Travail pratique 1

INF1008 SESSION HIV-22

ANALYSE ET CONCEPTION D'ALGORITHMES

PREPARER PAR :

- **ULRICH JIOGANG DONGMO**
- **KETSIA MBUYI KAZADI**
- **XINYE GE**

UNIVERSITE DE QUEBEC A TROIS-RIVIERES

Table de matières

Présentation du travail

Notre programme est composé de 3 classes, qui sont :
La classe main qui est la classe principale qui représente le programme.
La classe graphe, qui représente un graph à l'intérieur de cette classe on implémente un graphe sous la forme de matrice adjacente.
Ensuite la classe labyrinthe, qui va générer notre labyrinthe a l'intérieure il y'a la logique de pondération de notre graph et une méthode qui représente l'algorithme de prim qui va générer le labyrinthe se basant sur une entre et une sortie qui sera générer aussi aléatoirement, au niveau de la pondération va se faire aléatoirement.
Aussi dans la classe labyrinthe on a la méthode tostring qui permet d'afficher notre graphe selon les exigences demander.

Difficultés rencontrées

Première difficulté qu'on a eu était sur le choix du deux chemins ayant le même poids, quand l'algorithme arrivé sur deux chemins qui ont le même poids, le choix du nœud idéal se passé mal cela ramener à des cycles.
Deuxième difficulté était sur comment résoudre le problème de cycle qui se formé, l'algorithme choisissait toujours
le mauvais chemin entre le deux. Et à chaque fois on a eu ce même problème de cycle qui as était pour nous difficile de résoudre.

Complexité algorithmique

- **Algorithme de prim**

```
    public void algorithmPrim()
        {
            int iG = deijAJgraphe(entreeLabyr[0], entreeLabyr[1]);

            nouedVisiter.add(iG);

            ArrayList<Integer[]> lesPlusPetitsElementsdesLignes = new ArrayList<>();
            Integer[] lienMinimal;
    //        int o = 0;
            compteurLabyr += 4;

            //Dans notre matrice adjacente chaque ligne reprensete un noeud
            //La maniere dont l'algorithme fonction est que on parcout chaque noeud(ligne dans la
    matrice adjacente)
```

```
        //on regarde la liaison la plus petit du noeud on la conserve et on lui compare a la
liaison la plus petite de
        //chaque noeud qui on ete visiter pour selectionner la plus petite liaison parmi tout
les candidates
        //cette liaison et donc inscrit dans le graphe finale
        for (int i = 0; i < nbNoueds - 2; i++)
        {
            compteurLabyr++;
            for (Integer ligne: nouedVisiter)
            {

lesPlusPetitsElementsdesLignes.add(plusPetitDeLigne(grapheLabyr.getRows(ligne), ligne,
false));
                compteurLabyr++;
            }

            lienMinimal = plusPetitParmisLesPlusPetits(lesPlusPetitsElementsdesLignes,
nouedVisiter);
            grapheLabyr.liason(lienMinimal[2], lienMinimal[0], 0);
            grapheLabyr.liason(lienMinimal[0], lienMinimal[2], 0);

            labFInal.liason(lienMinimal[2], lienMinimal[0], 1);
            compteurLabyr += 4;


            if(labFInal.getElement(lienMinimal[0], lienMinimal[2]) == 0 )
            {
                labFInal.liason(lienMinimal[0], lienMinimal[2], 1);
                compteurLabyr ++;
            }

            if (!nouedVisiter.contains(lienMinimal[0]))
            {
                nouedVisiter.add(lienMinimal[0]);
                compteurLabyr++;
            }

            lesPlusPetitsElementsdesLignes.clear();
            compteurLabyr++;
        }

        Integer[] lienversortie =
plusPetitDeLigne(grapheLabyr.getRows(deijAJgraphe(sortieLabyr[0], sortieLabyr[1])),
deijAJgraphe(sortieLabyr[0], sortieLabyr[1]), true);
        labFInal.liason(lienversortie[2], lienversortie[0], 1);
        compteurLabyr += 2;

        if(labFInal.getElement(lienversortie[0], lienversortie[2]) == 0 )
        {
            labFInal.liason(lienversortie[0], lienversortie[2], 1);
            compteurLabyr++;
        }
        grapheLabyr = labFInal;
        compteurLabyr++;
    }
```

L’algorithme de génération du labyrinthe possède un boucle imbriquée ce qui implique (n*k) * (n*k’) nombre d’opération, ou k est le nombre d’opération de la première boucle et k’ celle de la deuxième boucle ce qui donne un nombres d’opérations de n^2 de plus il y’a k’’ nombres d’operations après les boucles donc le nombre d’operations est de n^2 + k’’ qui équivaut à n^2 . D’où une traduit une complexité $O(n^2)$.

- **Affichage Labyrinthe**

```
public String toString()
{
    String affichage = "";
    String firstLigne = "";
    String secondLigne = "";

    compteurAffichage += 3;
    for (int i = 0; i < mLab; i++)
    {
        compteurAffichage++;
        for (int j = 0; j < nLab; j++)
        {
            if (i == entreeLabyr[0] && j == entreeLabyr[1])
            {
                firstLigne += " E ";
            }
            else if (i == sortieLabyr[0] && j == sortieLabyr[1])
            {
                firstLigne += " S ";
            }
            else
            {
                firstLigne += " 0 ";
            }
            compteurAffichage++;

            if (j + 1 < nLab)
            {
                if (grapheLabyr.getElement(deijAJgraphe(i, j + 1),
deijAJgraphe(i, j)) != 0)
                {
                    firstLigne += " 0 ";
                }
                else
                {
                    firstLigne += " # ";
                }
            }
            compteurAffichage++;

            if (i + 1 < mLab)
            {
                if (grapheLabyr.getElement(deijAJgraphe(i + 1, j),
deijAJgraphe(i, j)) != 0)
                {
                    secondLigne += " 0 ";
                }
                else
                {
                    secondLigne += " # ";
                }
            }
            compteurAffichage++;
```

```
            if ((j + 1 < nLab ))
            {
                secondLigne += " # ";

            }
            compteurAffichage++;

        }
        affichage += "\n" + firstLigne + "\n" + secondLigne;
        firstLigne = "";
        secondLigne = "";
        compteurAffichage += 3;
    }

    compteurAffichage++;
    return affichage;
}
```

L'algorithme de génération du labyrinthe possède une boucle imbriquée ce qui implique (n*k) * (n*k') nombre d'opération, ou k est le nombre d'opération de la première boucle et k' celle de la deuxième boucle ce qui donne un nombre d'opération de n^2 ce qui traduit une complexité $O(n^2)$.

- **Pondération**

Code à voir sur projet.

L'algorithme de pondération du graphe possède une boucle ce qui implique n*k nombres d'opérations, ou k est le nombre d'opération de la première boucle ce qui traduit une complexité O(n).

- **Complexité générale**

La complexité générale de notre programme peut être depuis des complexité des algorithmes principaux : ComplexiteTotale = ComplexitePonderation + ComplexiteAffichage + ComplexiteAlgorithmePrim

⇨ ComplexiteTotale = $O(n) + O(n^2) + O(n^2)$

⇨ ComplexiteTotale = $O(n^2)$

Car $O(n) + O(n^2) = O(n^2)$ et $O(n^2) + O(n^2) = O(n^2)$