1: LES ABR

Arbre rouges noires

insert:

- cas 0:

Le noeud père p est la racine de l'arbre Le noeud père devient noir

- cas 1:

Le frère y du père de z est rouge Les noeuds p et f deviennent noirs et leur père devient rouge

- cas 2:

Le frère y du père de z est noir et z est le fils droit de son père rotation gauche depuis son père -> cas 3

- cas 3:

Le frère de y du père de z est noir recoloration et rotation droite depuis le père de son père

delete:

- cas 1:

Le frère de x est rouge recoloration et rotation gauche - cas 2:

Le frère de x(w) est noir et le fils droit de w est noir recoloration (le double noir remonte, seul cas qui peut se répéter)

- cas 3:

Le frère de x (w) est noir et le fils droit de w est noir (le fils gauche est rouge) recoloration et rotation à droite -> cas 4

- cas 4:

Le frère de x (w) est noir et le fils droit de w est rouge recoloration et rotation à gauche

2: LES GRAPHES

Roy Warshall:

```
A = matrice d'adjacence du graphe
```

```
for(k=0 ; k < ordre du graphe ; ++k)
    for(i=0 ; i < ordre du graphe ; ++i)
        if(A[i][k])
            for(j=0 ; j < ordre du graphe ; ++j)
                A[i][j]=A[i][j] or A[k][j]
```

lorsque k = 0, recherche d'un chemin de longueur 2 entre les sommets i et j passant par le sommet 0, si il n'existe pas de chemin de longueur 1 *complexité maximale*: O(n^3)

3: CYCLES ET CONNEXITE

complexité: tous les algos de ce chapitre sont en O(|S| + |A|)

composantes connexes:

```
void parcours()
{
    for(int k=0 ; k < ordre du graphe ; ++k)
        val[k]=0
    for(int k=0 ; k < ordre du graphe ; ++k)
        if(val[k] == 0)
            explore(k)
}
void explore(int k)
{
    ++id
    val[k]=id
    inval[id-1]=k
    for(sommet \*t=adj[k] ; t != null ; t=t->suivant)
        if(val[t->sommet] == 0)
            explore(t->sommet)
}
```

complexité: O(|S| + |A|)

points d'articulation:

```
int explore(int k)
{
    int min

    ++id
    val[k]=id
    min=id
    for(t=1er adjacent de k ; t existe ; t=adjacent suivant)
    {
        if(val[t->sommet] == 0)
        {
            m=explore(t->sommet)
            if(m < min)
```

```
                min=m
            if(m >= val[k])
                cout << k
        }
        else
            if(val[t->sommet] < min)
                min=val[t->sommet]
    }
    return(min)
}
```

fonctionnement:

Liste de successeurs : adj vecteur de pointeurs avec

sommet : numéro du sommet,

suivant : pointe vers l'élément suivant de la liste . Vecteur val : numéro d'ordre des sommets lors du parcours en profondeur du graphe.

L'algorithme détermine le « plus haut »sommet dans l'arborescence que l'on peut atteindre depuis chaque descendant du sommet k. On teste, pour chaque sommet k, si le sommet le plus « élevé », ayant le numéro d'ordre le plus petit, accessible depuis un successeur du sommet k est situé plus haut que ce sommet k dans l'arborescence. Un sommet s est situé plus haut que le sommet k si val[s] est plus petit que val[k].

complexité: O(|S| + |A|)

détection des cycles:

```
int id=0,cnt=0
int pre[dim]={-1,...,-1}
int post[dim]={-1,...,-1}
void main()
{
    for(int v=0 ; v < ordre du graphe ; ++v)
        if(pre[v] == -1)
            cycle(v)
}

void cycle(int k)
{
    int t
    ++id
    pre[k]=id
    for(t=1er adjacent de k ; t existe ; t=adjacent suivant)
    {
        if(pre[t] == -1) cycle(t)
        else if(post[t] == -1) cout << ''Cycle détecté''
    }
    ++cnt
    post[k]=cnt
```

```
}
```

fonctionnement:

l'idée ici est de prendre un sommet et tous ses descendants. Lorsqu'on traite un sommet, on modifie la valeur qu'il y a dans pre. Ensuite on va traiter ses descendants. Une fois qu'on a traiter complètement un sommet et tous ses descendants, on change la valeur de post Cependant, si pendant le traitement des descendants on retombe sur le sommet, post ne sera pas encore modifier. Il y a donc un cycle.

complexité: O(|S| + |A|)

Composantes fortement connexes:

```
int id=0,sid=0
int pre[dim]={-1,...,-1}
int low[dim]={0,...,0}
int comp[dim]={0,...,0}

void main()
{
    for(int v=0 ; v < ordre du graphe ; ++v)
        if(pre[v] == -1) explore(v)
}
void explore(int k)
{
    int t,min
    ++id
    pre[k]=id
    low[k]=id
    min=id
    push(k)
    for(t=1er adjacent de k ; t existe ; t=adjacent suivant)
    {
        if(pre[t] == -1) explore(t)
        if(low[t] < min) min=low[t]
    }
    if(min < low[k]) low[k]=min
    else
    {
        do
        {
            t=pop()
            comp[t]=sid
            low[t]=infini
        }
        while(t != k)
        ++sid
    }
}
```

variables:

pre: indique le marquage

low: indique les plus "hauts" descendants

comp: reprend les composantes (les sommets d'une même composante ont la même valeur)

fonctionnement:

explore stocke dans low à l'indice du successeur considéré un sommet dont le numéro d'ordre est plus petit que le numéro d'ordre du sommet k celui-ci j'aurais un peu de mal à l'expliquer mais le tout est de bien lire l'algo en essayant sur un graphe et vous comprendrez très bien l'idée.

complexité: O(|S| + |A|)

4: TRI TOPOLOGIQUE

Tri topologique:

```
void tri(graphe G)
{
    int pred[dim]={0,...,0}
    liste L;
    queue Q

    for(int i=0 ; i < ordre du graphe ; ++i)        //initialisation du nombre de
prédecesseur pour chaque noeud
        for(int j=adj[i] ; j != null ; j=adjacent suivant)
            ++pred[j]

    for(int i=0 ; i < ordre du graphe ; ++i)        //recherche des noeud n'ayant
aucun prédecesseur
        if(pred[i] == 0) Q.put(i)                   //si trouvé, on le met dans la
queue

    while(Q n'est pas vide)
    {
        int i=Q.get()
        //insérer le sommet i en fin de liste L
        for(int j=adj[i] ; j != null ; j=adjacent suivant)
        {
            --pred[j]
            if(pred[j] == 0) Q.put(j)               //si en supprimant le
prédecesseur il est == 0, on l'ajoute dans la queue
        }
    }
}
```

complexité: O(|S| + |A|)

5: LES MST (minimal spanning tree)

Prim (arbre sous-tendant minimal):

```
wt vecteur d'entiers initialisés à l'infini
mst vecteur de pointeurs vers arête initialisés à null
edge vecteur de pointeurs vers arête initialisés à null

min=1
for(v=0 ; min != 0 ; v=min)
{
    min=0
    for(w=1 ; w < nombre de sommets ; ++w)
    {
        if(mst[w] == null)
        {
            if(arête(v,w) existe)
            {
                if(poids arête(v,w) < wt[w])
                {
                    wt[w]=poids arête(v,w)
                    edge[w]=pointeur vers l'arête(v,w)
                }
            }
            if(wt[w] < wt[min])
                min=w
        }
    }
    if(min != 0)
        mst[min]=edge[min]
}
```

Prim va prendre un quelconque sommet et regardé parmis toutes les arêtes entre ce sommet et tous les autres laquel est la moins lourdes On ajoute donc le sommet à l'arbre et on recommence avec toutes les arêtes entre l'arbre en construction et le graphe (et ainsi de suite).

Complexité: O(|S|^2)

Si heap maximale: O(|A|*log|S|)

Kruskal:

```
mst vecteur de pointeurs vers arête initialisé à null

k=1
création du heap d'arêtes sur base des poids
for(i=0 ; i < nombre d'arêtes du graphe ; ++i)
{
    min=suppressionMinHeap()
    if(min ne produit pas de cycle avec les arêtes
        déjà dans mst[1...k-1])
```

```
    {
        union(min->origine,min->destination)
        mst[k]=min
        ++k
    }
}
```

Kruskal va prendre l'arête ayant le poids le plus faible et va l'ajouter si elle ne crée pas un cycle dans l'arbre en construction.

Complexité: O(|A|*log|S|)

--> Si graphe dense, Prim(car ne passe en revue que les sommets), si bcp de sommets et peu d'arêtes, Kruskal.

6: LES PLUS COURTS CHEMINS:

Dijkstra:

```
M=ensemble vide
for(i=1; i<= ordre du graphe; ++i)
{
dist[i]=poids arc(s,i)
prec[i]=s
rajouter le sommet i à l'ensemble M
}
retirer le sommet s de l'ensemble M
while(l'ensemble M n'est pas vide)
{
    m=sommet x appartenant à M tel que dist[x] est minimum
    retirer le sommet m de l'ensemble M
    if(dist[m] == infini)
        M=ensemble vide
    else
    {
        for(a=1er arc sortant de m ; a existe ; a=arc suivant)
        {
            y=extremité de a
            if(y appartient à M)
            {
                v=dist[m]+poids arc(m,y)
                if(v < dist[y])
                {
                    dist[y]=v
                    prec[y]=m
                }
            }
        }
    }
}
```

dist[i] indique la plus courte distance connue entre le sommet s et le sommet i

prec[i] reprend le numéro du sommet précédent le sommet i sur le plus court chemin en construction etre les sommet s et i

initialement:

dist[i] = poids de l'arc |e| i et s (si il n'existe pas: infini)

prec[i] = s (pour tout i)

M = ensemble des tous les sommets

fonctionnement:

on prend le sommet compris dans M et ayant la dist la plus petite;

on retire ce sommet de m;

on regarde tous ses voisins qui sont aussi dans M;

si dist[m] + poids de l'arete avec son voisin < dist[voisin], on update dist[voisin] et prec[voisin] = m;

On recommence avec tous les sommets dans M;

Si distance m = infini, on arête la recherche, cela signifie qu'on a cherché le plus loins possible

complexité: O(|A| log |S|)

Floyd:

```
for(k=0 ; k < ordre du graphe ; ++k)
    for(x=0 ; x < ordre du graphe ; ++x)
        if(mat[x][k] < infini)
            for(y=0 ; y < ordre du graphe ; ++y)
                if(mat[k][y] < infini)
                    if(mat[x][k]+mat[k][y] < mat[x][y])
                        mat[x][y]=mat[x][k]+mat[k][y]
```

fonctionnement:

l'idée ici est est de se dire que si on a la distance la plus courte entre x et k et entre k et y, la distance la plus courte entre x et y est la distance entre x et k+ la distance entre k et y (sauf si la distance actuelle entre x et y est plus petite).

remarque:

ici il n'y a que les distances, on ne peut pas savoir par où passe le chemin le plus court

complexité: O(|S|^3)

Bellman:

```
dist[0 ... n-1] = +infini
dist[s] = 0
renuméroter les sommets dans l'ordre topologique
for(int k=1 ; k < ordre du graphe ; ++k)
{
    j = i tel que pour tous les arcs (i,k) dans le graphe
          on ait dist[i]+mat[i][k] qui soit minimum
    dist[k] = dist[j]+mat[j][k]
    pred[k]=j
}
```

fonctionnement:

on réorganise les sommets dans l'orgre topologique, ensuite on traite les sommets dans ce nouvel ordre (le premier sommet étant la source donc dist[source] = 0). Pour chaque sommet, étant donné qu'on a trouvé le chemin le plus cours pour tous ces prédecesseurs, il suffit de choisir le prédecesseur dont la dist[prédecesseur]+poids(prédecesseur, actuel) est la plus petite.

complexité: O(|S| + |A|)

Bellman et Ford:

```
mettre le poids de tous les sommets à +infini
mettre le poids du sommet initial à 0
for(int i=1 ; i < ordre du graphe ; ++i)
{
    for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
    {
        val=poids du sommet u + poids de l'arc (u,v)
        if(val < poids du sommet v)
        {
            pred[v]=u
            poids du sommet v=val
        }
    }
}
for((u,v)=1er arc du graphe; (u,v) existe ; (u,v)=arc suivant)
    if(poids du sommet u + poids de l'arc (u,v) < poids du sommet v)
        return(false)
return(true)
```

fonctionnement:

on passe en revue tous les arcs n-1 fois (n = nombre de sommets). On recherche donc d'abord les chemins de longueur 1 depuis la source puis de deux,... un dernier test est fait pour savoir si il y a

un cycle négatif (ce qui implique qu'il n'y a pas de plus cours chemin)

complexité: O(|S| + |S|*|A|) approximé à O(|S|*|A|)

7: ALGORITHMES GENERIQUES SUR LES GRAPHES:

Canevas générique pour certains algorithmes gloutons:

```
for(int i=0; i < ordre du graphe ; ++i)
    val[i]=-infini
prec[0]=0
mettre (0,0) sur le heap

for (int n=1 ; heap n'est pas vide ; ++n)
{
    s=extraire le minimum du heap
    val[s]=-val[s]
    t=premier successeur du sommet s
    while(t existe)
    {
        if(val[t] < 0)
        {
            insérer (t,Prior) sur le heap
            if(heap a été modifié)
            {
                val[t]=-Prior
                prec[t]=s
            }
        }
        t=successeur suivant de s
    }
}
```

principe des algorithmes gloutons:

ajouter successivement des éléments à une solution en cours de construction, suivant des critères précis jusqu'à obtenir une solution complète optimale (en fait un grand nombre des algos qu'on a vu sont des algos gloutons) cependant tous les algorithmes gloutons ne s'instancient pas aisément à partir de ce canevas.

complexité: O(|S| + |S| log |S| + |A| log |S|)

9: PROGRAMMATION DYNAMIQUE:

Il semblerait que l'algo de Bellman(chapitre 6) en est un bon exemple