IUT Nancy - Charlemagne

Projet Labythin

2021

Bernardoni Erin – Orion Antoine
21/02/2021

Table des matières

Algorithme

Avant d'arriver à notre algorithme final, nous avons réfléchi autour de la création manuelle d'un labyrinthe et des options implémentables. Ainsi, nous avons décidé de se baser sur une entrée et une sortie définies au début. Pour la création du labyrinthe, nous partons d'un labyrinthe rempli de mur et nous créons des chemins au fur est à mesure. L'idée que l'on avait au départ pour la création du chemin entre le point d'entrée et jusqu'à une sortie est relativement simple : en partant du point de départ, un chemin se trace de proche en proche jusqu'à se retrouver dans un cul-de-sac. Le curseur remonte alors en backtracking jusqu'à la dernière cellule ne se trouvant pas dans un cul-de-sac et recommence à tracer un chemin.

Récursif

En réalisant des recherches sur internet autour des algorithmes de créations de labyrinthes, nous avons trouvé un algorithme du nom de *Recursive Backtracker* qui reprend l'idée que nous avions. Pour la création des chemins, nous avons considéré chaque cellule de notre labyrinthe comme soit un mur, soit un chemin. Nous nous déplaçons donc de 2 en 2 afin de ne pas se retrouver avec un bloc de plusieurs chemins. Ainsi, nous nous sommes retrouvés avec l'algorithme suivant :

```
Fonction RecursiveGeneration(Maze maze, Point activeNode) :
        Début
        change(maze, activeNode)
        neighbor <- pickRandomNeighbor(maze, activeNode)
        Tant que (neighbor != Point(-1,-1)) faire
                change(maze, ((neighbor.x + activeNode.x)/2, (neighbor.y+activeNode.y)/2))
                RecursiveGeneration(maze, next)
        Ftant
        Fin
```

Lexique :
maze : Maze → labyrinthe sur lequel nous travaillons
activeNode : Point→ point au hasard à partir duquel nous travaillons
neighbor : Point → point au hasard, voisin de activeNode, qui n'a pas encore été visité
Point<x : entier, y : entier> → variable composite représentant un point

Afin de réduire la taille de l'algorithme et de le rendre plus modulable, nous avons utilisé deux fonctions : *pickRandomNeighbor(Maze maze, Point node)* et *change(Point node*) décrites ci-dessous.

La fonction *change()* permet de transformer un point donné en chemin.

```
Fonction change(Maze maze, Point node) :
        Début
        maze[y][x] <-  caractère de chemin
        Fin
```

Lexique :
maze : Maze → labyrinthe sur lequel nous travaillons
node : Point → point à changer en chemin

La seconde fonction est *pickRandomNeighor*() qui, comme son nom l'indique, permet de choisir au hasard un voisin du point passé en paramètre, à deux cellules de distance. Cette méthode retourne par défaut (-1,-1), à savoir un point qui n'existe pas dans le labyrinthe, si aucun voisin n'existe. Ainsi, nous pouvons le savoir dans l'algorithme principal et le traiter.

```
Fonction pickRandomNeighbor(Maze maze, Point node) :
        Début
        unvisitedNeighbors <- lisvide()
        indice <- 0
        step[][] <- {{2,0}, {0,2}, {-2,0}, {0,-2}}
        Pour chaque item[] de step faire
                x <- node.x + item[0]
                y <- node.y + item[1]
                Si (x ≥ 0 et x < maze.getWidth() et y ≥ 0 et y < maze.getHeight() et maze.isUnvisited
                (x,y))
                        Alors
                                unvisitedNeighbor[indice] <- Point(x,y)
                                indice <- indice + 1
                Fsi
        Fpour
        Si (unvisitedNeighbors.taille > 0)
                Alors Retourne un point au hasard de unvisitedNeighbors
        Fsi
        Retourne un point (-1,-1)
        Fin
```

Lexique :
maze : Maze → labyrinthe sur lequel nous travaillons
node : Point → point pour lequel nous cherchons des voisins
indice : entier → indice courant du tableau
Point<x : entier, y : entier> → variable composite représentant une cellule du labyrinthe
unvisitedNeighbors : Liste<Point> → liste des voisins pas encore visités

Itératif

Nous avons aussi trouvé un second algorithme plus complet que le premier, du nom de *Growing Tree*. La particularité de ce second est que les cellules actives sont stockées au fur et à mesure dans une liste. A chaque itération, une cellule est alors choisie dans la liste. Ce qui est intéressant avec cet algorithme est la possibilité de modifier significativement le comportement du générateur selon la méthode utilisée pour récupérer une cellule active. Ainsi, en sélectionnant à chaque itération la dernière cellule entrée dans la liste, nous retombons sur le *Recursive Backtracker* vu dans la partie précédente. Si l'on choisi une cellule au hasard dans la liste, il s'agit de l'algorithme de Prism. L'algorithme alors obtenu est le suivant.

Dans cet algorithme, nous retrouvons les fonction change() et pickRandomNeighbor() qui sont identiques aux fonctions vues dans la partie précédente. Une nouvelle fonction est cependant utilisée : pickActiveNode(), qui permet de choisir une cellule active dans l'activeSet selon la méthode de récupération utilisée.

```
Fonction iterativeGeneration(Maze maze, Mode mode, Point activeNode) :
        Début
        activeSet <- lisVide()
        indice <- 0
        activeSet[indice] <- activeNode
        indice <- indice + 1
        change(activeNode)
        Tant que !(activeSet vide) faire
                activeNode <- activeSet[pickActiveNode(mode, activeSet)]
                change(activeNode)
                neighbor <- pickRandomNeighbor(maze, activeNode)
                Si (neighbor = Point(-1,-1)
                        Alors enlever activeNode de activeSet
                        Sinon
                                change(maze, ((neighbor.x + activeNode.x)/2,
                                        (neighbor.y+activeNode.y)/2))
                                activeSet[indice] <- neighbor
                                indice <- indice + 1
                Fsi
                change(neighbor)
        Ftant
        Fin
```

Lexique :
maze : Maze → labyrinthe sur lequel nous travaillons
activeNode : Point → point pour lequel nous cherchons des voisins
activeSet : Liste<Point> → liste des cellules actives, ayant déjà été visitées et ayant encore au moins un voisin non visité
indice : entier → indice courant du tableau
Point<x : entier, y : entier> → variable composite représentant une cellule du labyrinthe

Implémentation Java

Classe Maze

Avant de procéder à l'implémentation des algorithmes de génération de labyrinthe, nous avons réalisé une classe Maze afin de regrouper toutes les fonctions majeures de manipulation d'un labyrinthe. Cette classe contient trois attributs : width et height qui sont tous deux de type entier, contenant respectivement la largeur et la hauteur du labyrinthe et tab, un tableau bi-dimensionnel de caractères associant l'état d'une cellule (chemin ou mur) à des coordonnées.

```
public class Maze {
	private int width;
	private int height;
	private char[][] tab;

	public Maze(int width, int height, MazeElement default_element) {
		this.width = width;
		this.height = height;
		this.tab = new char[this.height][this.width];
		for (int y = 0; y < this.height; y++) // row by row
				Arrays.fill(tab[y], default_element.getChar());
	}
}
```

Cette classe comporte cinq méthodes et utilise le principe d'overloading afin notamment d'éviter la création d'objet de type Point en passant directement les coordonnées lorsque nécessaire.
Ainsi, nous avons une première méthode *change()* permettant de modifier l'état d'une cellule. Par défaut, la cellule est changée en élément *VISITED_PATH* de l'énumération MazeElement, ce qui correspond à un chemin visité. On peut remarquer que dans la première méthode écrite ci-dessous, l'appel de la cellule dans le tableau à deux dimensions se fait d'abord par l'appel de y et ensuite de x, contrairement à ce qu'on a l'habitude de faire dans un graphique. Nous avons décidé de considérer les colonnes en x et les lignes en y, cependant, dans l'appel d'un tableau la première valeur est l'indice de la ligne et le second l'indice de la colonne, d'où l'inversement dans cette première méthode.

```
public void change(int x, int y, MazeElement value) {
   this.tab[y][x] = value.getChar();
}

public void change(int x, int y) {
   change(x, y, MazeElement.PATH_VISITED);
}

public void change(Point node, MazeElement value) {
   change(node.x, node.y, value);
}

public void change(Point node) {
   change(node, MazeElement.PATH_VISITED);
}
```

La seconde et la troisième méthode sont surtout nécessaires pour l'affichage et pour certaines vérifications en fin d'algorithme. Ces deux méthodes permettent respectivement de vérifier si une cellule donnée est un chemin et si elle est un mur. Comme la première méthode présentée, ces deux méthodes se déclinent de deux manières différentes selon les arguments passés : un objet de type Point ou deux coordonnées x et y.

```
public boolean isPath(Point node) {
return (isWall(node.x, node.y));
}

public boolean isPath(int x, int y) {
return (getElement(x, y) == MazeElement.PATH);
}

public boolean isWall(Point node) {
return (isWall(node.x, node.y));
}

public boolean isWall(int x, int y) {
return (getElement(x, y) == MazeElement.WALL);
}
```

La quatrième méthode, *isUnvisited()* est elle aussi une méthode de test, qui permet de savoir si une cellule a déjà été visitée ou non, ce qui est utile pour le choix des voisins non visités lors de l'implémentation de l'algorithme. Une fois encore, la méthode se décline de deux manières différentes, selon qu'un Point ou deux coordonnées lui sont passées, mais réalise la même chose dans les deux cas.

```
public boolean isUnvisited(Point node) {
return (this.getValue(node.x, node.y) == MazeElement.PATH_UNVISITED.getChar() ||
this.getValue(node.x, node.y) == MazeElement.WALL_UNVISITED.getChar());
}

public boolean isUnvisited(int x, int y) {
return (this.getValue(x, y) == MazeElement.PATH_UNVISITED.getChar() ||
this.getValue(x, y) == MazeElement.WALL_UNVISITED.getChar());
}
```

Implémentation

Itérative

Dans un premier temps, nous avons implémenté l'algorithme itératif afin de bien comprendre le comportement de la génération d'un labyrinthe. Les deux implémentations, itérative et récursive, se basent sur une même méthode *generate()*, qui gère les l'appel de l'algorithme principal ainsi que l'appel des méthodes plus générales, comme pour le calcul du temps de génération, l'achèvement des bordures, le contrôle de la sortie et le polissage du labyrinthe.

```
public void do_iterative(Maze maze, Mode mode) {
	ArrayList<Point> activeSet = new ArrayList<Point>();

	// Originel node
	activeSet.add(this.activeNode);
	this.printer.stepDisplay(maze);
	maze.change(this.activeNode);
	while (!activeSet.isEmpty()) {
		this.printer.stepDisplay(maze);
		// Select node from active set
		try {
			this.activeNode = activeSet.get(pickActiveNode(mode, activeSet));
		} catch (Exception e) {
			this.printer.print(MessageLevel.FATAL, e.getMessage());
		}
		maze.change(activeNode);
		// Choose a random neighbor node, if none available remove active node from active set
		if ((neighbor = pickRandomNeighbor(maze, this.activeNode)).equals(new Point(-1, -1))) {
			activeSet.remove(this.activeNode);
			continue;
		} else {
			// inbetween node
			maze.change((this.activeNode.x + neighbor.x) / 2,
					(this.activeNode.y + neighbor.y) / 2);
			activeSet.add(neighbor);
		}
		// neighbor
		maze.change(neighbor);
	}
}
```

La méthode de génération itérative du labyrinthe en elle-même, *do_iterative()*, prend 2 arguments : *maze* correspondant au labyrinthe de travail et *mode*, le mode de génération du Growing Tree, qui va influer sur le choix de chaque cellule dans l'activeSet. Nous avons dans un premier temps défini un objet de type ArrayList, nommé activeSet, qui contiendra les cellules « actives », à savoir les cellules déjà visitées, qui possèdent encore des voisins non visités. Nous avons fait le choix d'un ArrayList au lieu d'un tableau classique afin, entre autres, de pouvoir utiliser toutes les méthodes de cet objet pour gérer les éléments et ne pas avoir à les déplacer manuellement au sein du tableau lorsqu'on en enlève ou qu'on en ajoute. Cet ArrayList contient des objets de type Point, amenés avec le package java.awt disponible de base dans l'environnement de développement.
Les différentes étapes de la méthode itérative suivent ensuite l'algorithme expliqué plus haut. En premier, nous ajoutons l'activeNode, à savoir ici la cellule d'entrée du labyrinthe, dans l'activeSet et nous la changeons en chemin visité. Ensuite, tant que l'activeSet n'est pas vide, et donc qu'il y a encore des cellules présentant un voisin non visité, nous choisissons une cellule dans activeSet, puis nous procédons au changement d'état des cellules comme décrit dans l'algorithme. Pour le choix de cette cellule, nous utilisons une méthode *pickActiveNode()*, qui se base sur le mode passé en paramètre au début.

```
private int pickActiveNode(Mode mode, ArrayList<Point> activeSet) throws Exception {
```

```
	switch (mode) {
		case RECURSIVE_BACKTRACKER :
			return(activeSet.size() - 1);
		case PRISM :
			return(this.rand.nextInt(activeSet.size()));
		case OLDEST :
			return(0);
		case FIFTY_FITY :
			if (Math.random() > 0.5)
				return(pickActiveNode(Mode.RECURSIVE_BACKTRACKER,
activeSet));
			return(pickActiveNode(Mode.PRISM, activeSet));
		default:
			throw new Exception("Issue when picking active node : mode unknow");
	}
}
```

Cette méthode utilise l'énumération Mode, afin de rendre plus propre le code et de ne pas passer par des String et des comparaisons de String. Nous avons implémenté 4 modes différents, correspondants à 4 comportements différents de l'algorithme du Growing Tree : le recursive backtracker, prism, le plus veux et 50/50 sur le récursive backtracker et le prism. Procéder de cette manière peut permettre d'ajouter facilement de nouveaux comportements avec un nombre très réduit de lignes.

Récursive

Nous avons ensuite dans un second temps implémenter l'algorithme récursive, à savoir celui correspondant au Recurive backtracker uniquement. Contrairement à l'implémentation itérative, nous n'avons pas besoin ici de créer une liste contenant l'activeSet, puisque cela est compris dans la récursivité.

```
public void do_recursive(Maze maze, Point activeNode) {
	Point next = null;

	this.printer.stepDisplay(maze);
	maze.change(activeNode);
	while (!(next = this.pickRandomNeighbor(maze, activeNode)).equals(new Point(-1, -1))) {
		this.printer.stepDisplay(maze);
		maze.change((activeNode.x + next.x) / 2,
					(activeNode.y + next.y) / 2, MazeElement.PATH);
		do_recursive(maze, next);
		this.printer.stepDisplay(maze);

	}
	maze.change(activeNode, MazeElement.PATH);
	this.printer.stepDisplay(maze);
}
```

Le reste de l'implémentation correspond à l'algorithme récursif que nous avons décrit plus haut. Ici, contrairement à l'implémentation itérative où nous bouclions tant que l'activeSet n'était pas vide, nous bouclons tant que le voisin n'est pas un point de coordonnées (-1,-1), ce qui signifie que plus aucun voisin est disponible. Dans ce cas, l'algorithme ne peut plus continuer à « avancer » et est donc contraint à retourner en arrière. Cet algorithme permet donc de se passer d'une liste contenant des objets Point et qui prend donc un espace non négligeable en mémoire pour de grands labyrinthe. Cependant, la récursivité demande elle aussi un certain espace en mémoire afin de stocker toutes les étapes précédentes. Notre algorithme remontant uniquement lorsqu'il se trouve dans un cul-de-sac, beaucoup d'étapes peuvent alors être retenues en mémoire. C'est pour cette raison qu'à partir d'un labyrinthe aux alentours de 215 sur 215 cellules, le programme renvoie une erreur d'overflow.

Amélioration

L'un des principaux problèmes inhérents de l'algorithme du Recursive Backtracker est la création de « chemin ». En effet, l'algorithme avançant jusqu'à un cul-de-sac, les premières itérations de celui-ci forment de grand chemin, desquels naissent par la suite des « branches » plus petites. Ce comportement implique que, par défaut, aucun chemin ne peut en rejoindre un autre, donnant alors un aspect particulier au labyrinthe mais empêchant surtout d'y « tourner en rond ». En effet, dans cette configuration, chaque chemin mène soit à une impasse, soit à l'arrivée.

Pour apporter un correctif à ce problème, nous avons décidé non pas de modifier les algorithmes eux-mêmes, mais plutôt de modifier le labyrinthe une fois constitué. Ainsi le comportement par défaut du programme est que, une fois le labyrinthe fini, une fonction examine chacune des lignes de celui-ci et, si le pourcentage de mur est trop élevé vis-à-vis de la longueur du labyrinthe, ajoute aléatoirement des murs. Cette méthode présente un double avantage : tout d'abord, elle crée des trous dans les murs et apporte une réponse au problème des chemins mais elle assure aussi que la répartition des murs dans le labyrinthe se fera de manière équitable. Enfin, comme la fonction ajoute des chemins et non des murs la sortie du labyrinthe restera quoi qu'il arrive accessible depuis l'entrée.

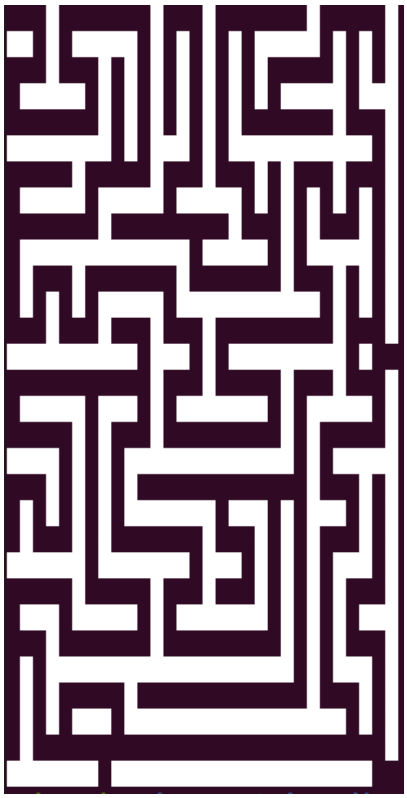

Figure 2 : Labyrinthe crée par Récursive Backtracking

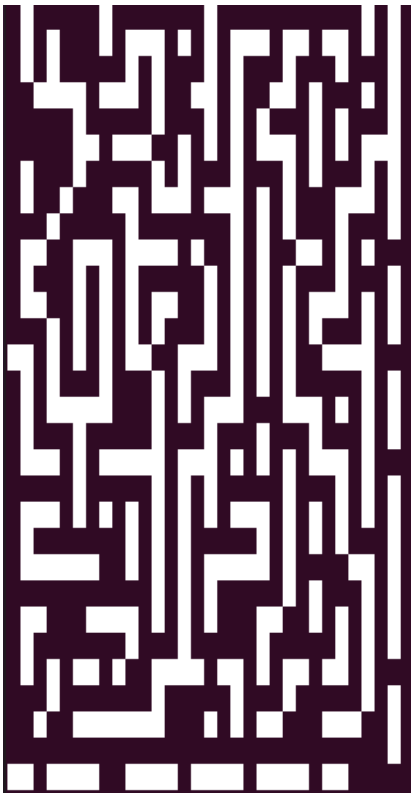

Figure 1 : Labyrinthe crée par Récursive Backtracking avec fonction d'amélioration

L'une des améliorations qui pourrait-être faite serait alors simplement d'appliquer une deuxième fois la même fonction mais de manière verticale cette fois. De même, la désactivation du charactère obligatoire de l'application de cette fonction pourrait être ajouté en tant qu'option.

Librairie Picocli

Dès le début du programme nous avons décidé d'utiliser une librairie de parsing pour récupérer les informations passées au programme via les arguments en ligne de commande. Le choix de Picocli comme librairie plutôt que d'autre CLI tel que Commons CLI ou JSAP qui sont pourtant bien plus utilisé s'est fait pour plusieurs raisons principales :

- D'apparence plus facile à maitriser que les autres, tout en disposant quand même de toutes les options que nous cherchions.
- Gestion des couleurs rendue plus facile.
- Très bonne symbiose avec GraalVM native image, qui était au début du projet une option que nous nous laissions (abandonné par manque de temps).

L'utilisation de la librairie Picocli, bien qu'ayant nécessité un temps conséquent de notre part pour arriver à l'intégrer à notre projet, s'est révélé extrêmement bénéfique. Le parsing de la ligne de commande est celui d'un logiciel professionnel, tandis que le programme a pu bénéficier par effets de bord de fonctionnalité amenées par la librairie mais non liées au parsing.

Parsing de l'entrée standard

L'utilisation d'une bibliothèque de parsing nous a permis d'ajouter très facilement n'importe quelle idée d'option qui pouvait nous passer par la tête tout en étant assuré que celle-ci serait tout à fait bien gérée en cas d'erreur par l'utilisateur. Picocli supportant de nombreux format d'options, l'utilisation de celle si ne paraît ni frustrant, ni contraignant ; ce qui était l'un des moteurs de notre décision d'utiliser un parser CLI.

```
erin@erin:~/Documents/prog/java/Labythin$ ./Labythin 20 20 -m -cs 1 -o=Labyrinthe.txt -a PRISM
```

Affichage

Picocli rend non seulement le parsing plus facile, il en va en effet de même pour l'affichage. L'aide, la version du programme ou l'affichage des couleurs en général peuvent être gérés par Picocli, rendant leur utilisation et leur intégration au programme bien plus aisée.

Aide

La librairie va d'elle-même mettre en forme la plupart des éléments à afficher dans l'aide. Ainsi l'affichage des options, paramètres, valeurs par défaut et espacements, s'ils peuvent êtres personnalisés, sont automatiques.

L'affichage de l'aide peut être adapté pour être contextuel et plus précis en fonction du problème rencontré. Ainsi, si un utilisateur fait une erreur en utilisant une option, Picocli donne la possibilité au développeur d'afficher alors une partie seulement de l'aide, avec des précisions supplémentaires concernant uniquement cette option tel que des exemples. Par manque de temps, c'est quelque chose qui n'a pu être fait.

Version

L'affichage de la version du programme est là aussi rendu plus simple par Picocli, et est personnalisable. Elle permet notamment de vérifier quelle version du programme et de la librairie est utilisée rendant plus facile le suivit de beugs/options au fur et à mesure que le programme évolue.

Couleurs

Picocli permet un affichage coloré, et ce plus facilement qu'en utilisant les codes couleurs utilisés habituellement pour enjoliver le terminal. De plus, la librairie vérifie d'elle-même si oui ou non le terminal supporte les caractères d'échappement ANSI pour ainsi éviter un affichage proche du Mojibake.

```
erin@erin:~/Documents/prog/java/Labythin$ ./Labythin --help

Name:
 ##          ###    ########   ##    ##  ########  ##      ##  ####  ##    ##
 ##         ## ##    ##     ##  ##  ##      ##     ##      ##   ##   ###   ##
 ##        ##   ##   ##     ##   ####       ##     ##      ##   ##   ####  ##
 ##       ##     ##  ########     ##        ##     ##      ##   ##   ## ## ##
 ##       #########  ##           ##        ##     ##      ##   ##   ##  ####
 ##       ##     ##  ##           ##        ##     ##      ##   ##   ##   ###
 ########  ##     ##  ##           ##        ##     ##      ##  ####  ##    ##

Usage:
Labythin [-chmptvV] [-a[=<mode>]] [-s[=<step>]] [-o=FILE] width height

Description:
Create mazes using the growing tree algorithm.

Parameters:
      width                 width  of the maze to create. default : 30
      height                height of the maze to create. default : 30

Options:
  -a, --algorithm[=<mode>]   Algorithm to use to generate the maze with.
                             If specified, the maze will be generated
                               iteratively, otherwise it will be done
                               recursively
                             values : NONE, RECURSIVE_BACKTRACKER, PRISM,
                               OLDEST, FIFTY_FITY
                             default : NONE
  -s, --step[=<step>]        will display the maze at every step of the maze's
                               creation
  -v, --verbose              verbose mode of display
  -c, --color                color the output, makes it look fabulous
  -o, --output=FILE          file to output the maze to
  -p, --pretty-print         display wall as wall with special characters
  -t, --time                 time took to generate the maze
  -m, --memory               display the allocated memory/memory used by the
                               program
  -h, --help                 Show this help message and exit.
  -V, --version              Print version information and exit.

Authors:
BERNARDONI Erin
ORION Antoine
```

Figure 3 : Affichage de l'aide du programme Labythin