Chapitre 4 : Graphes

Léo SAMUEL

21 décembre 2020

1 Vocabulaire

Un graphe est formé de :
— un ensemble, appelé ensemble des sommets (Noté S ou V)
— un sous-ensemble de S^2, appelé ensemble des arrêtes (Noté E)

Exemple 1 *$S = 1, 2, 3, 4$ et $A = (1, 2), (1, 3), (2, 1)$ Alors (S, A) est un graphe. Représentation graphique :*

Soit $k = (S, A)$ un graphe.
— Un chemin est une suite $(s_0, ..., s_n) \in S^{n+1}$ tq $\forall i \in [\![0, n[\![, (s_i, s_{i+1} \in A$
— Deux sommet s,$t \in S^n$ sont dits reliés lorsqu'il existe un chemin de s à t.
— La longueur d'un chemin est le nombre d'arretes empruntées
— $\forall(s, t)) \in S^2)$ deux sommets reliés. On note $d(s, t)$ la longueur du plus court chemin de s à t.
— On dit que G est non orienté lorsque $\forall(s, t) \in S^2, (s, t) \in A \equiv (t, s) \in A$.
— Supposons G non-orienté
  — on dit que G est connexe lorsque tous ses sommets sont reliés
  — $\forall s \in S$, on appelle composant connexe de s l'ensemble des sommets accessible depuis s

Propriété 1 *Supp G connexe et non orienté. La distance vérifie :*
— $\forall(s, t) \in S^2, d(s, t) = d(t, s)$
— $\forall(s, t, u) \in S^3, d(s, t) \leq d(s, u) + d(u, t)$ *(Inégalité triangulaire)*
— $\forall(s, t) \in S^2, d(s, t) = 0 \equiv s = t$
— *si G est orienté, les deux derniers points restent valides*
  Si G n'est pas connexe, on pose $\forall(s, t) \in S^2$ non connectés : $d(s, t) = +\infty$

Démonstration 1 *Soit $(s, t) \in S^2$, notons $a = d(s, t)$ et $b = d(t, s)$. Montrons que $a = b$.*
Soit γ un plus court chemin de s à t. Il est de longueur a. Soit $s_o, ..., s_a$ les sommets traversé par γ.

$(s_a, s_{a-1}, ..., s_o)$ est aussi un chemin. Notons le γ^T. Sa longueur est a.
Donc il existe un chemin de longueur a de t vers s. Donc $d(t, s) \leq l$, cad $b \leq a$.
Puis avec le même raisonement, on a $a \leq b$

Soit $(s, t, u) \in S^3$. Soit γ_1 un chemin de s à t et γ_2 un chemin de t à s.

Notons @ la concatenation des chemin et $|.|$ la longueur.
γ_1@γ_2 est un chemin de s à u de longueur $|\gamma_1| + |\gamma_2|$
Donc il existe un chemin de s à u de longueur $|\gamma_1| + |\gamma_2|$.
Donc $d(s, u) \leq |\gamma_1| + |\gamma_2| = d(s, t) + d(t, u)$

Soit $(s, t) \in S^2$:
Si $s = t$, soit $\gamma = (s)$. γ relie s à t et sa longueur est 0.

Si $d(s, t) = 0$. Soit γ un plus court chemin de s à t. $|\gamma| = 0$. Donc le sommet de départ est le sommet d'arrivé donc $s = t$

2 Implémentation

Soit G un graphe et (S, E) ses composants
Soit $n = |S|$.
On supp dans la suite que $S = [\![0, n[\![$
On utilise principalement deux méthodes pour enregistrer G.

— *Matrice d'adjacence : C'est la matrice $M \in M_n(N)$ tq $\forall (i, j) \in [\![0, n[\![^2, M_{i,j} = (1 si (i, j) \in A)(0 sinon)$*
— *Tableau de liste d'adjacences : C'est le tableau g de longueur n tq $\forall i \in [\![0, n[\![$. $g.(i)$ contient la liste des voisins de i*

```
type graphe1 = int array array;;
type graphe2 = int list array;;
```

3 Parcours d'un graphe

Souvent, on a besoin de parcourir les sommets de proche en proche à partir d'un sommet de départ.

3.1 Vocabulaire et invariants de boucle

— *Un sommet est dit *blanc* s'il n'est pas découvert*
— *Un sommet est dit *noir* s'il a été traité*
— *Un sommet est dit *gris* s'il est découvert mais pas traité (Sommet alors à traité)*
— *(V N) : Les voisins des sommet noir sont noirs ou gris*
— *(V G) : Tout sommet gris a au moins un voisin noir*
*Les seuls changement de couleurs autorisés sont de *blanc vers gris* et de *gris vers noir**

NB : Pour que le programme termine, il faudra éviter de revenir à un noir.

Consequences des invariants de boucle :

Lemme 1 *On suppose (V N) et (V G) vérifiés. On suppose que l'algo a aussi satisfait à (C C). On suppose qu'il n'y a plus de sommet gris.*
Alors les sommets blancs sont déconnectés des noirs cad : $\nexists$ des chemins d'un noir vers un blanc

Démonstration 2 *Supposons qu'il existe un chemin γ tq en notant $n = |\gamma|$ et $s_0, ...s_n$ ses sommets, s_0 est noir et s_n est blanc. Soit $i = \max\{k \in [\![0, n]\!], s_k noir\}$. On a $i \leq n$ car s_n est blanc. Donc s_{i+1} existe, et est différent de noir. Donc s_{i+1} est blanc car pas de gris par hypothese. Donc (V N) n'est pas vérifiée en s_i et s_{i+1}*

Lemme 2 *On suppose (V N), (V G), (C C). Notons D l'ensemble des gris ou noir au debut de l'algorithme. Alors tout sommet noir ou gris est relié à D $(*)$*

Démonstration 3 *Notons $(*)$ la propriété et verifions que c'est un invariant de boucle.*
Initialisation : Soit s un sommet noir ou gris au debut de l'algo alors $s \in D$. Prendre $\gamma = (s)$ de longueur 0 il relie s à l.*
Heredité : Supposons $(*)$ vraie à un instant de l'algo. Effectuant une étape qui respecte (V N), (V G), et (C C).*
S'il y a un changement de couleurs de gris vers noir, l'ensemble des sommets noirs ou gris n'est pas changé donc $(*)$ reste vrai.
S'il y a un changement blanc vers gris, Soit s le sommet concerné. Par (V G), s a un voisin noir,disons t. Par hypothese de récurrence, t est relié à l. Alors s est relié à l

On suppose qu'il n'y a plus de (V N), (V G), (C C). On suppose qu'au début de l'algo, $N = \emptyset$ et un seul sommet est gris, notons le s_d. On suppose que à la fin, $G = \emptyset$

Alors à la fin de la boucle, $N =$ (la composante connexe de s_d)

Rappel : La composante connexe de s_d est l'ensemble des sommets reliés a s_d

C'est aussi la plus petite composante connexe de G contenant s_d

Démonstration 4 *Montrons que $N \subset CC$ de s_d. Soit $s \in N$, par le lemme 2, s est rellié à s_d*
Montrons que $N \not\subset CC$ de s_d. Soit s relié a s_d, par le lemme 1, $s \notin B$ et $G = \emptyset$ alors $s \in N$

Ainsi un algo vérifiant nos 3 props permet de trouver la composante connexe de s_d. Ce sera notre premier exemple. Une simple modification permettra de trouver un chemin de s_d vers un autre sommet s_a. Puis un plus court chemin.

3.2 Squelette de programme impératif

3.2.1 Algorithme

Entrée :

- *Un Grapge (S, A)*
- *Un sommet $s_d \in S$*

```
Créer 3 ensembles $N$,$G$ et $B$
$N$ <- $\emptyset$
$B$ <- S
$G$ <- $\emptyset$

Peindre $s_d$ en gris

Tant que $G \not= \emptyset$:

    extraire un sommet $s$ de $G$
    Faire quelque chose
    Peindre $s$ en noir

    $\forall t$ voisin de $s$:
        Si $t \in B$, le peindre en gris
Renvoyer le resultat
```

3.2.2 Les invariants de boucle

- *(V N) est verifié*
- *(C C) est vérifié*
- *(V G) est vérifié*

3.2.3 En pratique

Comment enregistrer N,G,B ?

En général,

- *B n'est pas enregistré. Ce sont les sommets ni noirs, ni gris*
- *N : Un tableau "$deja_v u$"$tq \forall i \in S$, $deja_v u.(i) <=> i \in N$*
- *G : ça dépend de l'ordre dans lequel on veut traiter les sommets*

3.2.4 En autorisant les doublons dans G

Il est souvent plus pratique et parfois obligatoire d'utiliser à la place de G une structure qui autorise les doublons.
Exemple : $(1, 4, 0, 2, 4)$ *Après avoir traité* 4*, celui-ci devient noir, et la file contient donc des noirs.*

Ainsi :

- *La structure utilisée ne s'appellera plus G mais aVisiter*
- *Quand on sort un sommet de aVisiter, il faut vérifier qu'il est gris*
  Alors l'algo devient

```
Créer 3 ensembles $N$,$G$ et $B$
$N$ <- un tab de |S| bool initialement faux
aVisiter <- Une structure contenant initialement $s_d$
Peindre $s_d$ en gris
Tant que $G \not= \emptyset$:
    extraire un sommet $s$ de Avisiter
    Si Non N.(s):
        Faire quelque chose
        Peindre $s$ en noir
        $\forall t$ voisin de $s$:
            Si NON N.(t):
                Mettre t dans aVisiter
Renvoyer le resultat
```

Exemple 2 *Calcul de composante connexe. Avec pour aVisiter une file d'attente*

```
let composante_connexe_largeur g sd=
let n= Array.length g in
let deja_vu= Array.make n false in
let a_Visiter= Queue.create () in
Queue.add sd a_Visiter;
let rec visite_voisins = function
    | [] -> ()
    | t::q -> Queue.add t a_Visiter;
            visite_voisins q
in
while  not (Queue.is_empty a_Visiter) do
    let s= Queue.take a_Visiter in
    if not (deja_vu.(s)) then
    (
    visite_voisins g.(s);
    deja_vu.(s) <- true;
    )
done;
(* Maintenant, la composante connexe de sd correspond aux sommetsde deja_vu *)
let res = ref [] in
for i=0 to n-1 do
    if deja_vu.(i) then
    res:= i::(!res)
done;
!res
;;
```

3.2.5 Terminaison

Variant de boucle :

— *Pour la version 1 (sans doublons dans G), le nombre de sommets non noirs est un variant de boucle.*
— *Pour la version 2 (avec* $A_visiterquipeutcontenirdesdoublons), onpeutprendreelecouple(nombredesommetsnonnoir, |aVisiter|)po$

3.2.6 Complexité

Méthode de l'exercice 1 :
Prendre chaque ligne, voir ce qu'elle coute et combien de fois max elle est executée.

3.3 Parcours en largeur

Un parcours en largeur traite d'abord les sommets les plus proches du sommet de départ.

Pour réaliser un parcours en largeur, on prend aVisiter de type file d'attente.
Les deux exemples précedents étaient des parcours en largeur.

3.3.1 Invariant de boucle du parcours en largeur

Propriété 2 *A un certain instant d'un parcours en largeur. Soit n le nombre de sommets dans la file et $s_0, ..., s_{n-1}$ les sommets dans l'ordre. Alors $\exists d \in N$ et $k \in [\![0, n[\![$ tq $(s_{n-1}, ..., \vec{s}_k, ..., s_0)$*
— *$s_0, ..., s_{k-1}$ sont à distance inferieur a d de s_d*
— *$s_k, ..., s_n$ sont a distance inferieur a $d+1$ de s_d*
En outre, les noirs sont les sommets à distance inferieur a $d-1$ ainsi que les sommets à distance d qui ne sont pas dans la file.

3.4 Parcours en profondeur

Dans un parcours en profondeur, on poursuit un chemin autant que possible avant de partir sur un autre. Plus précisement, on visite en priorité les voisins du dernier sommet visité. Il suffit de remplacer la file par une pile.

Exemple 3 *Calcul d'un composante connexe*

```
let composante_connexe_profondeur g sd=
let n= Array.length g in
let deja_vu= Array.make n false in
let a_Visiter= Stack.create () in
Stack.push sd a_Visiter;
let rec visite_voisins = function
    | [] -> ()
    | t::q -> Stack.push t a_Visiter;
             visite_voisins q
in
while  not (Stack.is_empty a_Visiter) do
    let s= Stack.pop a_Visiter in
    if not (deja_vu.(s)) then
    (
    visite_voisins g.(s);
    deja_vu.(s) <- true;
    )
done;
(* Maintenant, la composante connexe de sd correspond aux sommetsde deja_vu *)
let res = ref [] in
```

```
for i=0 to n-1 do
    if deja_vu.(i) then
    res:= i::(!res)
done;
!res
;;
```

Le parcours en largeur est équivalent au parcours en profondeur (complexité : $(O(|S| + |A|))$

Cependant, il est plus facile à programmer en récursif.

3.4.1 Révision de parcours en profondeur d'un arbre de valence non bornée

```
type 'a arbre = Noeud of 'a * ('a arbre) list;;

let rec somme = function
  | Noeud(e,fils) -> e + somme_foret fils
and somme_foret = function
  | [] -> 0
  | t::q -> (somme t)+somme_foret(q)
;;

let rec etiquettes_arbre = function
  | Noeud(e,fils)-> e::(etiquettes_foret fils)
and etiquettes_foret = function
  | [] -> []
  | t::q -> etiquettes_arbre(t)@(etiquettes_foret q)
;;
```

En général, on ecrit deux fonctions récursives : une sur les arbres et une sur les forêts (liste d'arbres)

3.4.2 Parcours en profondeur d'un graphe, version recursive

On utilise deux fonctions mutuellement récursive :

- *une qui traite un sommet*
- *une qui traite la liste des sommets*

Squellette du programme :

```
let parcours_prof g sd =
  let n = Array.length g in
  let deja_vu = Array.make n false in

  let rec visite_sommet s=
    deja_vu.(s) <- true;
    (* Faire quelque chose avec s... *)
    visite_voisins s g.(s)
  and visite_voisins s= function
    | [] -> (* Renvoyer quelque chose *)
    | t::autre_Voisin when not(deja_vu.(t)) -> (* Renvoyer quelque chose avec visite_sommet et visite_voisins *)
    | t::autre_Voisin -> visite_voisins autre_Voisin

  in visite_sommet sd
;;
```

Exemple 4 *Composantes connexes*

```
let composante_connexe_rec g sd=
  let n = Array.length g in
  let deja_vu = Array.make n false in

  let rec visite_sommet s=
    deja_vu.(s) <- true;
    s::(visite_voisins g.(s))
  and visite_voisins = function
    | [] -> []
    | t::autre_Voisin when not(deja_vu.(t)) -> (visite_sommet t)@(visite_voisins autre_Voisin)
    | t::autre_Voisin -> visite_voisins autre_Voisin

  in visite_sommet sd
;;
```

Avantage de cette version :

- *plus court*
- *On peut facilement lorsqu'on traite un sommet d'où on vient : mettre l'argument (s) en plus dans la fonction* $visite_voisins(cfexercice11et12)$

4 Graphe pondérés

4.1 Notation

On suppose maintenant qu'à chaque arêtes G *est associé un flottant. Dans les applications, ce nombre sera la longueur de l'arête. On note* $\forall(s,t) \in A, l_{s,t}$ *la longueur de st*
Soit $n \in N$ *et* $(s_0, ..., s_n) \in S^{n+1}$ *tq* $\gamma = (s_0, ..., s_n)$ *est un chemin.*
Dans cette partie, on appelle longueur de γ *le nombre* $\sum_{i=1}^{n} l_{s_{i-1}s_i}$
$\forall s,t \in S^2$ *on note* $d(s,t) = min\{|\gamma|, \gamma : s- > t\}$ *s'il existe, sinon* $+\infty$

On ne suppose pas que $\forall(s,t) \in A, l_{s,t} = l_{t,s}$ *(même si* G *est non orienté) donc* d *ne sera à priori pas symétrique. En général, on prendra* $\forall(s,t) \in Al_{s,t} > 0$

4.2 Floyd-Warshall

But : Calculer $d(s,t)\forall(s,t) \in S^2$*. Ce qui fait* $|S|^2$ *flottants à calculer.*
On suppose que G *n'a pas de cycle de longueur nulle ou negative.*

Lemme 3 — *Soit* γ_1 *et* γ_2 *deux chemins tq la fin de* γ_1 *est le début de* γ_2 *alors* $\gamma_1 @ \gamma_2 = |\gamma_1| + |\gamma_2|$

- *Soit* γ_1, γ_2 *tq* $\gamma = \gamma_1 @ \gamma_2$ *et* $\gamma_1 : s- > u$, $\gamma_2 : u- > t$ *alors* γ_1 *est un plus court chemin de* s *à* u*. De meme entre* u *et* t*.*

Pricipe de l'algorithme :

$\forall(i,j,k) \in [\![0,n[\![^2 * [\![0,n]\!]$ *on pose* $d_{ij}^k)min\{|\gamma|, \gamma : i- > j$ *qui ne passe que par des sommets dans* $[\![0,k[\![\}$

Ainsi, $\forall(i,j) \in [\![0,n[\![^2$, $d_{i,j}^n = d(i,j)$ *et* $d_{i,j}^0 = l_{i,j}, 0sii = j, +\infty(i,j) \notin A$

Cherchons une relation de recurrence pour calculer les $(d_{i,j}^{k+1})_{i,j}$ *à partir des* $(d_{i,j}^k)_{i,j}$

Soit $k \in [\![0,n[\![$*, on suppose connue* $(d_{i,j}^k)_{i,j \in [\![0,n[\![}$
Soit $(i,j) \in [\![0,n[\![^2$ *Calculon* $d_{i,j}^{k+1}$*.*

Soit γ un plus court chemin de i à j avec étapes dans $[\![0, n[\![$
On distingue deux cas :

1 - Si γ passe par k
Mq $|\gamma| = d_{i,k}^k + d_{k,j}^k$
Comme γ est un plus court chemin, il n'a pas de cycle (les cycles sont de longueur >= 0) Donc il ne passe qu'une seul fois par k.

Soit $\gamma_1, \gamma_2 tq \gamma = \gamma_1 @ \gamma_2$ i->k->j
- γ_1 et γ_2 sont à étapes dans $[\![0, k[\![$
- γ_1 est un plus court chemin de i à k à étape dans $[\![0, k[\![$. Donc $|\gamma_1| = d_{i,k}^k$
- De même avec $|\gamma_1| = d_{i,k}^k$

D'où $|\gamma| = d_{i,k}^k + d_{k,j}^k$

2 - Si γ ne passe pas par k :
Alors $|\gamma| = d_{i,k}^k$

Pour conclure : $d_{i,j}^{k+1}$ vaut $d_{i,k}^k + d_{k,j}^k$ OU $d_{i,j}^k)$

Montrons alors que $d_{i,j}^k = min(d_{i,k}^k + d_{k,j}^k, d_{i,j}^k$

Cas 1 : si $d_{i,j}^k \leq d_{i,k}^k + d_{k,j}^k$
Soit $\gamma, \gamma_1, \gamma_2$ des chemins correspondant

γ est un plus court chemin de i à j avec étapes dans $[\![0, k[\![$ et il ne passe pas par k. On est alors dans le cas 1 précedent.

Cas 2 : si $d_{i,j}^k \geq d_{i,k}^k + d_{k,j}^k$
Alors γ ne peut pas être un plus court chemin de i à j avec étapes dans $[\![0, k+1[\![$ car $\gamma_1 @ \gamma_2$ est strictement plus court. Donc on n'est pas dans le cas 1 prec. donc on est dans le cas 2 et $d_{i,j}^k = d_{i,k}^k + d_{k,j}^k$

Bilan : $\forall (i,j) \in [\![0, n[\![^2, d_{i,j}^k = min(d_{i,k}^k + d_{k,j}^k, d_{i,j}^k)$

Il suffit d'utiliser un tableau de format $n * n$ et une boucle for sur k.

L'invariant de boucle sera "En entrée de l'itération k, $\forall (i,j) \in [\![0, n[\![^2, dist.(i).(j) = d_{i,j}^k$ "

On peut prouver qu'on a aussi $\forall i, jk \in [\![0, n[\![d_{i,j}^k = min(d_{i,k}^{k+1} + d_{k,j}^k, d_{i,j}^k) = d_{i,j}^k = min(d_{i,k}^k + d_{k,j}^k, d_{i,j}^{k+1})$

En effet, $d_{i,k}^{k+1} = d_{i,k}^k$ et $d_{k,j}^{k+1} = d_{k,j}^k$. On peut alors s'épargner la copie de dist

```
let floyd_warshall m=
  (* Entrée : m matrice d'adjacence d'un graphe pondéré G *)
  let n = Array.length m in
  let dist = copie_mat m in
  for k=0 to n-1 do
    (* Ici dist contient les d_{i,j}^{k} *)
    let sauv = copie_mat dist in
    for i=0 to n-1 do
      for j=0 to n-1 do
        dist.(i).(j) <- min (sauv.(i).(j)) (sauv.(i).(k)+.sauv.(k).(j));
      done;
    done;
    (* Maintenant, dist contient les d_{i,j}^{k+1} *)
```

```
  done;
  dist
;;
```

```
let floyd_warshall_chemin m sd sa=
  let n = Array.length m in
  let chemin = Array.make_matrix n n [] in
  let dist = copie_mat m in
  for i=0 to n-1 do
    for j=0 to n-1 do
      if m.(i).(j) <> infinity && i<>j then
        chemin.(i).(j) <- [i];
    done;
  done;
    for k=0 to n-1 do
        let sauv = copie_mat dist in
        for i=0 to n-1 do
            for j=0 to n-1 do
                if (sauv.(i).(k) +. sauv.(k).(j)) < dist.(i).(j) then
                    (
                    dist.(i).(j) <- sauv.(i).(k) +. sauv.(k).(j);
                    chemin.(i).(j) <- chemin.(i).(k)@chemin.(k).(j);
          )
            done;
        done;
    done;
  chemin.(sd).(sa)@[sa]
;;
```

Complexité de l'algo : $O(|S|^3)$ *Sauf pour les chemins qui est plus lent à cause des @*

4.3 Dijkstra

But : Calculer le plus court chemin entre deux points.
Fixons $(s_d, s_a) \in S^2$.

Même principe que pour un plus court chemin dans un graphe non pondéré :on effectue un parcours en largeur en partant de s_d. *Gardons le vocabulaire des sommets noirs, blancs et gris. On maintient un tableau dist tq* $\forall t \in S$ *dist.(t)contient à chauque instant la longueur d'un plus court chemin "connu" de* s_d *à* t. *S'il n'y a pas de telle chemin, cad si* t *est blanc. Dans ce cas on met dist.(t)=* $+\infty$.

Invariants de boucle précis :

- *(IG), (IN), (CC) restent valide (cf debut chapitre)*
- *(DN) :* $\forall s \in N$ *dist.(s)=* $d(s_d, s)$ *: On connait la distance à* s_d *pour les sommets noirs*
- *(DG) : dist.(s) contient la longueur d'un pcc de* s_d *à* t *avec étapes dans* N
- *(DB) :* $\forall s \in B$ *dist.(t)=* $+\infty$.

4.3.1 Lemme fondamental

Lemme 4 *Soit* s *un sommet gris pour lequel dist.(s) est minimal. On suppose les invariants de boucle vérifiés. Alors dist.(s)=*$d(s_d, s)$ *ainsi on peut peindre* s *en noir.*

Démonstration 5 *Soit* γ *un pcc de* s_d *à* s *avec étapes noires. Donc par (D N)* $|\gamma|$ *=dist.(s)*

Mq $|\gamma| = d(s_d, s)$ cad que γ est un pcc de s_d à s .
Supposons η un chemin de s_d à s tq $|\eta| < |\gamma|$. Donc par (D N), η ne passe que par des sommets noirs. Soit t le premier sommet non noir traversé par η. t est gris par (I N).
Notons η_1 la partie de η de s_d à t. η_1 a ses étapes dans N.
Donc dist.(t)$\leq |\eta| < |\gamma| =$dist.(t) D'où la contradiction

Ainsi à chaque étape on va chercher $s \in G$ tq dist.(t) est minimal et on va peindre s en noir
Il faudra alors pour tout voisins de s

Auparavant, dist.(t) contenait la longueur d'un pcc de s vers t à étapes dans N. On utilise le même raisonnement que pour FW : soit γ un pcc de s_d à t avec étapes dans $N\{s\}$

- *si γ ne passe pas par s c'est un pcc de s_d à t à étapes dans N. Donc dist.(t)= $|\gamma|$, on ne fait rien*
- *si γ passe par s. Soient γ_1, γ_2 tq $\gamma = \gamma_1 @ \gamma_2$. $\gamma_1 : s_d - > s$ et $\gamma_2 : s - > t$. γ_1 est un pcc de s_d à s à étapes dans N donc $|\gamma_1 =$dist.(s)*
- *si γ_2 passe par un sommet u. Soit η un pcc noir de s_d à u. On peut remplacer le début de γ par η : $\exists$ un pcc noir de s_d à t qui ne passe pas par s. Donc dist.(t) contient déja la bonne valeur (comme dans le cas précedent).*
- *γ_2 est juste une arete. Donc $|\gamma_2|$ est connue c'est $l_{s,t}$ et dans dist.(t) il faut mettre $|\gamma_1| + |\gamma_2|$ cad dist.(s)+$l_{s,t}$*

Ainsi dans dist.(t) il faut mettre dist.(s)+$l_{s,t}$. Pour resumer tous les cas, il faut effectuer :
dist.(t) <- min (dist.(t), dist.(s)+$l_{s,t}$)

En effet : notons d la valeur à mettre dans dist.(t)
- On a que d =dist.(t) ou d = $l_{s,t}$+dist.(t) donc d $\geq$ min (dist.(t), dist.(s)+$l_{s,t}$)
- De même on a d $\leq$ min (dist.(t), dist.(s)+$l_{s,t}$)

4.3.2 Algorithme simplifié

```
Créer un tableau dist de longueru $|S|$ contenant initialement des $\infty$
Créer une structure pour les sommets gris contenant $s_d$.
dist.(sd)<-0
$s$ blanc -> dist.(s)=$\infty$
$s$ est noir s'il n'est pas gris et s'il n'est pas blanc

Tant que $s_a$ n'est pas noir et gris pas vide:

    extraire de gris le sommet pour lequelle dist est minimal
    pour tout t voisin de s:
        si t est blanc:
            on le met en gris
            dist.(t)<-dist.(s)+l_st
        sinon si t est gris:
            dist.(t) <- min(dist.(t), dist.(s)+l_st
        sinon
            ()
Renvoyer dist.(s_a)
```

4.3.3 Avec des tas mutables

- *pour choisir la place dans le tas, il faut utiliser dist.(s) et non s lui même donc fun s t -> dist.(s) <= dist.(t)*
- *mettre dans le tas des couples (dist.(s),s)*

Le problème est qu'a chaque fois que l'on change dist.(s) il faudra le changer dans le tas puis remonter à sa place l'élément. Pour retrouver s il faut aussi enregistrer sa position donc faire par exemple un tableau position

4.3.4 Complexité

Le tas contient à chaque instant <= |S| éléments. Donc les opérations sont en $O(log|S|)$*,*
La complexité est la même que le parcours en largeur car même sutructure : mais enfile et défile sont en O(1) et entasse et detasse en O(log|S|). On a alors une complexité en $O((|A|+|S|)log(S))$

4.3.5 Avec des tas persistants

On y mettra des couples de la forme (dist.(s),s). Si on met a jour dist.(s), on va entasser (nouvelle valeur de dist.(s),s). Le sommet sera en double avec sa version plus récente au dessus.
Lorsqu'on aura traité la premiere version, ne pas retraiter la deuxième.

Utiliser un tableau dejaVu pour les noirs

Problèmes : le tas peut contenir plus d'élément, donc les operations de base seront plus lente...
L'opération "entasser un élément" est effectué au plus $O(|A|)$
Donc les operations sont en $O(log|S|)$*. Mais* $|A| \leq |S|^2$

Donc $log|A| \leq 2log(|S|)$
$O(log|A|) = O(log(|S|))$