Algorithmes pour les graphes

1 Définitions

Un graphe est représenté par :

- V : L'ensemble des noeuds ou sommets.
- E : L'ensemble des arcs ou arrêtes.

E est un sous-ensemble de VxV.

On note $G = (V, E)$.

Si G est connecté, alors $|E| \geq |V| - 1$.

2 Structures de Données

Dans cette partie, nous allons étudier les structures de données permettant de représenter des graphes.

2.1 Matrice d'adjacence

C'est une matrice définie comme suit :

$$\begin{cases} A_{i,j} = 1 \text{ si } (i,j) \in E \\ A_{i,j} = 0 \text{ si } (i,j) \notin E \end{cases}$$

C'est une structure facile à utiliser, qui convient aux graphes denses, c'est à dire lorsque $|E|$ approche $O(|V|^2)$.

L'inconvénient majeur est la consommation de mémoire quelque soit le graphe, puisqu'il faut une taille en $O(|V|^2)$.

2.2 Liste d'adjacence

On utilise un tableau de $|V|$ listes. Pour chaque sommet v, on stocke $Adj(v)$.

Cette structure convient aux graphes peu denses, et la quantité de mémoire utilisée est $O(|V| + |E|)$. Toutefois, en contrepartie, sa mise en oeuvre n'est pas aisée.

2.3 Matrice d'incidence

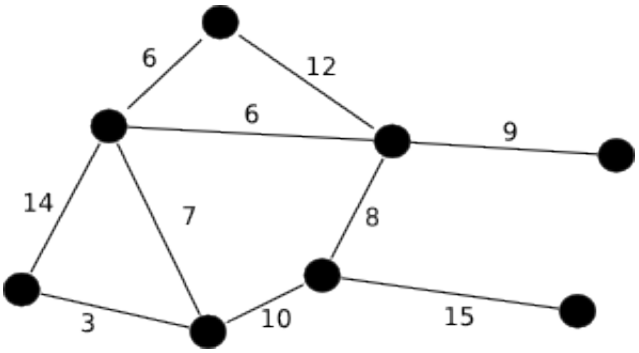
Il s'agit d'une matrice de taille $|E| \cdot |V|$. Une entrée $b_{i,j}$ de cette matrice vaut soit 0, soit 1 si l'arc commence en j, et -1 si l'arc se termine en j.

La mise en oeuvre est très facile, mais la mémoire utilisée est $O(V^3)$.

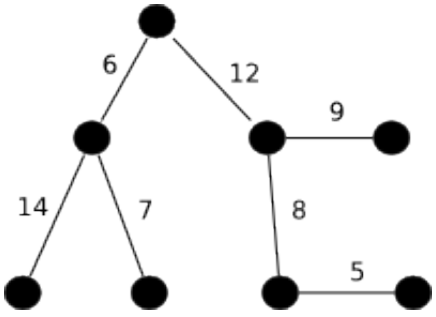
3 Arbre de recouvrement minimum

ou Minimum Spanning Tree (MST).
Soit $G = (V, E)$, un graphe indirect pondéré. Un arbre de recouvrement est un arbre qui contient tous les sommets de G.

Exemple :



L'arbre de recouvrement minimum obtenu donne :



Un arbre de recouvrement minimum cherche à minimiser :

$$w(T) = \sum_{u,v \in V} w(u, v)$$

3.1 Algorithme de Kruskal

- En entrée : le graphe G
- En sortie : L'entrée T

Etape 1 : T est initialisé à Vide.
On trie les arbres de G en ordre croissant selon leur poids dans une liste L.

Etape 2 : On sélectionne le premier arc de L et on l'inclut dans T.

Etape 3 :

```
Si tous les arcs de L on été examinés Alors
  Retourner "Graphe non connecté"
Sinon
  On prend le premier arc non examiné de L
  S'il se forme un égal dans T
    On l'ajoute à T et on va à l'étape 4.
  Sinon
    On supprime cet arc de L et on recommence l'étape 3
```

Etape 4 : On arrête le traitement si T a $(n-1)$ arcs, sinon on retourne à l'étape 3.

Le coût de cet algorithme sera étudié en TD.

3.2 Algorithme de Prim

- En entrée : G
- En sortie : T

Etape 1 : On sélectionne un sommet quelconque de V et on l'insére dans T.

Etape 2 : Soit S l'ensemble de sommets de T. Si les deux ensembles S et $(V-S)$ sont connectés alors on renvoie "Graphe non convexe".
On cherche un arc de poids minimum connectant S et $(V-S)$ s'ils ne forment pas un cycle dans T, on l'insére dans T et on va à l'étape 3.

Etape 3 : Si T a $n-1$ arcs on retourne T. Sinon, on retourne à l'étape 2.

La encore, se réferrer au TD correspondant pour l'analyse du coût.

3.3 Parcours en largeur

Le P.L. est la base de nombreux algorithmes importants sur les graphes. Dijkstra pour le calcul de plus court chemin a pour origine unique Prim afin de déterminer l'arbre de recouvrement minimum.

Principes : Etant donné un graphe $G = (S, A)$, et une origine s. Le P.L. emprunte systématiquement les arcs G pour découvrir tous les sommets accessibles depuis s. Il calcule la distance entre s et tous les sommets accessibles. Il construit également une arborescence de P.L. de racine s.

L'algorithme de P.L. découvre d'abord tous les sommets situés à une distance k de s, avant de découvrir tous les sommets situés à la distance $k + 1$.

Le P.L. colorie chaque sommet en blanc, gris ou noir. Tous les sommets sont blancs au départ. Un sommet est découvert la première fois qu'il est rencontré au cours de la recherche.

Les sommets gris ou noirs ont été déjà découverts. Si $(U, V) \in A$ et si le sommet U est noir alors le sommet V est gris ou noir. Les sommets gris peuvent avoir des sommets adjacents blancs. c'est ma frp,toère entre les sommets découverts et les sommets non découverts.

Algorithme :

```
Pour chaque sommet u dans S[G]
  Faire couleur[u] <- blanc
  j[u] <- infini
  p[u] <- NULL
FinPour

s <- gris
j[s] <- 0
p[s] <- NULL
Tant que F != 0 Faire
  u <- tete[p]
  Pour chaque v dans Adj(u) Faire
    Si couleur[v] <- gris Alors
      d[v] <- d[u] + 1
      p[v] <- u
    Finsi
  FinPour
FinTantQue
Enfile(F,V)
Defile(F)
```

3.4 Parcours en profondeur

C'est un algorithme de recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.

Le nom d'algorithme en profondeur est du au fait que, contrairement à l'algorithme de parcours en

largeur, il explore en fait “à fond” les chemins un par un : pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'aie plus de voisins (ou que tout ses voisins soient marqués), et revient alors au sommet père.

Si G n'est pas un arbre, l'algorithme pourrait tourner indéfiniment, c'est pour cela que l'on doit en outre marquer chaque sommet déjà parcouru, et ne parcourir que les sommets non encore marqués.

Enfin, on notera qu'il est tout à fait possible de l'implémenter itérativement à l'aide d'une pile LIFO contenant les sommets à explorer : on dépile un sommet et on empile ses voisins non encore explorés.

Code de l'algorithme :

```
DFS (graphe G, sommet s):
{
Marquer(S);
Debut
PourTout éléments fils de Voisin(s) Faire
   Si NonMarqué(sfils) Alors
     DFS(G,sfils);
   FinSi
FinPour
Fin
}
```

3.5 Algorithme de Dijkstra

Cet algorithme resoud le problème de la recherche d'un plus court chemin à origine unique pour un graphe orienté pondéré $G(S, A)$ dans le cas où tous les arcs sont de poids positifs ou nuls.

L'algorithme de Dijkstra gère un ensemble E de sommets dont les longueurs finales de plus court chemin à partir de l'origine s ont été calculées.

A chaque itération, l'algorithme choisit le sommet dont l'estimation de plus court chemin est minimale.

Détail de l'algorithme :

- Les poids de tous les sommets sont initialiés à ∞
- Les sommets sont mis à blanc
- Les prédecesseurs sont mis à NULL

File d'attente priorité F. Le poids du sommet u initial est mis à 0. Ajouter u à f.

```
TantQue F != 0 Faire
  u = sortir le premier sommet de F
  mettre u à noir
```

```
  Pour Tous les successeurs s de u Faire
    mise à jour du poids du sommet d'arrivée
    Si le sommet s n'est pas dans F
      Ajouter s dans F
    FinSi
  FinPour
FinTantQue
```

3.6 Algorithme de Bellman-Ford

Cet algorithme permet de résoudre le problème des plus courts chemins à origine unique dans le cas général où les arcs peuvent avoir un poids négatif.

Principe : Soit $G = (S, A)$ un graphe orienté pondéré. soit $w : A \longrightarrow \mathbb{R}$, fonction de pondération et s une origine.

L'algorithme utilise la technique de relachement diminuant progressivement une estimation $d[v]$ du poids d'un plus court chemin depuis l'origine s vers chaque sommet $v \in s[G]$ jusqu'à atteindre la valeur réelle du poids de plus court chemin $S(s, V)$.

L'algorithme retourne VRAI si et seulement si le graphe ne contient aucun circuit de longueur strictement négatif accessible depuis l'origine.

Code :

```
initialisation ( G, s)  // les poids de tous les sommets sont mis à +infini
                          // le poid du sommet initial à 0 ;
  pour i=1 jusqu'à Nombre de sommets -1 faire
   |    pour chaque arc (u, v) du graphe faire
   |     |  paux := poids(u) + poids(arc(u, v));
   |     |  si paux <poids(v) alors
   |     |    |  pred(v) := u ;
   |     |    |  poids(v) := paux;
  pour chaque arc (u, v) du graphe faire
   |   si poids(u) + poids(arc(u, v)) <poids(v) alors
   |     retourner faux
  retourner vrai
```

4 Analyse des algorithmes

4.1 Parcours en largeur (Breadth First Search)

Nous avons réutilisé une queue de priorité FIFO. Les opérations d'enfilage et de défilage coûtent $O(1)$.

Le coût total du parcours entier est linéaire.

Remarque : BFS permet de construire un BFSTree, i.e. un arbre de pracours en largeur d'abord. Il permet en outre de calculer la plus petite distance séparant le sommet source vers un autre sommet du graphe lorsque celui-ci n'est pas pondéré. L'algorithme de Dijkstra et de Prim utilisent le même principe.

4.2 Parcours en profondeur (Depth First Search)

Nous pouvons utiliser une pile (LIFO) ou un algorithme récursif.

Contrairement au BFS, le DFS permet de construire une forêt d'arbres de DFS.

4.3 Algorithmes du MST

Soit $G = (V, E)$, il s'agit de trouver $T = (V, E)$. i.e. : connecte tous les sommets du graphe tout en minimisant le coût $w(t)$.

En ce qui ceoncerne L'algorithme de Kruskal, cela dépend de la structure de données utilisée et des coûts associés aux opérations de base, et du graphe (s'il comporte des ensembles disjoints).

Pour Prim, le coût de l'algorithme dépend de la S.D. implémentant le graphe (soit un tableau, soit un tas binaire, soit un tas de Fibonacci). S'il s'agit d'un tas binaire, le coût est $O(E \cdot log(V))$.