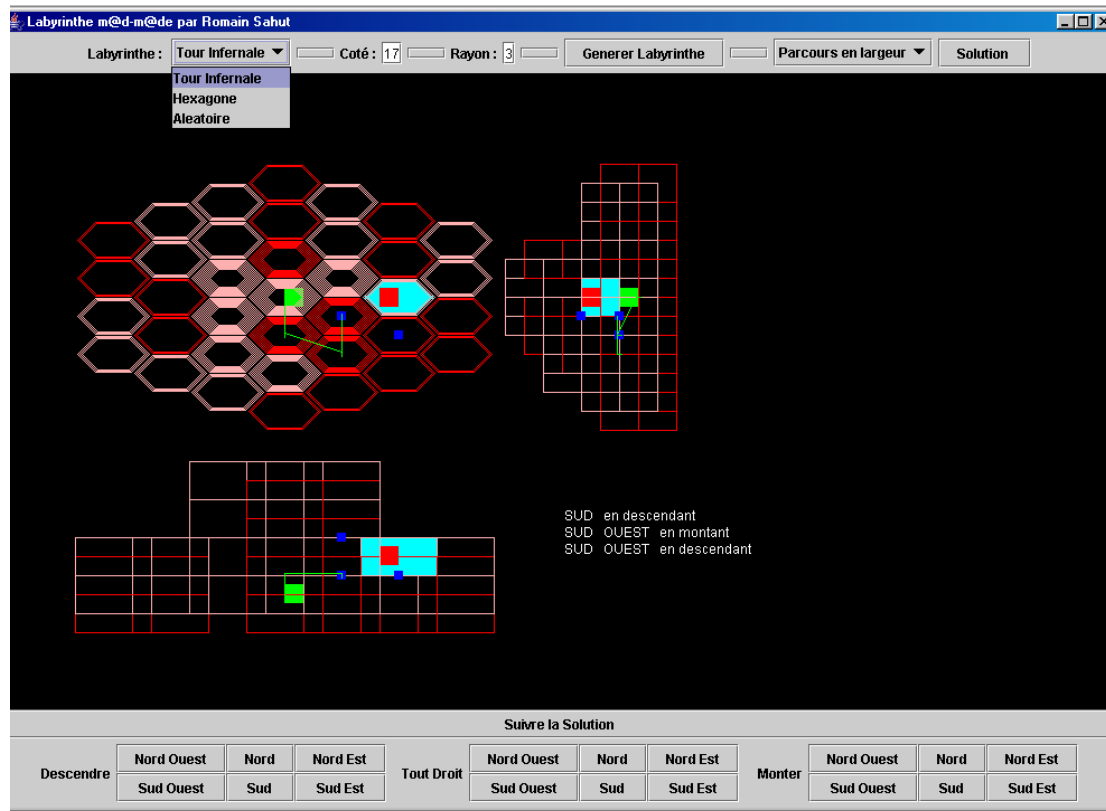


Utilisation des Graphes en Java :

Labyrinthe 3D



Sommaire

Sommaire	1
Introduction.....	2
Objectifs	3
Structure	4
Choix d'implémentation	4
Pièce	4
Empilement	4
Graphe.....	4
Explications du code source	5
Parcours en largeur	5
Explication de la structure hexagonale	6
Initialisation	7
Placement des pièces de départ et d'arrivée	7
Conclusion	9
Annexe : Dictionnaire des méthodes	10
Graphe.....	10
Empilement.....	11
Piece	12

Introduction

L'objectif du projet de LO42 est de programmer un logiciel permettant de résoudre différents problèmes classiques sur les graphes. Le sujet est ici de programmer un labyrinthe à base hexagonale en trois dimensions.

Le but de ce projet est de nous faire utiliser les connaissances acquises en cours de LO42 sur les graphes et leur implémentation, ainsi que les algorithmes les concernant.

Ce rapport a pour but d'expliquer les fonctions importantes utilisées dans ce projet, ainsi que son implémentation.

Ce logiciel sera programme en Java.

Objectifs

Le labyrinthe que nous voulons représenter est une structure en trois dimensions. C'est une construction constituée d'empilements de salles hexagonales. Chaque empilement peut être décalé par rapport à ces voisins d'un demi niveau.

Chaque salle peut être ou non en communication avec une de ces voisines. C'est-à-dire qu'une salle peut avoir de 0 à 12 issues (Il n'y a pas de communication par le plafond ou le plancher). Un passage d'une salle à l'autre peut se faire au même niveau, en descendant ou en montant un demi niveau.

Ce projet est la suite logique des séances de TP suivies au cours du semestre, par conséquent il utilise les structures de données étudiées.

L'utilisateur doit avoir la possibilité de visualiser le chemin optimal vers la pièce finale. Au démarrage (ou après avoir généré un nouveau labyrinthe) on affiche la solution optimale du labyrinthe sur la sortie standard.

Structure

Choix d'implémentation

La représentation structure chaînée

La représentation en liste chaînée est sans aucun doute la plus simple car elle ne nécessite pas de redimensionnement de tableaux. J'ai choisi d'utiliser le type LinkedList existant déjà en Java car il semblait adapté à l'utilisation que je lui destinais.

Les graphes non orientés

L'implémentation des graphes non orientés à partir des graphes orientés est une opération simple. Il suffit lorsque l'on ajoute un arc dans un graphe non orienté d'ajouter deux arcs dans le graphe orienté, et de faire de même pour la suppression.

Donc lors de la liaison entre deux Empilements ou deux Pièces, on les ajoute mutuellement dans la liste chaînée des adjacents. Cela permet de passer d'une pièce à l'autre, puis de revenir en arrière.

Pièce

La classe Pièce représente une salle du labyrinthe. Une pièce est liée à d'autres pièces et l'ensemble des liaisons des pièces entre elles constitue le labyrinthe.

Empilement

La classe Empilement représente un empilement d'objets Piece. Cette classe permet de générer récursivement d'autres empilements. Elle respecte les contraintes du sujet, c'est-à-dire que les empilements sont naturellement juxtaposés. Elle comporte une liste chaînée d'empilements adjacents de type LinkedList.

Graphe

La classe Graphe permet de conserver les références des deux graphes (empilement pour la structure, et Pièces pour le labyrinthe). Elle stocke le chemin pris et le chemin optimal, les pièces courante, de départ et d'arrivée, l'empilement originel...

Explications du code source

Parcours en largeur

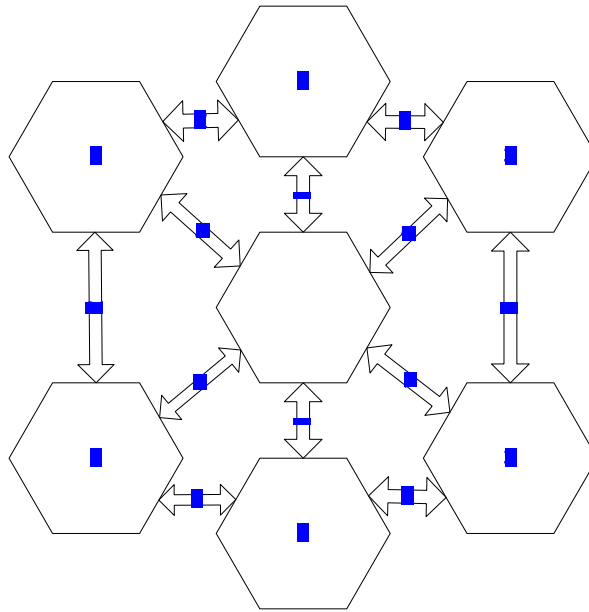
Un parcours du graphe G à partir de s est dit en largeur si, à chaque étape, l'arête de liaison x, y choisie est telle que le sommet x soit le premier sommet visite ouvert.

Une implémentation possible de cet algorithme consiste à utiliser une file de sommets comme structure de stockage des sommets. On obtient ainsi l'algorithme qui suit.

```
Procédure LARGEUR(Graphe g, Sommet s)
  var : file : file
  N : Entier correspond au nombre de sommets
  marque [1..N] tableau de Booleen

  Debut
    Initialisation de marque [1..n] = faux
    Enfiler (s) dans file
    marque[s] <- vrai
    tant que (non file_vide)
      s <- Defiler() de file
      pour tout successeurs k de s dans G faire
        si (marque[k] = faux) alors
          marque[k] <- vrai
          Enfiler (k) dans file
        finsi
      finpour
    fintantque
  Fin
```

Explication de la structure hexagonale



```

generer(int profondeur,Empilement origine,int x,int y)
{
    //Génère en cascade des empilements
    Empilement tmp;
    int a=x,b=y;
    Si (profondeur>0)
1 :      Pour j de 0 a 6 //Les adjacents de l'empilement
        Si (origine.getAdjacent(j)=null) Alors//S'il n'existe pas
            tmp=new Empilement(c,type,profondeur,x,y);
            origine.setAdjacent(j,tmp);
            tmp.setAdjacent((j+3)%6,origine);
        FinSi
        FinPour
2 :      Pour j de 0 a 6//MaJ des arêtes latérales communes aux nouveaux
empilements
        //(j+1)%6 c'est l'arête suivante dans le sens horaire
        //(j+5)%6 c'est l'arête précédente dans le sens horaire
        Empilement e=origine.getAdjacent(j);
        origine.getAdjacent((j+1)%6).setAdjacent((j+5)%6,e);
        e= origine.getAdjacent((j+1)%6);
        origine.getAdjacent(j).setAdjacent((j+2)%6,e);
        FinPour
3 :      Pour j de 0 a 6
empilements fils
        generer(profondeur-1,tmp); //Appels récursifs aux
        FinPour
    FinSi
}

```

Initialisation

```
graphe.newOriginel(250,200,3); //Génère le premier empilement à une
position donnée
graphe.generer(3,graphe.getOriginel(),250,200); //Génère récursivement les
empilement autour de l'originel
graphe.MaJPieces(); //Choisi les points de départ et d'arrivée du
labyrinthe
while(!graphe.listePieces(graphe.getDepart()).contains(graphe.getFin()))
//On refait tant qu'il n'y a pas de solution
{
    graphe.generer(3,graphe.getOriginel(),250,200);
    graphe.MaJPieces();
}
graphe.trouverChemin(); //On va écrire le plan du parcours
graphe.afficheCheminOptimalTxt();
graphe.setChemin(new LinkedList());
```

Placement des pièces de départ et d'arrivée

Le principe est de choisir le couple de pièce dont le chemin optimal est le plus grand (=le plus dur).

```
int max=1;

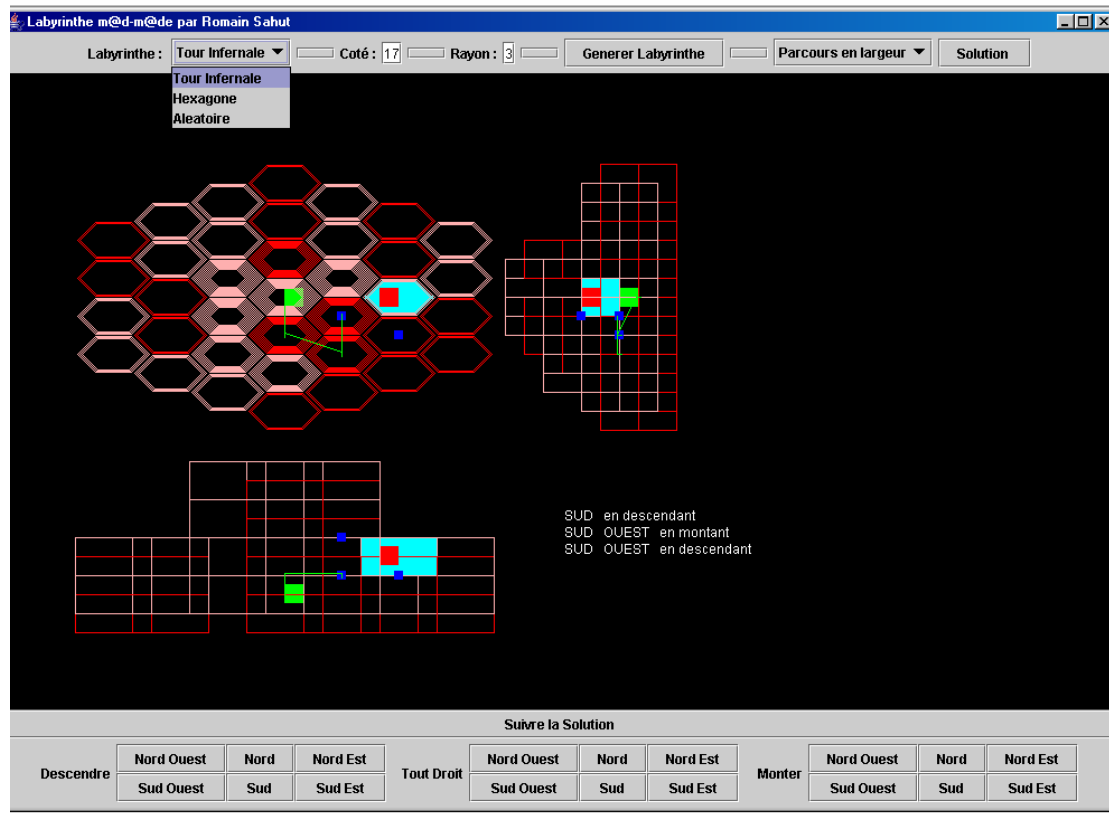
originel.genererLab(); //génère les chemins du labyrinthe

Piece pa=placementAleatoire();
Piece pb=(Piece) originel.getPiece().getFirst();
pieceFin=pa;
pieceCourante=pb;

l=listePieces(pa);
while(!(l.isEmpty())) //Pour chaque piece
{
    pa=(Piece) l.removeFirst();
    lj=l;
    while(!lj.isEmpty()) //Pour chaque piece
    {
        pb=((Piece) lj.removeFirst());
        trouverChemin(pa,pb);
        if(cheminOptimal.size(>max)
        {
            pieceDepart=pa;
            pieceFin=pb;
            max=cheminOptimal.size();
        }
    }
}

pieceCourante=pieceDepart;
```

Jeu d'essai



Sortie standard : la solution du labyrinthe

Cap au SUD OUEST en descendant
Cap au SUD au même niveau
Cap au NORD OUEST au même niveau
Cap au NORD en descendant

Conclusion

Le projet de LO42 nous a permis d'appliquer des connaissances acquises en cours en ce qui concerne la représentation des graphes. D'autre part, tous les algorithmes à implémenter n'ayant pas été vu en cours, ce projet nous a permis de nous familiariser avec la compréhension des algorithmes proposés dans les ouvrages spécialisés.

Par ailleurs, ce projet nous a permis de nous initier au langage Java et de comprendre le fonctionnement général des langages orientés objet.

Annexe : Dictionnaire des méthodes

Graphe

```
public class GrapheEmpilement
{

private Empilement originel;

private Graphics g=null;

private Piece pieceDepart;
private Piece pieceFin;
private Piece pieceCourante;

private LinkedList cheminPris;
private LinkedList cheminOptimal;

private int c=0; //Taille en pixels d'un coté
private int type; //Type du labyrinthe - 0:Tour, 1:Hexagone, 2:Aleatoire

public GrapheEmpilement(int type,int profondeur)
public void setType(int type) //Fixe le type du labyrinthe à créer
public void setChemin(LinkedList li)
public void setCote(int cote) //Fixe la taille d'un coté
public void allerEn(Piece p) //passer de la pièce courante à une des pièces
adjacentes
public LinkedList getCheminOptimal()//Retourne le chemin optimal
public void trouverChemin()//fonction qui lance les fonctions qui trouvent
un chemin court
public LinkedList parcours_largeur()//Retourne la liste afin d'avoir le
parcours depuis le point courant au point de fin
public int trouver_num_Piece(Piece s) //Retourne le numéro d'une pièce
qu'on cherche
public Piece trouver_Piece(int i) //Retourne la ieme pièce.
public Piece placementAleatoire()//retourne une pièce au hasard, mais
située en hauteur (car le point d'arrivée est tjs le point le plus bas)
public LinkedList listePieces(Piece p) //Idem fonction suivante mais plus
facilement appelable
public LinkedList listePieces(Piece p,LinkedList liste) //Retourne une
liste constituée de toutes les pièces du labyrinthe accessible depuis la
pièce p
public Piece getDepart()//Retourne la pièce de départ
public Piece getCourante()//Retourne la pièce courante
public Piece getFin()//Retourne la pièce de fin
public void newOriginel(int x, int y,int profondeur) //On change tout et on
recommence
public void MaJPieces()//Définit les pièces de départ et d'arrivée
public void generer(int profondeur,Empilement origine,int x,int y) //Génère
en cascade des empilements
public Empilement getOriginel()//Retourne l'empilement central, celui dont
tout part!
public void affiche()//Affiche les élément du labyrinthe
public void afficheCheminPris//Affiche les 10 dernières pièces parcourues
public void afficheCheminOptimal()//Affiche le chemin optimal calculé
public void afficheCheminOptimalTxt()//Affiche le chemin optimal calculé en
mode textuel (carte solution du labyrinthe)
public void setGraphics(Graphics g)
}
```

Empilement

```
public class Empilement
{

private int profondeur;
private int decalage;

private boolean marque;

private LinkedList piece;
private LinkedList adjacents;

private Graphics g;

private int c=0;

private int x,y;

public Empilement(int cote,int type,int profondeur,int x,int y)
public void setXY(int x,int y)
public Empilement getAdjacent(int i)
public LinkedList getPiece()
public void setAdjacent(int i, Empilement e)
public void setGraphics(Graphics g)
public void affiche(Piece pieceDepart,Piece pieceCourante,Piece pieceFin)
public void deMarquer()
public void setCote(int cote)
public void genererLiaisons(Empilement a,Empilement b)
public void genererLab()
public void genererLabyrinthe()
public void afficheGfx(Piece pieceDepart, Piece pieceCourante, Piece
pieceFin)

}
```

Piece

```
public class Piece
{

private int hauteur;

private LinkedList ArcPiece; //Une liste chaînée d'arcs

private int c;
private Graphics g; //Le panneau sur lequel on dessine

private int x,y,z;

public Piece(int hauteur,int x, int y,int c)
public boolean existeArc(Piece a) //Test retournant l'existant c'un arc
entre la piece courante et celle passée en argument
public void ajouterArcAleatoire(Piece a) //Ajoute des arcs de manière
aleatoire autour d'une piece et faisant attention à ce qu'ils n'existent
pas deja
public void setGraphics(Graphics g) //Dit à la Piece sur quel objet
Graphics elle doit afficher ses infos
public void ajouterArc(Piece a,Piece b) //Ajoute deux arc (on travaille en
graphe non orienté)
public void ajouterArc(Piece a) //Ajoute un arc entre la pièce courante et
la piece en argument
public int getHauteur()//Retourne la hauteur de la pièce
public LinkedList getArcPiece()//Retourne une linkedList de Pièces vers
lesquelles pointe cette Piece
public void afficheDeplacementPossible()//Affiche les déplacements
possibles depuis une piece donnée
public void afficheBingo(Piece p,int nbcoups) //Affiche la bonne arrivée !
public void dessineAraignee()//Dessine les petits carres bleus de
l'araignee
public void afficheAraignee()//Affiche des petits carres bleus sur les
Pieces accessibles directement
public Piece suivre(int i) //Suit le ieme arc de la pièce
public void afficheCourante()//Dessine la pièce courante
public void affiche(Color couleur) //Dessine une Pièce quelconque avec une
couleur (dependant du decallage)
public void afficheDepart()//Dessine la pièce de départ un carré rouge
public void afficheFin()//Dessine la pièce de fin un carré vert
public void setXY(int x,int y) //MaJ coords X et Y de la Pièce
public int getZ()//Accesneur pour la hauteur du centre de la pièce
public int getX()//Accesneur pour l'affixe du centre de la pièce
public int getY()//Accesneur pour l'ordonnée du centre de la pièce
public void dessinerHexagone()//Dessine un hexagone classique
public void dessinerHexagone(int k) //Dessine un hexagone avec un trait
epais
public void remplirHexagone()//Rempli un hexagone de coté c

}
}
```