée pourra être corrigée et lue plus facilement ;

— Dans le cas où, suite à l'ajout d'une amélioration, votre programme ne fonctionne plus, vous aurez toujours une version fonctionnelle en plus de votre tentative d'amélioration.

Cette quatrième partie est facultative, ces suggestions sont là si vous souhaitez améliorer votre programme et essayer de nouvelles choses. Elles ne peuvent influer sur la note du projet que positivement.

4.1 Esthétique

Il serait intéressant d'améliorer un peu le rendu dans la console pour que ce soit plus joli à regarder mais aussi plus compréhensible pour de très grand labyrinthe (dont le problème peut être réglé avec l'amélioration 4.3). Pour cela, on peut utiliser les caractères suivants :

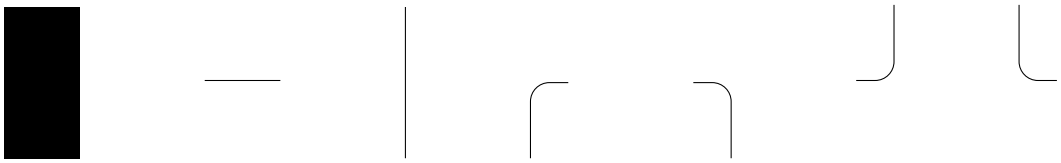


FIGURE 3 – Les caractères pouvant être utilisé pour les murs et le trajet de la solution

De gauche à droite, les codes *Unicode* (ainsi que la séquence permettant de les afficher dans une chaîne) de ces caractères sont :

1. `U+2588` (affiché avec `"\u2588"`), il servira pour les murs ;
2. `U+2500` (affiché avec `"\u2500"`), il servira quand le trajet traverse la case horizontalement ;
3. `U+2502` (affiché avec `"\u2502"`), il servira quand le trajet traverse la case verticalement ;
4. `U+256D` (affiché avec `"\u256D"`), il servira quand le trajet vient d'en bas et tourne à droite ;
5. `U+256E` (affiché avec `"\u256E"`), il servira quand le trajet vient d'en bas et tourne à gauche ;
6. `U+256F` (affiché avec `"\u256F"`), il servira quand le trajet vient d'en haut et tourne sur sa gauche ;
7. `U+2570` (affiché avec `"\u2570"`), il servira quand le trajet vient d'en haut et tourne sur sa droite.

Essayez ces commandes dans la console, la séquence `\uxxxx` à l'intérieur d'une chaîne de caractère devrait être remplacé par le caractère Unicode correspondant. Cette séquence est compatible avec la commande `print`.

4.2 Des cases spéciales

On peut imaginer que notre labyrinthe comporte des *pièges* très pénalisant, des *passages sombres* où la progression est difficile, ou bien des couloirs considérés comme *passage secret* ou *raccourci*. Ces nouvelles cases pourraient affecter le trajet de la solution en modifiant le coût (le nombre de pas) nécessaire pour les traverser. Les pièges pourraient par exemple valoir 100 pas, un passage sombre 2 pas et un passage secret en revanche ne coûterait rien (0 pas).

Pour ajouter ces nouvelles cases, il faudra modifier le chargement du fichier contenant le labyrinthe, car il contiendra désormais plus de symboles ; mais aussi l'affichage et ajouter une gestion du coût de déplacement.

4.3 Sauvegarder la solution

Pour le moment la solution est imprimée dans la console, mais une fois `Pyzo` fermé, notre solution est perdue et doit être recalculée. Il serait pratique de pouvoir sauvegarder la solution dans un fichier, qu'il soit séparé du fichier labyrinthe ou qu'il le complète en écrivant la solution sous le labyrinthe vierge.

Conseil : Pour éviter d'écraser sans le vouloir les différents labyrinthes fournis avec le projets, pensez à sauvegarder une copie dans un autre dossier.

En plus de sauvegarder la solution, cela permet de ne plus être limité par la taille de la console, on peut dorénavant traiter des labyrinthes très larges !

4.4 Générer un labyrinthe

Nous avons un programme permettant de résoudre des labyrinthes, mais à quoi sert-il si nous n'avons pas de labyrinthe ? On peut parfaitement en générer aléatoirement grâce au module `random`. L'utilisation de ce module est donnée ici :

`http://python-simple.com/python-modules-math/random.php`

La génération aléatoire permettra de placer les murs, il faudra tout de même faire attention à ce qu'une des sorties soit accessible depuis le point de départ.

4.5 Autres améliorations

Si il vous vient d'autres idées d'amélioration qui vous semble intéressante à nous montrer, n'hésitez pas à les mettre en œuvre et à nous les présenter !