université de BORDEAUX

Collège Sciences et Technologies
UF Mathématiques et Interactions

Projet Labyrinthe

Ethan Facca
Lucien Glasson
Aloyse Arnaud

CMI Optim L2

2024–2025

Projet de programmation

Table des matières

1 Introduction

2 Algorithmes de génération

2.1 Backtracking

Cet algorithmes créer un labyrinthe en se déplaçant aléatoirement dans le labyrinthe en créant et détruisant des murs pour faire un chemin. Le labyrinthe est intitialisé avec tous les murs.

Son fonctionnement est le suivant :
- On commence par choisir un cellule dans le labyrinthe, ce sera notre point de départ.
- Ensuite, on marque cette cellule comme visitée et on se déplace aléatoirement dans une cellule voisine non visitée en supprimant le mur entre les deux cellules.
- Si toutes les cellules voisines ont été visitées, on revient en arrière jusqu'à trouvé une cellule avec des voisins non visités.
- L'algorithme se termine quand on est revenu à la cellule de départ.

Algorithme 1 : Backtracking

Entrée : Un labyrinthe *maze* de dimensions $width \times height$ et un booléen $parfait$
Sortie : Un labyrinthe généré

```
Créer un tableau de structures pour stocker les coordonnées des cellules visitées.
Initialiser un index pour suivre la position dans le tableau.
Marquer la cellule de départ (0, 0) comme visitée et l'ajouter au tableau.
tant que l'index est supérieur à 0 faire
    Récupérer les coordonnées de la cellule courante à partir du tableau.
    Trouver un voisin non visité de la cellule courante.
    Si un voisin non visité est trouvé Alors
        Récupérer tous les voisins non visités de la cellule courante.
        pour chaque voisin non visité sauf celui sélectionné faire
            Ajouter un mur entre la cellule courante et ce voisin.
        fin
        Marquer le voisin sélectionné comme visité.
        Ajouter les coordonnées du voisin sélectionné au tableau.
        Incrémenter l'index.
    Sinon
        Compter le nombre de voisins non visités relatifs à la cellule courante.
        Si aucun voisin non visité n'est trouvé Alors
            Décrémenter l'index.
        Sinon
            Récupérer les coordonnées des voisins non visités.
            Retirer un mur entre la cellule courante et un de ses voisins non visités.
            Ajouter les coordonnées de ce voisin au tableau.
            Incrémenter l'index.
        FinSi
    FinSi
fin
Marquer le labyrinthe comme généré.
```

2.1.1 Preuve formelle de la perfection du labyrinthe

Nous devons démontrer que l'algorithme produit des labyrinthes parfaits. Pour cela, notre algorithme doit satisfaire ces deux propriétés pour toutes les configurations possibles de labyrinthe.

Connexité :
- L'algorithme commence à une position de départ et parcourt le labyrinthe en utilisant une approche de backtracking.
- À chaque étape, il avance d'une cellule non visitée et crée un passage vers cette nouvelle cellule.
- En parcourant le labyrinthe de cette manière, toutes les cellules finissent par être connectées, car chaque cellule est atteinte au moins une fois.
- L'algorithme utilise un tableau pour stocker les coordonnées des cellules visitées, garantissant que chaque cellule est visitée et connectée à une autre cellule au moins une fois.
- Lorsqu'une cellule courante n'a plus de voisins non visités, l'algorithme revient en arrière pour explorer d'autres chemins possibles, assurant ainsi que toutes les cellules seront visitées et connectées.

Absence de cycles :
- L'algorithme avance systématiquement d'une cellule non visitée à la fois, en créant des passages uniquement entre la cellule actuelle et la cellule suivante.
- En choisissant un voisin non visité de manière aléatoire et en marquant chaque cellule visitée, l'algorithme évite de revisiter les cellules déjà visitées, ce qui empêche la formation de cycles.
- Chaque cellule est connectée de manière unique à ses voisines, garantissant qu'il n'y a qu'un seul chemin entre deux cellules quelconques.
- Lorsqu'un voisin non visité est trouvé, l'algorithme crée un passage entre la cellule courante et ce voisin, et marque ce voisin comme visité. Cela assure que chaque connexion est unique et ne forme pas de cycle.
- Si une cellule courante n'a plus de voisins non visités, l'algorithme revient en arrière pour explorer d'autres chemins, ce qui garantit que chaque cellule est connectée de manière unique.

L'algorithme génère des labyrinthes parfaits car il garantit la connexité et l'absence de cycles. Chaque cellule est accessible depuis n'importe quelle autre cellule, et il n'y a qu'un seul chemin entre deux cellules quelconques. En utilisant une approche de backtracking et en marquant chaque cellule visitée, l'algorithme assure que toutes les cellules sont connectées sans former de cycles, produisant ainsi un labyrinthe parfait.

2.2 Fractal

2.2.1 Fonctionnement de l'algorithme

Cet algorithme génère un labyrinthes d'une manière qui fait penser aux fractales, d'où son nom. Le labyrinthe est initialisé comme un labyrinthe de taille 1*1 qui ne contient donc aucun mur.
Ensuite à chaque itération de l'algorithme, le labyrinthe carré de taille $n \times n$ est dupliqué 3 fois afin d'obtenir un labyrinthe de taille $2n \times 2n$.

Son fonctionnement est le suivant : - On double la taille du labyrinthe en le dupliquant à droite, en bas, et en bas à droite. On se retrouve avec un labyrinthe composé de 4 partie identique. - On ajoute des murs sur les ligne et colonne du milieu du labyrinthe i.e. on délimite les 4 parties de notre labyrinthe avec des mur. On a donc 4 délimitations, une entre les deux partie du haut, du bas, de gauche et de droite. - On supprime, aléatoirement, 3 murs dans ces délimitations pour créer un chemin entre les 4 parties. Il faut pour cela que les mur choisis soit sur des délimitations différente à chaque fois. - On répète le processus un nombre déterminé de fois.

Algorithme 2 : Fractal

Entrée : Un entier n et un labyrinthe *maze* de taille 1×1, et un booléen $parfait$
Sortie : Un labyrinthe généré de taille 2^n

```
pour c allant de 1 à n faire
    pour i allant de 1 à n faire
        tmp ← maze
        new_maze ← maze
    fin
fin
Marquer le labyrinthe comme généré.
```

2.2.2 Preuve de perfection

le labyrinthe 1

2.3 Diagonal

2.3.1 Fonctionemnet de l'algorithme

Cet algorithme génère un labyrinthe en le parcourant en diagonale. Le labyrinthe est initialisé sans aucun murs.

Son fonctionnement est le suivant :
- On commence par choisir un coins du labyrinthe aléatoirement.
- À chaque étape, il place des murs horizontalement ou verticalement, en fonction de la position actuelle. Une ouverture est laissée à un endroit choisi aléatoirement pour permettre un passage.

Algorithme 3 : Diagonal

Entrée : Un labyrinthe *maze* de dimensions $width \times height$ et un booléen $parfait$
Sortie : Un labyrinthe généré

```
Initialiser la largeur et la hauteur du labyrinthe.
Déterminer aléatoirement le point de départ (coin supérieur ou coin inférieur).
Initialiser les coordonnées x et y en fonction du point de départ.
tant que les coordonnées x et y sont dans les limites du labyrinthe faire
    On défile une cellule qu'on note cell de la file.
    Si un nombre aléatoire est pair Alors
        Créer une sortie verticale.
        Créer une sortie horizontale.
    Sinon
        Créer une sortie horizontale.
        Créer une sortie verticale.
    FinSi
fin
```

Algorithme 4 : création de sortie

```
si la position actuelle est dans les limites du labyrinthe alors
    Déterminer les positions de début et de fin pour la sortie.
    si les positions de début et de fin sont égales alors
        Mettre à jour la position actuelle.
        Retourner.
    fin
    Générer une position aléatoire pour la sortie.
    pour chaque position entre le début et la fin faire
        si la position n'est pas la position de sortie alors
            Ajouter un mur à cette position.
        fin
    fin
    Mettre à jour la position actuelle.
fin
```

2.3.2 Preuve formelle de la perfection du labyrinthe

Nous devons démontrer que l'algorithme produit des labyrinthe parfait. Pour cela notre algorithme doit satisfait ces deux propriétés pour toutes les configurations possibles de labyrinthe.

Connexité :
- L'algorithme commence à une position de départ et parcourt le labyrinthe en alternant entre les directions horizontale et verticale.
- À chaque étape, il avance d'une cellule et crée un passage vers cette nouvelle cellule.
- En parcourant le labyrinthe de cette manière, toutes les cellules finissent par être connectées, car chaque cellule est atteinte au moins une fois.

Absence de cycles :
- L'algorithme avance systématiquement d'une cellule à la fois, en créant des passages uniquement entre la cellule actuelle et la cellule suivante.
- En alternant entre les directions horizontale et verticale, l'algorithme évite de créer des chemins redondants qui pourraient former des cycles.
- Chaque cellule est connectée de manière unique à ses voisines, garantissant qu'il n'y a qu'un seul chemin entre deux cellules quelconques.

En conclusion, l'algorithme génère des labyrinthes parfaits car il garantit la connexité et l'absence de cycles. Chaque cellule est accessible depuis n'importe quelle autre cellule, et il n'y a qu'un seul chemin entre deux cellules quelconques.

2.4 Wallmaker

Cet algorithme génère un labyrinthe en créant des murs aléatoirement. Le labyrinthe est initialisé sans aucun murs.

Son fonctionnement est le suivant : - On part d'une grille vide et on va créer les murs. - On crée un tableau avec tous les murs possible de la grille. - On a choisi un mur aléatoirement dans la liste et on vérifie : - si le mur appartient à une chaine de mur (côte à côte) : - si oui : si la chaine de mur touche deux fois le bord du labyrinthe → le mur ne peut pas être mis là, donc on l'enlève ce mur du tableau. - sinon : on pose le mur et on l'enlève du tableau - On sait le nb de mur au maximum que l'on doit poser (L-1)*(l-1)

3 Algorithmes de résolution

3.1 Depth First Search

Le parcours en profondeur (Depth-First Search) est un algorithme de recherche qui explore un chemin en allant aussi loin que possible dans une direction avant de revenir en arrière pour explorer d'autres chemins.

Son fonctionnement est le suivant :
- Commencer à partir de la cellule de départ du labyrinthe. Marquer cette cellule comme visitée.

Exploration des chemins :
- Depuis la cellule courante, sélectionner une cellule voisine qui n'a pas encore été visitée et qui n'est pas bloquée par un mur.
- Aller dans cette cellule et marquer ce nouveau point comme visité.

Retour en arrière :
- Si la cellule courante n'a aucune cellule voisine accessible et non visitée, revenir à la cellule précédente (retour en arrière dans la pile des cellules visitées).
- Continuer ce processus jusqu'à trouver la cellule de sortie ou épuiser toutes les options.
- L'algorithme se termine une fois que la sortie du labyrinthe est atteinte, ou qu'il n'existe plus de cellules non visitées accessibles, signalant qu'il n'y a pas de solution.

Entrée : Un labyrinthe $maze$ de dimensions $width \times height$ et un booléen $parfait$

Sortie : Un labyrinthe généré

Algorithme 5 : Depth First Search

```
On crée une pile vide et on y empile la cellule de départ.
tant que la pile n'est pas vide faire
    On dépile une cellule qu'on note cell de la pile.
    Si cell a déjà été visitée Alors
        On passe au tour de boucle suivant
    SinonSi cell est la sortie Alors
        On reconstitue le chemin menant à cell et on le retourne.
    Sinon
        On ajoute cell à l'ensemble des cellules visitées.
        On empile toutes les cellules accessibles depuis cell dans la pile.
    FinSi
fin
```

3.2 Breadth First Search

Le parcours en largeur (Breadth-First Search) est un algorithme de recherche qui visite d'abord tous les voisins directs d'un point avant de passer aux voisins de ces voisins, garantissant ainsi de trouver le chemin le plus court vers une cible si elle existe.

Son fonctionnement est le suivant :
- Commencer à partir de la cellule de départ. Placer cette cellule dans une file d'attente et marquer comme visitée.

Exploration par niveaux :
- Tant que la file d'attente n'est pas vide, retirer la cellule en tête de la file (cellule actuelle).
- Examiner toutes les cellules voisines accessibles (sans murs).
- Pour chaque voisine non encore visitée, l'ajouter à la file d'attente, marquer comme visitée.

Trouver la sortie :
- Ce processus se poursuit jusqu'à atteindre la cellule de sortie (si elle est accessible) ou jusqu'à ce que la file soit vide.
- À chaque étape, l'algorithme explore d'abord les cellules les plus proches, garantissant ainsi de trouver le chemin le plus court.
- L'algorithme se termine lorsqu'il atteint la sortie, ou qu'il a exploré toutes les cellules accessibles sans trouver de solution.

Algorithme 6 : Breadth First Search

```
On crée une file vide et on y enfile la cellule de départ.
tant que la file n'est pas vide faire
    le On défile une cellule qu'on note cell de la file.
    Si cell a déjà été visitée Alors
        On passe au tour de boucle suivant.
    SinonSi cell est la sortie Alors
        On reconstitue le chemin menant à cell et on le retourne.
    Sinon
        On ajoute cell à l'ensemble des cellules visitées.
        On enfi toutes les cellules accessibles depuis cell dans la file.
    FinSi
fin
```

4 Validation de labyrinthes

Cet algorithme utilise une approche récursive pour vérifier la validité d'un labyrinthe et sa perfection (absence de cycles). En fonction du type de vérification souhaité (simple ou parfaite), il utilise différentes stratégies de parcours pour déterminer si toutes les cellules du labyrinthe sont accessibles et correctement connectées.

Son fonctionnement est le suivant :

- L'algorithme commence par vérifier chaque cellule du labyrinthe en utilisant une approche récursive. Il vérifie d'abord si le labyrinthe est valide et, si nécessaire, s'il est parfait.

Exploration des cellules :
- Chaque cellule est marquée comme visitée et ses voisins non visités sont explorés.
- Si une cellule n'a plus de voisins accessibles, elle est marquée comme un cul-de-sac.
- Pour un labyrinthe parfait, l'algorithme vérifie également qu'il n'y a pas de cycles (plus de 2 voisins déjà visités).

Vérification de la validité : - L'algorithme parcourt toutes les cellules et s'assure qu'elles sont toutes accessibles. - Si un cycle est détecté dans un labyrinthe parfait, il est marqué comme non parfait.

- Lorsque toutes les cellules sont explorées, l'algorithme affiche si le labyrinthe est valide et, si vérifié, s'il est parfait.