1 Graphes valués

On appelle graphe valué (ou pondéré) un couple (G, v) où $G = (S, A)$ est un graphe et v une application de A dans $\mathbb{R}$. Si a est un arc (ou une arête), le nombre $v(a)$ est la valuation (ou valeur ou poids) de l'arc.

Exemples

a) Sur un graphe de carte routière, les valuations sont les distances en kilomètres.

b) Sur un graphe de carte de randonnée, les valuations sont les temps de parcours pour un marcheur moyen.

c) Sur le réseau internet, les valuations peuvent être la bande passante de la ligne ou la probabilité qu'un message soit transmis sans altération.

d) Sur un circuit électrique, les valuations peuvent être les intensités ou les tensions.

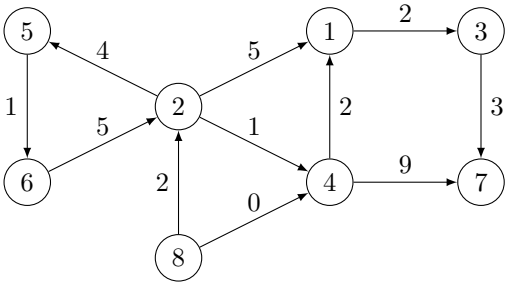
En fonction des valuations, on calcule la valeur d'un chemin : dans le cas des distances, la somme des distances ; dans le cas des probabilités de bonne transmission, le produit des probabilités.

Un problème courant sur les graphes valués est de déterminer le meilleur chemin d'un sommet à un autre : le plus court en distance, en temps, celui avec le plus gros débit, la meilleure qualité, etc.

Nous étudions ici le cas de la somme des valuations et on cherche le chemin de valuation minimale entre deux sommets, les autres cas ne sont que des adaptations de ce cas particulier.

On représente souvent les graphes valués par des matrices d'adjacence légèrement modifiées :

$m_{i,j} = \infty$ s'il n'y a pas d'arc de i vers j, $m_{i,j}$ la valeur de l'arc sinon.



est representé par la matrice $M = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & 1 & 4 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & \infty & \infty & \infty & \infty & 9 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & 5 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & \infty & 0 & \infty & \infty & \infty & \infty \end{pmatrix}$

Ou alors par des listes d'adjacence du type suivant : à chaque sommet i, on associe la liste des couples (j, v) tels que j soit un sommet accessible depuis i par une arc (une arête) de valeur v.

Le graphe précédent est représenté par le tableau de listes d'adjacence

1	2	3	4	5	6	7	8
$(3,2)$	$(4,1);(1,5);(5,4)$	$(7,3)$	$(1,2);(7,9)$	$(6,1)$	$(2,5)$		$(2,2);(4,0)$

2 Cas d'un graphe non valué

Un graphe non valué est intrinsèquement valué par l'application constante égale à 1 : tous les arcs sont de valuation 1. La longueur d'un chemin est donc le nombre d'arcs du chemin (c'est la longueur naturelle définie dans le premier chapitre sur les graphes).

Un algorithme simple pour déterminer le plus court chemin dans un tel graphe est d'effectuer un parcours en largeur et de l'arrêter prématurément dès qu'on arrive sur le sommet but recherché (si c'est possible) : le nombre d'itération est la longueur d'un chemin minimal. Si on veut en plus donner le chemin, il suffit de noter à chaque mise en file d'un sommet le sommet en cours (son prédécesseur sur le chemin) et de remonter de prédécesseur en prédécesseur jusqu'au sommet origine.

3 Cas d'un graphe valué

3.1 Cycle absorbant

Soit G un graphe valué. On appelle cycle absorbant un cycle de valeur strictement négative. La présence d'un cycle absorbant est une calamité : en effet, si le chemin passe par ce cycle, alors en bouclant autant de fois qu'on veut sur ce cycle, on peut faire diminuer la valeur du chemin arbitrairement, donc il n'existe pas de chemin optimal.

Nous supposerons donc toujours que les graphes ne possèdent pas de cycle absorbant : une condition suffisante pour cela est que les valuations soient toutes positives ou nulles (cas le plus courant en pratique). Notons qu'il existe des algorithmes qui permettent la détection des cycles absorbants.

3.2 Algorithme de Floyd-Warshall

Comme d'habitude, on numérote les sommets d'un graphe valué par les entiers de 1 à n et on appelle $M = (m_{i,j})$ la matrice des valuations des arcs (avec la convention précédente : ∞ si l'arc n'existe pas).

a) Algorithme

L'algorithme de Floyd-Warshall est fondé sur le principe de sous-optimalité de Bellmann :

si un chemin est optimal, alors tout sous-chemin est lui-même optimal

On reconnaît là le signe d'une possible programmation dynamique.

Si i, j sont deux indices distincts compris entre 1 et n, alors on note $L(i, j, k)$ la longueur du chemin optimal entre i et j ne passant que par des sommets **intermédiaires** de numéros inférieurs ou égaux à k s'il existe, ∞ sinon.

Donnons une définition récursive de $L(i, j, k)$.

- Si $k = 0$, alors $L(i, j, 0) = m_{i,j}$: seul un chemin direct peut être pris en compte, celui qui suit l'arc (i, j) si cet arc existe ;
- Si $1 \leqslant k \leqslant n$, alors considérons un chemin optimal de i à j ne passant que par des sommets **intermédiaires** de numéros inférieurs ou égaux à k . Deux cas se présentent :
  - soit le chemin ne passe pas par le sommet numéroté k et dans ce cas, tous ses sommets intermédiaires sont de numéros inférieurs ou égaux à $k - 1$, donc $L(i, j, k) = L(i, j, k - 1)$;
  - soit il passe par le sommet numéroté k, donc comme il est optimal, il ne passe qu'une fois par ce sommet ; le chemin se décompose donc en un chemin optimal de i à k ne passant que par des sommets de numéros inférieurs ou égaux à $k - 1$ et d'un chemin optimal de k à j ne passant que par des sommets de numéros inférieurs ou égaux à $k - 1$, donc $L(i, j, k) = L(i, k, k - 1) + L(k, j, k - 1)$

  On calcule donc ces deux quantités $L(i, j, k - 1)$ et $L(i, k, k - 1) + L(k, j, k - 1)$:
  - si les deux sont infinies, il n'y a pas de chemin entre i et j ne passant que par des sommets de numéros inférieurs ou égaux à k, donc $L(i, j, k) = +\infty$;
  - sinon il existe au moins un chemin entre i et j : l'optimal est alors le plus court des deux, donc $L(i, j, k) = \min\big(L(i, j, k - 1),\ L(i, k, k - 1) + L(k, j, k - 1)\big)$

  On constate que cette expression est valable aussi dans le premier cas, donc en toute généralité :

$$L(i, j, k) = \min\big(L(i, j, k - 1),\ L(i, k, k - 1) + L(k, j, k - 1)\big)$$

La solution du problème est donc la valeur $L(i, j, n)$.

À partir de cette définition récursive, on peut appliquer un principe de mémoïsation, car on risque sinon de devoir recalculer souvent les mêmes nombres plusieurs fois.

b) En pratique

En pratique, on appelle M_k la matrice dont le coefficient d'indice (i, j) vaut $L(i, j, k)$. On calcule donc une suite de matrices, sachant que $M_0 = M$.

Donnons un exemple avec le graphe précédent.

$$M_0 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & 1 & 4 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & \infty & \infty & \infty & \infty & 12 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & 5 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & \infty & 0 & \infty & \infty & \infty & \infty \end{pmatrix} \quad M_1 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ 5 & \infty & 7 & 1 & 4 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 12 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ \infty & 5 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & \infty & 0 & \infty & \infty & \infty & \infty \end{pmatrix}$$

$$M_2 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ 5 & \infty & 7 & 1 & 4 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 12 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ 10 & 5 & 12 & 6 & 9 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 7 & 2 & 9 & 0 & 6 & \infty & \infty & \infty \end{pmatrix} \quad M_3 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & 5 & \infty \\ 5 & \infty & 7 & 1 & 4 & \infty & 10 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ 10 & 5 & 12 & 6 & 9 & \infty & 15 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 7 & 2 & 9 & 0 & 6 & \infty & 12 & \infty \end{pmatrix}$$

$$M_4 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & 5 & \infty \\ 3 & \infty & 5 & 1 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ 8 & 5 & 10 & 6 & 9 & \infty & 13 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 2 & 2 & 4 & 0 & 6 & \infty & 7 & \infty \end{pmatrix} \quad M_5 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & 5 & \infty \\ 3 & \infty & 5 & 1 & 4 & 5 & 8 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & \infty & 1 & \infty & \infty \\ 8 & 5 & 10 & 6 & 9 & 10 & 13 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 2 & 2 & 4 & 0 & 6 & 7 & 7 & \infty \end{pmatrix}$$

$$M_6 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & 5 & \infty \\ 3 & 10 & 5 & 1 & 4 & 5 & 8 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 7 & \infty \\ 9 & 6 & 11 & 7 & 10 & 1 & 14 & \infty \\ 8 & 5 & 10 & 6 & 9 & 10 & 13 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 2 & 2 & 4 & 0 & 6 & 7 & 7 & \infty \end{pmatrix} \quad M_7 = \begin{pmatrix} \infty & \infty & 2 & \infty & \infty & \infty & 5 & \infty \\ 3 & 10 & 5 & 1 & 4 & 5 & 8 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 3 & \infty \\ 2 & \infty & 4 & \infty & \infty & \infty & 7 & \infty \\ 9 & 6 & 11 & 7 & 10 & 1 & 14 & \infty \\ 8 & 5 & 10 & 6 & 9 & 10 & 13 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 2 & 2 & 4 & 0 & 6 & 7 & 7 & \infty \end{pmatrix}$$

La dernière matrice donne les résultats voulus : par exemple, le plus court chemin du sommet 6 vers le sommet 7 est de longueur 13.

Remarque. Une implémentation efficace en utilisation de la mémoire n'a besoin que de calculer une matrice à partir d'une autre, donc on ne retient pas la suite des matrices.

Si on veut obtenir en plus un chemin optimal, alors lors chaque calcul de coefficient de la matrice, on enregistre par exemple dans une autre matrice C le chemin associé au calcul :

- $C(i,j,0) = [i;j]$ si $m_{i,j} < \infty$, [ ] sinon
- si $L(i,j,k) = L(i,j,k-1)$, alors $C(i,j,k) = C(i,j,k-1)$
- si $L(i,j,k) = L(i,k,k-1) + L(k,j,k-1)$, alors $C(i,j,k) = C(i,k,k-1) \bullet C(k,j,k-1)$ où $\bullet$ désigne la concaténation de deux chemins dont le but du premier est égal à l'origine du second

c) Complexité

L'algorithme calcule n matrices $n \times n$, chaque coefficient étant calculé de manière élémentaire, donc la complexité de l'algorithme de Floyd-Warshall est en $O(n^3)$.

Mais on peut remarquer que l'algorithme de Floyd-Warshall calcule les longueurs de tous les chemins optimaux entre deux sommets du graphe. Si on n'a besoin que de la longueur du chemin optimal entre deux sommets donnés, on a fait trop de travail pour rien.

3.3 Algorithme de Dijkstra

On suppose que toutes les valuations sont cette fois-ci positives (l'algorithme précédent fonctionne même avec quelques valeurs négatives, l'important étant qu'il n'y ait pas de cycle absorbant). Ceci implique l'absence de cycles absorbants.

Ici encore, on applique le principe de sous-optimalité de Bellmann :

si un chemin est optimal, alors tout sous-chemin est lui-même optimal

Le problème est ici plus précis : étant donné un sommet i, on veut calculer les chemins optimaux d'origine i et seulement ceux-là.

L'algorithme de Floyd-Warshall construit peu à peu la solution en ajoutant à chaque étape le sommet de numéro suivant. Cependant, il est possible que ce sommet ne serve à rien dans le calcul qui nous intéresse, car il calcule tous les chemins optimaux.

L'algorithme de Dijkstra (prononcez `deille-k-stra`) ajoute seulement un sommet parmi les sommets vraiment pertinents : ceux qui sont les successeurs des sommets précédents (en partant de l'unique sommet i). L'algorithme avance dans le graphe d'un arc à la fois : à chaque étape, on va un arc plus loin et si on revient sur un sommet déjà atteint, on regarde si on a fait mieux. Le point clef de l'algorithme de Dijkstra est le choix du prochain sommet à étudier : parmi tous ceux disponibles, on choisit le plus proche du sommet initial.

a) Algorithme

L'algorithme de Dijkstra fait lui aussi partie de la catégorie des algorithmes de marquage : à chaque étape, les sommets ont un état (libre / en cours / ouvert / fermé).

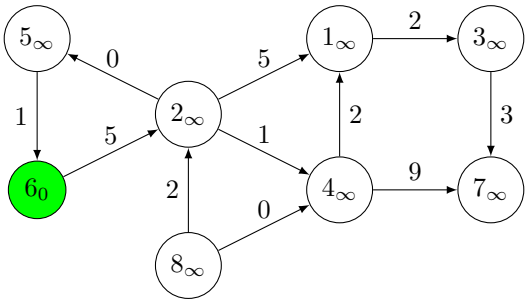
On part d'un sommet donné i et on veut calculer la longueur des plus courts chemins issus de i.

Pour cela, on tient à jour deux ensembles dit « ouvert » et « fermé » : l'ensemble ouvert contient les sommets à traiter, l'ensemble fermé ceux qui ont déjà été traités, ce qui signifie ici « ceux dont on connaît définitivement la distance à i ».

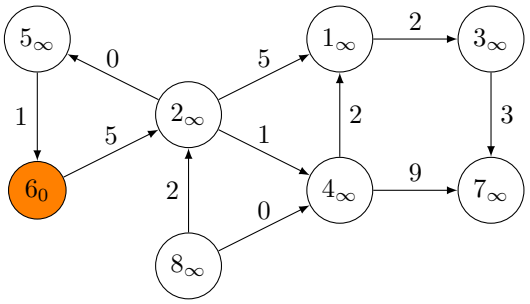
En plus des deux ensembles ouvert Ω et fermé F, on considère un tableau t de longueur n tel que $t[i] = 0$ et les autres cases valent ∞ : t contient à la fin de l'étape e la longueur du plus court chemin **calculé à cet instant** de i à tout autre sommet. Quand un sommet est dans l'ensemble fermé, alors ça signifie que la longueur calculée est définitive.

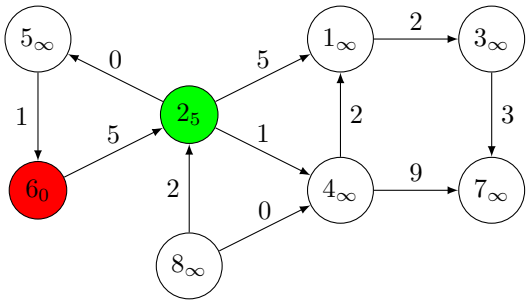
Donnons d'abord un exemple sur un dessin. On ajoute en indice de chaque sommet la longueur optimale d'un chemin depuis un sommet (ici, on part du sommet 6).

Initialement, le graphe est dans l'état suivant (en orange le sommet en cours, en rouge les sommets de l'ensemble fermé, en vert ceux de l'ensemble ouvert, en vert citron un sommet de l'ensemble ouvert mis à jour) :

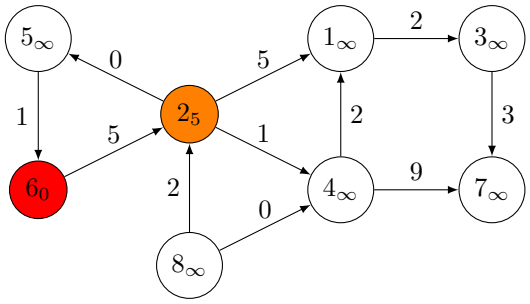


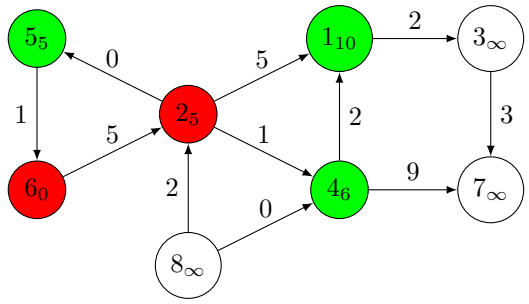
À la première étape, on choisit parmi les sommets verts le plus proche de l'origine (ici 6), on marque en vert ses successeurs pas encore marqués et si nécessaire, on met à jour les distances des points verts, enfin on passe le point orange au rouge.



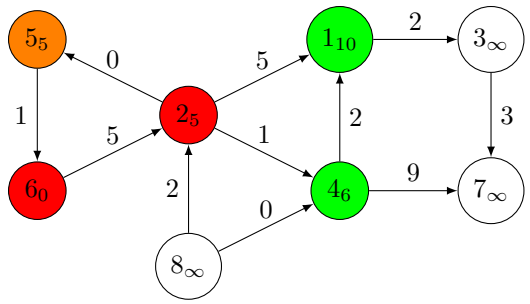


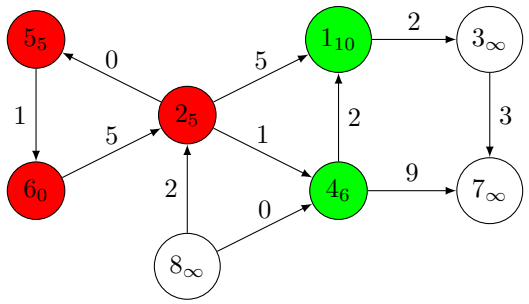
À la deuxième étape, on choisit parmi les sommets verts le plus proche de l'origine (ici 2), on marque en vert ses successeurs pas encore marqués et si nécessaire, on met à jour les distances des points verts, enfin on passe le point orange au rouge.



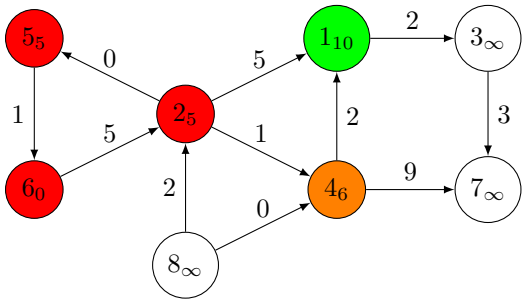


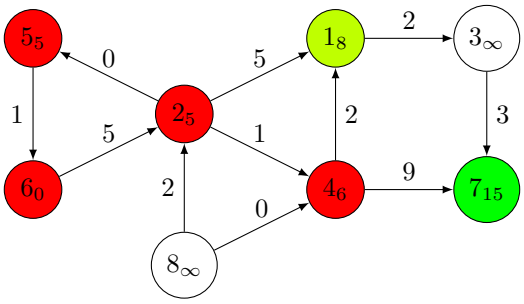
À la troisième étape, il y a plusieurs choix pour le point suivant à choisir parmi les verts : le plus proche étant le sommet 5, il passe à l'orange, mais comme son unique successeur est déjà marqué rouge, il ne se passe rien (on est revenu sur un sommet déjà vu avant donc forcément on a fait un cycle, donc un chemin plus long).



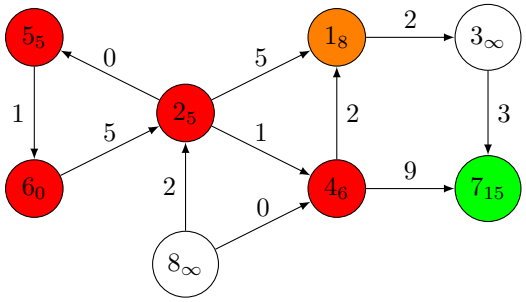


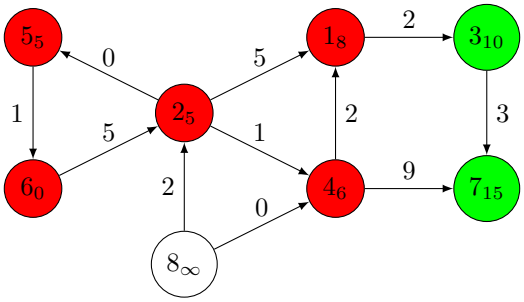
À la quatrième étape, il y a plusieurs choix pour le point suivant à choisir parmi les verts : le plus proche étant le sommet 4, il passe à l'orange, ses successeurs non marqués au vert et on met à jour les points verts (ici, on trouve un chemin vers 1 plus court, il est mis à jour).



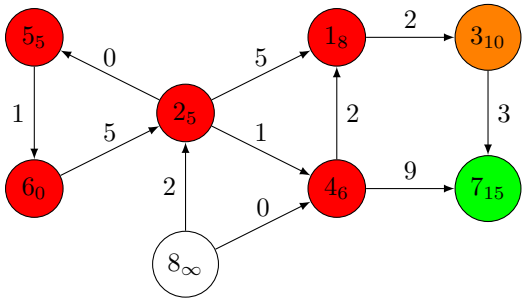


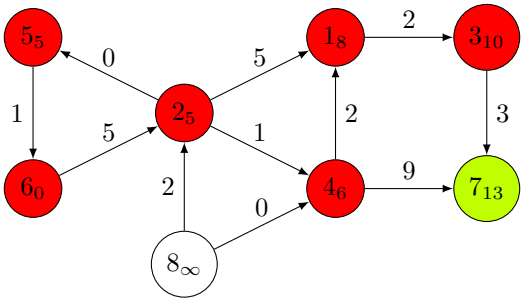
Et on recommence.



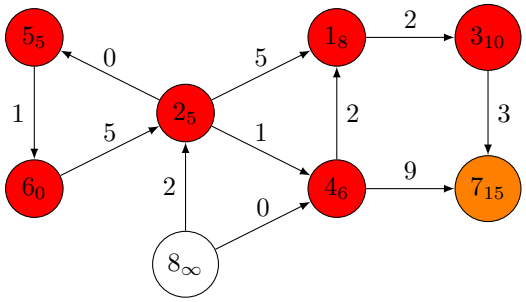


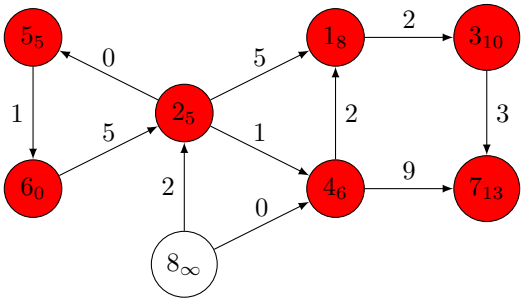
Encore une fois.





Enfin, dernière étape : le sommet 7 passe à l'orange, mais comme il n'a pas de successeur, l'ensemble ouvert est désormais vide, l'algorithme est terminé.





Finalisons la description de l'algorithme.

- On initialise l'ensemble ouvert Ω au sommet i, l'ensemble fermé F à $\varnothing$
- Tant que Ω est non vide,
  - parmi tous les éléments de Ω, on choisit k tel que $t[k]$ soit minimal, on le retire de Ω et on l'ajoute à F ;
  - puis on passe en revue tous les successeurs ℓ de k :
    si ℓ n'est pas dans F (*), alors
    - si ℓ n'est pas dans Ω, on l'y ajoute ;
    - si $t[k] + m_{k,\ell} < t[\ell]$, alors on remplace $t[\ell]$ par $t[k] + m_{k,\ell}$ (**)

    (si ℓ est dans F, on ne fait rien)

La contrainte (*) est pour éviter de réétudier des points par lesquels on est déjà passé (sinon on reprendrait leurs successeurs et on finirait par boucler sur les mêmes sommets).

Le test (**) permet de mettre à jour un sommet par lequel on est déjà passé : en passant par le point k, on a un chemin plus court que celui précédemment calculé, donc on remplace (remarque : comme initialement, les longueurs sont infinies, même si on n'est pas vraiment passé par ce sommet, il est mis à jour quand même).

b) En pratique

La description précédente ne dit rien sur la façon d'implémenter les ensembles. Comme pour les parcours, le choix de l'implémentation influe fortement sur la complexité.

On ajoute sans contrainte des éléments dans Ω, en revanche on retire toujours l'élément minimal pour l'ordre induit par le tableau t : pour éviter le calcul du minimum à chaque retrait, il faudrait que le minimum « vienne tout seul ». La structure de **file de priorité** est toute indiquée pour ce genre de traitement. Concrètement, on l'implémente donc par un tas-min, avec une opération supplémentaire : lorsqu'on modifie la distance calculée jusqu'à un sommet (**), la priorité du sommet dans la file est modifiée à la hausse (il doit en sortir plus vite), il faut donc le remonter dans la file pour le mettre au bon rang de sortie.

Remarque. Pour obtenir un chemin optimal, il suffit de travailler avec un deuxième tableau, qui contient dans la case j le numéro du sommet précédent sur le chemin qui va de i à j : à la fin de l'algorithme, il suffit de remonter de proche en proche vers le sommet initial pour connaître le chemin optimal.

c) Complexité

L'ensemble ouvert Ω contient au plus n éléments (les n sommets). Rappelons que
- l'insertion dans une file de priorité de longueur au plus n est de complexité $O(\log n)$;
- l'extraction de l'élément minimal d'une telle file de priorité est aussi de complexité $O(\log n)$;
- la modification de la priorité d'un élément dans la file est encore de complexité $O(\log n)$, à condition de savoir où se trouve cet élément dans la file (si on doit en plus le trouver avant, il faut compter alors $O(n)$).

L'extraction des sommets coûte donc au plus $O(n \log n)$.

Lors du traitement du sommet k, l'insertion des successeurs ou la mise à jour de la file de priorité coûte donc $O(a_k \log n)$ où a_k est le nombre de successeurs de k (appelé aussi arité du sommet k). En réitérant cette opération pour chaque sommet au maximum, on obtient donc un coût de $\sum_{k \text{ sommet}} O(a_k \log n)$. Or $\sum_{k \text{ sommet}} a_k$ est le nombre total d'arcs du graphe, donc les insertions coûtent $O(m \log n)$.

Au total, on obtient une complexité en $O((n+m)\log n)$ ou encore $O(n^2 \log n)$.

d) Autre version

L'algorithme précédent dans son implémentation à base de file de priorité impose de devoir modifier la priorité des sommets à l'intérieur de la file. On peut éviter ces modifications en inversant l'ordre des tests et en admettant qu'un sommet peut se retrouver plusieurs fois dans la file avec des priorité différentes. Ceci n'est pas gênant, à condition de tester en sortie de file si un élément est déjà dans l'ensemble fermé (ce qui est immédiat), car un sommet présent plusieurs fois ne sera traité qu'une fois et avec la priorité maximale.

On peut donc formuler l'algorithme de la façon suivante :
- On initialise l'ensemble ouvert Ω au sommet i, l'ensemble fermé F à $\varnothing$
- Tant que Ω est non vide,
  - parmi tous les éléments de Ω, on choisit k tel que $t[k]$ soit minimal, on le retire de Ω ;
  - s'il n'est pas dans F, alors on passe en revue tous les successeurs ℓ de k :
    si ℓ n'est pas dans F (*), alors
    - si $t[k] + m_{k,\ell} < t[\ell]$, alors on remplace $t[\ell]$ par $t[k] + m_{k,\ell}$ (**)
    - on ajoute ℓ dans Ω avec la priorité associée à cette distance ;

    (si ℓ est dans F, on ne fait rien)
  - sinon on ne fait rien

Bien sûr, ceci a un coût. La file de priorité peut alors être beaucoup plus longue que dans la version précédente, puisque on insère des éléments lors de chaque passage par une arête (au plus), donc on peut avoir une file de longueur m. Dans ce cas, les complexités précédentes sont modifiées : les facteurs $\ln n$ deviennent des facteurs $\ln m$. Ceci dit, ça ne change pas l'ordre de grandeur, car $m \leqslant n^2$, donc $\ln m \leqslant 2 \ln n$. La complexité est toujours en $O((n+m)\log n)$, avec une constante cachée plus grande.