INFOF203 - Algorithmique 2
Rapport

Rodrigue Van Brande

14 décembre 2014

Table des matières

1 Introduction

On nous demandait de produire un algorithme efficace pour résoudre le problème du labyrinthe de *M. Pakkuman*. L'algorithme devait produire un fichier représentant l'état initiale du labyrinthe et un autre fichier pour l'état finale. La difficulté étant qu'il existe de petits *monstres* sur le chemin de la sortie que *M. Pakkuman* qui lui empêche de continuer son chemin sans lui donner un *bonbon* en échange. Ainsi la notion de **demi-tour** est imposée et a été la plus grande difficulté du projet.

2 Étapes

Les étapes de l'algorithme sont simples :

1. On récupère le fichier d'entrée contenant les détails du labyrinthe (voir énoncé).
2. On décrypte ce fichier en une matrice.
3. On forme un arbre binaire.
4. On élimine les voies sans issue.
5. On remonte l'arbre de la fin jusqu'au début en calculant le nombre de bonbons besoins.
6. Le nombre de bonbons devient alors le nombre de demi-tour permis.
7. On recherche le chemin le plus court mais le calcul devient moins long car les impasses sont retirés et les demis-tours sont limités.

3 Chargement

La classe *FileLoader* permettait de charger le fichier de la situation initiale. Comme précisé dans l'énoncé, on suppose que le fichier d'entrée ne comporte aucune erreur. Ainsi un simple try/except général est implémenté.

Voici le labyrinthe de l'énoncé :

```
+---+---+---+---+---+---+---+---+ est traduit en 00000000000000000000000000000000
|   |                           |                 01110111111111111111111111111110
+   +---+   +---+---+---+   +   +                 01110000011100000000000011101110
|       |   |           |   |   |                 01111111011101111111111011101110
+---+   +   +   +---+   +   +   +                 00000111011101110000011011101110
|   |   |   |       |   |   |   |                 01110111011101111111011011101110
+   +   +   +---+   +   +   +   +                 01110111011100000111011011101110
|       |       |   |   |   |   |                 01111111011111110111011011101110
+   +---+---+---+   +   +   +---+                 01110000000000000111011011100000
|   |           |   |   |       |                 01110111111111110111011011111110
+   +   +---+   +   +   +---+   +                 01110111000001110111011000001110
|       |       |   |       |   |                 01111111011111110111011111101110
+---+---+   +---+   +---+   +   +                 00000000011100000111000001101110
|       |   |       |   |       |                 01111111011101111111011011111110
+   +---+   +   +---+   +---+   +                 01110000011101110000011000001110
|               |               |                 01111111111111110111111111111110
+---+---+---+---+---+---+---+   +                 00000000000000000000000000001110
```

FIGURE 1 – Le labyrinthe traduit en vrai et faux.

On remarque que ce labyrinthe, en terme de texte brute, est de la taille 33x17 [1]. Seulement nous voulons former une matrice 8x8 avec, à chaque case, une classe qui contiendrait si les directions du haut, du bas, de droite ou de gauche sont ouvertes ou non.

Dans une première étape, nous formons donc cette matrice 17x33 [2] et on reprend les éléments pertinents se trouvant aux bonnes position.

1. Remarquons que dans l'exemple de résolution de l'énoncé, la largeur va de haut en bas et la hauteur de gauche à droite. Nous gardons ce choix dans notre algorithme.
2. La formule utilisé est $WIDTH = WIDTH_{REAL} * 2 + 1$ pour la largeur et $HEIGHT = HEIGHT_{REAL} * 4 + 1$ pour la hauteur. Pour exemple avec la matrice 8x8, $17 = 8 * 2 + 1$ et $33 = 8 * 4 + 1$.

On résout ce petit problème mathématique en respectant les propriétés suivantes [3] :

1,2 On prend soin de retenir les x et y de notre nouvelle matrice.

3 On ne prends que les lignes impaires.

4 On ne prends que 1 colonnes toutes les 4, en partant de la 3ème.

5,6,7,8,9,10 On définit la nouvelle case en indiquant les directions ouvertes ou non.

```
int x_real = 0;
int y_real = 0;
for( int x = 1; x < matrice.length-1; x+=2 ) {
  for( int y = 2; y < matrice[0].length-2; y+=4 ) {
    Case newCase = new Case();
    newCase.top   = matrice[x][y+2];
    newCase.right = matrice[x+1][y];
    newCase.down  = matrice[x][y-2];
    newCase.left  = matrice[x-1][y];
    _labyrinthe.set( x_real, y_real, newCase);
    y_real += 1;
  }
  x_real += 1;
  y_real = 0;
}
```

FIGURE 2 – Code source pour réduire la matrice.

De façon plus visuelle, la position des cases, la position des directions et les cases inutiles :

FIGURE 3 – Représentation visuelle de l'opération.

Ainsi est enregistré le labyrinthe. Les autres données indiquant la position de *M. Pakkuman*, des *monstres* et des *bonbons* sont très facilement récupérable, le développement n'est donc pas utile à être expliqué ici.

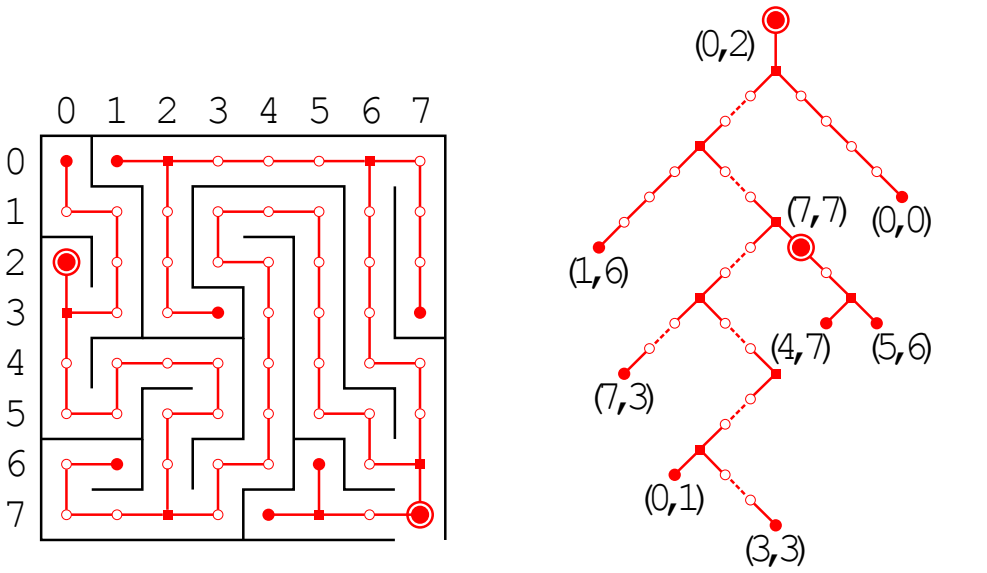
4 Arbre binaire

Le labyrinthe est une simple matrice où chaque élément est une *Case*. Mais on peut aussi le décomposer en un arbre binaire car le labyrinthe imposé dans le projet est dit "parfait" [4].

Cette arbre est donc construit par la classe *Path* sous forme de plusieurs *Tree* comme vu au cours.

3. Les lignes et les colonnes sont nommés par rapport à la vue du labyrinthe sur le rapport, on n'inverse pas les axes dans l'explication.

4. Cette notion a été reprise de *fr.wikipedia*.



(a) Un labyrinthe parfait contenant son arbre. (b) Représentation en arbre du labyrinthe.

FIGURE 4 – Le labyrinthe et son arbre associé.

5 Recherche du chemin le plus court

Il n'y a pas d'alternative, si on veut trouver le chemin le plus court ; il faut essayer toutes les solutions. Mais il existe une alternative intéressante qui permet d'éviter un grand nombre de calculs inutile. Reprenons l'arbre binaire crée précédemment en prenant soin de garder en mémoire le *Tree* contenant la sortie ainsi que les *Tree* finaux représenté par des ronds rouges pleins.

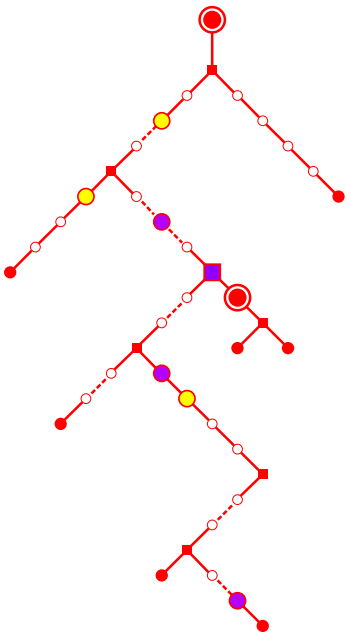


FIGURE 5 – Arbre binaire avec les bonbons et les monstres.

Parcourons chaque *Tree* final et remontons l'arbre jusqu'à un *Tree* possédant soit la sortie, soit un bonbon, soit un autre fils et en supprimant les autres *Tree* inutiles c-à-d ceux qui ne contiendrait rien ou un monstre.

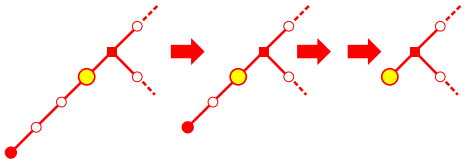


FIGURE 6 – Simplification de l'arbre binaire étape par étape.

Ainsi l'arbre binaire se simplifie et devient nettement plus clair.

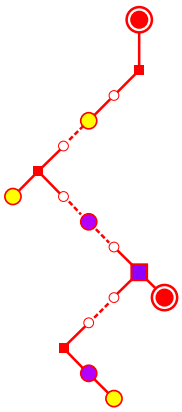


FIGURE 7 – Arbre binaire simplifié.

Il nous reste plus qu'à remonter du *Tree* de sortie jusqu'à la racine (l'entrée) et de compter le nombre de bonbon on a besoin. Ici on croise 2 monstres et 1 bonbon, il nous manque donc 1 bonbon pour arriver à la sortie.

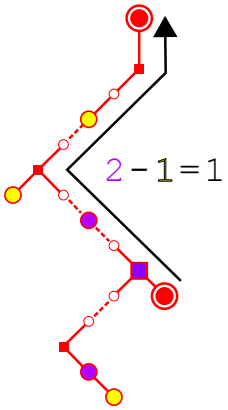


FIGURE 8 – Recherche du nombre de bonbon qu'on a besoin.

Ensuite nous pouvons faire le parcours récursif depuis la racine en autorisant autant de demi tour que de bonbon manquant. Ici, un bonbon est manquant donc on a droit qu'à un seul demi tour lorsqu'on arrive à un *Tree* final. On remarque que le parcours récursif est beaucoup plus rapide que si nous avions pas fait les simplifications.