SAE Exploration Algorithmique d’un problème

BEAL Lucas

ROUYER Hugolin

Représentation d’un graphe

Classes produites :

- Nœud (Constructeur, equals, ajouterArc)
- Arc (Constructeur, getter)
- Interface Graphe (listeNoeuds, suivants)
- GrapheListe (Constructeur, ajouterArc, listeNoeuds, suivants, toString, toGraphViz)
- Main

La classe GraphListe a été tester (et donc arc et nœud au passage)

Calcul du plus court chemin par point fixe

Ecriture de l’algorithme :

Question 13

fonction resoudre(Graphe g, Chaine depart): valeur

début

v <- initialisation(g,depart)

finis <- faux

iteration <- 0

tant que non finis et itaration< g.listeNoeuds().size() faire

pour sommet de 0 à g.listeNoeuds().size() faire

```
                        pointfixe.add(v.getValeur(g.listeNoeuds().get(sommet)))
                fpour

                pour sommet de 0 à g.listeNoeuds().size() faire
                        listeParents <- listeAntecedentSommet(g,g.listeNoeuds().get(sommet))
                        si listeParents.size()!=0 faire
                                min <- v.getValeur(g.listeNoeuds().get(sommet))
                                si min < 0
                                        min <- Plus l'infinie
                                fsi
                                placeP <- 0

                                pour parent de 0 à listParents.size() faire
                                        valeur <- v.getValeur(listParents.get(parent)) + this.valeurArc(v, g, g.listeNoeuds().get(sommet), listParents.get(parent))
                                        si valeur > 0
                                                si min > valeur faire
                                                        min <- (int) v.getValeur(listParents.get(parent)) + this.valeurArc(v, g, g.listeNoeuds().get(sommet), listParents.get(parent));
                                                        placeP = parent;
                                                fsi
                                        fsi
                                fpour

                                si min != v.getValeur(g.listeNoeuds().get(sommet) faire
                                        v.setValeur(g.listeNoeuds().get(sommet), min);
                  v.setParent(g.listeNoeuds().get(sommet), listParents.get(placeP));
              fsi
                        fsi
```

```
        fpour

        i <- 0
        finis <- vrai
        tant que i<pointfixe.size() et finis faire
            si v.getValeur(g.listeNoeuds().get(i)) != pointfixe.get(i) faire
                finis <- false
            fsi
            i <- i + 1
        ftant

        iteration <- iteration + 1
    ftant
    retourne v
Fin
```

```
fonction initialisation(Graphe g,Chaine depart) : Valeur
Début
    listeNoeuds <- g.listeNoeuds()
    pour i de 0 listeNoeuds.size() faire
        si listeNoeuds.get(i) == depart faire
            v.setValeur(listeNoeuds.get(i), 0)
        sinon
            v.setValeur(listeNoeuds.get(i), Double.MAX_VALUE)
        fsi
    fpour
    retourne v
```

```
    Fin


fonction listeAntecedentSommet (Graphe g, String sommet): ArrayList<String>
    Début
        pour i de 0 à g.listeNoeuds().size()
            j <- 0
            trouver <- faux
            tant que j<g.suivants(g.listeNoeuds().get(i)).size() et  non trouver faire
                si g.suivants(g.listeNoeuds().get(i)).get(j).getDest() == sommet faire
                    trouver <- vraie
                fsi
                j <- j+1
            ftant
            si trouver faire
                list.add(g.listeNoeuds().get(i)
            fsi
        fpour
        retourne list
    Fin


fonction valeurArc ( Valeur v , Graphe g , Chaine arriver ,   Chaine depart):entier
    Début
        listArcs <- g.suivants(depart)
        i<- 0
        trouver <- faux
```

```
        tant que non trouver et i<listArcs.size() faire
            si listArcs.get(i).getDest() == arriver faire
                trouver = vrai
            fsi
            i <- i+1
        ftant

        retourne listArcs.get(i-1).getCout()
    Fin
```

Lexique de resoudre :

v : Valeur / valeur du graphe retourné

finis : booleen / booleen qui regarde si il y a un pointfixe

iteration : entier / nombre d iteration

pointfixe : ArrayList<Integer> / sauvergarde de la valeur courante des sommets

sommet : entier / variable d iteration

parent : entier / variable d iteration

min : entier / qui calcule le minimum pour chaque sommet

placeP : entier / qui retourne la place du parent du sommet courant

Classe valeur implémenter

Ecriture de la classe BellmanFord

Ecriture d’un main pour cette classe

Ecriture d’une classe de test pour BellmanFord

Ecriture de calculerChemin dans valeur

Calcul du meilleur chemin par Dijkstra

Ecriture de la classe Dijkstra

Ecriture de l'interface Algorithme

Ecriture de MainDijkstra

Ecriture de la classe de test pour Dijkstra

Validation et expérimentation

Question 21

Algorithme de BellmanFord :

(Initialisation)

A -> V:0.0 p:null

B -> V:1.7976931348623157E308 p:null

C -> V:1.7976931348623157E308 p:null

D -> V:1.7976931348623157E308 p:null

E -> V:1.7976931348623157E308 p:null

F -> V:1.7976931348623157E308 p:null

G -> V:1.7976931348623157E308 p:null

(1^{er} tour)

A -> V:0.0 p:null

B -> V:20.0 p:A

C -> V:7.0 p:D

D -> V:3.0 p:A

E -> V:38.0 p:F

F -> V:35.0 p:G

G -> V:30.0 p:B

(2ème tour)

A -> V:0.0 p:null

B -> V:9.0 p:C

C -> V:7.0 p:D

D -> V:3.0 p:A

E -> V:27.0 p:F

F -> V:24.0 p:G

G -> V:19.0 p:B

(3ème tour)

A -> V:0.0 p:null

B -> V:9.0 p:C

C -> V:7.0 p:D

D -> V:3.0 p:A

E -> V:27.0 p:F

F -> V:24.0 p:G

G -> V:19.0 p:B

On remarque que l'algorithme de BellmanFord mets 3 tours de boucle pour résoudre le graphe + un pour l'initialisation.

(Initialisation)

A -> V:0.0 p:null

B -> V:1.7976931348623157E308 p:null

C -> V:1.7976931348623157E308 p:null

D -> V:1.7976931348623157E308 p:null

E -> V:1.7976931348623157E308 p:null

(1ère Iteration)

A -> V:0.0 p:null

B -> V:12.0 p:A

C -> V:1.7976931348623157E308 p:null

D -> V:87.0 p:A

E -> V:1.7976931348623157E308 p:null

(2ème Iteration)

A -> V:0.0 p:null

B -> V:12.0 p:A

C -> V:1.7976931348623157E308 p:null

D -> V:87.0 p:A

E -> V:23.0 p:B

(3ème Itération)

A -> V:0.0 p:null

B -> V:12.0 p:A

C -> V:1.7976931348623157E308 p:null

D -> V:66.0 p:E

E -> V:23.0 p:B

(4ème Itération)

A -> V:0.0 p:null

B -> V:12.0 p:A

C -> V:76.0 p:D

D -> V:66.0 p:E

E -> V:23.0 p:B

(5ème Itération)

A -> V:0.0 p:null

B -> V:12.0 p:A

C -> V:76.0 p:D

D -> V:66.0 p:E

E -> V:23.0 p:B

On remarque que l'algorithme de Dijkstra mets 5 tours de boucle pour résoudre le graphe + un pour l'initialisation. On remarque que le nombre de tour de boucle correspond au nombre de sommet.

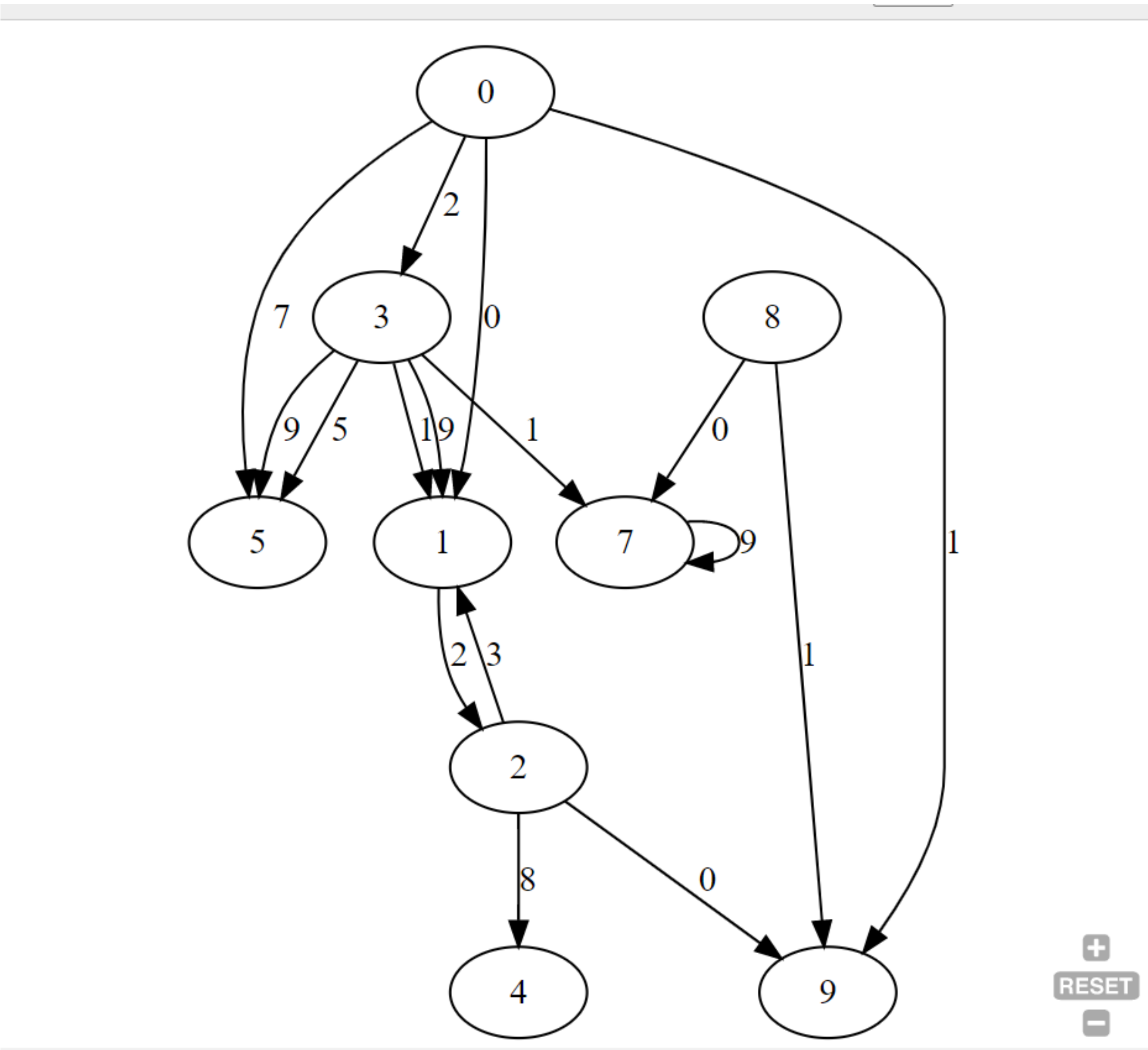
Question 22

On peut donc conclure que l'algorithme de BellmanFord fait moins de tour de boucle que Dijkstra.

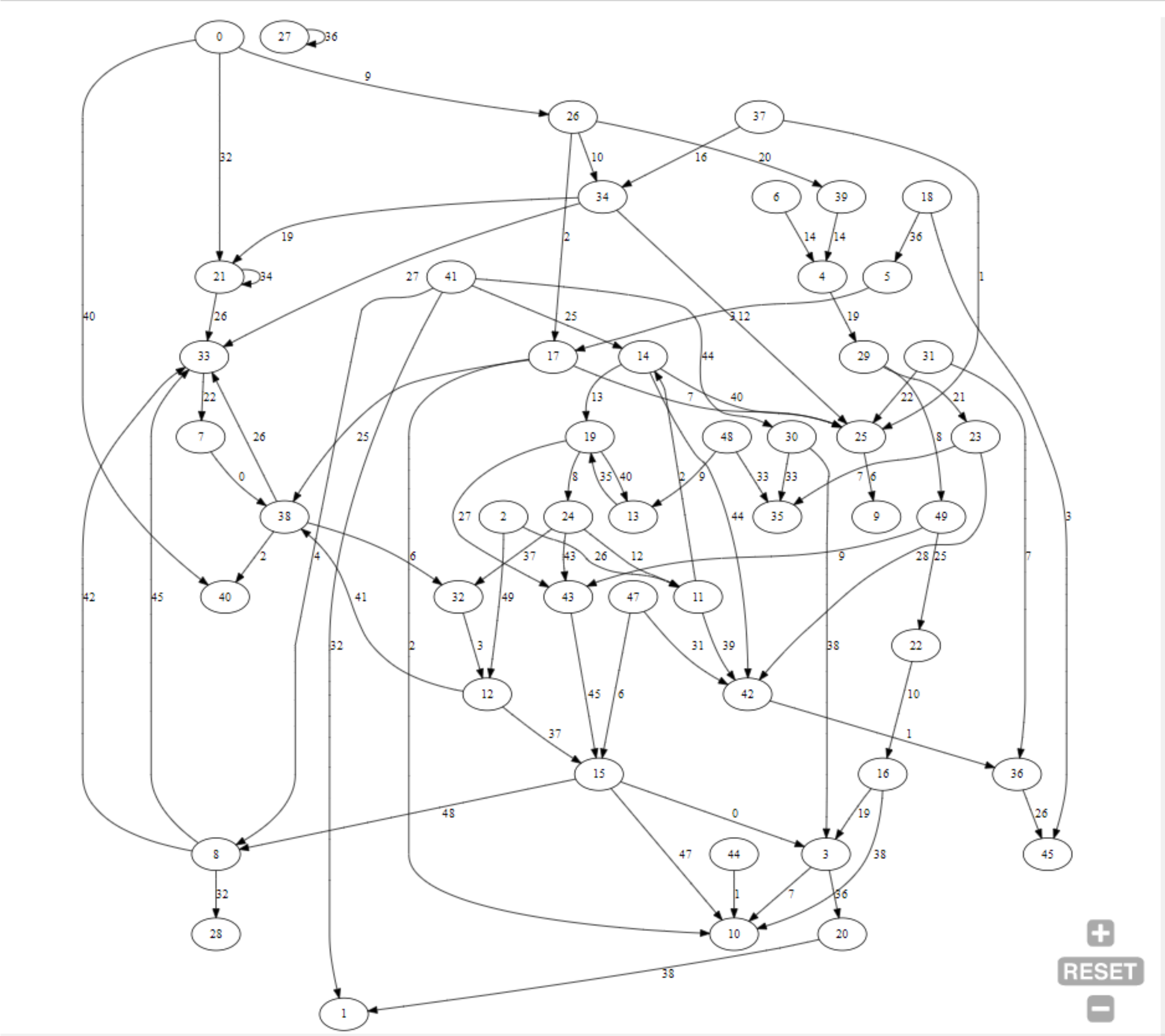
Question 22

Ecriture de la classe GenererGraphe et d'un main associer

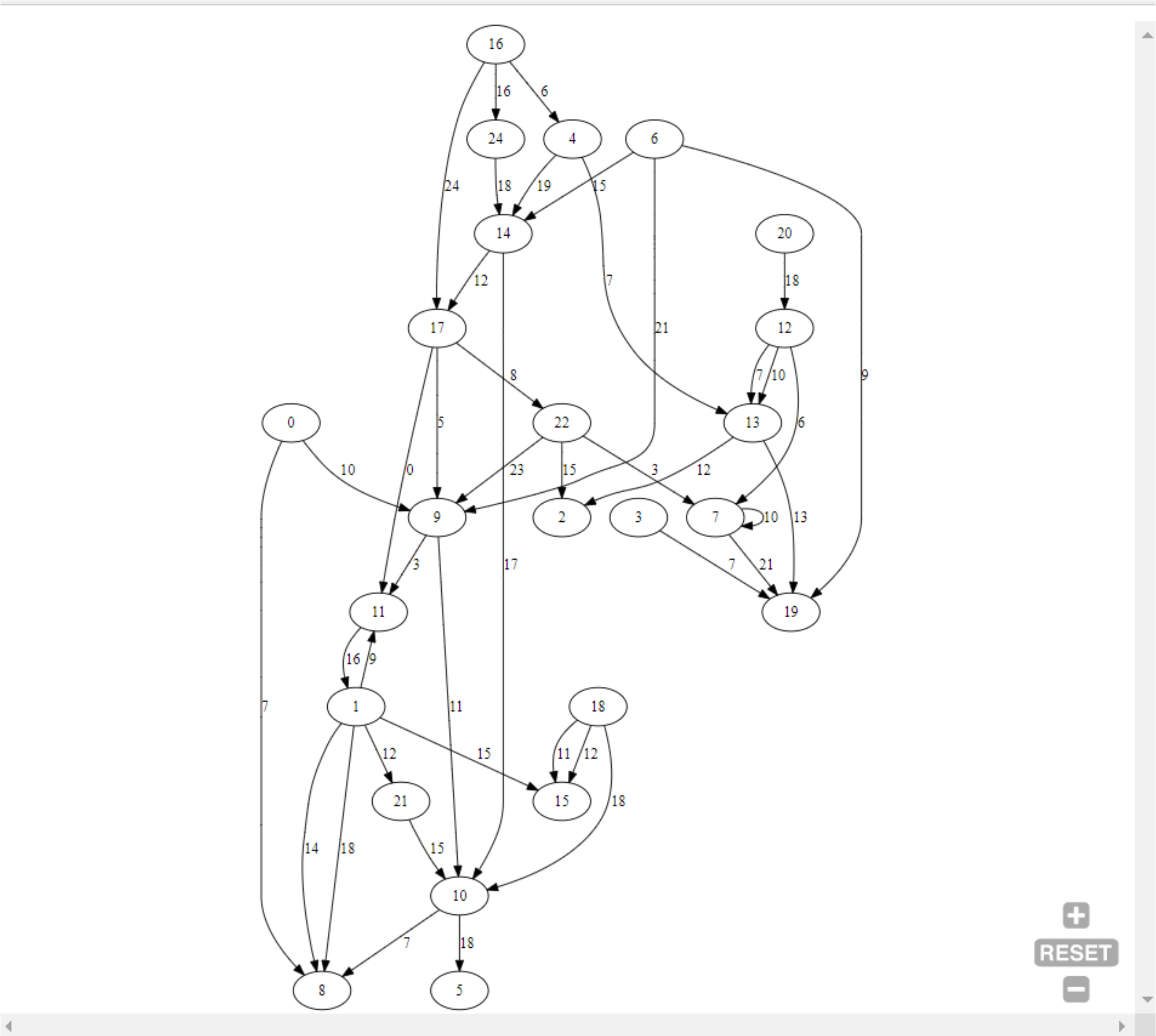
Question 25



Graphe généré avec 10 nœuds avec départ 0 et arrivée 9



Graphe généré avec 50 nœuds avec départ 0 et arrivée 45

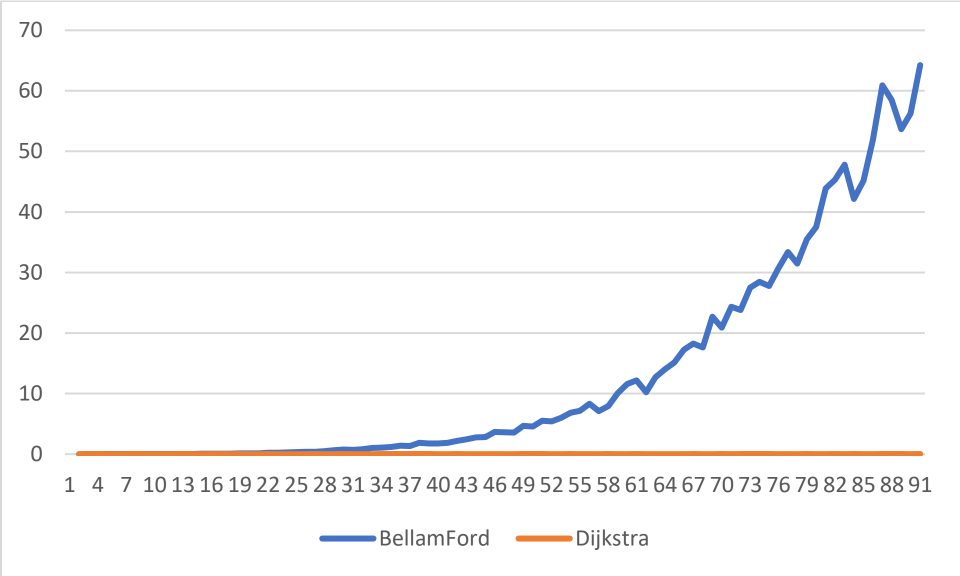


Graphe généré avec 25 nœuds avec départ 0 et arrivée 15

Pour les questions suivantes, on a écrit un petit paragraphe qui répond à toutes les questions à la fois

Questions 23/ 26/27

Lors de petit graphe, la différence de temps pour la résolution de celui-ci est très faible (Dijkstra est plus rapide de quelques millisecondes par rapport à BellmanFord). En revanche, on remarque que le temps de résolution pour BellmanFord va augmenter de manière exponentielle à chaque fois que l'on rajoute un sommet. A l'inverse, Dijkstra reste quasiment constant (il augmente très lentement) cf graphe si dessous.



Comparaison entre l'algorithme de BellamFord et celui de Dijkstra en seconde en fonction du nombre de nœud.

Questions 28

On peut conclure que Dijkstra est beaucoup plus efficace sur les grands graphe que le point fixe qui lui peut être utilisé mais que sur des petits graphes

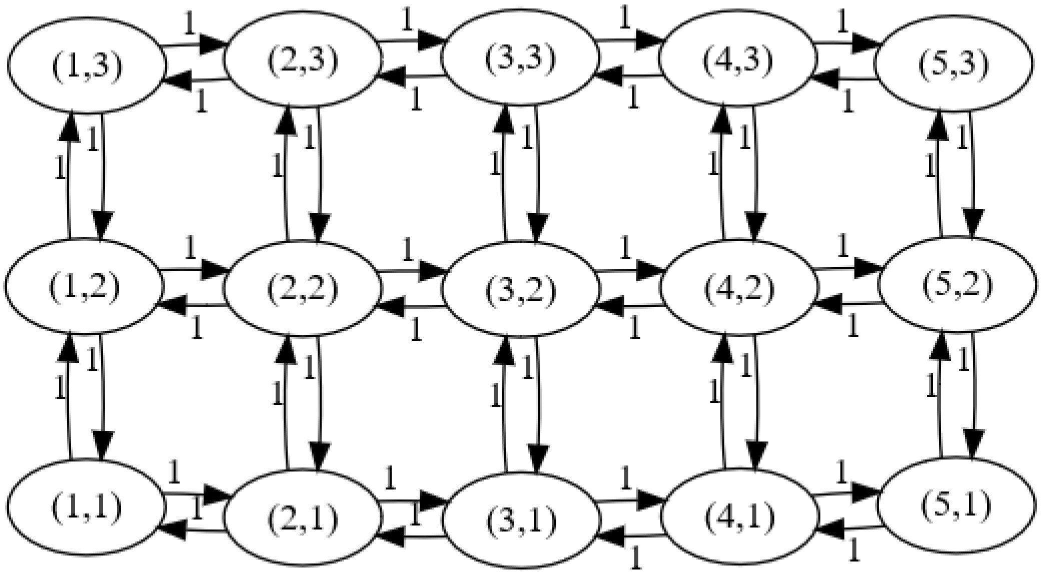
Les difficultés rencontrées lors de cette SAE sont les suivantes :

- Ecriture de l'algorithme de point fixe car nous avons uniquement les parents avec la classe valeur alors que pour résoudre cet algorithme on a besoin des antécédents (5 heures au lieu de 2)

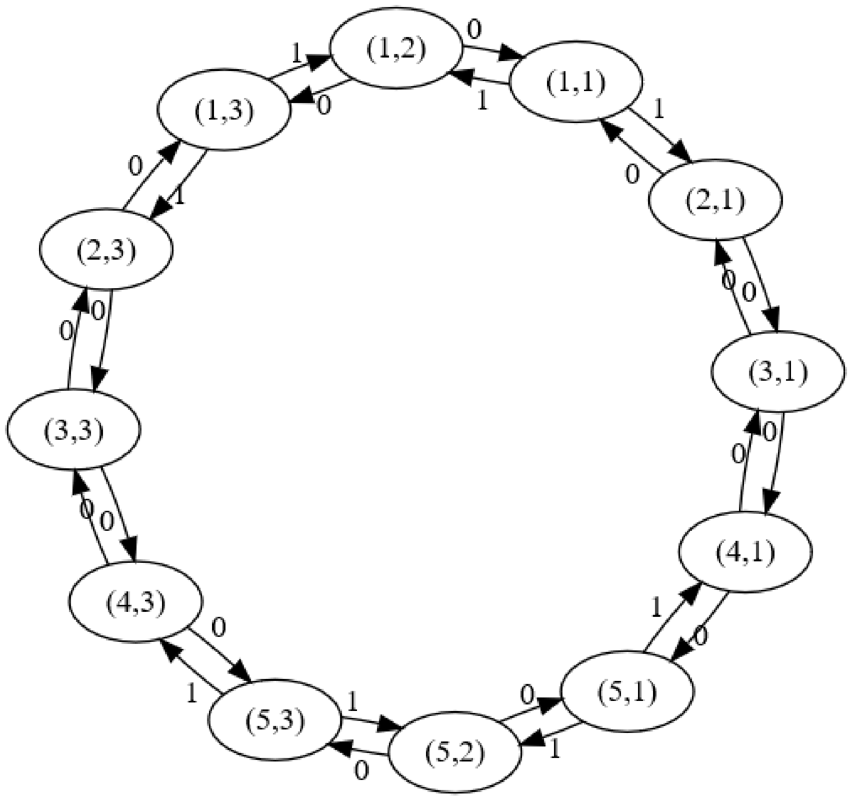
Bilan à tirer de cette SAE :

- Ce n'est pas parce qu'un algorithme fait moins de boucle qu'il est plus rapide
- Les algos ont une vitesse qui dépend de la taille du graphe
- L'algorithme de Dijkstra est l'algorithme le plus efficace et donc à utiliser pour faire différentes tâches

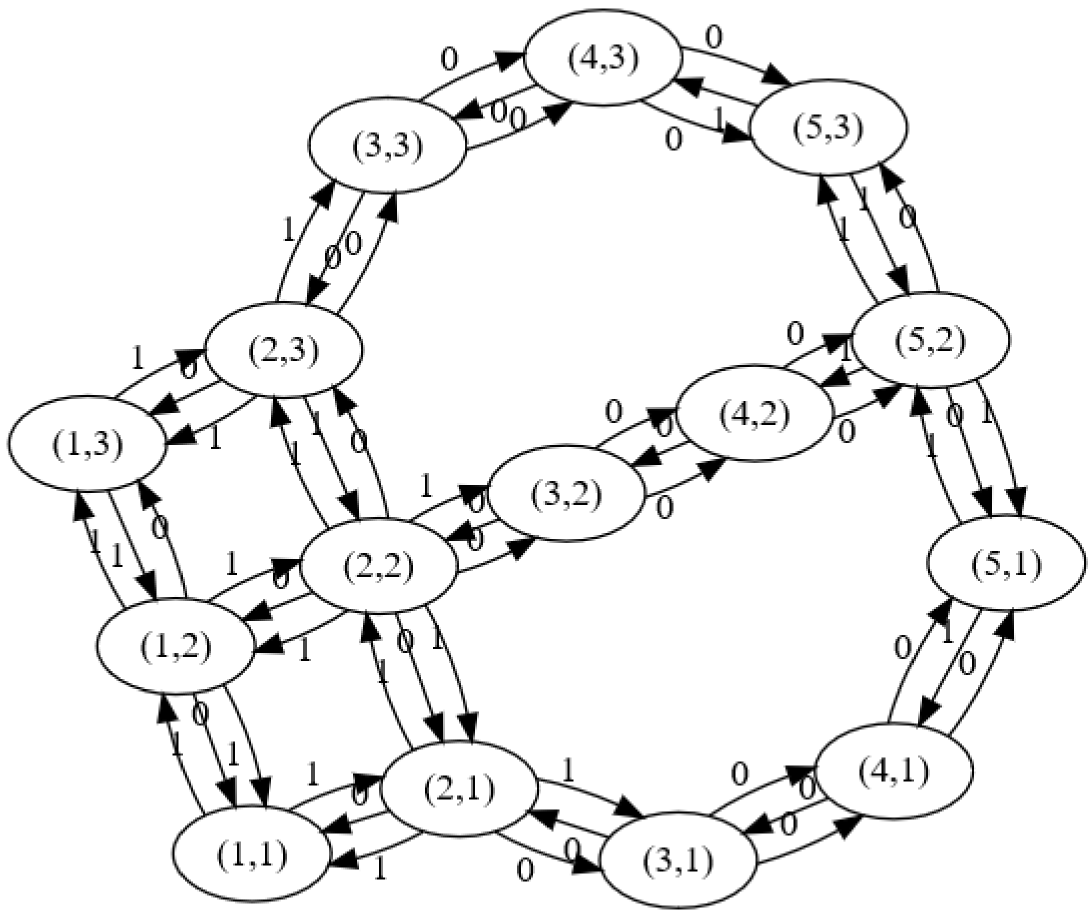
Extension : Intelligence Artificielle et Labyrinthe :



Graphe généré avec le labyrinthe laby0 fournit :



Graphe généré avec le labyrinthe laby0 fournit mais le labyrinthe est fait en glace en partant de la case (1 ;1) :



Graphe généré avec le labyrinthe laby0 fournit mais le labyrinthe est fait en glace en partant de la case (2 ;2) :