Rapport : Projet Java Labyrinthe

ING 1 GI

Projet CYnapse Groupe 3

Tuteur : M. Haddache

Thomas Pisaneschi

Ivan Chantepie

Gaspard Malmezat

Rémi Mesbahy

Thibaut Laheurte

Projet JAVA

Table des matières

I. Introduction et Contexte

Ce projet a pour objectif de développer une application Java permettant la génération et la résolution de labyrinthes à travers une interface graphique, tout en respectant les contraintes et fonctionnalités définies dans le cahier des charges. La génération est procédurale, basée sur des algorithmes classiques de graphes, et la résolution utilise plusieurs techniques algorithmiques, permettant une exploration approfondie des méthodes d'algorithmique sur graphes.

Initialement, notre approche s'est basée sur la modélisation du labyrinthe comme un graphe sans arêtes, où les sommets étaient connectés par adjacence. Cette représentation, bien que simple, nous a rapidement semblé limitée, notamment car elle ne facilitait pas la gestion des murs et passages typiques d'un labyrinthe. La nécessité de mieux représenter les connexions entre les cellules du labyrinthe, et notamment les murs, nous a conduits à adopter un modèle plus classique de graphe composé de sommets (nodes) reliés par des arêtes (edges). Ce modèle facilite la gestion des murs comme des arêtes présentes ou absentes entre les sommets.

Ainsi, tout au long du projet, nous avons conservé cette structure classique de graphe avec sommets et arêtes, ce qui a permis une gestion claire de la topologie du labyrinthe, indispensable pour la génération et la résolution.

II. Répartition des tâches entre les membres du groupe

- Gaspard et Thomas ont assuré l'ensemble de la partie affichage via JavaFX. Thomas s'est principalement focalisé sur la représentation graphique du labyrinthe, en développant les éléments visuels liés à la structure même du labyrinthe (cases, murs, chemins). De son côté, Gaspard a pris en charge le reste de l'interface utilisateur, incluant les composants annexes, la gestion des interactions utilisateur (boutons, menus) et la mise à jour dynamique de l'affichage lors des opérations de génération ou de résolution du labyrinthe.

- Ivan a été responsable des algorithmes fondamentaux du projet. Il a conçu et implémenté la logique de fonctionnement pas à pas (step-by-step) permettant de visualiser la progression des algorithmes. Il a également développé les mécanismes de sauvegarde et de restauration des labyrinthes, en veillant à ce que la structure des données soit correctement enregistrée. Par ailleurs, Ivan a initialisé les classes essentielles aux graphes, notamment les classes Edge

(arêtes), Node (nœuds) Maze (labyrinthe), qui forment la base de la représentation interne du labyrinthe.

- Rémi et Thibaut ont concentré leurs efforts sur la génération et la résolution des labyrinthes. Ils ont implémenté plusieurs méthodes de création de labyrinthes parfaits, en particulier en utilisant les algorithmes BFS et Kruskal, garantissant qu'il existe un unique chemin entre deux points quelconques du labyrinthe. Ils ont aussi développé la génération de labyrinthes imparfaits, où des cycles et des chemins multiples peuvent exister. En plus de cela, ils ont créé et modifié les classes Node et Edge pour permettre l'implémentation efficace des différentes méthodes de création et de résolution des labyrinthes. Enfin, ils ont intégré six algorithmes de résolution du labyrinthe, permettant à l'utilisateur de choisir entre BFS, DFS, A*, Dijkstra, ainsi que deux variantes du Wall Follower (à gauche et à droite). Ces algorithmes permettent de trouver des chemins dans le labyrinthe et sont visualisés pas à pas ou en mode complet selon le choix de l'utilisateur. Par ailleurs, ils ont réalisé un main interactif servant à l'affichage en console, facilitant ainsi l'interaction et la visualisation des différentes étapes de génération et de résolution des labyrinthes dans la console.

III. Description technique et étapes réalisées

1. Modélisation du labyrinthe

Le labyrinthe est représenté sous forme d'un graphe non orienté où chaque case correspond à un sommet. Les connexions possibles entre cases adjacentes sont représentées par des arêtes. Cette représentation facilite la manipulation des murs (arêtes absentes) et des passages / chemins (arêtes présentes).

2. Génération des labyrinthes parfaits

Nous avons implémenté deux méthodes principales pour générer des labyrinthes parfaits (sans cycles et avec un seul chemin entre deux points) :

- Algorithme de Kruskal
  Cette méthode s'appuie sur une approche d'arbre couvrant minimal. Chaque sommet du labyrinthe est initialement assigné à une pondération unique. On sélectionne aléatoirement des arêtes entre sommets adjacents, et si ces

sommets appartiennent à des ensembles différents (pondérations différentes), on ajoute cette arête au graphe et on fusionne leurs pondérations en une seule. Ce processus continue jusqu'à ce que tous les sommets soient connectés, garantissant un labyrinthe parfait sans cycle.

- Algorithme BFS
  Ici, on commence par un sommet initial aléatoire et on explore ses voisins de manière itérative en largeur, en ajoutant les arêtes nécessaires pour connecter les sommets visités sans créer de cycle. Cette méthode assure également la construction d'un labyrinthe parfait, mais avec une structure légèrement différente générée par l'ordre d'exploration en largeur.

3. Génération de labyrinthes imparfaits

Pour obtenir des labyrinthes imparfaits, nous sommes partis d'un labyrinthe parfait généré par l'une des deux méthodes précédentes. Nous avons ensuite appliqué deux types de modifications aléatoires :

- Suppression d'un certain nombre de murs (arêtes) pour créer des cycles dans le labyrinthe.
- Ajout aléatoire de murs à des endroits choisis au hasard, créant ainsi des zones inaccessibles et des cycles complexes.

Cette méthode permet d'obtenir des labyrinthes avec des caractéristiques plus variées, incluant des cycles multiples et des passages bloqués, renforçant la complexité pour la résolution.

4. Résolution des labyrinthes

Le projet intègre six algorithmes de résolution permettant à l'utilisateur de trouver un chemin entre une position de départ et une position d'arrivée dans le labyrinthe. Ces algorithmes offrent différentes approches, chacune avec ses caractéristiques propres.

BFS

L'algorithme BFS explore le labyrinthe en largeur, visitant les sommets niveau par niveau à partir du point de départ. Il garantit de trouver le chemin le plus court en nombre d'arêtes.

Fonctionnement :

- On utilise une file d'attente pour les sommets à explorer.
- Le point de départ est inséré dans la file.
- Tant que la file n'est pas vide, on retire un sommet, on visite ses voisins non encore explorés, qu'on ajoute à la file.
- On mémorise les parents des sommets pour reconstruire le chemin final.

Pseudo-code :

```
BFS(start, goal):
  queue ← nouvelle file d'attente
  queue.enqueue(start)
  visited ← ensemble vide
  parent ← dictionnaire vide

  visited.add(start)

  while queue is not empty:
    current ← queue.dequeue()
    if current == goal:
      return reconstruct_path(parent, start, goal)

    for each neighbor in neighbors(current):
      if neighbor not in visited:
        visited.add(neighbor)
        parent[neighbor] ← current
        queue.enqueue(neighbor)

  return "Pas de chemin trouvé"
```

DFS

L'algorithme DFS explore le labyrinthe en profondeur, suivant un chemin jusqu'à une impasse avant de revenir en arrière pour explorer d'autres chemins. Il ne garantit pas le chemin le plus court.

Fonctionnement :

- Utilise une pile pour gérer les sommets à explorer.

- On pousse le point de départ sur la pile.
- Tant que la pile n'est pas vide, on dépile un sommet, visite ses voisins non explorés, et les empile.

Pseudo-code :

```
DFS(start, goal):
  stack ← nouvelle pile
  stack.push(start)
  visited ← ensemble vide
  parent ← dictionnaire vide

  visited.add(start)

  while stack is not empty:
    current ← stack.pop()
    if current == goal:
      return reconstruct_path(parent, start, goal)

    for each neighbor in neighbors(current):
      if neighbor not in visited:
        visited.add(neighbor)
        parent[neighbor] ← current
        stack.push(neighbor)

  return "Pas de chemin trouvé"
```

A* (A Star)

L'algorithme A* combine le coût réel parcouru depuis le départ et une estimation heuristique du coût restant jusqu'à l'arrivée pour prioriser les nœuds à explorer. Il est efficace pour trouver un chemin optimal rapidement.

Fonctionnement :

- Maintient une liste ouverte priorisée selon la fonction f(n) = g(n) + h(n), où g(n) est le coût depuis le départ et h(n) l'heuristique estimant la distance restante.
- Explore toujours le nœud avec la plus petite valeur f.

Pseudo-code :

```
A_Star(start, goal):
  open_set ← structure de priorité
  open_set.add(start, f=0)
  parent ← dictionnaire vide
  g_score ← dictionnaire avec valeurs infinies
  g_score[start] ← 0

  while open_set is not empty:
    current ← open_set.pop_lowest_f()
    if current == goal:
      return reconstruct_path(parent, start, goal)

    for neighbor in neighbors(current):
      tentative_g_score ← g_score[current] + cost(current, neighbor)
      if tentative_g_score < g_score[neighbor]:
        parent[neighbor] ← current
        g_score[neighbor] ← tentative_g_score
        f_score ← tentative_g_score + heuristic(neighbor, goal)
        if neighbor not in open_set:
          open_set.add(neighbor, f_score)
        else:
          open_set.update_priority(neighbor, f_score)

  return "Pas de chemin trouvé"
```

Dijkstra

L'algorithme de Dijkstra calcule le plus court chemin dans un graphe pondéré avec des poids non négatifs. Dans notre contexte avec des poids uniformes, il se comporte comme BFS, mais il peut aussi gérer des coûts variables.

Fonctionnement :

- Initialise les distances à l'infini sauf pour le départ à zéro.
- Tant qu'il existe des sommets non visités, on choisit celui avec la plus petite distance connue, on met à jour les distances de ses voisins.

Pseudo-code :

```
Dijkstra(start, goal):
  dist ← dictionnaire avec valeurs infinies
```

```
  dist[start] ← 0
  parent ← dictionnaire vide
  unvisited ← ensemble de tous les sommets

  while unvisited is not empty:
    current ← sommet avec dist minimale dans unvisited
    if current == goal:
      return reconstruct_path(parent, start, goal)

    unvisited.remove(current)

    for neighbor in neighbors(current):
      if neighbor in unvisited:
        alt ← dist[current] + cost(current, neighbor)
        if alt < dist[neighbor]:
          dist[neighbor] ← alt
          parent[neighbor] ← current

  return "Pas de chemin trouvé"
```

Wall Follower Left (Suiveur de mur gauche)

Cette méthode consiste à avancer dans le labyrinthe en gardant toujours le mur situé à gauche. Elle fonctionne sur certains labyrinthes mais ne garantit pas toujours une solution.

Fonctionnement :

- À chaque étape, on tente de tourner à gauche si possible.
- Sinon, on avance tout droit.
- Sinon, on tourne à droite.
- Sinon, on fait demi-tour.

Pseudo-code simplifié :

```
Wall_Follower_Left(start, goal):
  current ← start
  direction ← direction initiale
  path ← liste contenant current

  while current != goal:
```

```
    if left_cell_free(current, direction):
      direction ← tourner à gauche
      current ← avancer dans direction
    else if front_cell_free(current, direction):
      current ← avancer dans direction
    else if right_cell_free(current, direction):
      direction ← tourner à droite
      current ← avancer dans direction
    else:
      direction ← demi-tour
      current ← avancer dans direction
    path.append(current)

  return path
```

Wall Follower Right (Suiveur de mur droit)

Identique à la méthode précédente, mais en gardant toujours le mur situé à droite.

Pseudo-code simplifié :

```
Wall_Follower_Right(start, goal):
  current ← start
  direction ← direction initiale
  path ← liste contenant current

  while current != goal:
    if right_cell_free(current, direction):
      direction ← tourner à droite
      current ← avancer dans direction
    else if front_cell_free(current, direction):
      current ← avancer dans direction
    else if left_cell_free(current, direction):
      direction ← tourner à gauche
      current ← avancer dans direction
    else:
      direction ← demi-tour
      current ← avancer dans direction
    path.append(current)
```

return path

Chaque algorithme est implémenté avec une possibilité d'exécution en mode pas à pas, permettant la visualisation dynamique de la résolution, ou en mode complet où le résultat s'affiche une fois terminé.

5. Fonctionnalités supplémentaires

- Modularité du labyrinthe

  L'utilisateur peut définir la seed (graine) du labyrinthe, ce qui permet une génération reproductible du même labyrinthe à partir de cette valeur. Il peut également choisir la taille du labyrinthe, décider s'il souhaite générer un labyrinthe parfait ou imparfait, et activer un mode de création pas à pas. Ce mode pas à pas permet de visualiser la construction du labyrinthe étape par étape directement sur la grille.

- Choix du départ et de l'arrivée

  L'utilisateur a la possibilité de sélectionner librement une position de départ et une position d'arrivée dans le labyrinthe avant de lancer la résolution.

- Mode pas à pas et contrôle de la résolution

  La création et la résolution du labyrinthe peuvent se faire en mode step by step. L'utilisateur peut ainsi observer chaque étape de la génération ou de la résolution affichée sur la grille. Ce mode permet aussi de mettre en pause le processus à tout moment et de progresser manuellement d'une étape à la suivante. Cette fonctionnalité facilite la compréhension des algorithmes en action.

- Mesure du temps de résolution
  Lors de la résolution, l'utilisateur peut consulter le temps nécessaire pour trouver une solution, ce qui permet d'évaluer la performance des différents algorithmes appliqués.
- Sauvegarde et chargement

  Les labyrinthes générés peuvent être sauvegardés dans un fichier sur le disque

dur et rechargés ultérieurement, assurant ainsi la continuité de l'utilisation et la possibilité de reprendre ou d'analyser des labyrinthes spécifiques.

- Modification locale

  L'utilisateur peut modifier ponctuellement la topologie du labyrinthe en ajoutant ou supprimant un mur (arête) à un endroit précis. Cependant, contrairement à une mise à jour dynamique, le recalcul de la solution ne se fait pas de manière incrémentale : la résolution est relancée entièrement après chaque modification pour garantir l'exactitude du chemin trouvé.

IV. Conclusion et Perspectives

Ce projet a permis de réaliser une application Java complète et fonctionnelle pour la génération et la résolution de labyrinthes, respectant tous les critères du cahier des charges. Notre choix d'une modélisation classique sous forme de graphe avec sommets et arêtes a permis une manipulation claire et efficace des labyrinthes, ainsi qu'une intégration facile des algorithmes classiques de génération et de résolution.

La mise en œuvre de deux méthodes de génération pour les labyrinthes parfaits, combinée à une approche aléatoire de modification pour créer des labyrinthes imparfaits, offre une grande variété de configurations pour l'utilisateur.

L'intégration de six algorithmes de résolution, dont des méthodes classiques et des approches plus heuristiques ou manuelles, permet une exploration pédagogique des différentes stratégies de parcours de graphe.

Enfin, la modularité de l'application choix de la taille, du type de labyrinthe, sauvegarde, modification dynamique garantit une expérience utilisateur riche et adaptée à divers besoins.