LE LABYRINTHE DE MINOS

ALGORITHME DE KRUSKAL

NASSIM KISSI - SIBORY GUEYE

SOMMAIRE

A) GÉNÉRALITÉS

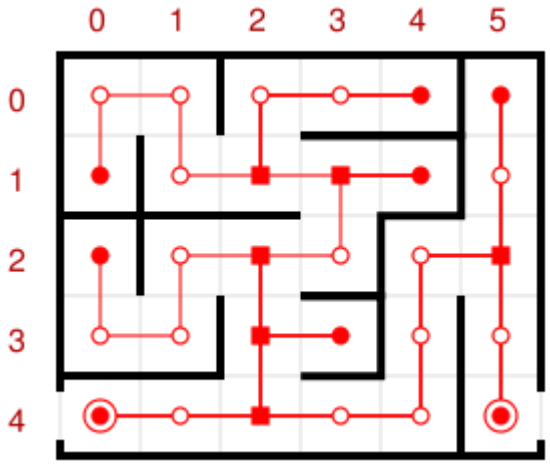
1) GRAPHES ET LABYRINTHES

Le labyrinthe est un « motif » apparu dès la préhistoire, son but premier a toujours été de perdre ou de ralentir quiconque ose s'y aventurer. En latin, labyrinthe vient de *labyrinthus* ce qui signifie « enclos de bâtiments dont il est difficile de trouver l'issue ».De tout temps, le labyrinthe a souvent été utilisé comme symbole de réflexion et de choix méthodiques dans les mythologies, ou dans des œuvres culturelles.

Le lien entre un labyrinthe et un graphe ne semble pas évident à trouver mais pourtant il existe. Rappelons qu'un graphe est un ensemble de points (ou nœuds ou sommets) reliés ensemble par des lignes (ou arêtes). Un graphe peut être connexe (regroupé en un seul objet) ou non-connexe (comprenant plusieurs objets), il peut également être cyclique lorsque ses points sont reliés en formant une boucle ou acyclique. Un graphe peut aussi être orienté (les arêtes sont à sens unique) ou non-orienté.

Afin d'établir le lien entre un labyrinthe et un graphe supposons que l'on peut considérer le labyrinthe comme une grille de cellules qui sont reliées ou non entre elles. Deux cellules sont reliées entre elles par une porte, ou séparées par un mur. Tout labyrinthe a une entrée et une sortie, et quelle que soit l'entrée ou la sortie, le chemin entre ces deux cellules est unique.

Il est alors possible d'intégrer un graphe au sein d'un labyrinthe, pour parvenir à effectuer cette tâche il faut placer un point sur chaque cellule du labyrinthe.



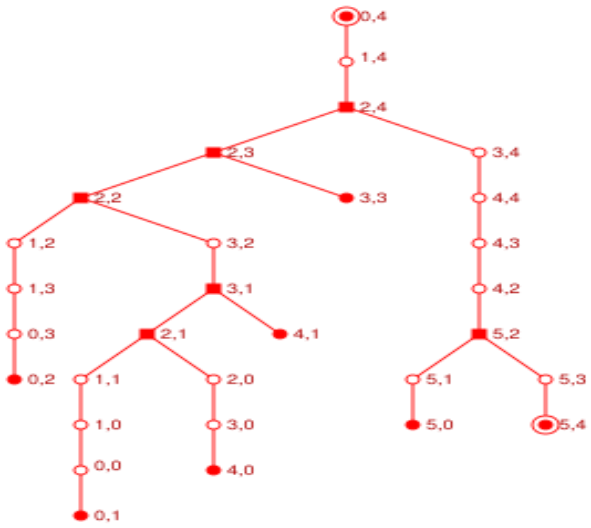
Le point correspondra donc à un endroit précis au sein du labyrinthe, les croisements de chemins peuvent alors être traduis comme un croisement de points au sein d'un graphe.

La sortie d'un labyrinthe relève plus généralement de la recherche de chemin dans le graphe du labyrinthe. On fait face alors à deux configurations différentes :

Dans le premier cas, on dispose d'une vue globale et on est capable de distinguer sans ambiguïté deux positions. De plus, on sait localiser la sortie. On peut alors utiliser toute ces connaissances pour mesurer une distance (comme Dijkstra ou A*) par rapport à la sortie et déterminer le chemin pour l'atteindre.

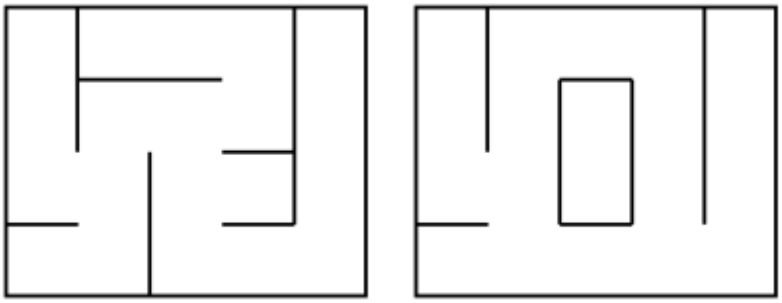
Le second cas est celui de la vue locale, celle qu'aurait la personne qui serait placée dans le labyrinthe (toute autre perception que les murs avoisinants étant négligée) en une immersion en trois dimensions. Cette personne ne dispose alors plus de moyen de distinguer un couloir ou un carrefour d'un autre. La sortie du labyrinthe s'apparente alors surtout à de la chance pure.

Selon la configuration la recherche de chemins peut donc être traduite en problème mathématique. On s'aperçoit rapidement que les choix de chemins disponibles peuvent correspondre à un embranchement de plusieurs arêtes, ce qui nous donne une figure ressemblant à un arbre de probabilités.



2) MODÉLISATION MATHÉMATIQUE DE LABYRINTHE

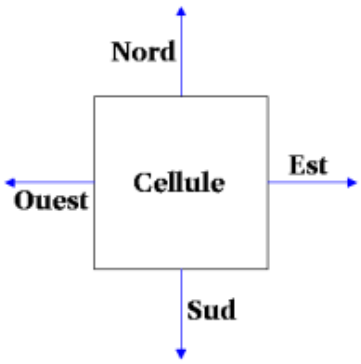
Les labyrinthes peuvent donc être étudiés comme des objets mathématiques, on nomme cela la modélisation mathématique de labyrinthe. La génération automatique de labyrinthe et la résolution de labyrinthe sont des aspects centraux de ce concept. Il existe deux types de labyrinthes, les labyrinthes dits « parfaits » et les labyrinthes « imparfaits » :



Les labyrinthes parfaits sont des labyrinthes où chaque cellule est reliée aux autres de manière unique.

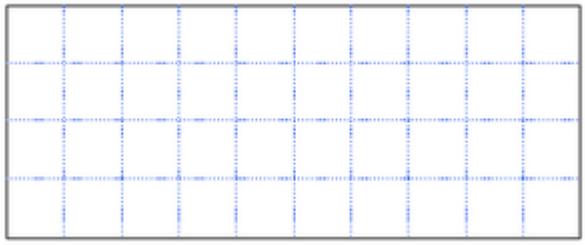
Les labyrinthes imparfaits sont tous ceux qui ne rentrent pas dans la première catégorie. Ce sont les labyrinthes qui peuvent contenir des boucles, des îlots ou même certaines cellules peuvent être inaccessibles.

Les algorithmes dédiés à la création de labyrinthe s'appuient en général sur l'espace discrétisé dont les cellules carrées sont initialement remplies et séparées par des murs, selon les quatre directions (nord, sud, est et ouest).



Soit un labyrinthe rectangulaire parfait de colonnes **H** par lignes **L** est un ensemble de cellules **HL** reliées les unes aux autres par un chemin unique.

Le nombre de murs ouverts pour permettre un chemin unique dans un labyrinthe de **HL** cellules est identique au nombre de murs ouverts pour un chemin droit de HL cellules, soit **HL** – 1 mur. Dans un rectangle **H** × **L** cellules, le nombre total de murs internes possible est : 2**HL** − **H** − **L**.



Pour obtenir ce résultat, on compte deux murs par cellule, par exemple celui du bas et celui de droite (on évite ainsi de recompter deux fois les mêmes) et on retire le nombre de murs limitant le rectangle en bas **H** et à droite **L**. Le nombre de murs fermés dans un labyrinthe parfait est donc : 2**HL** – **H** – **L** – (**HL** – 1) = (**H** – 1)(**L** – 1).

3) PROBLÉMATIQUE

Comment répartir toutes ces portes de façon à ce que toutes les cellules soient accessibles et qu'il n'existe qu'un unique chemin entre l'entrée et la sortie ? De nombreux algorithmes existent, dont le parcours en profondeur est l'algorithme de Kruskal. Chaque algorithme est différent et produit des labyrinthes visuellement différents.

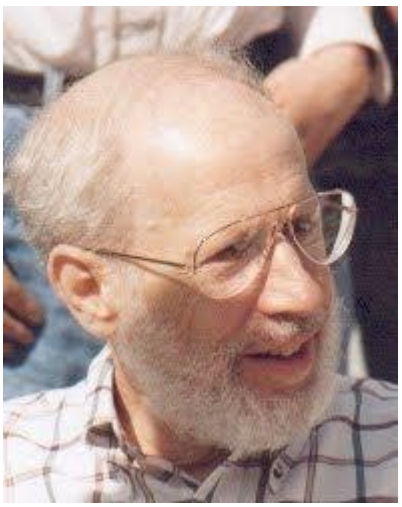
L'algorithme de Kruskal est un algorithme qui n'est à la base pas destiné à générer des labyrinthes. Cet algorithme est utilisé afin de trouver l'arbre couvrant ayant un poids inférieur à tous les autres arbres couvrant du graphe. Un arbre couvrant est un arbre qui couvre tous les sommets d'un graphe. Lorsqu'un tel arbre est démontré, le poids de cet arbre est égal au poids de tous les sommets qui le compose.

Toutefois le générateur de labyrinthe permet lui de créer des labyrinthes parfait. Cet un générateur qui diffère légèrement de l'algorithme notamment durant la phase d'itération. Le choix d'un tel algorithme dans l'optique de créer un labyrinthe soulève plusieurs interrogations.

B) ALGORITHME DE KRUSKAL

1) DÉFINITION

Joseph Kruskal était un mathématicien, statisticien, chercheur en informatique et psychométricien. Il est né le 29 janvier 1928 à New York et est décédé le 19 septembre 2010 à Princeton. Son algorithme a été conçu en 1956.



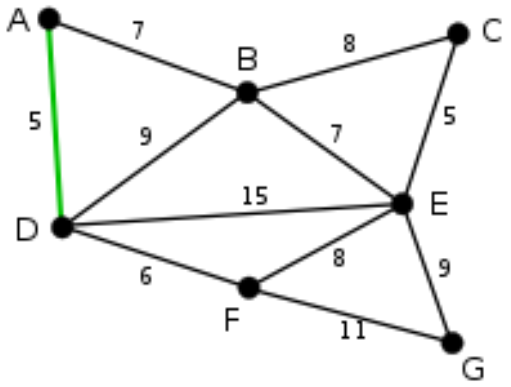
L'algorithme de Kruskal est donc un algorithme qui se charge de trouver un arbre couvrant de poids minimum. Les arbres couvrants de poids minimum interviennent dans la conception et le tarifage des réseaux de communication et de distribution les plus généraux.

2) INITIALISATION, ITÉRATION ET CONDITION DE FIN

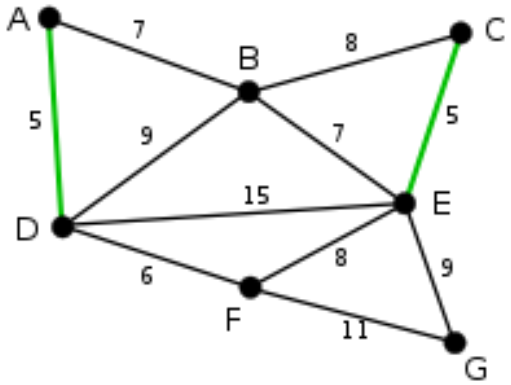
a) Initialisation

On considère un graphe non orienté évalué (A,B,C,D,E,F,G), chaque arête du graphe possède un poids. Rappelons que l'objectif de l'algorithme est de trouver un arbre couvrant de poids minimal.

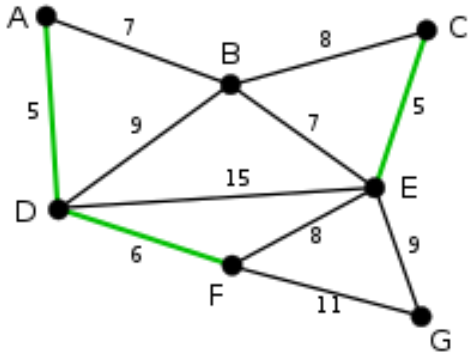
b) Itération



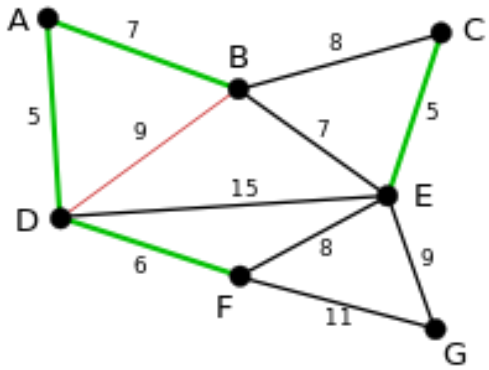
Avec 5, AD et CE sont les arêtes avec les poids les plus faibles. On choisit en premier AD de façon totalement arbitraire.



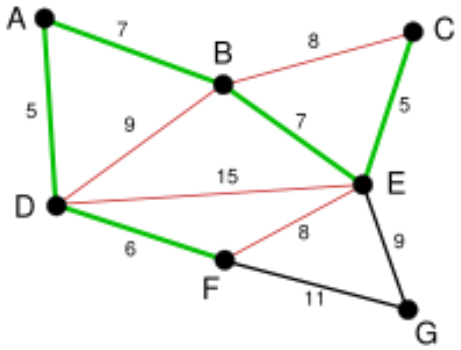
Logiquement l'arête suivante à être sélectionner est CE, d'autant plus que sa sélection ne créée pas de cycle.



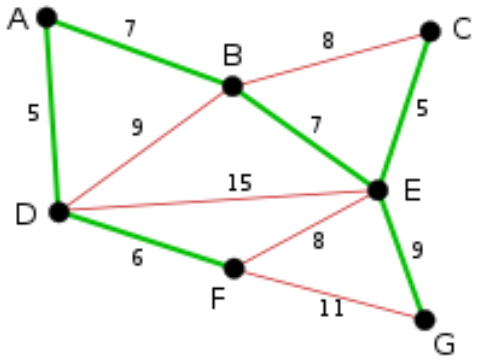
Puis on sélectionne l'arête DF de poids 6, suivante sur la liste.



Les arêtes suivantes à être sélectionnées sont AB BE. Le premier sommet est toujours choisit de manière arbitraire. BD apparait en roue car il ne sera jamais sélectionné car il existe déjà un chemin entre ces deux points (AD et AB).



Après avoir sélectionné BE on peut exclure BC de l'arbre car cela formerait le cycle BCE. Le même raisonnement pouvant être appliqué à FE et à DE.



Il ne reste alors que l'arête EG avec un poids de 9, l'arbre couvrant minimum est alors trouvé, en vert sur le schéma.

c) Condition de fin

La condition de fin de l'algorithme est obtenue lorsque l'arbre couvrant minimum est trouvé, c'est-à-dire lorsqu'il passe par tous les sommets.

A noter également que la découverte d'un sommet qui engendrerait la création d'un cycle mettrait fin à l'algorithme. Dans ce cas précis, la solution n'a pas été trouvée.

3) MÉTHODOLOGIE

Processus de rangement des arêtes du graphe par leur poids par ordre croissant. On enlève les unes après les autres les arêtes selon ce même ordre croissant. On les ajoute alors à l'arbre couvrant en cours de création tout en veillant à ce que l'intégration d'une arête ne créé pas de cycle. L'arbre couvrant

créé doit couvrir tous les sommets et son poids sera toujours inférieur à celui d'un autre arbre trouvé dans le même graphe.

4) COMPLEXITÉ

En algorithmique, la complexité en temps est une mesure du temps utilisé par un algorithme, exprimé comme fonction de la taille de l'entrée. Le temps compte le nombre d'étapes de calcul avant d'arriver à un résultat.

Pour que l'algorithme s'exécute jusqu'au bout, il faut ajouter des arêtes (notées **S**), ce qui peut prendre jusqu'à **N** itérations. A chaque itération, la recherche de la plus petite arête, tout comme la détection d'un cycle, se font en $O(\boldsymbol{N})$. La complexité est donc de $O(\boldsymbol{N^2})$. Il est en fait possible d'améliorer cette complexité sur deux points :

On peut trier les arêtes par poids croissant une seule fois en début d'algorithme, et pointer ensuite sur l'arête courante. Ce tri peut se faire en $O(\boldsymbol{N}\ log\ \boldsymbol{N}) = O(\boldsymbol{N}\ log\ \boldsymbol{S})$.

La recherche de cycle peut se faire de façon bien plus efficace que par une exploration dans notre cas. En effet, on crée un cycle si on rejoint deux sommets d'un même arbre. En tenant à jour les indices des composantes connexes de chaque sommet de la forêt via une structure adaptée (la structure Union-Find), il est possible de vérifier en $O(log\ \boldsymbol{S})$ si une arête est éligible ou non. En combinant ces deux améliorations, on obtient une complexité de $O(\boldsymbol{N}\ log\ \boldsymbol{S})$ pour l'algorithme de Kruskal.

5) UNION-FIND

En informatique, union-find est une structure de données qui représente une partition d'un ensemble fini (ou de manière équivalente une relation d'équivalence). Elle a essentiellement deux opérations trouver et unir et est appelée union-find, suivant en cela la terminologie anglo-saxonne :

• **Find** (trouver) : détermine la classe d'équivalence d'un élément ; elle sert ainsi à déterminer si deux éléments appartiennent à la même classe d'équivalence ;

• **Union** (unir) : réunit deux classes d'équivalence en une seule.

Une autre opération importante, MakeSet (dans notre programme cette méthode s'appelle creeEnsemble), construit une classe d'équivalence contenant un seul élément, autrement dit un singleton.

Afin de définir ces opérations plus précisément, il faut choisir un moyen de représenter les classes. L'approche traditionnelle consiste à sélectionner un élément particulier de chaque classe, appelé le représentant, pour identifier la classe entière.Lors d'un appel, Find(x) retourne le représentant de la classe de x.

Il existe plusieurs variantes d'Union-Find, dont notamment la compression de chemin, qui permet de n'avoir qu'un ou deux éléments à parcourir pour obtenir le représentant d'un élément.

Union-Find fonctionne comme un arbre. 'a' a un représentant 'A', etc..., qui lui même a un représentant 'α', qui est son propre représentant.

Union-Find est composé de deux méthodes :

• union(a, b) : associe à 'a' et 'b' le même représentant, par exemple 'A' le représentant de 'a'

• find(a) -> α : tant que l'élément n'est pas son propre représentant, on applique find au représentant de l'élément. On commence donc par find(a), puis find(A) (si 'A' est le représentant de 'a'), etc... La méthode de la compression de chemin permet d'éviter trop de récursivité en écrivant : représentant de 'a' = α.

6) SPÉCIFICITÉ DE L'ALGORITHME AVEC LE LABYRINTHE

Le générateur de labyrinthe de Kruskal est une version aléatoire de l'algorithme de Kruskal : une méthode pour produire un arbre couvrant de poids minimum à partir d'un graphe pondéré.

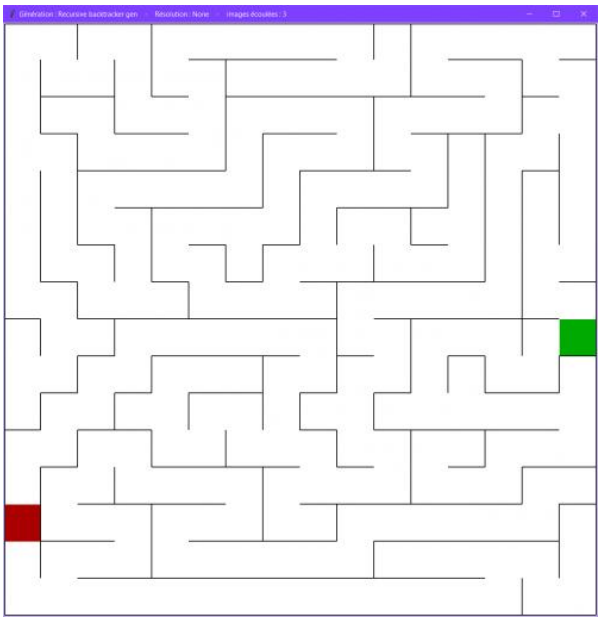
L'algorithme est intéressant car il ne "pousse" pas comme un arbre : il sculpte au hasard des passages dans tout le labyrinthe, cela le rend très amusant à regarder. Malgré tout, il en résulte un labyrinthe parfait à la fin.

Une fois que nous avons tous les passages possibles dans un grand "répertoire" et un identifiant unique associé à chaque cellule, tout ce dont nous avons besoin de faire est de choisir un des passages au hasard, vérifier si les cellules voisines appartiennent à un identifiant (sous-ensemble) différent et les unifier dans le cas échéant.

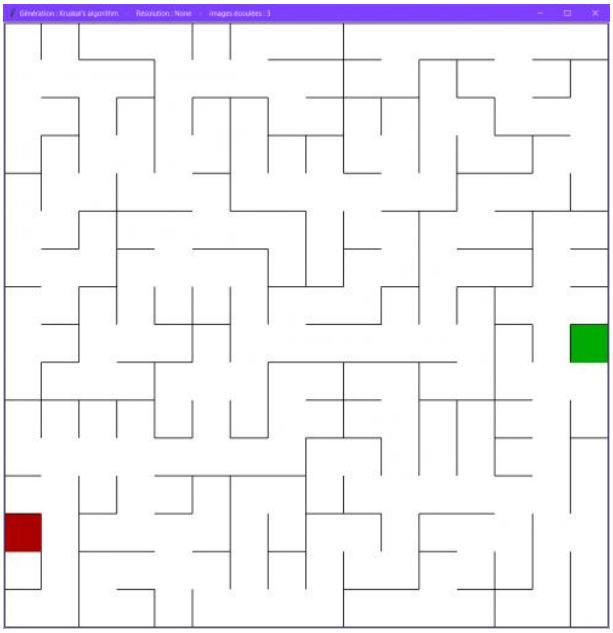
Dans sa version de générateur de labyrinthe, l'algorithme de Kruskal reprend presque points pour points la méthodologie de

l'algorithme. Cependant, lors de l'étape du choix des arêtes, l'algorithme dans sa version originale incite à choisir l'arête avec le poids le plus faible. Le générateur de labyrinthe va lui prendre un passage au hasard à chaque itération.

Exemple de labyrinthe créé avec l'algorithme du parcours en profondeur :



Exemple de labyrinthe créé avec l'algorithme de Kruskal :



On remarque que le premier algorithme marque des chemins proprement dessinés alors que l'algorithme de Kruskal a construit quelques murs d'une longueur de « un ».

L'algorithme du parcours en profondeur donne une image finale d'homogénéité alors que l'algorithme de Kruskal donne vraiment un aspect de chemins « entremêlées » et d'issue sans sortie.

C) PROJET DU LABYRINTHE

1) CONCEPT

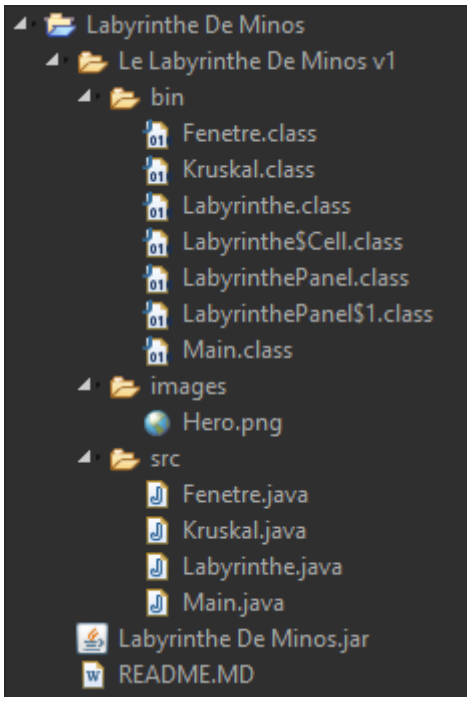
Dans la mythologie grecque, Minos est un roi légendaire de Crète. Selon le mythe, désireux de montrer à son peuple le crédit dont il jouissait auprès des dieux, Minos pria Poséidon de faire surgir de la mer un superbe taureau, lequel lui serait aussitôt sacrifié. Poséidon répondit à cette demande en lui envoyant un magnifique taureau blanc que Minos trouva si beau qu'il décida de tromper le dieu, il épargna le taureau et le plaça parmi son troupeau et immola une autre bête. Courroucé par l'imposture de Minos, Poséidon anima le taureau de fureur et lui fit dévaster les terres de Crète.

En outre, il inspira à Pasiphaé, l'épouse du roi, un amour passionné pour l'animal. Elle sollicita Dédale, l'architecte du roi, de lui construire un objet lui permettant d'accomplir ses desseins. Dédale lui construisit une vache en bois et la reine pu s'accoupler avec le taureau. De cette union naquit le Minotaure, une créature à tête de taureau, et au corps d'homme. Minos, suivant les conseils de certains oracles, confia à Dédale la construction du Labyrinthe, dans lequel il fit enfermer le monstre.

Quelques temps plus tard le héros Thésée, fils du roi d'Athènes, Égée, fut tiré au sort pour faire partie du tribut annuel. Avec l'aide d'une des filles de Minos éprise de lui, Ariane, il parvint à tuer le monstre, puis à sortir du labyrinthe.

L'idée principale du projet est de reprendre le mythe de Thésée et du Minotaure et de l'inclure au sein d'un jeu vidéo. Le joueur devra s'enfuir la plus rapidement du labyrinthe tout en évitant la confrontation avec le minotaure.

2) ARCHITECTURE DU PROJET



Le projet est structuré de la façon suivante, il y a quatre classes, la classe main, la classe fenêtre, la classe labyrinthe et la classe Kruskal. La classe Fenetre définie la taille de la fenêtre de l'application et comment le labyrinthe sera intégré. La classe Labyrinthe elle définie l'objet labyrinthe, pour cela elle a recours à la classe Kruskal. La classe Kruskal elle contient trois méthodes union(), find() et creerEnsemble().

3) LE LABYRINTHE VU DE L'EXTÉRIEUR

Le labyrinthe généré par l'algorithme est un labyrinthe parfait, il n'y a pas d'îlots ni de boucles et toutes les cellules sont accessibles. On remarque également que comme dit précédemment, des murs d'une longueur minimale (« un ») ont été créés, cela montre explicitement des chemins menant à des culs de sac mais renforce dans le même temps l'aspect complexe du labyrinthe.

4) LE LABYRINTHE VU DE L'INTÉRIEUR

1) Classe Fenêtre

C'est une classe qui créé une fenêtre d'application à l'aide d'une JFrame, la fenêtre est définie par son titre, sa taille, un objet est placé eu centre de la zone, on y intègre également un panel qui va contenir un labyrinthe.

```
public class Fenetre extends JFrame{
    Labyrinthe labyrinthe = new Labyrinthe(20); // Construit l'objet labyrinthe
    /**
     * @param args
     * Constructeur de la classe Fenetre
     */
    public Fenetre(){

        // Définit un titre pour notre fenêtre
        setTitle("Le labyrinthe de Minos");
        // Définit sa taille : 1000 pixels de large et 1000 pixels de haut
        setSize(800, 800);
        // Nous demandons maintenant à notre objet de se positionner au centre
        setLocationRelativeTo(null);
        // Termine le processus lorsqu'on clique sur la croix rouge
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        LabyrinthePanel panel = new LabyrinthePanel(labyrinthe); // Construit le panel pour contenir le labyrinthe
        JScrollPane scrollPane = new JScrollPane(panel);
        add(scrollPane, BorderLayout.CENTER);
        // Et enfin, la rendre visible
        setVisible(true);

    }
}
```

2) Classe Labyrinthe

```
public class Labyrinthe {
    public static final int CELL_WIDTH = 30; // format carré du labyrinthe
    public static final int MARGIN = 50; // tampon entre le bord de la fenêtre et le labyrinthe
    public static final int DOT_SIZE = 10; // taille du carré de la solution du labyrinthe
    public static final int DOT_MARGIN = 10; // espace entre les murs et la solution
    private int N;
    private Cell[] cells; // tableau contenant toutes les cellules du labyrinthe

    /*
     * Constructeur de la classe labyrinthe
     */
    public Labyrinthe(int n) {
        N = n;
        cells = new Cell[N * N]; // crée un tableau de cellules

        // Boucle for qui va initialisé le tableau avec les objets Cell
        for(int i = 0; i < N * N; i++) {
            cells[i] = new Cell();
        }

        if(N > 0) {
            faireMurs(); // met à jours les informations sur le mur à l'intérieur de chaque objet Cell
            detruitMurs(); // détruit le mur jusqu'à la formation d'un labyrinthe
        }
    }
```

La classe labyrinthe définit la taille du labyrinthe, son format et son implémentation dans la fenêtre de l'application.

Méthode faireMurs()

```
    private void faireMurs() {
        // définit les murs nord, sud, est et ouest
        for(int i = 0; i < N * N; i++) {
            cells[i].murs[NORD] = i - N;
            cells[i].murs[SUD] = i + N;
            cells[i].murs[EST] = i + 1;
            cells[i].murs[OUEST] = i - 1;
        }

        for(int i = 0; i < N; i++) {
            cells[i].murs[NORD] = -1; // situé dans les cellules nord de la frontière, mur nord à -1
            cells[N * N - i - 1].murs[SUD] = -1; // situé dans les cellules frontalières sud, mur sud à -1
        }

        for(int i = 0; i < N * N; i += N) {
            cells[N * N - i - 1].murs[EST] = -1; // situé dans les cellules est de la frontière, mur est à -1
            cells[i].murs[OUEST] = -1; // situé dans les cellules frontalières ouest, mur ouest à -1
        }
    }

    // Détruit les murs avec une version modifiée de l'algorithme de Kruskal
```

La méthode faireMurs() définie les côtés nord, sud, ouest, et est de chaque cellule. Ce sont ces côtés qui vont donner

naissance à des murs ou à des portes entre les différentes cellules (avec une autre méthode).

Méthode detruitMurs()

```
private void detruitMurs() {
    int NumElements = N * N;

    Kruskal ensembleDisjoint= new Kruskal(NumElements); // crée un ensemble disjoint pour représenter les cellules

    for(int k = 0; k < N * N; k++) {
        ensembleDisjoint.find(k); // ajoute chaque cellule à un seul ensemble
    }

    // Générateur aléatoire
    Random generateur = new Random();

    // Tant que tout les éléments de l'ensemble disjoint ne sont pas connectés
    while(ensembleDisjoint.creeEnsemble() == false) {
        int cell1 = generateur.nextInt(N * N); // Choisis une cellule aléatoire
        int mur = generateur.nextInt(4);

        int cell2 = cells[cell1].murs[mur]; // Choisis une seconde cellules aléatoire

        // S'il existe un mur entre ces deux cellules
        if (cell2 != -1 && cell2 != N * N) {
            // Si les cellules n'appartiennent pas au même ensemble
            if(ensembleDisjoint.find(cell1) != ensembleDisjoint.find(cell2)) {
                cells[cell1].murs[mur] = N * N; // Détruit les murs entre ces deux cellules. N*N ne représentera aucun mur
                cells[N * N - 1].murs[EST] = N * N; // détruit le mur en bas à droite pour crée la sortie

                if(mur == NORD || mur == EST) {
                    cells[cell2].murs[mur + 1] = N * N;
                }

                if(mur == SUD || mur == OUEST) {
                    cells[cell2].murs[mur - 1] = N * N;
                }

                // Fait une union de l'ensemble de ces deux cellules, par laquelle un deplacement vient d'être crée
                ensembleDisjoint.union(ensembleDisjoint.find(cell1), ensembleDisjoint.find(cell2));
            }
        }
    }
}
```

C’est la méthode qui va détruire les murs de chaque cellule avec les cellules adjacentes.

4) IMPLÉMENTATION DE L'ALGORITHME

```java
public class Kruskal {
    public int[] s;
    /*
     * Constructeur de l'objet Kruskal
     * @param g le nombre initial d'ensembles disjoints
     */
    public Kruskal(int g) {
        s = new int [g];
        for (int i = 0; i < s.length; i++) {
            s[i] = -1;
        }
    }
```

La classe Kruskal est définie par l'objet Kruskal avec en paramètre 'g', g représente le nombre initial d'ensembles disjoints.

Union-Find

```java
/*
 * Union de deux ensembles disjoints en utilisant l'heuristique de hauteur.
 * Pour simplifier, nous supposons que racine1 et racine2 sont distinctes et représentent des noms d'ensembles
 * @param racine1 est la racine de l'ensemble 1.
 * @param racine2 est la racine de l'ensemble 2.
 */
public void union(int racine1, int racine2) {
    if(s[racine2] < s[racine1]) { // racine2 est plus profond
        s[racine1] = racine2; // Faire de racine2 une nouvelle racine
    } else {
        if(s[racine1] == s[racine2]) {
            s[racine1]--; // Mettre à jour la hauteur si identique
        }
        s[racine2] = racine1; // Faire de racine1 une nouvelle racine
    }
}
```

La méthode union fait le lien entre deux ensembles disjoints en utilisant l'heuristique de hauteur. Elle repose sur deux paramètres, racine1 et racine2. Celles-ci sont distinctes et représentent des noms d'ensembles.

```java
/*
 * Effectue une recherche avec compression de chemin.
 * Les contrôles ont été omis à nouveau pour plus de simplicité
 * @param x l'élément recherché
 * @return l'ensemble contenant x
 */
public int find(int x) {
    if(s[x] < 0) {
        return x;
    } else {
        return s[x] = find(s[x]);
    }

}
```

La méthode find elle recherche les cellules correspondantes de la méthode union, en temps normal l'algorithme effectue des contrôles afin de ne pas créer de cycle mais nous l'avons enlevée dans une optique de simplicité.

D) CONCLUSION

En conclusion, nous pouvons dire que les labyrinthes sont un sujet très passionnant en théorie des graphes. Les labyrinthes soulèvent de nombreuses questions d'ordre mathématiques. Les labyrinthes peuvent être un sujet d'étude déclinable en une multitude de façons différentes.

On a pu voir au cours des recherches que la création d'un labyrinthe va surtout dépendre du choix de l'algorithme, certains comme l'algorithme A* qui permet également de trouver le chemin le plus court vers la sortie mais tous algorithmes possèdent des inconvénients. Par exemple, A* ne donnera pas toujours le chemin optimal vers la sortie. L'algorithme du parcours en profondeur permet aussi de générer un labyrinthe, nous avons trouvé que Kruskal était plus adéquat au projet car il créé un labyrinthe qui apporte vraiment un « aspect complexe ».

Que ce soit la création de labyrinthe, le choix de chemins possibles ou encore la recherche du chemin de sortie. Les labyrinthes semblent être des terrains favorables pour l'utilisation des algorithmes.

De possibles axes d'améliorations pour le projet serait d'intégrer plusieurs joueurs dans le labyrinthe, ainsi le jeu prendrait l'aspect d'un « survival game » en multi-joueurs. Une autre amélioration possible serait d'indiquer lorsque le joueur passe par le chemin parfait. L'idée serait que la cellule où est passé le joueur devienne blanche, ainsi il sait qu'il est sur la bonne voie. On pourrait appeler cela « le chemin d'Arianne ».

E) SOURCES/ANNEXES

1) RESSOURCES INFORMATIONNELLES

- https://fr.wikipedia.org/wiki/Algorithme_de_Kruskal
- https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe
- https://fr.qwe.wiki/wiki/Kruskal%27s_algorithm
- https://www.lama.univ-savoie.fr/mediawiki/index.php/G%C3%A9n%C3%A9ration_et_r%C3%A9solution_de_labyrinthes_II

2) RESSOURCES VISUELLES

- https://www.fontspace.com/gelio-font-f11521

3) OUTILS/LOGICIELS

- Java