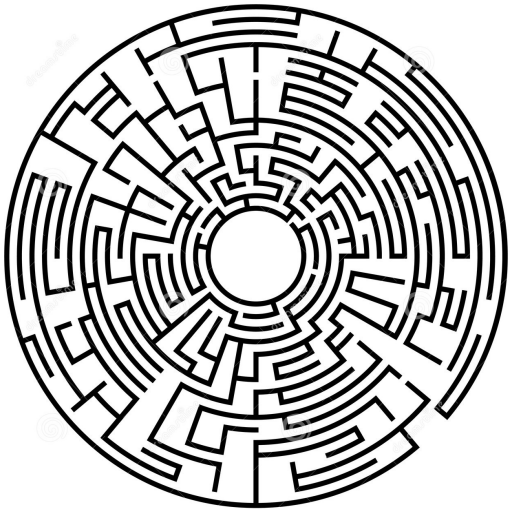
Institut Universitaire de Technologie
Aix-Marseille Université
Département Informatique, Aix

413 avenue Gaston Berger, 13625
Aix-en-Provence cedex 1
Tél. 04 42 93 90 43
Télécopie 04 42 93 90 74
www.iut.univ-amu.fr
sylvie.seropian@univ-amu.fr

Module 3202C / Modélisation de labyrinthe



Andrei Serguienko & Maxime Souchet

Groupe A - 2e année

Responsable
Morgan Morancey & Patricia Gaitan

6 septembre 2017

Table des matières

1 Introduction **2**
1.1 Choix du sujet . . . 2
1.2 Différent type de labyrinthe . . . 2
1.2.1 Labyrinthe parfait . . . 2
1.2.2 Labyrinthe imparfait . . . 3

2 Géneration du labyrinthe **4**
2.1 Notre algorithme de génération de labyrinthe parfait . . . 4
2.2 Pourquoi cet alorgithme produit toujours un labyrithe parfait ? . . . 7

3 Résolution du labyrinthe **8**
3.1 Algorithme de résolution dans un labyrinthe parfait . . . 8
3.1.1 Démonstration . . . 8
3.2 Backtracking . . . 9
3.2.1 Fonctionnement de l'alogirthme . . . 9
3.2.2 Démonstration . . . 9
3.2.3 Pseudo-code . . . 9
3.3 Algorithme de *Pledge* . . . 10
3.3.1 Introduction . . . 10
3.3.2 Explication . . . 10

4 Problèmes recontrés **13**

5 Remerciement **13**

6 Conclusion **13**

1 Introduction

1.1 Choix du sujet

Notre choix s'est porté sur les labyrinthes, car nous avions déjà abordé le sujet lors du projet de `C++` l'an dernier. En effet pendant nos recherches nous avons été fascinés par cet espace bi-dimensionnel plein de surprises. Notre projet de modélisation était donc tout trouvé ! Nous allons donc vous présenter dans ce dossier la modélisation des Labyrinthes et leurs aspects mathématiques.

1.2 Différent type de labyrinthe

Les mathématiciens ont défini les labyrinthes grâce à des concepts, il s'agit d'une structure connexe d'un seul tenant pouvant avoir 2 formes différentes : parfait ou imparfait.

1.2.1 Labyrinthe parfait

Dans un labyrinthe parfait chaque cellule est reliée à toutes les autres et cela de manière unique. Cela veut dire qu'on peut représenter un labyrinthe de taille *m*n* par un rectangle de largeur 1 et de longueur *(m*n)*. Donc cela forme une surface connexe, c'est-à-dire qu'à partir de n'importe quelle cellule du labyrinthe, on peut accéder à toutes les cellules du labyrinthe.

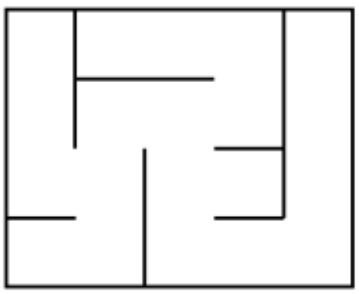


FIGURE 1 – Labyrinthe parfait

1.2.2 Labyrinthe imparfait

Les labyrinthes imparfaits sont tous les labyrinthes qui ne sont pas parfaits (ils peuvent donc contenir des boucles, des îlots ou des cellules inaccessibles).

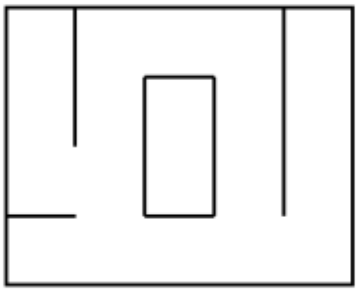


Figure 2 – Labyrinthe imparfait

2 Géneration du labyrinthe

2.1 Notre algorithme de génération de labyrinthe parfait

Exploration exhaustive
Nous nous sommes penchés uniquement sur la création de labyrinthe parfait. L'algorithme que nous avons implémenté se base sur l'exploration exhaustive :

L'alogirthme se déroule de manière suivante :

— On génère un rectangle rempli de cellules où tous les murs sont fermés. Chaque cellule contient une variable booléenne qui indique si la cellule a déjà été visitée ou non.

— Au départ, toutes les cellules sont marquées comme non visitées (faux).

— Nous avons fixé la cellule de départ *(l'entrée du labyrinthe)* en haut à gauche et la cellule d'arrivée *(sortie du labyrinthe)* en bas à droite.

— On choisit arbitrairement une cellule, on stocke la position en cours et on la marque comme visitée (vrai).

— Puis on regarde quelles sont les cellules voisines possibles et non visitées.

— S'il y a au moins une possibilité, on en choisit une au hasard, on ouvre le mur et on recommence avec la nouvelle cellule.

— S'il n'y en pas, on revient à la case précédente et on recommence.

— Lorsque l'on est revenu à la case de départ et qu'il n'y a plus de possibilités, le labyrinthe est terminé. On sait au total qu'il y aura *m*n-1* murs à détruire.

— L'historique des emplacements des cellules précédentes est géré de façon à ce qu'elles soient stockées uniquement si les cellules voisines non visitées sont supérieures à 2. On stocke ainsi seulement les cellules nécessaires afin qu'on puisse revenir sur cette cellule pour détruire le mur manquant.

Voici le pseudo code de l'algorithme :

```
1  //On initialise différents paramètres pour l'algorithme
2  CreerLeLabyrinthe(); //Initialise un tableau de cellule (par défaut à false)
3  InitialiserHistorique(); //Initialise un tableau de position
4  InitialiserPosition(); //Initialise la PositionActuelle
5
6  Cellule(PositionActuelle).etat = true; //Met à true la case actuelle
7  Tant que (LabyrintheComplet() == false)
8  Faire
9          InitialiserChoixSuivant(); //Initialise un tableau de position
10                                    des cellules non visitées
11         Si (ChoixSuivant.taille() >= 2) //Si on a 2 cellules non visitées
12         Faire
13                 Historique.Ajouter(PositionActuelle) //Ajoute au tableau la position
14                                                      de la cellule
15         Tant que (ChoixSuivant.taille() == 0) //Si aucune cellule est accessible
16         Faire
17                 PositionActuelle = Historique[Historique.taille() - 1]; //Retourne à la dernière
18                                                                         cellule de l'historique.
19                 InitialiserChoixSuivant(); //Re initialise un tableau de position
20                                    des cellules non visitées
21         FinFaire
22         PositionActuelle = ChoixSuivant[Random(ChoixSuivant.taille()-1]; //Se déplace à la
23                                                                          nouvelle position
24         CasserLeMur(); //Casse le mur entre la nouvelle position et l'ancienne
25         Cellule(PositionActuelle).etat = true; //Met à true la case actuelle
26 FinFaire
```

Nous avons fait plusieurs variantes de cet algorithme afin d'essayer de créer des labyrinthes plus ou moins différents. Les variantes consistent seulement à la modification de la ligne *13* de notre pseudo-code. Dans ce cas nous piochons la dernière case possible de l'historique. Cependant nous pouvons appliquer 3 autres variantes :

— *random :* On tire aléatoirement la case dans le tableau de l'historique. Ce qui rajoute un facteur aléatoire de plus.

— *last or random :* Cette fois on tire aléatoirement pour décider si on choisit de revenir à la dernière case du tableau ou une tirée aléatoirement dans le tableau. Dans ce cas le labyrinthe aura un facteur "rivière" élevé mais une solution souvent courte et directe.

— *last n :* On tire au hasard parmi les n-dernières cellules ajoutées, dans ce cas le labyrinthe aura un facteur "rivière" bas et une solution longue et sinueuse.

En bleu le parcours entier de l'algorithme
*En rouge le **(seul)** chemin le plus court*

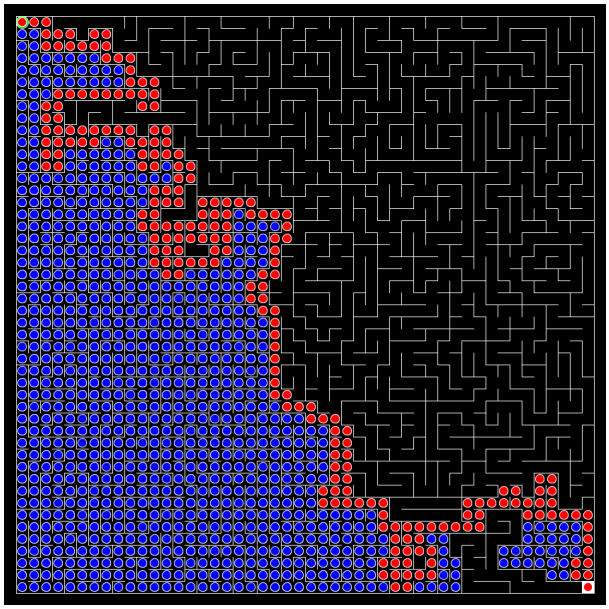


FIGURE 3 – Chemin de last or random

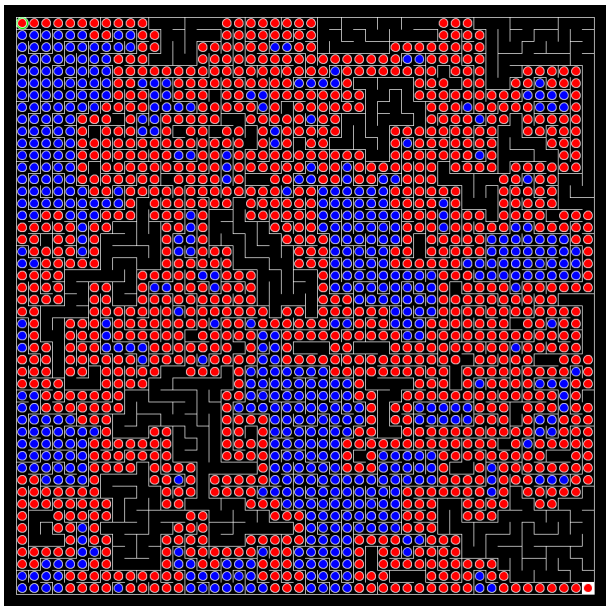


FIGURE 4 – Chemin de last N

2.2 Pourquoi cet alorgithme produit toujours un labyrithe parfait ?

Cet algorithme produit toujours un labyrinthe parfait pour deux raisons :

— Toute les cellules sont connectées entre elles car c'est le principe même de l'agorithme de l'exploration exhaustive, parcourir tout le labyrinthe tant que toutes les cases n'ont pas été visitées.

— Cet algorithme ne produira jamais de boucles ou d'îlots car on vérifie systématiquement de ne pas passer deux fois sur la même cellule.

3 Résolution du labyrinthe

3.1 Algorithme de résolution dans un labyrinthe parfait

Nous avons optés pour un algorithme, simple d'utilisation et réalisable par un humain. L'algorithme consiste seulement dès l'entrée du labyrinthe de longer le mur de droite, c'est-à-dire mettre la main sur le mur de droite et ne jamais le lâcher.

3.1.1 Démonstration

Par définition du labyrinthe parfait
Comme vu précédemment, dans un labyrinthe parfait toutes les cellules sont connexes, reliées avec un seul chemin possible. Donc en longeant le mur choisit, définit au départ (droite ou gauche) mais qu'on gardera jusqu'à la sortie, on accédera à toutes les cellules du labyrinthe et donc forcément par la cellule qui mène à la sortie.

Par création du labyrinthe parfait
Comme vu précédemment, notre algorithme gènere un labyrinthe parfait donc une succession de création de murs qui sont forcément reliés entre eux. On peut voir sur le schéma ci-contre, que lorsque le labyrinthe est vide, c'est à dire sans murs à l'intérieur. On observe que le mur de droite de celui de l'entrée et celui de la sortie sont les mêmes, il est donc facile de sortir d'un tel labyrinthe avec notre résolution.
Regardons le cas si on ajoute un mur *Figure 10*, par définition du labyrinthe parfait il n'y a ni boucle ni îlot dans le labyrinthe, donc aucun mur ne peut se trouver isolé et donc, comme on voit sur le schéma, le chemin sera juste plus long pour sortir mais le mur est forcément raccordé au mur de l'extérieur et donc on tombera toujours sur la sortie.

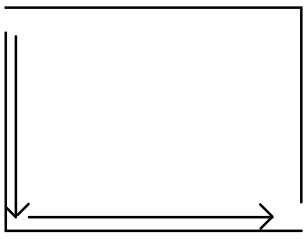


Figure 5 – Labyrinthe vide

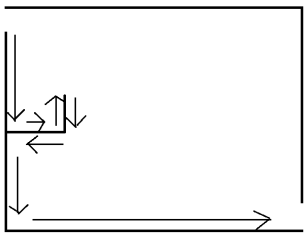


Figure 6 – Labyrinthe avec un mur

3.2 Backtracking

Dans notre algorithme de résolution nous avons implémenté un algorithme de "Backtracking", qui repére quel est *le (unique)* chemin le plus court.

3.2.1 Fonctionnement de l'alogirthme

Pendant toute la résolution du labyrinthe on enregistre un tableau de positions par toutes les cases où l'algorithme est passé. Donc à la sortie du labyrinthe on possède un ensemble $\mathbb{E}$ de toutes ces positions.

$$\mathbb{E} = \{\text{pos1, pos2, pos3, ...}\}$$

Une fois cet ensemble récupéré, l'algorithme vas parcourir chaque élément un par un afin de répérer les doublons, et supprimer toute l'intervalle entre ces deux positions.

$$\mathbb{E} = \{\text{pos1, pos2, pos3, pos1, pos4, ...}\}$$
$$\mathbb{E} = \{\text{pos1, } \sout{\text{pos2, pos3, pos1,}} \text{ pos4, ...}\}$$
$$\mathbb{E} = \{\text{pos1, pos4, ...}\}$$

3.2.2 Démonstration

Nous savons que dans un labyrinthe parfait, il existe seulement un chemin unique entre deux cellules. En parcourant notre ensemble de positions et en cherchant deux positions identiques dans tout l'intervalle, cela montre que l'intersection de ces deux positions est inutile car on se retrouve sur la même position, l'algorithme est donc rentré dans un embranchement sans issue et est retourné sur ses pas.

3.2.3 Pseudo-code

On récupère un tableau de toute les positions visitées de nom BHistorique.

```
1 POUR i = 0 JUSQU'A i < BHistorique.taille() INCREMENT  i = i + 1
2 Faire
3         Si (BHistorique[i] == (tmp = BHistorique.hasValue(BHistorique[i + 1])))
4         Faire
5                 BHistorique.supprimer(BHistorique[i], tmp);
6         FinFaire
7 FinFaire
```

Explication :

- On parcourt tout le tableau de BHistorique
- On regarde si la valeur du tableau actuel existe dans tout le sous tableau suivant à partir de BHistorique[i+1]
- Si oui, on supprime l'intervalle exclus entre BHistorique[i] et inclus BHistorique.hasValue(BHistorique[i+1])

3.3 Algorithme de *Pledge*

3.3.1 Introduction

Dans le cas d'un labyrinthe imparfait notre algorithme de résolution ne fonctionnera pas car, il se retrouvera coincé dans une boucle au moindre ilot rencontré.

Pour palier à ce problème John Pledge a trouvé la solution : il faut s'éloigner de ce genre de pilier !

3.3.2 Explication

Pour cela nous allons choisir une direction et avancer tout droit jusqu'à un mur, puis choisir une direction (droite ou gauche).

Il faudra utiliser un "compteur" pour chaque changement de direction (par exemple +1 pour la droite et -1 pour la gauche en commençant bien sûr à 0). Dès que le "compteur" retombe à 0, nous avançons tout droit jusqu'au prochain mur, puis on choisira à nouveau la même direction que la première fois sans oublier d'augmenter ou de diminuer le "compteur" en fonction. On répétera ces étapes jusqu'à trouver la sortie du labyrinthe.

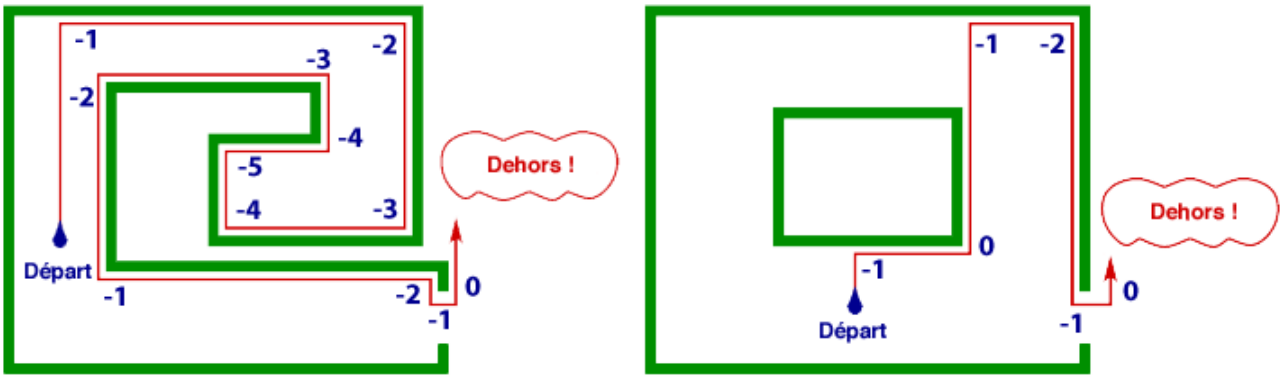


Figure 7 – Déroulement de l'algorithme de Pledge

L'algorithme de Pledge permet de toujours trouver une solution si elle existe. La démonstration rigoureuse étant assez complexe elle ne sera pas abordée dans ce dossier mais, nous allons tenter de la simplifier en suivant l'esprit de la démonstration. Elle part du principe que si on ne trouve pas la sortie avec l'algorithme de Pledge c'est que l'on tournerait en rond dans une boucle infinie.

3 cas pourraient se présenter :

— le premier est impossible dans un labyrinthe : une boucle avec croisement

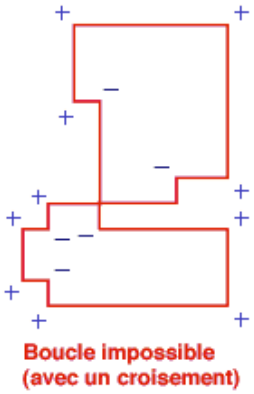


FIGURE 8 – Labyrinthe impossible

— les 2 autres presque similaires : une boucle dans le sens des aiguilles d'une montre et une autre dans le sens inverse.

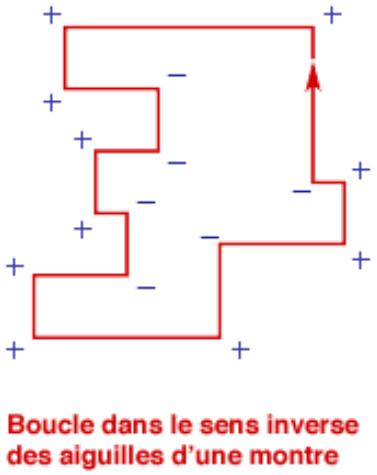


FIGURE 9 – Labyrinthe avec boucle

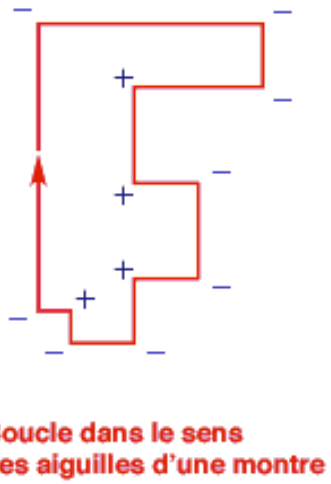


FIGURE 10 – Labyrinthe avec boucle

On se place dans le cas où on tourne à droite la première fois qu'on arrive à un mur. Si on effectuait une boucle sans fin dans le sens inverse des aiguilles d'une montre, il y aurait quatre changements de direction vers la gauche de plus que vers la droite. Le décompte des changements de direction pourrait donc prendre des valeurs positives. Or ceci n'est pas possible, car au moment où on rejoint un mur, le décompte passe à -1. Et dès qu'il atteint la valeur 0, on s'éloigne à nouveau du mur !

Si on pouvait suivre une boucle sans fin, celle-ci devrait donc se dérouler dans le sens des aiguilles d'une montre. Alors, à chaque nouveau passage au même sommet, le décompte diminuerait de 4, de sorte que sa valeur resterait toujours négative et ne prendrait jamais la valeur 0. On longerait donc

constamment le même mur en le gardant à main gauche, sans jamais pouvoir s'en écarter. Cela signifie que dans ce cas, on aurait buté sur un mur d'enceinte sans aucune ouverture. Il n'y aurait aucun moyen de sortir, ni d'ailleurs d'entrer dans le labyrinthe par un chemin classique.

4 Problèmes recontrés

Nous avons eu du mal à choisir le language utilisé pour implémenter nos différents algorithmes (à la fois visuel et rapide). Le démarrage de la programmation a été compliqué mais une fois en place tout s'est bien enchaîné. Le code a été plutôt long à écrire mais le temps de réflexion pour la conception nous a permis d'aller droit au but.

5 Remerciement

Nous tenons à remercier Morgan Morancey pour son aide apportée tout au long de ce projet de modélisation.

6 Conclusion

Il est donc possible de générer une infinité de labyrinthes tous plus complexes les uns que les autres mais les mathématique trouveront toujours la sortie tant qu'elle existe ! Alors, si vous vous retrouvez perdus dans un labyrinthe vous avez toutes les clés en main pour pouvoir en sortir vivants !