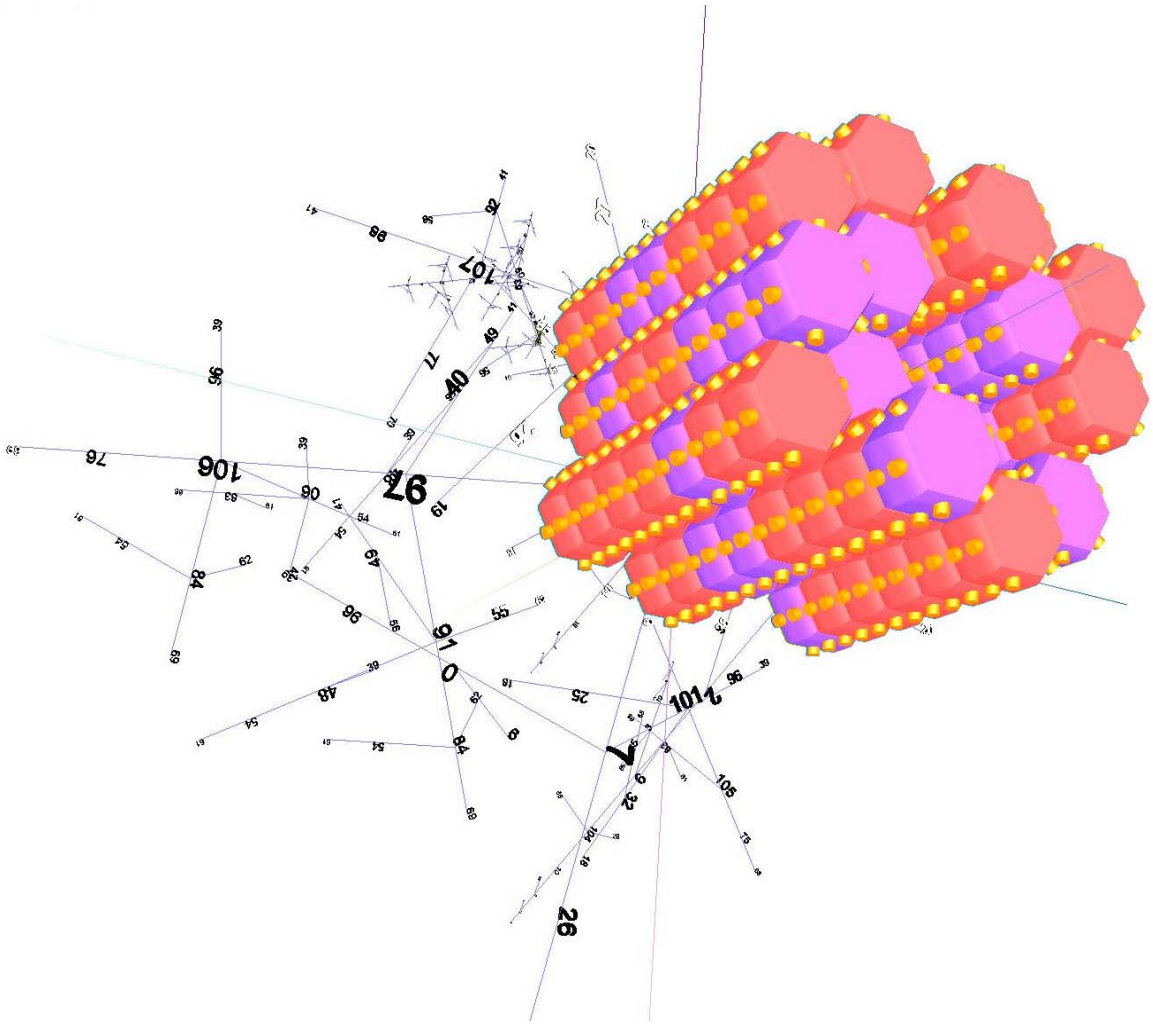
Projet Labyrinthe



Sommaire

Introduction

Le but ce projet de *LO42* était de manipuler les graphes et les arbres de manière à créer et parcourir un *labyrinthe 3D* composé de salles hexagonales. Nous avons réalisé une application en JAVA pour traiter le problème qui nous a été soumis. Les salles sont disposées en empilements afin de faciliter la construction et de garantir la cohérence géographique de l'ensemble. Les salles adjacentes peuvent être ou non reliées entre elles ce qui crée des chemins dans le labyrinthe et peut engendrer des salles secrètes. La complexité de ce *labyrinthe* est d'autant plus importante que les salles ne sont pas forcément au même niveau, ce qui fait entre 0 et 12 portes possibles par salle.

Notre démarche a été tout au long de ce projet de créer un code à la fois efficace et structuré pour en assurer une certaine évolutivité et *généricité* du code.

Dans une première partie, nous allons présenter les principales structures de données mises en œuvre puis nous expliquerons les algorithmes majeurs que nous avons implémentés pour assurer l'efficacité du programme et résoudre les problèmes posés et enfin nous présenterons rapidement l'interface utilisateur mise en oeuvre.

1. Génération du labyrinthe

1.1 Les empilements

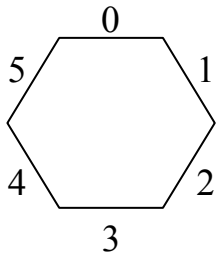
La génération des salles et de leurs liens se fait en deux étapes, on commence par générer un graphe d'empilements qui permettent ensuite de créer les salles de manière cohérente géographiquement dans l'espace. Ces empilements peuvent êtres décalés les uns par rapport aux autres d'1/2 niveau en hauteur, cela étant généré de manière aléatoire pour chaque empilement. Les dimensions du graphe sont déterminées lors de l'initialisation de la construction. Toutes nos constructions utilisent une architecture objet, chaque élément étant représentés par une classe contenant les méthodes pour construire les ensembles auxquels ils appartiennent. Un empilement possède un tableau de NB_LIENS références (6 dans notre cas) vers ses empilements voisins. La création d'un empilement provoque récursivement la création du graphe autour de ce dernier.

```
public class Empilement {
	//Constructeur de départ
	Empilement(int t[], int haut)
	//Construction récursive
	Empilement(Empilement pere, int dirPere, int haut)

	//renvoi le numero de lien correspondant dans un empilement lié
	private int reciprok(int dir)
	//Renvoi l'indice du lien précédent correspondant dans empilement lié au lien dir
	private int indiceDansPerePrec(int dir)
	//Renvoi l'indice du lien suivant correspondant dans empilement lié au lien dir
	private int indiceDansPereSuiv(int dir)

	//Mise a jour des coordonnées de l'empilement
	private void majCoord(Empilement pere, int dirPere)
	//utilise pour calculer les coordonnées d'un nouvel empilement
	private int[] calcCoord(int co[], int dir)

	//Détecte si un empilement sort de la zone à construire
	boolean dehors(int coord[])
	boolean dehors()
	//Construction de l'empilement
	void build()
	//Pour créer un lien réciproque entre empilements
	Empilement creeLien(Empilement fils, int dirFils)
	//indice lien precedent
	int prec(int i)
	//indice lien suivant
	int suiv(int i)
	//initialise la construction des voisins restants à construire de l'empilement
	void creeFils(int dirPere)
	//Retourne l'empilement lié selon direction
	Empilement getLien(int dir)
	//Coordonnees empilement
	int[] getCoord()
	//Creation des salles de l'empilement
	void creeSalles()
	//Cree les liens entre les salles de l'empilement et ses voisines
	void buildSallesLiens()
	//Initialise la creation du graphe de salles
	void buildSallesGraph()
	//Renvoi la salle à la hauteur haut dans l'empilement
	Salle getSalle(int haut)
	//Renvoi la salle dans un empilement voisin selon une direction et une hauteur
	Salle getSalle(int dir, int haut)
}
```

1.2 Génération du graphe d'empilements

La génération de ce graphe est réalisé a partir d'une vision totalement locale du graphe. Chaque empilement ne connaissant simplement que les empilements situés au bout des liens qu'il possède.

Les liens étant numérotés de la façon suivante :



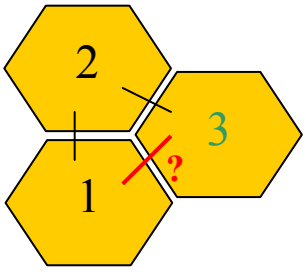
On appellera réciproque d'un lien (implémenté par la méthode *reciprok* de la classe *Empilement*) le lien situé sur sa face opposée, par exemple le réciproque de 5 est 2.

Le nœud premier est créé par la méthode *Build* de la classe Labyrinthe, classe de départ de l'application. Le constructeur prenant en paramètre un tableau de *int* représentant la taille maximum du labyrinthe et un *int* représentant la hauteur du nœud premier est alors utilisé pour paramétrer le Labyrinthe.

La méthode *build* de l'empilement est ensuite appelée ce qui provoque l'appel à la méthode *creeFils* et initie la création récursive du graphe d'empilements. Cette méthode fonctionne de la manière suivante :

Elle commence par détecter le sens de parcours de ses liens, il faut en effet non seulement créer de nouveaux empilements mais également les liens vers les empilements existant car créés par d'autres empilements. Il est fondamental que tous les liens d'un empilement vers les empilements générés précédemment soient créés avant d'en générer de nouveaux. En effet, pour savoir si l'empilement situé au bout d'un lien doit être créé ou si il existe déjà, l'empilement se réfère au nœud situé au bout d'un lien connaissant celui à générer.

Par exemple dans le cas suivant :



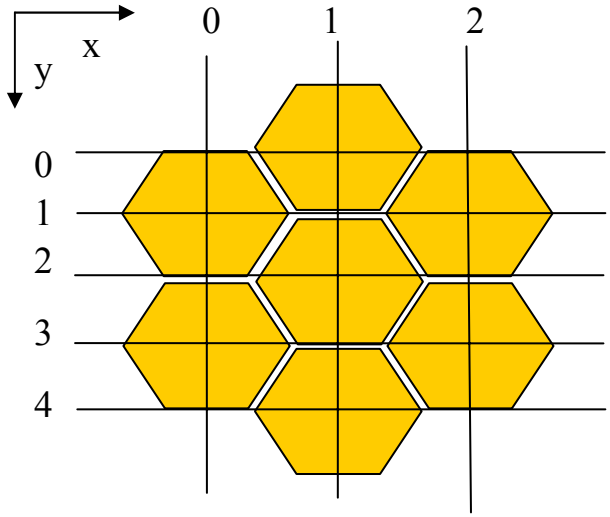
Pour créer le liens 4 (cf. schéma précédent) dans l'empilement 3, il faut aller voir le liens 3 dans l'empilement 2. Si il existe, c'est qu'il faut simplement le relié, sinon on le crée. On test donc le liens précédent et suivant le liens courant afin de savoir quoi faire. Pour déterminer le lien à tester dans la salle 2 on utilisera la méthode *indiceDansPereX* de la classe Empilement.

1.3 Les Salles

La génération du graphe de salles se fait à travers la classe *Salle* qui fonctionne sur le même principe que la classe *Empilement* avec cette fois 2*NB_LIENS références sur des salles voisines numérotées comme pour les empilement, l'étage inférieur succédant à l'étage supérieur. Les Salles sont stockées dans des *ArrayList* Java à l'intérieur de leurs empilements respectifs et ce sont ces derniers qui initient la création du graphe de Salles. Les salles déterminent aléatoirement à leur création si il s'agit d'une entrée. La création des liens entre salles se fait à travers les empilements ce qui est efficace d'un point de vue complexité et garanti la cohérence géographique de l'ensemble.

```
public class Salle {
	//Nb de salles total
	static int nbSalles()
////pour DJSka////
	//Renvoi le nœud de l'arbre associé lors du parcours
	Noeud getNoeud()
	//Associe un nœud de l'arbre
	void setNoeud(Noeud n)
	//Indique si un lien existe avec la salle s
	boolean lienVers(Salle s)
/////////////////
	//constructeurs
	Salle(Empilement e, int h, int ent)
	//Même principe que dans Empilement + gestion du niveau
	private int reciprok(int dir, boolean croise)
	//Construction des liens et portes
	void build()
	//cree un lien vers une salle existante
	Salle creeLien(Salle fils, int dirFils, int ouverture)

	//Cf Empilement
	Salle getLien(int dir)
	int[] getCoord()
	//Porte ouverte ou non
	int getPorte(int dir)
	//Ouvre/ferme une porte
	int setPorte(int dir, int val)
}
```

Les salles possèdent également une cordonnée spatiale pour pouvoir en réaliser l'affichage, ces coordonnés fonctionnes selon le principe suivant :



Notre démarche avait été dans un premier temps de vouloir générer directement le graphe de Salles sans passer par des Empilements en utilisant le principe des liens relatifs que nous avons employé pour les Empilements. Seulement il c'est avéré que la construction en une seule passe des liens et des salles en 3 dimensions était impossible à cause des décalages qui font perdes la connaissance de nombreuses salles. De nombreuses Salles se trouvaient ainsi dupliqués car crées plusieurs fois, les liens vers des Salles existantes n'ayant pas été créés au bon moment dans les salles pères. Une connaissance cohérente simplement locale des salles créées semble donc impossible à maintenir avec ce principe. La méthode *créFils* des *Empilements* a tout de même héritée de cette recherche et son architecture permettrait facilement de générer un graphe 3D mais sans décalages.

Les empilements ne sont donc plus qu'un moyen de généré le graphe de Salles et ne sont plus utiles par la suite, chaque salle possédant une référence directe vers chacun de ses voisins. Pour tenir compte du fait qu'une porte vers une salle adjacente puisse être ouverte ou fermée, nous avons également utilisé un tableau de 2*NB_LIENS entiers placés à 0, 1 ou de signifiant respectivement, porte pour le moment indéfinie, porte ouverte, et porte fermée.

1.4 Génération du graphe de Salles

Cette génération se fait comme nous l'avons précisé précédemment en créant les 12 liens de chaque à l'aide des empilements. Pour cela, les empilements fournissent une méthode *getSalle* qui renvoie une référence vers une salle située dans un empilement voisin à partir d'une direction et d'une hauteur à l'intérieur de lui-même. Nous appelons ici direction un nombre entre 0 et 2*NB_LIENS-1 afin de tenir compte de la hauteur du lien.

```
//Renvoi Salle dans l'empilement courant
Salle getSalle(int haut){
	//Si la sale n'existe pas
	if(haut>=hauteur || haut<0)
		return null;
	return (Salle)sallesEmp.get(haut);
}
//Renvoi salle dans un empilement adjacent
Salle getSalle(int dir, int haut){
	//Si il n'y a pas d'empilement dans cette direction
	if(liens[dir%NBLIEN]==null)
		return null;
	//Si les deux empilements ont le même décalage
	if(decal==liens[dir%NBLIEN].decal)
		return liens[dir%NBLIEN].getSalle(haut);
	//Si le decalage est différent
	//Si c'est un liens haut et que que l'on est décalé
	if(dir<NBLIEN && decal)
	//On renvoi la salle de l'empilement adjacent à la hauteur haut+1
		return liens[dir%NBLIEN].getSalle(haut+1);
	//Si c'est un lien bas et que l'on est pas décalé
	if(dir>=NBLIEN && !decal)
	//On renvoi la salle de l'empilement adjacent à la hauteur haut-1
		return liens[dir%NBLIEN].getSalle(haut-1);

	//Autrement on renvoi la salle à la meme hauteur
	return liens[dir%NBLIEN].getSalle(haut);
}
```

2. Promenade au sein du labyrinthe

2.1 L'algorithme de Dijkstra

A partir d'une entrée donnée nous devions établir une carte sous la forme d'une arborescence donnant le chemin le plus court pour accéder à chaque salle.

Pour construire ce graphe des plus courts chemins, notre première idée fut de parcourir le graphe du labyrinthe en profondeur c'est à dire visiter systématiquement tous les nœuds fils de chaque nœud par un parcours récursif. Cette idée nous est rapidement apparue être extrêmement coûteuse car chaque nœud est traité un grand nombre de fois, ce qui entraîne une complexité très élevée ($O(n^x)$ ou approchant). On procède en effet par raffinement successif du chemin le plus court à chaque salle. Nous nous sommes donc mis à la recherche d'un algorithme plus performant permettant de trouver le chemin le plus court à chaque nœud avec une complexité acceptable. Nous sommes ainsi partis de l'algorithme de *Dijkstra* qui donne la longueur des chemins minimaux pour générer un arbre de parcours de manière efficace en l'optimisant et l'adaptant à notre problème.

Voici l'algorithme original de *Dijkstra* :

```
E : ensemble de tous les sommets du graphe
V : ensemble des sommets ayant déjà été visités
L :tableau d'indice les successeurs de x, et dont les valeurs
représentent le poids des arcs (x, successeur)
Pi,j : poids de l'arc (i,j)

V ←{1};
L(1) ← 0;
Pour i de 2 à N
     Si i est successeur de 1 alors
           L(i) ← P1,j sinon L(i) ← +infini
Fin pour
Tant que V ≠ E
     Trouver j pas dans V tel que L(j)=min{L(i),i pas dans V}
     V ← V union {j}
     Pour tout successeur i de j
         Si i pas dans V alors
              L(i) ← min{L(i), L(j)+Pj,i}
         Fin si
     Fin pour
Fin tant que
```

Le principe général de l'algorithme est de parcourir le graphe par niveau de profondeur et non plus en profondeur. Il s'agit de ce que l'on appel un algorithme "glouton".

Dans cet algorithme, on part du sommet 1 : initialisation de l'ensemble V à l'élément 1 et de l'élément L(1) à 0.

Ensuite pour tous les sommets qui restent, on regarde si le sommet est un successeur de 1. Si c'est le cas on met le poids de l'arc (1, successeur) à la position L [successeur], sinon on met +infini pour signaler que cet élément du tableau n'a pas encore reçu de valeur (sommet pas encore visité).

Une fois cette étape terminée, on relance une seconde boucle qui s'arrête lorsque tous les sommets ont été traités.

On récupère alors le sommet S qui n'a pas encore été visité (le sommet n'appartient pas à V) et dont le poids est le plus petit dans le tableau des poids L.

On rajoute alors ce sommet à l'ensemble des sommets déjà visités.

Ensuite, on regarde les successeurs de ce sommet S : s'ils n'ont pas encore été visités et si la valeur de l'arc entre le sommet S et son successeur est inférieure à la valeur présente à la position L[successeur], on remplace la valeur de L[successeur] par L(S) + poids entre les sommets S et successeur. On continue ainsi pour tous les successeurs de S, puis on cherche à nouveau le plus petit poids du tableau L.

Grâce à cette méthode, on trouve le plus court chemin entre un sommet de départ et tous les autres sommets. En effet, on stocke dans un tableau le poids ou longueur du chemin à partir du sommet de départ, poids qui change si le passage par un sommet intermédiaire permet de diminuer sa valeur (chemin plus court).

2.2 L'algorithme de *DJSka*

Pour optimiser notre recherche des plus courts chemins dans notre graphe, nous avons choisi d'implémenter l'algorithme de *Dijkstra* de la manière suivante :

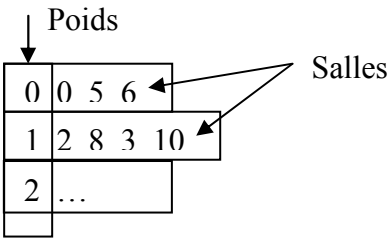
Nous avons tout d'abord mis en place des structures de données efficaces pour stocker les nœuds traités ainsi que les successeurs. Pour cela, nous avons utilisé des tableaux associant à chaque niveau (distance par rapport à l'entrée) une liste de salles. Ceci permet d'accéder rapidement à toutes les salles du niveau qui nous intéresse. Concrètement, nous avons deux ArrayList JAVA stockant des ArrayList de salles pour chaque niveau.

```
ArrayList sallesParcourues =new ArrayList();
ArrayList sallesPoids =new ArrayList();
```

Avec lors de l'ajout d'une Salle:

```
void addSalleArray(ArrayList al, Salle s, int decal){
        if(al.size()<(s.poids+1 -decal) )
              al.add(new ArrayList());
        ((ArrayList)al.get(s.poids-decal)).add(s);
}
```

Schématiquement:



Nous stockons également le poids attribué a chaque salle directement dans les objets associés pou y accéder directement. Tout ceci donne le code *Java* suivant dans la classe DJSka:

```
int build(Salle entree, int nbSalles, ArrayList sallesSecretes){
    entree.poids=0;

    //ajout de l'entree dans le tableau des poids de salles (L)
    addSalleArray(sallesPoids, entree, decalIndicesPoids);

    Salle current;  //Salle courante
    int poidsPere;  //Poids de la salle traitée à une étape de l'algo

    //Tq on a des salles liées, cad placees ds la tableau de successeurs
    while(sallesPoids.size()>0){
      //recherche du min poids dans sallesPoids
      current=(Salle)((ArrayList)sallesPoids.get(0)).get(0);
      poidsPere=current.poids;

      //recherche des successeurs de l'entree
      for(int i=0; i<Salle.NBLIEN*2; i++){
        //Si le lien existe
        if(current.portes[i]==1 && current.liens[i]!=null && !current.liens[i].parcouru)
          //et si le chemin courant est plus court
          if(current.liens[i].poids>poidsPere+1 || current.liens[i].poids==-1)
            chgPoids(current.liens[i],poidsPere+1 )    ; //on met a jour le poids
      }

      //enlever la salle parcourue de sallePoids
      rmSalleArray(sallesPoids);

      //insertion dans sallesParcourues de ce min
      addSalleArray(sallesParcourues, current, 0);
      nbSallesParcourues++;       //Incremente le nb de salles parcourues
      current.parcouru=true;            //On marque la salle comme parcourue
      sallesSecretes.remove(current);   //Et on la retire de la liste des salles secretes

      //ajout de l'entree dans l'arbre
      if(arbre==null) //Si arbre vide
        arbre=new Noeud(current);
      else{           //Creation du nouveau noeud ds l'arbre et liaison avec ses peres
        Noeud nouvNoeud=new Noeud(current);
        //On recherche au poids precedent des salles liées a la salle courante
        for(int j=0; j<((ArrayList)sallesParcourues.get(poidsPere-1)).size(); j++){
          //Puis on ajoute un fils a son noeud dans l'arbre
          Salle tmp=((Salle)((ArrayList)sallesParcourues.get(poidsPere-1)).get(j));
          if( current.lienVers( tmp ) ){
            tmp.getNoeud().addFils(nouvNoeud);
          }
        }
      }
    }

    //Clean up :) Permet de tt nettoyer pour le prochain parcours
    for(int j=0; j<Salle.sallesLaby.size(); j++){
      ((Salle)Salle.sallesLaby.get(j)).parcouru=false;
      ((Salle)Salle.sallesLaby.get(j)).poids=-1;
    }

    //Renvoi la profondeur de l'arbre
    return sallesParcourues.size()-1;
}
```

Cet algorithme se révèle d'une efficacité assez importante, sa complexité théorique étant quasiment en O(m*log(n)) avec m arcs et n sommets.

3. Recherche de l'entrée optimale

Il nous a été demandé de trouver en plus du chemin le plus court à chaque salle à partir de chaque entrée, un algorithme capable de déterminer l'"entrée optimale". Pour nous, une entrée optimale serait une entrée qui permet d'accéder à un maximum de salles le plus rapidement possible, c'est à dire, dont l'arbre de parcours serait le plus dense et le moins profond.

Nous avons tout d'abord répondu à cette demande en construisant l'arbre de parcours de chaque entrée, puis, en comparant les arbres obtenus.

Pour pouvoir faire mieux, c'est à dire utiliser l'arbre généré pour une entrée afin de déterminer l'entrée optimale, il faut absolument que le graphe ne soit pas découpé en "zones d'influence" c'est à dire des ensembles de salles (avec entrées) sans lien entre eux. Dans ce cas, l'arbre généré pour une entrée n'est d'aucune utilité pour avoir des informations sur les autres zones accessibles par d'autres entrées.

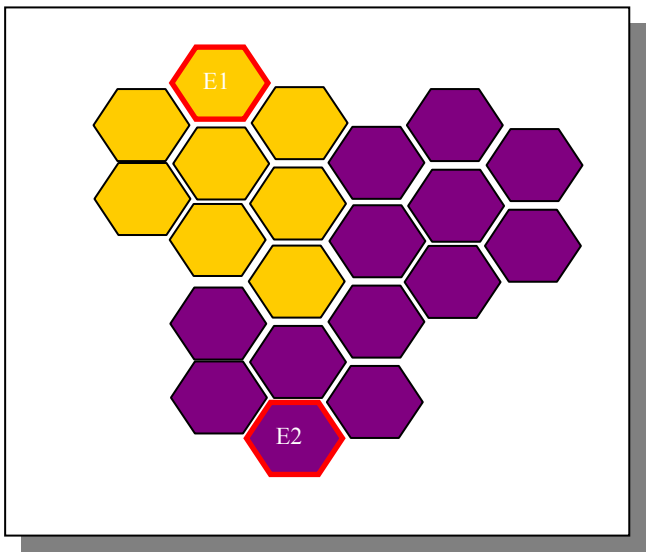
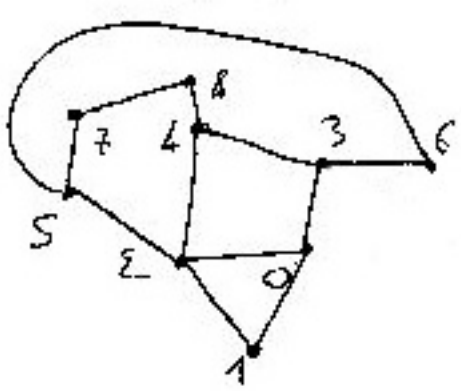


Illustration 2D des zones d'influence autour de deux entrées E1 et E2

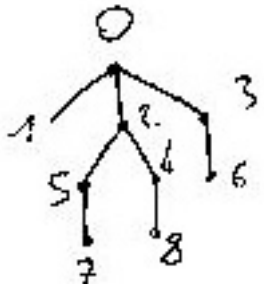
3.1 Algorithme par recherche de successeur

Nous avons tout de même réussi à concevoir un algorithme capable de déterminer cette entrée optimale dans le cas d'un graphe sans "zone d'influence" et voici à quoi notre réflexion a abouti :

On part par exemple du graphe suivant :



On génère ensuite l'arbre de parcours à partir du nœud 0 :



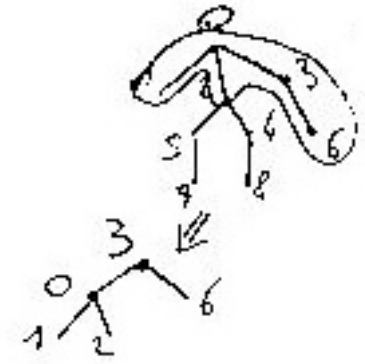
L'arbre obtenu a une profondeur maximale de 3.
La question est alors : y a -t il moins profond ?
On cherche des arbres de profondeur < 3.

1- On peut éliminer de l'étude la racine et les feuilles à une profondeur 3 (7 et 8) car il on sait que le chemin le plus court pour aller de 7 à 0 et de 8 à 0 est de longueur 3.
2- Il reste (1, 2, 3, 4, 5, 6) à étudier.
Prenons 3, V(3) = {0, 4, 6}.
On a déjà des chemins de longueur <= 2 pour les sommets (0 ,1, 2, 6) donc, il faut que les sommets qui restent aient au moins un voisin dans les sommets {0, 3, 6} car autrement leur distance au nœud 3 sera > 2.

V(5) = {2, 6, 7} donc c'est bon.

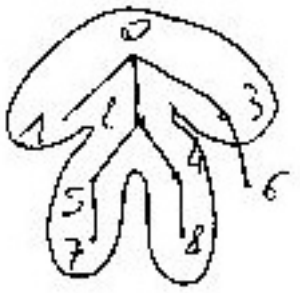
V(7) = {5, 8} donc ça ne marche pas car 5 et 8 ne sont pas dans {0, 3, 6}.

On arrête.



L'arbre de racine 3 sera de profondeur au moins 3.
On teste donc un autre sommet, par exemple : 2.

Les sommets (0, 1, 3, 4, 5, 7, 8) sont atteignables par un chemin de longueur 2 maximum.



Reste à tester le voisinage de 6 qui doit être inclus dans le voisinage de 2.

V(2) = {0, 1, 4, 5}

V(6) = {3, 5}

5 est commun aux deux listes, il y a donc un chemin de longueur 2 maximum entre 2 et 6.
Donc, on sait que l'entrée 2 est meilleure que l'entrée 0. Il suffit de re-générer son arbre de parcours à l'aide de *Dijkstra* par exemple.

En résumé :

1- Créer un arbre A_0 à partir d'une entrée quelconque.

2- Rechercher un arbre A_1 de profondeur inférieure : prof (A_1) = prof (A_0) - 1

Faire {

- prendre une entrée quelconque dans l'arbre.
- mettre les nœuds se trouvant à une profondeur inférieure à prof(A_0) - 2 dans une liste.
- vérifier que les sommets restant ont leur voisinage dans la liste.

Si oui, on, recommence sur cet arbre, sinon on retire le nœud de la liste, des nœuds à traiter. Pour réaliser cette opération, les matrices d'adjacence semblent être les structures de donnés les mieux adaptées et permettraient d'atteindre une complexité quasiment de l'ordre de n.

} Jusqu'à ce que tous les nœuds soient traités

Une fois un ou plusieurs nœuds optimaux obtenus, on calcule son arbre.

3.2 Algorithme par parcours concurrents

Une autre technique pour trouver l'entrée optimale dans un graphe sans "zone d'influences" isolées, consisterait à générer les arbres de parcours pour chaque entrée en parallèle c'est à dire étage par étage en s'arrêtant dès que l'on rencontre un nœud plus proche d'une autre entrée. Par contre les structures de données à mettre en place semblent assez complexes puisqu'il faut stocker dans chaque salle la distance à chaque entrée. On obtient ainsi des groupes de salles autours des entrées, l'entrée optimale étant alors l'entrée regroupant le plus de salles.

4. Représentation graphique et interface utilisateur

Afin d'assurer une bonne compréhension du problème posé dans l'espace, nous avons créé une représentation graphique tridimensionnelle du graphe de salles tout comme de l'arbre de chemin minimum à partir d'une entrée. Cette représentation a été réalisée à l'aide d'une bibliothèque 3D pour JAVA nommée *Java3D* et développée par Sun pour un grand nombre de plates-formes. Ceci a donc été également une bonne occasion pour nous de nous plonger dans une bibliothèque complexe d'affichage. Il est intéressant de noter que l'organisation d'une scène en Java3D est basée sur une arborescence d'objets.

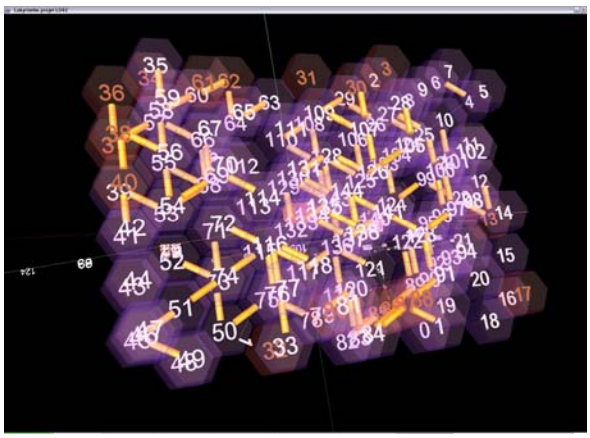
L'affichage du labyrinthe propose 3 modes de représentation:

- 1) Numéros de Salles+Liens+Salles 3D transparentes
- 2) Salles 3D
- 3) Numéros de Salles

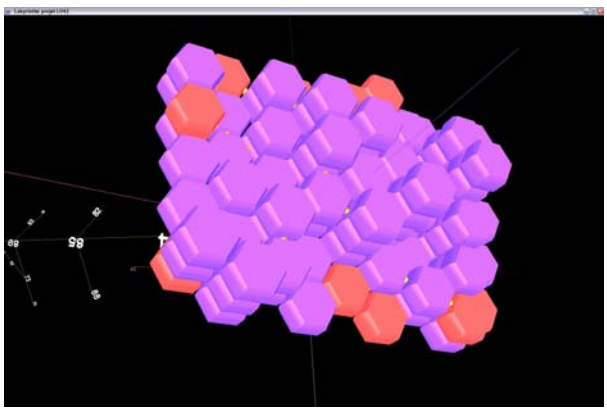
Dans tous les modes, l'arbre de parcours reste affiché sous le labyrinthe ce qui permet une observation relativement aisée.

Nous avons implémenté ce système d'affichage dans une classe nommée *J3D_DrawManager* qui implémente une interface appelée DrawManager qui fixe les méthodes d'affichage des différents éléments. Cette classe sert également à gérer les entrées utilisateurs avec les commandes suivantes. (Attention car tous les paramétrages se font au clavier dans la console et lors de l'attente d'une entrée, la fenêtre d'affichage se fige) :

- **Souris+bouton gauche** : Rotation de la vue – **Souris+bouton milieu** : Zoom vue
- **Souris+bouton droit** : Déplacement vue
- **touche *a*** : Affichage de l'arbre généré pour une entrée, numéro de l'entrée a préciser au clavier dans la console.
- **touche *s*** : Mode d'affichage des salles – **touche *r*** : Création d'un nouveau Labyrinthe
- **touche *t*** : Création d'un nouveau Labyrinthe avec paramétrage de la taille
- **touche *e*** : Création et affichage de l'arbre de parcours pour une entrée quelconque



Mode N°1 Le graphe de salles



Mode N°2 Représentation des salles en 3D

Conclusion

Ce projet nous a permis d'appliquer et de mettre en œuvre la manipulation et le parcours de graphes ainsi que la création d'arbres dans un langage objet. Nous avons été confronté au problème du coût en puissance de calcul que demandent les algorithmes sur ce genre de structures en particulier lorsque la masse d'information à traiter devient importante (dimensions du labyrinthe). Tous les algorithmes et structures de données ont été conçus de telle sorte à pouvoir être utilisés dans d'autres applications et généralisés pour différentes formes de salles, du fait que nous ayons pris soin de rendre le nombre de liens/voisins de chaque salle totalement paramétrable. Nous nous sommes également attachés à créer une interface utilisateur et plus particulièrement une représentation graphique permettant une bonne compréhension du problème et de sa résolution par le programme.