TP scilab : Arbre couvrant de poids minimal

IUT Paris Descartes, DUT Informatique, 1ère année

Théorie des graphes, 2017-18

Le but de ce TP est d'implanter sous scilab l'algorithme de Kruskal qui permet de déterminer un arbre couvrant de poids minimal associé à un graphe non orienté valué. Dans le TP n° 1, nous avons étudié différents formats de représentation d'un graphe non orienté et valué. Dans ce TP, nous considérerons qu'un graphe non orienté valué est défini par le nombre de ses sommets, la liste de ses arêtes et la liste des valuations associées aux arêtes.

Un fichier .sce pré-rempli est disponible sur Moodle, dans l'onglet 'TINF2U21 Graphes et langages' et dans la section 'TP MACHINE'.

1 Rangement des arêtes par ordre de poids croissant

En préambule, il faut classer les arêtes par ordre de poids croissant. Dans un script scilab, on pourra ainsi définir une fonction `rangement` qui réalise cette opération et la tester sur des graphes simples.

```
clear
function [A,V]=rangement(arete_A,value_A)
   // ENTREES :
   //   - arete_A : liste des aretes (sous forme matricielle)
   //   - value_A : liste des valuations
   // SORTIES :
   //   - A : liste des aretes (rangee par ordre croissant de valuations)
   //   - V : liste des valuations rangees par ordre croissant
   ...
   ...
endfunction
```

Note. On pourra utiliser la fonction `gsort` qui classe une liste de réels par ordre de poids décroissant (!). Pour en savoir davantage sur cette fonction de scilab, on pourra utiliser l'aide scilab en exécutant dans la console la commande : `> help gsort`

2 Algorithme de Kruskal

On dispose désormais d'une description des arêtes par ordre croissant des valuations. L'algorithme de Kruskal repose sur l'idée de construire l'arbre couvrant de poids minimal au fur et à mesure, en choisissant une arête de poids minima qui préserve le caractère *acyclique* de graphe qui en résulte.

À chaque étape, soit on rejette l'arête considérée, soit on l'ajoute. Si on l'ajoute, deux possibilités :

- soit l'arête se retrouve associée à une composante connexe déjà existante ;
- soit l'arête se retrouve isolée et constitue ainsi une nouvelle composante connexe.

Dans tous les cas, à chaque étape, le graphe provisoire (avant convergence de l'algorithme) est composée d'une ou plusieurs composantes connexes. Dès lors, *comment savoir si l'ajout de l'arête préserve ou*

Scanned by CamScanner

pas l'acyclicité ? La réponse est simple : si les deux extremités de l'arête considérée possèdent le même identifiant de composante connexe, alors l'ajout de l'arête crée un cycle et il faut donc rejeter l'arête.

En pratique, il faut construire une méthode qui définit au fur et à mesure un *identifiant de composante connexe* à laquelle appartient chaque sommet du graphe. Le programme pourrait être structuré de la façon suivante :

```
function [p,L]=Kruskal(N,A,V)

// ENTREE :
//     - N : nombre de sommets du graphe
//     - A : liste des aretes rangees par ordre croissant de poids
//     - V : liste des valuations des aretes rangees par ordre croissant
// SORTIE :
//     - p : poids de l'arbre couvrant de poids minimal
//     - L : liste des aretes qui definissent l'arbre couvrant de poids minimal

// INITIALISATION :
// Le graphe n'a aucune arete : les aretes n'appartiennent a aucune
// composante connexe et l'identifiant de chaque sommet est assigne a 0 :
...

// BOUCLE SUR LES ARETES
for i=1:nA
    // Pour chaque extremite de l'arete i, determiner l'identifiant de la
    // composante connexe a laquelle elle appartient
    ...

    // Cas 1 : les deux extremites de l'arete consideree ne sont pas associes a
    // une composante connexe existante.
    ...

    // Cas 2 : pour l'arete consideree, une extremite est associee a une
    // composante connexe existante tandis que l'autre extremite ne l'est pas.
    ...

    // Cas 3 : les deux extremites de l'arete appartiennent chacune a une
    // composante connexe existante (pas necessairement la meme)

        // Cas 3.1 : soit les deux composantes connexes sont differentes.
            ...
        // Cas 3.2 : soit les deux composantes connexes sont identiques.
            ...
end

...

endfunction
```

Implanter sous scilab l'algorithme de Kruskal et valider l'algorithme sur des exemples du cours ou des séances de TD.

Scanned by CamScanner

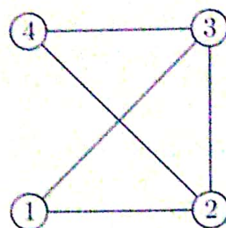
TP scilab : Stockage de graphes sur machine

IUT Paris Descartes, DUT Informatique, 1ère année

Théorie des graphes, 2017-18

1 Description d'un graphe non orienté

Pour stocker un graphe en machine, il est nécessaire de connaître le nombre de sommets et la liste des arêtes (sous une forme donnée). Considérons le graphe associé à cette représentation :



Nous allons décrire plusieurs façons de décrire ce graphe de référence sous scilab (dans la suite, le suffixe `_ref` renvoie explicitement à ce graphe de référence).

- **Matrice d'adjacence.** Le graphe peut être défini par la matrice d'adjacence

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

  Sous scilab, cette matrice est définie par la commande :

```
M_ref=[0,1,1,0; 1,0,1,1; 1,1,0,1; 0,1,1,0]
```

- **Liste d'arêtes.** Le graphe peut être défini par la donnée du nombre de ses sommets, `N_ref=4`, et de la liste d'arêtes $[(1,2),(1,3),(2,3),(2,4),(3,4)]$. Sous scilab, cette liste pourra être stockée sous la forme matricielle suivante (d'autres choix sont possibles !) :

```
A_ref=[1,2; 1,3; 2,3; 2,4; 3,4]
```

- **Liste d'adjacence.** Le graphe peut être défini par la donnée du nombre de ses sommets, `N_ref=4`, et de la liste d'adjacence $[[2,3],[1,3,4],[1,2,4],[2,3]]$. Cette liste de listes contient l'information suivante : la liste n° i de la liste d'adjacence décrit l'ensemble des voisins du sommet i. Sous scilab, cette liste peut être stockée sous la forme matricielle

```
L_ref=[2,3,0; 1,3,4; 1,2,4; 2,3,0]
```

  où, afin de conférer à `L_ref` une structure matricielle, nous avons ajouté des 0 en l'absence de voisins supplémentaires (**Note :** les sommets sont numérotés de 1 à `N_ref` ; 0 n'est donc pas un sommet et aucune confusion n'est donc possible dans la lecture de `L_ref`).

Remarque. Rigoureusement, le nombre de sommets n'est pas nécessaire si l'on définit les arêtes par la matrice d'adjacence. Mais l'information est nécessaire si on définit les arêtes par liste d'arêtes ou liste d'adjacence : en effet, le(s) dernier(s) sommet(s) pourrai(en)t être isolé(s).

Scanned by CamScanner

2 Stockages sur machine d'un graphe non orienté

Nous allons travailler avec des graphes non orientés représentés par liste d'adjacence, matrice d'adjacence ou liste d'arêtes. Le but de ce TP est de se familiariser avec ces représentations en définissant des procédures de conversion d'un format à un autre.

a) Dans un script `TP1.sce`, définir le graphe de référence dans les différents formats présentés :

```
clear
N_ref=4;                                      // nombre de sommets
M_ref=[0,1,1,0; 1,0,1,1; 1,1,0,1; 0,1,1,0]; // matrice d'adjacence
A_ref=[1,2;1,3;2,3; 2,4; 3,4];                // liste d'aretes
L_ref=[2,3,0;1,3,4; 1,2,4; 2,3,0];            // liste d'adjacence
```

Afficher ces formats dans scilab en lançant le script (exécuter la commande '`exec TP1.sce`' dans la console).

b) Construire une fonction scilab, `listadjTOmatadj.sci`, qui permet de renvoyer la matrice d'adjacence associée à une liste d'adjacence donnée. Pour ce faire, on intégrera la fonction dans le script puis on testera la fonction sur le graphe de référence :

```
function [M]=listadjTOmatadj(N,L)
      ...
endfunction
M=listadjTOmatadj(N_ref,L_ref)
```

Pour valider la fonction, on comparera `M` et `M_ref` (qui doivent être identiques!). De façon analogue, construire la fonction réciproque `matadjTOlistadj.sci` qui permet de renvoyer la liste d'adjacence associée à une matrice d'adjacence donnée.

c) Construire une fonction scilab, `listadjTOlistaretes.sci`, qui permet de renvoyer la liste des arêtes associée à une liste d'adjacence donnée. Construire la fonction réciproque `listaretesTOlistadj.sci`

d) Construire une fonction scilab, `matadjTOlistaretes.sci`, qui permet de renvoyer la liste des arêtes associée à une matrice d'adjacence donnée. Construire également la fonction `listaretesTOmatadj.sc`

Chaque fonction sera validée à l'aide du graphe de réféence.

3 Stockages sur machine des graphes valués

e) Comment définir, de manière simple, un graphe non orienté valué en conservant les structures définies précédemment ? Adapter les fonctions précédentes afin de traiter des graphes non orientés valués.

f) Reprendre les définitions et procédures précédentes afin de traiter le cas des graphes orientés valués.

Scanned by CamScanner

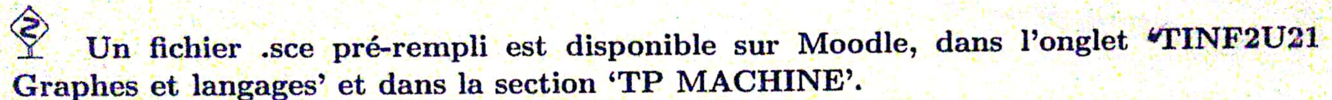
TP scilab : Algorithmes de plus courts chemins

IUT Paris Descartes, DUT Informatique, 1ère année

Théorie des graphes, 2017-18

Le but de ce TP est d'implanter sous scilab l'algorithme de Dijkstra qui permet de déterminer, sous conditions, un plus court chemin entre un sommet choisi et tous les autres sommets du graphe. Dans le TP n° 1, nous avons étudié différents formats de représentation d'un graphe orienté et valué. Dans ce TP, nous considérerons qu'un graphe orienté valué est défini par sa matrice d'adjacence et sa matrice de valuation (il est toujours possible de le définir avec un autre format et de le convertir dans le format souhaité à l'aide d'une fonction implantée dans le TP n° 1).

Implanter l'algorithme de Dijkstra en scilab. On validera la fonction `Dijkstra` avec un graphe dont les valuations sont toutes positives. On testera l'algorithme sur un graphe dont les valuations ne sont pas toutes positives.



Un fichier .sce pré-rempli est disponible sur Moodle, dans l'onglet 'TINF2U21 Graphes et langages' et dans la section 'TP MACHINE'.

Le script pourrait ainsi avoir la structure suivante :

```
clear

function [d,P]=Dijkstra(M,f)
   // ENTREES : 'M' matrice d'adjacence, 'f' matrice de valuation
   // SORTIES : 'd' tableau des distances, 'P' tableau des peres
   ...
   ...
   ...
endfunction

// Definition du graphe (matrice d'adjacence et matrice de valuation) :
Mat=...;
Val=...;

//Validation :
[d,P]=Dijkstra(Mat,Val)
```

Note. Un exemple issu du cours permet de définir un graphe dont les valuations sont toutes positives

$$\mathtt{Mat} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \quad \mathtt{Val} = \begin{pmatrix} +\infty & 8 & 6 & 2 \\ +\infty & +\infty & +\infty & +\infty \\ +\infty & 3 & +\infty & +\infty \\ +\infty & 5 & 1 & +\infty \end{pmatrix}$$

et un graphe possédant une valuation negative :

$$\mathtt{Mat} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \quad \mathtt{Val} = \begin{pmatrix} +\infty & 3 & 4 \\ +\infty & +\infty & +\infty \\ +\infty & -2 & +\infty \end{pmatrix}$$

Scanned by CamScanner