Des ensembles mathématiques aux distances pratiques

Essai sur l’histoire de la théorie des graphes et des ensembles et sur l’Algorithme de Dijkstra.

Table des matières

Introduction aux ensembles, opérations sur les ensembles, cardinal de l'ensemble des parties, exemples.

Dans cette partie, nous allons nous intéresser aux ensembles et aborder les théorèmes basiques autour de ceux-ci afin de comprendre leur raisonnement intrinsèque et leurs implications.
Pour cela, nous allons dans un premier temps exposer la logique préliminaire des ensembles, puis nous aborderons les opérations possibles sur ceux-ci. Par la suite nous étudierons le cardinal de l'ensemble des parties et nous finirons avec quelques exemples d'application.

Introduction aux ensembles

Définition 1a : Un ensemble est une collection d'objets.
Ces objets sont nommés élément de cette ensemble, et ils peuvent satisfaire un certain nombre de propriétés comme être des nombres réels, naturels, ou être paires par exemples. (1.c)

Notation :
On nomme x un élément de l'ensemble E, alors nous pouvons écrire $x \in E$ (qui signifie "x appartient à E").
Si x n'appartient pas à E, on note alors $x \notin E$.
De plus, il existe l'ensemble vide noté $\varnothing$, qui signifie que l'ensemble n'a aucun élément.

Propriétés de la comparaison des ensembles :

- Deux ensembles sont égaux s'ils contiennent exactement les mêmes éléments.
- Un ensemble A est inclus dans un ensemble B si tous les éléments de A appartienne à B (donc dire que A= B, revient à dire que $\forall x \in A, x \in B$ et $\forall x \in B, x \in A$).

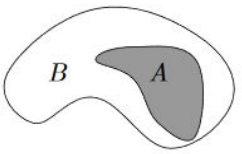
Remarques :

- L'ensemble vide est inclus dans tous les ensembles.
- Soient A et B deux ensembles. Si A est inclus dans B, on dit que A est un sous-ensemble de B. (1.c)

Maintenant que nous avons quelques notions de bases, nous pouvons passer aux opérations sur les ensembles.

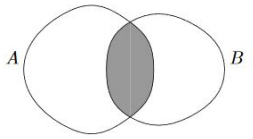
Opération sur les ensembles

Pour pouvoir étudier les opérations sur les ensembles, il faut avant tout faire un bref rappel sur les inclusions.
Soient A et B deux ensembles.
On définit :

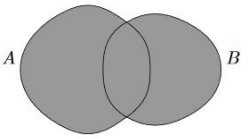


$A \subset B$

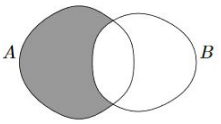
- $A \subset B$ (inclusion), si chaque élément de A est un élément de B.
- $A \cup B$, l'union de A et B, est l'ensemble des éléments qui sont dans A ou dans B ou dans les deux.
- $A \cap B$, l'intersection de A et B, est l'ensemble des éléments qui sont dans A et dans B.
- $A \setminus B$, la différence A moins B, est l'ensemble des éléments qui sont dans A, mais pas dans B.
- $A \Delta B$, la différence symétrique de A et B, l'ensemble des éléments qui sont soit dans A soit dans B, mais pas dans $A \cap B$.
- A^c ou $\bar{A}$, le complémentaire de A, l'ensemble des éléments qui ne sont pas dans A.



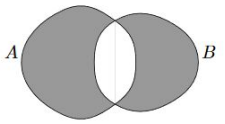
$A \cap B = \{x \mid x \in A \text{ et } x \in B\}$



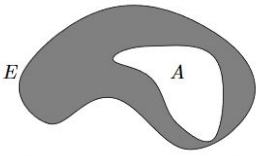
$A \cup B = \{x \mid x \in A \text{ ou } x \in B\}$



$A \setminus B = \{x \in A ; x \notin B\}$



$A \Delta B = (A \cup B) \setminus (A \cap B)$



$\complement_E A = \{x \in E ; x \notin A\}$

(extrait de (1.2))

Définition 2.a :
Soit E un ensemble et A et B deux sous-ensembles de E. On appelle intersection de A et de B l'ensemble des éléments qui appartiennent à A et à B. On note cet ensemble $A \cap B$ (lire «A inter B»). On a donc $A \cap B = \{x \in E ; x \in A \text{ et } x \in B\}$. (1.c)

Proposition 1 :
- commutativité : $A \cap B = B \cap A$, $A \cup B = B \cup A$.
- associativité : $A \cap (B \cap C) = (A \cap B) \cap C = A \cap B \cap C$,
  $A \cup (B \cup C) = (A \cup B) \cup C = A \cup B \cup C$.
- distributivité : $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$, $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.
- $(A \cup B)^c = A^c \cap B^c$, $(A \cap B)^c = A^c \cup B^c$

(1.3)

Définition 2.b : Produit Cartésien
Soient A et B deux ensembles non vides. On appelle produit cartésien de A et de B, que l'on note $A \times B$ (lire «A croix B»), l'ensemble des objets (a,b) où $a \in A$ et $b \in B$ et qui vérifie la propriété : $\forall a,a' \in A$, $\forall b,b' \in B$, $(a,b)=(a',b') \Leftrightarrow a=a'$ et $b=b'$.
On a donc $A \times B = \{(a,b) ; a \in A \text{ et } b \in B\}$. (1.c)

Notation : le couple d'objets a et b est noté : (a,b).

Cardinal de l'ensemble des parties

Définition 3.a : Cardinal

Si E est un ensemble, on appelle cardinal de E le nombre d'éléments distincts que contient A.

Par exemple, si E={0,1,2,3,4,5}, alors son cardinal est 6. Si A possède une infinité d'éléments, son cardinal est égal à $+\infty$.

Le Cardinal de E est noté : card(E).(1.c)

Propositions :

- Additivité : Soient A et B deux ensembles finis, disjoints (c'est-à-dire $A \cap B = \varnothing$).
  Alors $card(A \cup B) = card(A) + card(B)$
- Multiplicativité : Soient A et B deux ensembles finis, et $C = A \times B$.
- Alors $card(C) = card(A) \cdot card(B)$
  (1.3)

Théorème :

Si E est un ensemble qui possède n éléments alors, l'ensemble P(E) des parties de E contient 2^n éléments. (1.c2)

Corollaires :

- Principe du dénombrement. On réalise deux expériences qui peuvent produire respectivement n et m résultats différents. Au total, pour les deux expériences prises ensemble, il existe n.m résultats possibles.

- Soit A un ensemble fini de cardinal n. Le nombre de suites de longueur r constituées d'éléments de A est n^r.

(1.3)

Exemples

1) On donne A={0,1,2,3} et B={-1,2,3,10,55}, nous allons donc commenter les relations entre ces 2 ensembles.

L'intersection de A et B vaut : {2,3}
L'union de A et de B vaut: {-1,0,1,2,3,10,55}
Le cardinal de A, card(A)=4.
Et card(B)=5.
A≠B car ils n'ont pas les même cardinales. De plus tous les termes de A ne sont pas égaux aux termes de B, ce qui confirme l'inégalité.
D'une autre part, l'ensemble des parties de A : P(A)= 2^4 =16 et P(B)=2^5 =32.

On donne C=A ✕ B,
Ainsi card(C)=card(A) ✕ card(B)=4 ✕ 5=20

On donne également D=A+B,
Donc card (E)= card(A)+card(B)=4+5=9

2) On donne maintenant, X={1,2} et Y={a, b, c}
Calculons le produit cartésien de X ✕ Y :
X ✕ Y= {(1,a);(1,b);(1,c);(2,a);(2,b);(2,c)}

3) Pour Z={9,8,7,6,5,4}, avec un cardinal P(Z)=6, calculons le nombre de suites de longueur 7.

Nous obtenons par définition 6^7 = 279936 nombres de suite de longueur 7 pour l'ensemble Z.

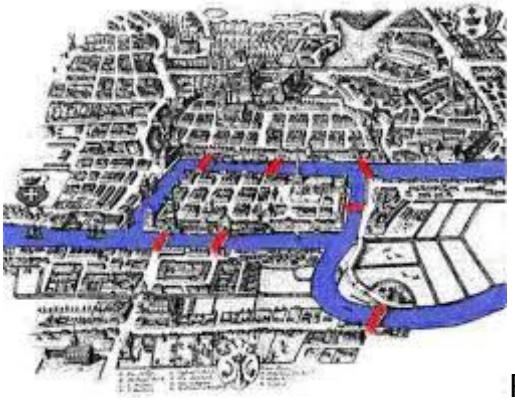
Les graphes, histoire, applications et illustrations.

La théorie des graphes est une branche des mathématiques récentes apparue à la fin du 18e siècle. Elle n'est que peu étudiée à l'enseignement secondaire mais quid à son utilité ? C'est pour cela que nous allons dans cette partie nous interroger sur l'origine de la théorie des graphes ainsi que ces répercussions sur les mathématiques et la société.

Les débuts de la théorie des graphes.

Cette théorie est présentée en 1735 à l'Académie de Saint-Pétersbourg dans un article du mathématicien suisse Leonhard Euler, qui sera publié plus tard en 1741. (1)
Cet article fondateur est basé sur un problème qu'Euler avait rencontré à Koenigsberg en Poméranie (Prusse Orientale).

En effet, cette ville est traversée par 2 rivières et au centre de la ville se trouve une île nommée Kneiphof. Cette île est reliée à la terre ferme par 7 ponts, cette particularité à mener des habitants à créer et tenter de résoudre un problème : trouver un chemin permettant de pouvoir partir d'un endroit et d'y retourner en parcourant tous les ponts une seule et unique foi tout en revenant au point de départ. Cependant personne ne savait si cela était possible ni n'avais réussi à démontrer le contraire. Euler décida de répondre à ce problème, et ainsi mis en place une méthode de résolution qui installa les bases de la théorie des graphes. (c) (2)



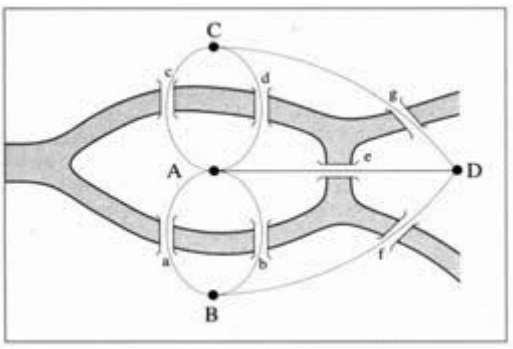
Plan de la ville(I.1)

Pour résoudre ce problème, Euler commença par faire une représentation simplifiée du problème :

- Transformation de chacune des 4 rives par un point.
- Représentation de chaque pont par une arête reliant 2 rives.

On obtient alors ce graphe simplifié (avec les bras de rivières pour mieux comprendre la transformation) :

(I.2)



Par la suite, nous pouvons compter le nombre d'arêtes(ici les ponts) partant de chaque sommet (ici les rives).
On obtient alors :

- n(A)=5
- n(B)=3
- n(C)=3
- n(D)=3

Nous constatons que chaque point porte un nombre impair d'arêtes.
Or pour passer par chaque point une seule fois, il faut un nombre pair d'arêtes par point.

Ainsi Euler pu conclure qu'il était impossible de faire une balade passant par tous les ponts une fois et qui nous ramène au point de départ. (c) (3)

Par ailleurs, la théorie des graphes fut approfondie au 19e siècle par de nombreux autres chercheurs, à l'image du mathématicien Arthur Cayley qui s'intéressa à un type particulier de graphe, les arbres.

Les progrès liés à la théorie des graphes et d'autres applications.

Les graphes ont également permis de faire des avancés dans les domaines des probabilités, comme avec les marches aléatoires (qui font parties du fonctionnement des bourses) qui se basent sur les graphes continus.(4)
Outre cet exemple nous pouvons aussi noter que la théorie des graphes est à l'origine des arbres de probabilité pondérés.

Une autre application de la théorie des graphes que tout le monde a déjà côtoyé et qui date du début 21e siècle concerne l'application de la théorie des graphes au système d'itinéraires (souvent couplés au GPS) et des "cartes" qui permettent d'aller d'un point A, à un point B selon des facteurs que l'utilisateur spécifie.
Ces facteurs vont du chemin le plus court au plus long en termes de distance, de temps ou de carburant.

Nous constatons que la théorie des graphes est omniprésente dans notre quotidien, et qu'elle nous permet de résoudre de nombreux problèmes.

Pour cela nous pouvons citer quelques algorithmes permettant de résoudre certains problèmes :

- L'algorithme de Welsh et Powell qui permet de colorer des graphes de telle sorte qu'il n'y ait pas. 2 fois la même couleur reliées par un même arc. (6)
- L'algorithme de Dijkstra qui permet de trouver le plus court chemin entre 2 sommets. (7)
- L'algorithme de Bellman Ford (1956-1958) permet de calculer des plus court chemin avec une origine unique dans le cas le plus général. Les poids des arcs(ou arêtes) peuvent avoir des valeurs négatives (ce qui n'est pas le cas de l'algorithme de Dijkstra). Cet algorithme permet aussi de savoir s'il existe un chemin à poids négatif. (8)

Intéressons-nous à l'algorithme de Welsh Powell. Il peut servir dans de l'organisation et l'attribution de position (concrète ou virtuelle) pour pouvoir attribuer des places en tenant compte des incompatibilités et ainsi pouvoir attribuer ces places en évitant tous problèmes d'incompatibilités tout cela grâce à la coloration d'un graphe représentant le problème. Concrètement cela peut se traduire en télécommunication pour l'attribution des fréquences pour éviter l'empiétement des fréquences ce qui nuirait à la qualité du réseau. (9)

De plus, les algorithmes de Dijkstra et de Bellman Ford peuvent être utiles dans différents domaines permettant de trouver un trajet plus court ou moins cher (dépendants de la valeur attribuée au poids des arêtes) quand nous nous intéressons aux déplacements d'un point à l'autre.

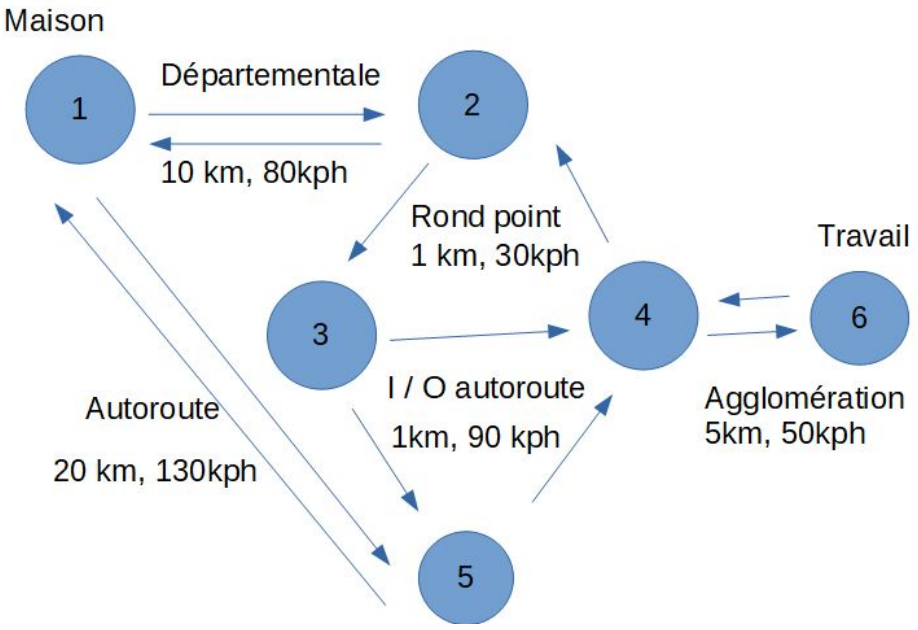
Il est aussi intéressant de noter que ces algorithmes sont implémentables sur ordinateurs.

Nous avons donc pus voir que la théorie des graphes, bien qu'étant une composante des maths récente (XVII e siècle), est néanmoins très utile dans de nombreux domaines, qu'ils soient mathématiques, économique et individuels pour les déplacements de la vie courante.

Description de l'Algorithme de Dijkstra, implémentation et traitement de quelques exemples.

Avant toute chose, partons d'un exemple concret afin de mieux comprendre l'utilité d'un tel algorithme :

En supposant que nous devions tous les jours réaliser le trajet Maison-Travail, quel chemin est le plus court en distance ?



L'algorithme de Dijkstra permettra de calculer par quels points (ici les croisements) passer afin d'aller d'un point à un autre avec un coût (ici la distance) le plus faible possible.

Afin de rendre les choses compréhensibles par le lecteur, nous allons d'abord expliquer le fonctionnement d'un tel algorithme, puis exposer quel a été le processus pour traduire cet algorithme du langage français vers le langage Processing de façon seyante.

Qu’est-ce que l’algorithme de Dijkstra ?

Selon Wikipedia[1];

> L'algorithme [de Dijkstra] prend en entrée un graphe orienté pondéré par des réels positifs et un sommet source. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.
>
> Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle. Le sous-graphe de départ est l'ensemble vide.
>
> Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de celui ajouté. La mise à jour s'opère comme suit : la nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté.
>
> On continue ainsi jusqu'à épuisement des sommets (ou jusqu'à sélection du sommet d'arrivée).

Approprions-nous les explications de Wikipédia avec notre exemple ci-dessus :

Premièrement, réalisons un graphe pondéré par des réels positifs, avec n la colonne des arêtes, i et f respectivement les sommets début et fin de chaque arrête, en sachant que 0 est ici notre sommet initial. Finalement, la colonne c correspond au poids de l'arête, ici il s’agit de nos distances, mais cela peut être la distance divisée par la vitesse autorisée afin d’avoir le chemin le plus rapide.
(Attention, le point numéro 1 dans notre schéma correspond au point numéro 0 dans notre algorithme.)

n	i	f	c
0	0	1	10.0
1	0	4	20.0
2	1	0	10.0
3	1	2	1.0
4	2	3	1.0
5	2	4	1.0
6	3	1	1.0
7	3	5	5.0
8	4	0	20.0
9	4	3	1.0
10	5	3	5.0

Secondement, comme indiqué dans la citation ci-dessus, il s'agit de construire un sous graphe classant les différents sommets par ordre croissant de leur distance minimale au sommet de départ.

	à **0**	à **1**	à **2**	à **3**	à **4**	à **5**
étape initiale	0 [1]	∞	∞	∞	∞	∞
0(0)		10.0 [2]	∞ [3]	∞	20.0	∞
1(10.0)		- [4]	11.0 [5]	∞	20.0 [6]	∞
2(11.0)		-	-	12.0 [7]	~~20.0~~ 12.0 [8]	∞
3(12.0)		-	-	-	13.0	17.0
4(13.0)		-	-	-	-	17.0
5(17.0)		-	-	-	-	- [9]

[1] : Il s'agit de notre sommet de départ.

[2] : Le chemin le plus court depuis 0 est vers 1.

[3] : Il n'y a pas de chemin de 0 à 2, la distance est donc virtuellement infinie.

[4] : Afin d'éviter de tourner en boucle on ne retraite plus ce sommet.

[5] : On additionne la valeur du chemin de 1 à 2 plus la valeur du chemin suivi actuellement.

[6] : On conserve la valeur car il n'y a pas de chemin de 1 à 4.

[7] : Nous avons deux chemins à valeurs égales, cela ne pose cependant aucun problème pour la suite de l'algorithme de choisir celui suivi au hasard.

[8] : Il y a un chemin de 2 à 4, qui, additionné au chemin suivi actuellement, est plus court que le chemin de 0 à 4.

[9] : On s'arrête car le sommet d'arrivée est dans le sous graphe.

Ainsi, la distance minimale de **0** à **5** est 17.0, et le chemin à suivre est trouvé grâce au prédécesseur de chaque sommet, soit : **5 - 3 - 2 - 1 - 0** pour aller de **0** à **5**.

Bien que peu visible sur cet exemple, les sommets sont ici rangés par distance croissante pour aller du sommet 0 au sommet s.

Implémentation de l'algorithme

Il s'agit maintenant d'implémenter cet algorithme sur Processing afin de pouvoir traiter plus efficacement les différents cas possibles, l'idée étant de n'avoir qu'à renseigner dans un tableau les différents chemins existants afin que le logiciel donne le chemin le plus court.

Afin de travailler efficacement, il a été important de diviser ce projet en étapes :

Cahier des charges :

1 : Usage des fonctions implémentées en TP

2 : Implémenter de nouvelles fonctions afin d'avoir un algorithme de Dijkstra fonctionnel

3 : Implémenter l'écriture et la lecture de matrices dans des fichiers

4 : Implémentation de l'interface graphique, du menu, et de la licence d'exploitation.

```
19
20  int[][] sum (int[][] M1, int[][] M2)
21  {
22    int[][] M3 = new int[M1.length][M1.length];
23    for (int i =0 ; i < M1.length ; i++)
24    {
25      for (int j =0; j < M1.length ; j++)
26      {
27        M3[i][j]=M1[i][j]+M2[i][j];
28      }
29    }
30    return M3;
31  }
32
33  boolean equality( int[][] M1, int[][] M2)
34  {
35    boolean ok = true;
36    for (int i =0 ; i < M1.length ; i++)
37    {
38      for (int j =0; j < M1.length ; j++)
39      {
40        if(M1[i][j]!=M2[i][j])
41        ok = false;
42      }
43    }
44    return ok;
45  }
46
47  int[][] transitivity_matrix(graph g)
48  {
49    int[][] multiplied = new int[g.n][g.n];
50    int[][] multiplier = new int[g.n][g.n];
51    int[][] after = new int[g.n][g.n];
52    multiplied = adjacency(g);
53    multiplier = multiplied;
54    after = multiplied;
55    for (int i = 0; i < g.n ; i++)//multiplications qum
56    {
57      if(i == 0) //Identity matrix
58      {
59        for (int j = 0; j < g.n ; j++)
60          for (int k = 0; k < g.n ; k++)
61            after[j][k]=0;
62        for (int j = 0; j < g.n ; j++)
63        {
64          after[i][i]=1;
65        }
66      }
67      else if(i == 1)
68      {
69        after = sum(multiplied,after);
70      }
71      else
72      {
73        multiplied = product(multiplied,multiplier);
74        after = sum(multiplied,after);
75      }
76    }
77    return after;
78  }
```

Les fonctions implémentées en TP.

Une première partie du travail fut achevé lors de la séance de TP et permis de créer les premières fonctions de notre programme.

Les matrices d'adjacence et de transitivité furent bien utile à l'algorithme de Dijkstra.

L’algorithme de Dijkstra.

Partie centrale de notre programme, cette fonction permet de retourner un tableau contenant le prédécesseur de chaque sommet.

Ce tableau est ensuite réutilisé pour afficher à l’utilisateur le plus court chemin d’un sommet à un autre.

```
int[] djikstra(float[][] source, int s, int S)
{
  //int s = 0;                 //s the starting vertex (S-1)
  //int S = 6;                 //S the number of vertexes
  float[] L = new float[S];    //L the table of n entries contraining distances from s to other edges
  int[] P = new int[S];        //P the table of n entries containing the pedecessor of each vertex in the shortest path of origin s
  IntList M = new IntList();   //M the liste of processed vertexes (so V\M the list of remaining vertexes to process)
  IntList V = new IntList();   //V the liste of all vertexes to process
  for(int v = 0; v < S;v++)    //fill V
    V.append(v);

  /*-----Initialisation-----*/

  L[s] = 0;
  P[s] = s;
  for(int i =0; i<S;i++)
  {
    if(i!=s)
    {
      if(source[s][i] != 1000000000)  // If i successor of s (if inequal to infinity(=1billion here))
      {
        L[i] = source[s][i];          // Enter weight of i in the distance table from s to the other vertexes
        P[i] = s;                     // i predecessor of s vertex
      }
      else
      {
        L[i]=1000000000;              //virtually infinity
        P[i]=-1;                      //virtually null
      }
    }
  }
  println(P);
  M.append(s);                        //s has been processed

  /*-----Reccurence-----*/

  while(!(M.size() == V.size()))      //let's process all vertexes
  {
    for(int v = 0; v<S ;v++)         // let's choose v in V\M
    {
      if(V.hasValue(v) && !M.hasValue(v))
      {
        for(int i = v; i<S ;i++)     // if i is in V\M such as i is a succesor of x -- this with i in V\M: L[v]≤L[i] (that's why in
        {
          if(V.hasValue(i) && !M.hasValue(i))//appartenance
          {
            if(source[v][i] != 1000000000);  //succession
            {
              if(L[i]>(L[v] + source[v][i]))
              {
                L[i] = L[v] + source[v][i];
                P[i] = v;
              }

            }
          }
        }
      M.append(v);
      }

      println(P);
      println("m:" + M);
      println(V);
    }
    println(M);
  }
  println(P);
  return(P);
}
```

L’écriture et la lecture de matrices dans des fichiers.

Le développeur a tenu à ce qu’il soit aisé de traiter rapidement plusieurs graphes.

Il est ainsi possible de copier coller les tableaux contenant les chemins possibles dans le dossier DATA.

```
// The following short CSV file called "StudiedGraph.txt" is parse
// in the code below. It must be in the project's "data" folder.
//
//n,i,f,c
//0,1,2,10.0
//1,1,5,20.0

Table StudiedGraph;

float[][] read_graph()
{
  StudiedGraph = loadTable("StudiedGraph.csv", "header");
  float[][] studiedTable = new float[StudiedGraph.getRowCount()][4
  println(StudiedGraph.getRowCount() + " total edges in table()");
  println("n_i_f_c");

  for (TableRow row : StudiedGraph.rows())
  {
    int n = row.getInt("n");
    int i = row.getInt("i");
    int f = row.getInt("f");
    float c = row.getFloat("c");
    studiedTable[n][0]=float(n);
    studiedTable[n][1]=float(i);
    studiedTable[n][2]=float(f);
    studiedTable[n][3]=c;
    println(n + "," + i + "," + f + "," + c);
  }

  return studiedTable;
}



void write_graph(graph g)
{
  StudiedGraph = new Table();
  StudiedGraph.addColumn("n");
  StudiedGraph.addColumn("i");
  StudiedGraph.addColumn("f");
  StudiedGraph.addColumn("c");

  TableRow newRow = StudiedGraph.addRow();
  for (int i =0; i< g.nbEdges;i++)
  {
    newRow.setInt("n",i);
    newRow.setInt("i",g.theEdges[i].initial);
    newRow.setInt("f",g.theEdges[i].finale);
    newRow.setFloat("c",g.theEdges[i].cost);
```

```
Button GNUGPL = new Button(width/2, int(4*height),3*width, height/4, #FF0077, #9605FF, "View more informations");
int menu = 0;
int[] graphic;
int nVertexes = 0;
int nEdges;
int start;

void draw()
{
  background(#FFF305);
  textSize(height/15);
  switch(menu) {
  case 0:             // first screen
    println("first");
    text("DJIKSTRA", width/3, height/5);  //title
    text("Choose what action to do", width/4, (2*height)/5);                  //question
    create_graph.display();                                                   //available cases
    read_graph.display();
    view_license.display();
    quit.display();
    if(create_graph.pushed())
      menu = 1;
    if(read_graph.pushed())
      menu = 2;
    if(view_license.pushed())
      menu = 3;
    if(quit.pushed())
      exit();
    break;
  case 1:
    println("create");   //create new graph
    char q = askChar("Are you sure ? (y/N)");
    if(str(q)=="y") || str(q)=="Y")
    {
      graph g = graph_entry();
      show_graph(g);
      show_matrix(adjacency(g), g);
      show_matrix(transitivity_matrix(g),g);
      write_graph(g);
    }
    menu = 0;
    break;
  case 2:
    println("read");   //read djikstra's shortest paths
    if(nVertexes == 0)
    {
      nVertexes = askInteger("Number of vertexes ? ~ points");
      nEdges = askInteger("Number of edges ? ~ lines");
      start = askInteger("Starting vertex ? (0 to " + (nVertexes-1) + ") for now must be 0");
    }
    float[][] edges = read_graph();
    //valuation(edges, nVertexes, nEdges);
    //djikstra(valuation(edges, nVertexes, nEdges));
    //predecesseur(djikstra(valuation(edges, nVertexes, nEdges)));
    graphic = djikstra(valuation(edges, nVertexes, nEdges), start, nVertexes);
    draw_predecessor(graphic);
    go_back.display();
    if(go_back.pushed())
      menu = 0;
    break;
  case 3:             //GNU Public licence
    println("license");
    textSize(height/20);
    text("This is program is under the GNU Public licence", width/20, height/5);
    textSize(height/40);
    text("Quick Summary : You may copy, distribute and modify the software as long as you track changes/dates in source files.", wi
    text("Any modifications to or software including (via compiler) GPL-licensed code must also be made available under the GPL". w
```

Implémentation de l'interface graphique, du menu, et de la licence d'exploitation.

Cette partie fut aisée étant donné que le développeur a repris le code d'un programme précédent.

Le développeur a en effet eu à réorganiser la position des boutons et à implémenter une fonction pour relier chaque sommet à son prédécesseur.

Difficultés rencontrées

Les principales difficultés furent d'aborder la notion de distance infinie, une perte de donnée et la boucle traitant tous les sommets.

Premièrement, il faut savoir que Processing n'a pas de fonction "infini", ainsi, pour des raisons pratiques nous considérons dans ce programme que l'infini est égal à un milliard, cependant cette valeur peut être augmentée.

Aussi, concernant la perte de données, son origine m'est encore inconnue, je suspecte un problème de mémoire ayant mis le désordre dans mes "annuler" et "refaire", j'ai en effet "perdu" quelques heures, mais cela a donné lieu à un programme selon moi plus clair. J'en ai aussi profité pour apprendre l'usage de l'outil "git" permettant de faire des sauvegardes du programme par branches de développement. L'usage de branches permet en effet de tester l'implémentation de nouvelles fonctionnalités sans toucher au programme principal tant que l'on n'est pas sûr de notre choix, l'avantage étant de pouvoir revenir en arrière sur un programme en gardant un code propre sans "code mort".

Enfin, j'ai eu quelques difficultés à savoir comment implémenter la sortie de la boucle traitant tous les sommets, le problème fut que je n'ai pas assez étudié le principe de l'algorithme de Dijkstra avant de commencer à le coder. D'où l'importance de bien procéder par étapes.

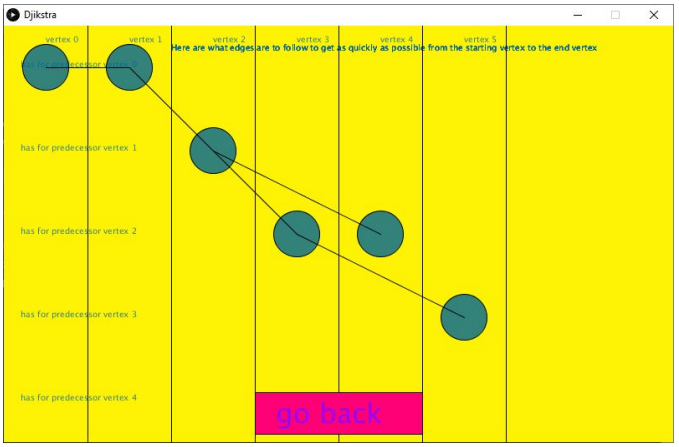
Finalement, c'est au moment de publier ce papier que je me rends compte avoir mal écrit Dijkstra dans tout mon programme, et tiens à présenter mes excuses à la personne ayant portée ce nom.

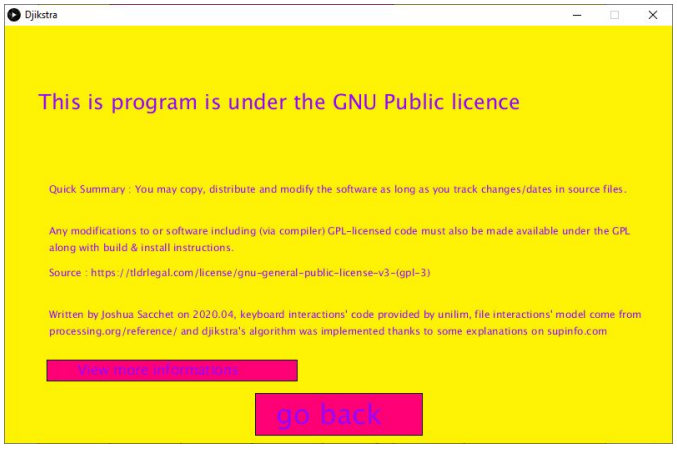
Le résultat final

L'utilisateur est accueilli par un menu proposant 4 actions :

- Créer un nouveau graphe (l'édition du fichier source permet de revenir en arrière sur les saisies : Dijkstra/data/StudiedGraph.csv)
- Lire le graphe
- Voir la licence
- Quitter l'application

Lire le graphe : Après renseignement du nombre de sommets, arêtes, et du sommet initial le plus cours chemin depuis le sommet initial vers chaque autre sommet sera affiché





Ce programme, tout comme ce mémoire, sont rédigés sous la licence publique GNU, toute personne est donc autorisée à utiliser, modifier et partager le logiciel et ses changements dans n'importe quel but. L'avis de licence est disponible avec les sources à cette adresse : https://github.com/josh-chez-sosh/Dijkstra

Gauches d’auteurs et sources :

Sources

Partie 1

(1.c) : Cours de fondements mathématique du semestre 1.

(1.c2) : Cours de Mathématiques appliqués à l’informatique.

(1.2): https://perso.univ-rennes1.fr/laurent.moret-bailly/docpedag/polys/MA2.pdf

(1.3): http://math.univ-lyon1.fr/~perrut/mass/cours1-4.pdf

(1.4): http://www.math.univ-toulouse.fr/~yak/Ens-Cardinaux.pdf

Partie 2

(c): Cours

(1): https://fr.wikipedia.org/wiki/Th%C3%A9orie_des_graphes

(2): http://www.bibmath.net/dico/index.php?action=affiche&quoi=./p/pont.html

(3): https://fr.vikidia.org/wiki/Probl%C3%A8me_des_sept_ponts_de_K%C3%B6nigsberg

(4): https://fr.wikipedia.org/wiki/Histoire_des_probabilit%C3%A9s

(5): Jean-Pierre Kahane, « Le mouvement brownien », *Société mathématique de France*, 1998, p. 123-155

(6): https://m6colorationgraphes.wordpress.com/2015/11/30/partie-iii-la-coloration-par-welsh-powel/

(7): https://www.maths-cours.fr/methode/algorithme-de-dijkstra-etape-par-etape/

(8): http://www.ens-lyon.fr/denif/data/algorithmique_enslyon/07-08-semestre1/td/td8_corrige.pdf

(9): https://members.loria.fr/JDumas/files/tipe/rapport.pdf

(I.1): http://www.guichetdusavoir.org/viewtopic.php?f=2&t=13858&view=print

(I.2): https://www.mathouriste.eu/Koenigsberg/koenigsberg.html

Partie 3

1 : https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

2 : https://dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc et https://www.maths-cours.fr/methode/algorithme-de-dijkstra-etape-par-etape/ afin d’implémenter l’algorithme.

Le programme et ce mémoire : https://github.com/josh-chez-sosh/Dijkstra

Auteurs

Riche Antonin
Sacchet Joshua

Date

le 16/05/2020