Introduction à l'informatique
Graphes

Benjamin Monmege

2019/2020

Nous savons désormais mieux comment on (ou une machine) peut calculer avec des algorithmes, sur des ensembles d'objects stockés dans des tableaux ou des données numériques. En particulier, on a vu comment trier des tableaux, ce qui permet de répondre à l'un des problèmes posé dans le chapitre d'introduction. Un autre problème que nous posions alors est le calcul du plus court chemin dans Google Maps : on avait dit alors qu'on pouvait abstraire le problème à l'aide d'un graphe avec des nœuds représentant les intersections de rue et des arcs reliant ces nœuds avec la durée en minutes du trajet correspondant. Ce chapitre et le suivant ont pour objectif de mieux comprendre cette structure de données supplémentaire que sont les graphes. Dans ce premier chapitre, nous allons voir que les graphes sont partout et nous allons répondre à la question que pose Google Maps. Le chapitre suivant permettra d'étudier d'autres problèmes en lien avec les graphes.

1 Les graphes sont partout

Commençons par observer différents graphes qu'on croise, consciemment ou pas, dans notre vie. Les réseaux de métro (cf Figure 1) sont sans doute l'un des exemples les plus visibles de graphes où des stations sont connectées par des rails de métro (de différentes lignes).

Nous sommes sans doute moins conscients qu'Internet est également un gigantesque graphe dans lequel des utilisateurs (représentés par une adresse, qu'on appelle *adresse IP*) sont reliés entre eux via des serveurs, comme illustré en Figure 2.

Toujours sur Internet, il existe des sous-réseaux spécifiques, par exemple ceux de réseaux sociaux tels que Twitter ou Instagram.

Mais nous n'avons pas attendu la création d'Internet pour créer des réseaux connectant des personnes : il existe ainsi des graphes représentant les collaborations entre chercheurs, les liens dans le graphe représentant un article écrit en commun. La Figure 4 montre un exemple de tel graphe, centré sur le chercheur mathématicien Paul Erdös.

2 Graphes : application au diamètre des réseaux sociaux

Définissons donc plus formellement ce qu'est un graphe, pour pouvoir mieux raisonner dessus.

Définition 1. Un graphe est la donnée d'un ensemble fini S de *sommets* (ou nœuds) et d'un ensemble fini A de paires de sommets qu'on appelle *arêtes* : si u et v sont deux sommets,

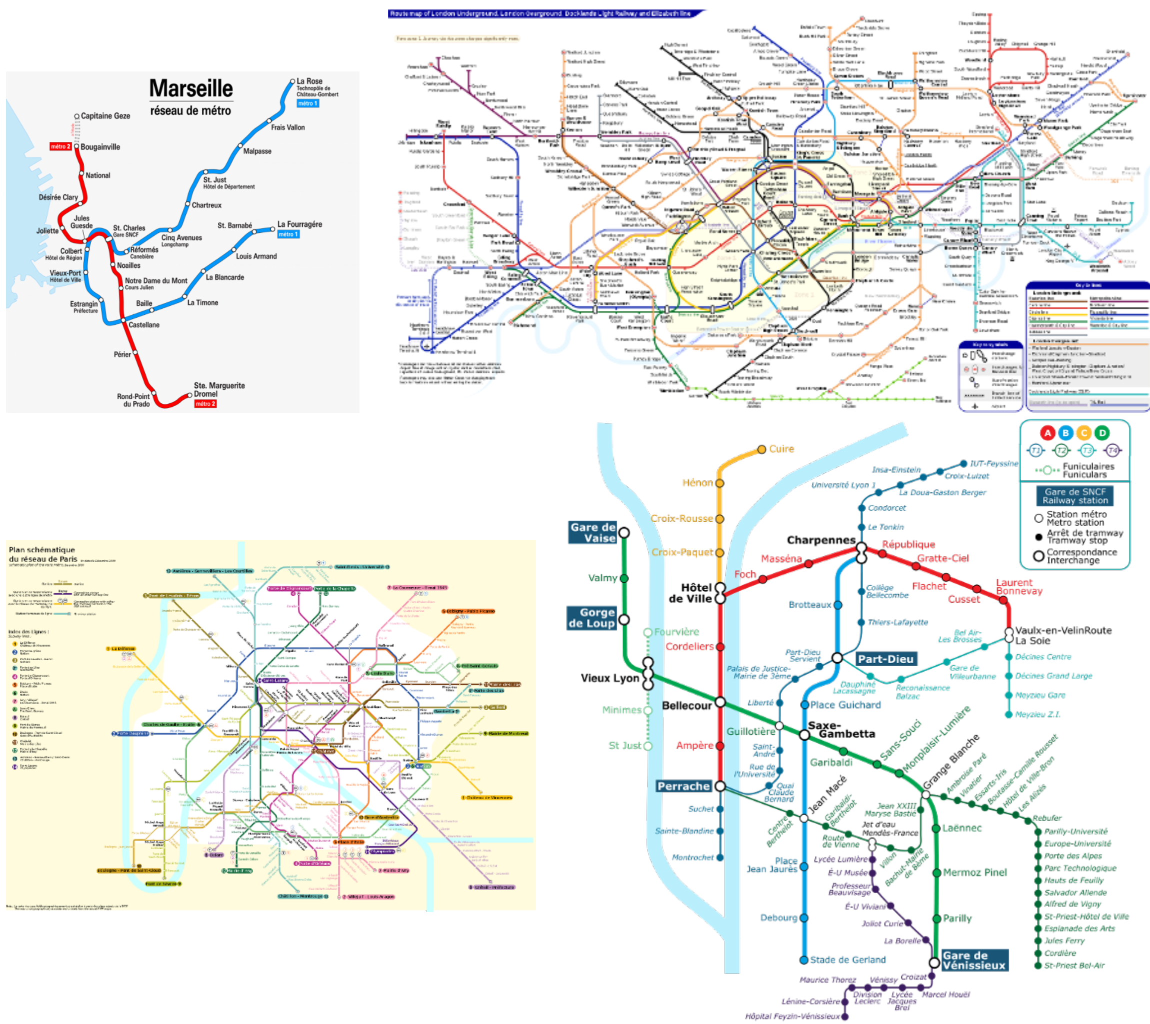


FIGURE 1 – Réseaux de métro

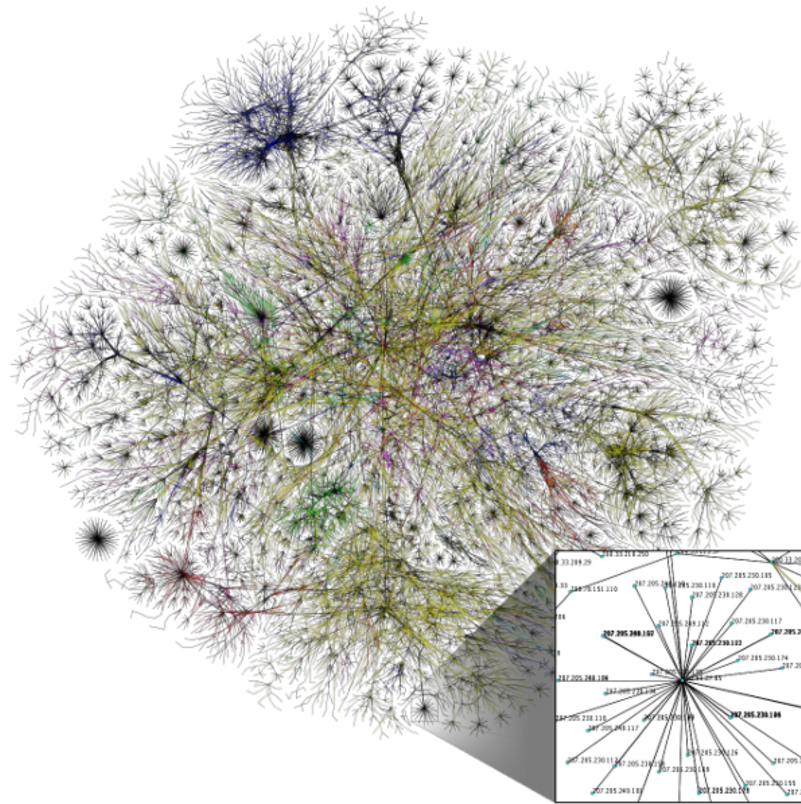


FIGURE 2 – Illustration d'une partie du graphe d'Internet

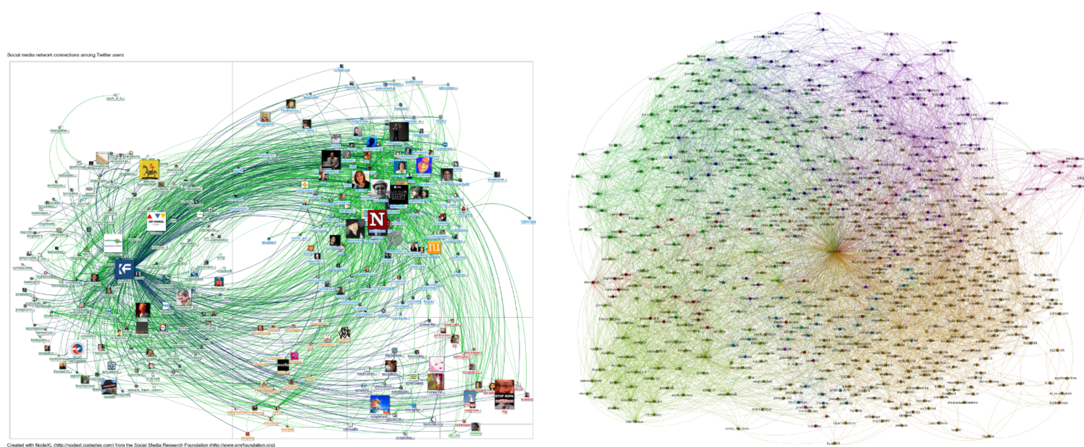


FIGURE 3 – Graphes tirés des connexions entre utilisateurs de réseaux sociaux tels que Twitter (à gauche) ou Instagram (à droite)

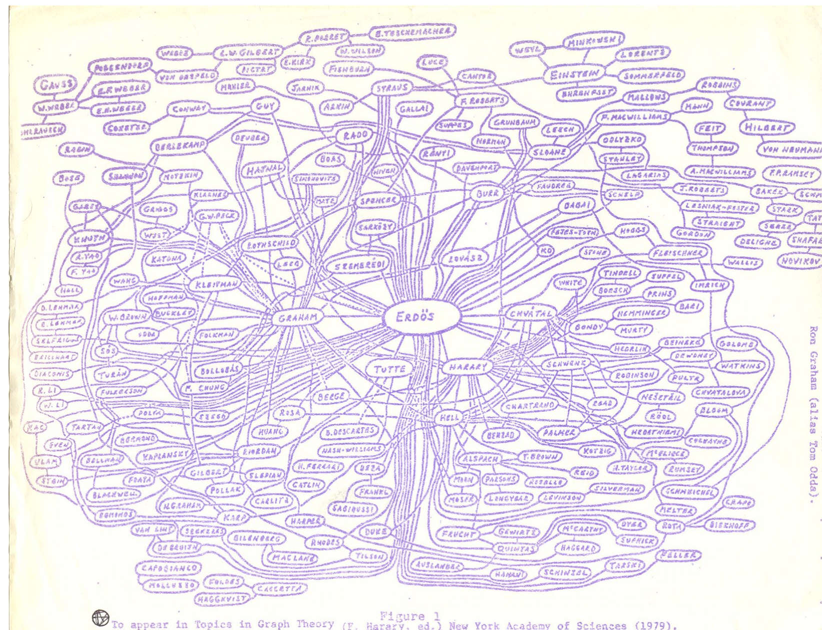
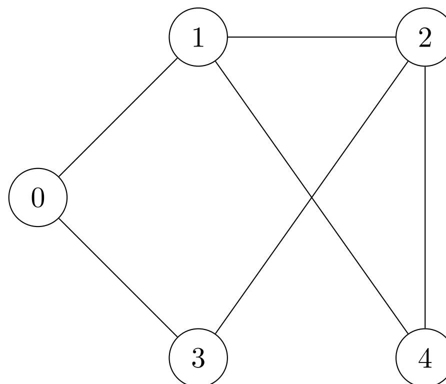


FIGURE 4 – Graphe de collaborations scientifiques centré sur Paul Erdös

l'arête $\{u, v\}$ représente le fait que les sommets u et v sont reliés dans le graphe. On note parfois $G = (S, A)$ le graphe.

Voici un exemple de graphe pour lequel l'ensemble de sommets est $S = \{0, 1, 2, 3, 4\}$ et l'ensemble d'arêtes est $A = \{\{0, 1\}, \{0, 3\}, \{1, 2\}, \{1, 4\}, \{2, 4\}, \{2, 3\}\}$:



Un *chemin* (on dit parfois aussi *chaîne*) dans un tel graphe $G = (S, A)$ est une suite de sommets $s_1, s_2, s_3, \ldots, s_{n-1}, s_n$ qui sont reliés par une arête, c'est-à-dire tel que $\{s_1, s_2\} \in A$, $\{s_2, s_3\} \in A$, …, et $\{s_{n-1}, s_n\} \in A$. La longueur d'un chemin est le nombre d'arêtes qu'il emprunte. Par exemple $0, 1, 2, 4, 2, 3$ est un chemin de longueur 5 dans le graphe précédent.

Un graphe non orienté permet, par exemple, de représenter les relations de connaissance ou d'amitié dans un réseau social (de telles relations sont généralement symétriques : Marc est ami avec Sarah si et seulement si Sarah est amie avec Marc…). La FIGURE 5 donne un petit aperçu d'un tel réseau social : les sommets sont les utilisateurs du réseau social et les arêtes représentent les liens de connaissance ou d'amitié.

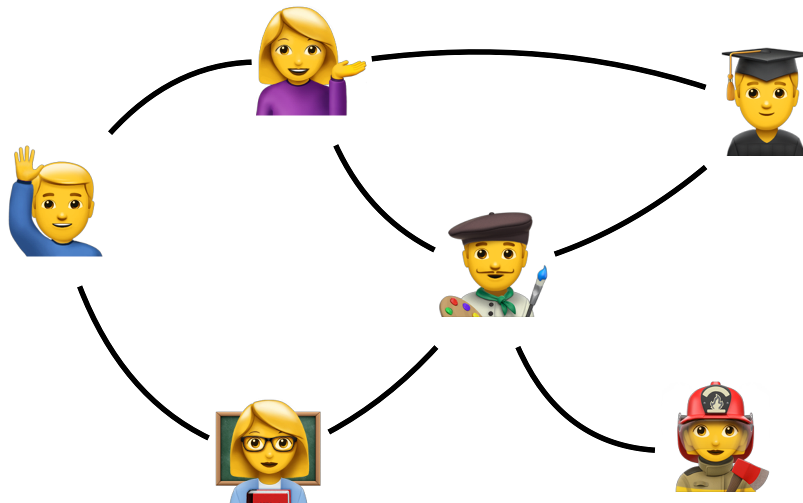


FIGURE 5 – Aperçu du graphe d'un réseau social

Modéliser un tel réseau social sous la forme d'un graphe permet de visualiser le réseau (comme dans les figures du début du chapitre...), mais aussi de raisonner sur le réseau. Par exemple, des recherches ont montré qu'« Un utilisateur de Facebook (parmi les 1,59 milliards d'utilisateurs) est connecté à n'importe quelle autre personne par le biais de 3,5 personnes en moyenne » (voir le post en anglais de Facebook Research `https://research.fb.com/three-and-a-half-degrees-of-separation/`) : cela veut dire que sur Facebook par exemple, il y a de fortes chances que Madonna soit amie avec un des amis des amis de vos amis. Cela fait référence à l'expérience que le psychologue social Stanley Milgram avait conduite dans les années 60, montrant qu'en moyenne, deux individus américains ne sont séparés que par 6 degrés de connaissance en moyenne.

Formellement, la distance entre deux sommets est la longueur minimale d'un chemin qui les relie. La distance moyenne entre deux sommets du graphe de Facebook est donc estimée à 3,5. Cela permet d'en déduire des informations sur le diamètre du graphe de Facebook : le *diamètre* d'un graphe est la distance maximale séparant deux sommets quelconques du graphe.

3 Graphes orientés : application aux graphes de configuration

Considérons désormais le plan d'une ville : il y a des routes qui se croisent à des intersections. On peut donc sur-imprimer au-dessus du plan un graphe dont les sommets sont les intersections et les arêtes sont les morceaux de route sans intersection. On a représenté une partie d'un tel graphe en FIGURE 6.

Trouver un chemin pour aller d'un point A à un point B dans la ville, revient donc à trouver un chemin du sommet A au sommet B dans le graphe. Si cela fonctionne parfaitement pour trouver un chemin pour un piéton, c'est nettement moins intéressant pour une voiture, puisqu'on n'a pas l'information des sens interdits ! Il nous faut donc ajouter cette information dans les graphes, qu'on appelle ainsi *orientés*. Un exemple de graphe orienté pour le plan de Marseille est donné en FIGURE 7.

Figure 6 – Graphe sur-imprimé sur un morceau du plan de Marseille

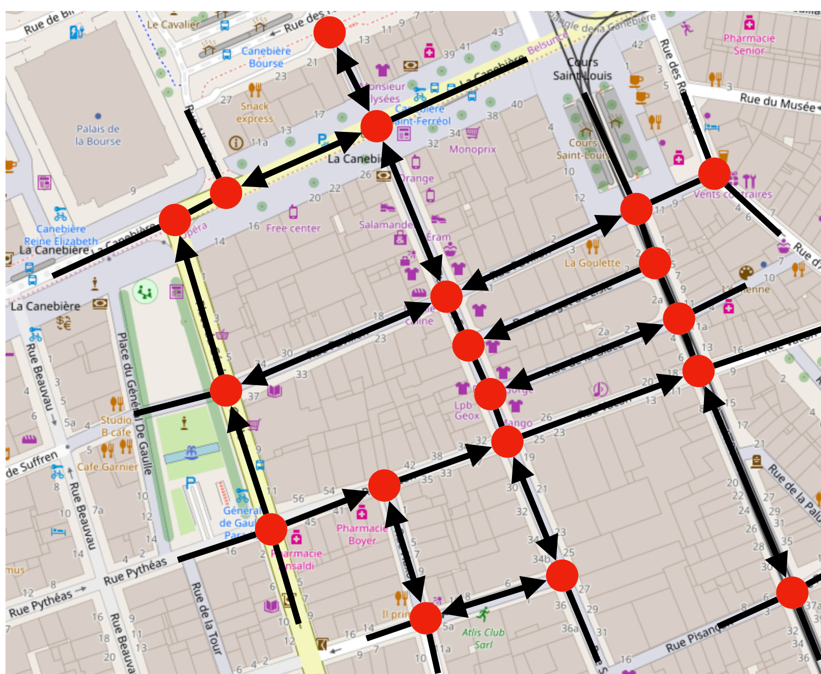
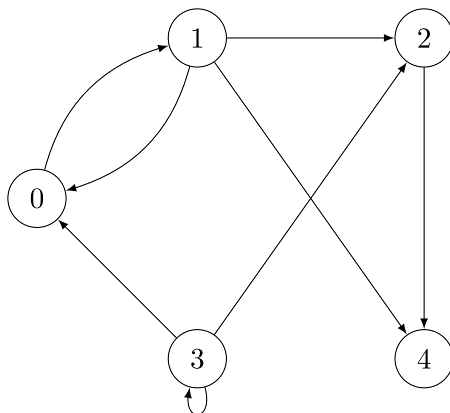


Figure 7 – Graphe orienté sur-imprimé sur un morceau du plan de Marseille

Définition 2. Un graphe orienté est la donnée d'un ensemble fini S de *sommets* (ou nœuds) et d'un ensemble fini A de couples de sommets qu'on appelle *arcs* : si u et v sont deux sommets, l'arc (u, v) représente le fait qu'il existe un arc partant du sommet u et allant au sommet v. On note parfois $G = (S, A)$ le graphe.

Notez qu'on utilise la notation $\{u, v\}$ pour une paire de sommets (différents), une arête dans un graphe non orienté, alors qu'on utilise la notation (u, v) pour un couple de sommets (potentiellement le même sommet deux fois), un arc dans un graphe orienté. En particulier, si 1 et 2 sont deux sommets du graphe, les arêtes $\{1, 2\}$ et $\{2, 1\}$ sont les mêmes, alors que les arcs $(1, 2)$ et $(2, 1)$ sont différents. Pour les rues dans un plan, une rue à sens unique sera représentée par un arc dans un seul sens, alors qu'une rue à double-sens est représentée par deux arcs, un pour chaque sens.

Voici un exemple de graphe orienté pour lequel l'ensemble de sommets est $S = \{0, 1, 2, 3, 4\}$ et l'ensemble d'arcs (qu'on représente par des liens avec des flèches) est $A = \{(0, 1), (1, 0), (1, 2), (1, 4), (2, 4), (3, 0), (3, 2), (3, 3)\}$:



Un *chemin* dans un tel graphe orienté $G = (S, A)$ est une suite de sommets $s_1, s_2, s_3, \ldots, s_{n-1}, s_n$ qui sont reliés par un arc, c'est-à-dire tel que $(s_1, s_2) \in A$, $(s_2, s_3) \in A$, ..., et $(s_{n-1}, s_n) \in A$. La longueur d'un chemin est le nombre d'arcs qu'il emprunte. Par exemple $3, 3, 0, 1, 4$ est un chemin de longueur 4 dans le graphe orienté précédent.

Les graphes orientés permettent de représenter des *graphes des configurations* qu'on utilise dans diverses applications. Si on cherche à modéliser un système qui évolue dans le temps, on appelle configuration d'un tel système son état à un instant donné : l'évolution du système, sa dynamique, peut alors parfois être représentée comme un graphe dans lequel les sommets abritent les configurations et un arc relie une configuration c_1 à une configuration c_2 s'il est possible de passer de manière élémentaire de c_1 à c_2 lors de l'évolution du système.

Exercice 1

Dans le film *Die Hard 3* (*Une journée en enfer*), les deux héros, John McClane et Zeus Carver, doivent résoudre l'énigme de Simon Gruber pour arrêter le compte à rebours d'une bombe. Voici l'énigme : « Sur la fontaine, il y a deux bidons : l'un a une contenance de 5 gallons, l'autre de 3 gallons. Remplissez l'un des bidons de 4 gallons d'eau exactement et placez-le sur la balance. La minuterie s'arrêtera. Soyez extrêmement précis : un gramme de plus ou de moins et c'est l'explosion ! ». Les nerfs de John McClane sont alors mis à rude épreuve pour trouver une solution. Il commence par remarquer très justement qu'on ne peut pas remplir le bidon de 3 gallons avec 4

gallons d'eau. Il faut donc trouver le moyen de mettre exactement 4 gallons d'eau dans le bidon de 5 gallons. Dans la scène du film (que vous pouvez consulter en français sur `https://www.youtube.com/watch?v=pmk2mNf9iqE`), John commence par donner une première idée peu convaincante puisqu'elle termine par la nécessité de remplir le bidon de 3 gallons au tiers, ce qu'on ne sait faire précisément... Le film propose ensuite une solution très partielle, coupée au montage. Appliquons donc des méthodes de graphes orientés pour retrouver la meilleure solution possible que les héros appliquent pour s'en sortir.

1. Une configuration du système correspond au volume d'eau contenu dans chacun des deux bidons. On peut donc représenter une telle configuration par une paire (a, b) où a est le volume d'eau contenu dans le bidon de 5 gallons et b le volume d'eau contenu dans le bidon de 3 gallons, avec $0 \leq a \leq 5$ et $0 \leq b \leq 3$. Les actions élémentaires possibles du système sont de remplir un des deux bidons (qu'il soit initialement vide ou pas), vider un des deux bidons (qu'il soit initialement plein ou pas) et transférer le contenu d'un des bidons dans l'autre jusqu'à ce que ce dernier soit plein. En partant de la configuration initiale $(0, 0)$, construire le graphe orienté décrivant entièrement l'ensemble des configurations et la dynamique du système. *Attention, certains mouvements ne sont possibles que dans un seule sens, raison pour laquelle nous utilisons un graphe orienté dans ce cas.*
2. En déduire une solution la plus courte possible permettant d'aller de la configuration $(0, 0)$ à une configuration de la forme $(4, b)$ où exactement 4 gallons d'eau se trouvent dans le gros bidon. Décrire à John McClane la suite d'opérations qu'il doit effectuer pour arrêter la bombe au plus vite.
3. Imaginons la suite de l'énigme désormais. Dans une autre fontaine, le terroriste a placé deux bidons, l'un de 6 gallons et l'autre de 15 gallons. S'il demande à John McClane de remplir l'un des deux bidons avec exactement 5 gallons d'eau, pourra-t-il éviter l'explosion de la bombe ?

4 Codage d'un graphe

Pour pouvoir calculer sur un graphe, il faut savoir comment on va coder un graphe. La façon la plus simple (il existe d'autres façons qu'on n'étudiera pas dans ce cours) consiste à stocker le graphe à l'aide d'une *matrice d'adjacence*. Il s'agit d'un tableau bidimensionnel M dont les lignes et les colonnes sont indexées par les sommets du graphe et tel que la case $M[u, v]$ en ligne u et en colonne v vaut 1 dès lors qu'il existe un arc (u, v) ou une arête $\{u, v\}$ dans le graphe, et vaut 0 sinon. Voici les matrices d'adjacence du graphe non orienté (à gauche) et orienté (à droite) présentés plus tôt :

$$M_1 = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

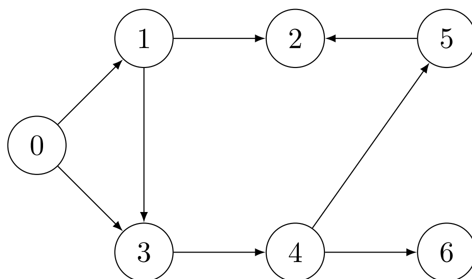
Par exemple, on a $M_1[2,3] = 1$ signifiant qu'il existe une arête reliant les sommets 2 et 3 dans le graphe non orienté, et $M_2[2,1] = 0$ signifiant qu'il n'y a pas d'arc allant de 2 à 1 dans le graphe orienté.

5 Parcours de graphe

Il est temps d'automatiser les calculs que l'on souhaite faire dans des graphes, que ce soit la recherche d'un itinéraire pour aller d'un point A à un point B, ou la résolution d'énigmes telles que celles de Pagnol ou de Die Hard 3. Le point commun de ces différents problèmes est la recherche d'un chemin allant d'un sommet source à un sommet cible. On résout ce problème d'*accessibilité* dans un graphe à l'aide d'un algorithme qui *parcourt* le graphe, c'est-à-dire visite les sommets du graphe petit à petit en suivant des arêtes/arcs du graphe. Pour simplifier, on donne les explications qui suivent uniquement dans le cas des graphes orientés, mais tout fonctionne de la même façon avec des graphes non orientés.

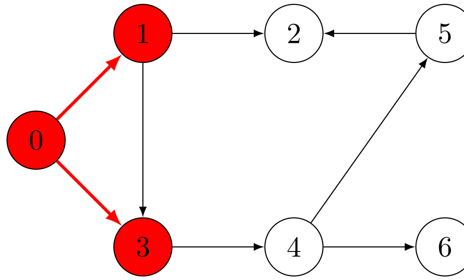
L'algorithme de parcours qu'on va considérer est le *parcours en largeur*. On part d'un sommet source s du graphe. L'idée est alors d'imiter le gonflement d'un ballon de baudruche depuis s : au fur et à mesure où le ballon se gonfle, il voit de plus en plus de sommets du graphe, ceux qui sont de plus en plus éloignés du sommet s. Sur le cours Ametice en ligne, je vous présente un exemple du parcours en largeur dans une vidéo. Si vous pouvez y accéder, faites-le avant de lire la suite.

Pour exécuter le parcours en largeur, nous allons utiliser une nouvelle structure de données, les files, qui permettent de stocker un ensemble de clients en se souvenant de l'ordre dans lequel ils sont arrivés afin de servir les clients dans leur ordre d'arrivée. Dans le cas du parcours en largeur, les clients seront les sommets du graphe. Considérons ainsi le graphe ci-dessous qu'on souhaite parcourir à partir de la source $s = 0$:



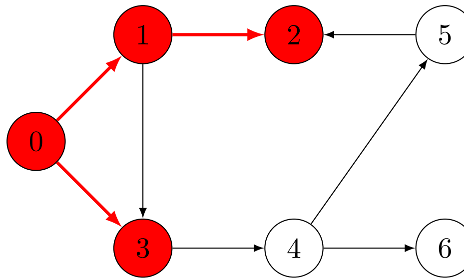
On va colorier les sommets avec la couleur rouge (en plus du blanc initial) pour enregistrer de l'information. La première chose qu'on fait est d'ailleurs de colorer en rouge la source $s = 0$ et de l'insérer dans la file d'attente. Ensuite, on traite l'unique sommet dans la file d'attente. Traiter un sommet u signifie regarder tous ses voisins, c'est-à-dire considérer tous les sommets v tels que (u, v) est un arc du graphe. En l'occurrence, puisqu'on traite le sommet 0, on va donc considérer ses voisins 1 et 3. Pour chacun de ses voisins, s'ils sont blancs, on les colorie en rouge et on les insère dans la file d'attente. Après avoir traité entièrement le sommet 0, on arrive donc dans la situation suivante, dans laquelle on colorie également en rouge les arcs qui nous ont permis d'aller de la source 0 aux différents sommets déjà découverts :

File : $1, 3$



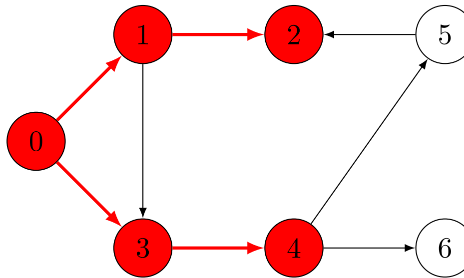
La file d'attente contient désormais les sommets 1 et 3, et on va supposer qu'on y a inséré d'abord 1, puis 3. Ainsi, on extrait d'abord de la file d'attente le sommet 1. Il a deux voisins, 2 et 3. Pour le sommet 2, comme avant, on le colore en rouge et on l'insère dans la file. Mais le sommet 3, lui, est déjà rouge et on ne le considère donc pas : en particulier, on ne colorie pas en rouge l'arc $(1, 3)$. Une fois le sommet 1 entièrement traité, on est donc dans la situation suivante :

File : $3, 2$



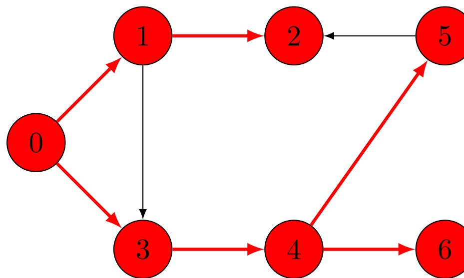
On traite ensuite le sommet inséré il y a le plus longtemps dans la file d'attente, c'est-à-dire le sommet 3. Il n'a qu'un seul voisin, le sommet 4, qu'on colorie donc en rouge :

File : $2, 4$

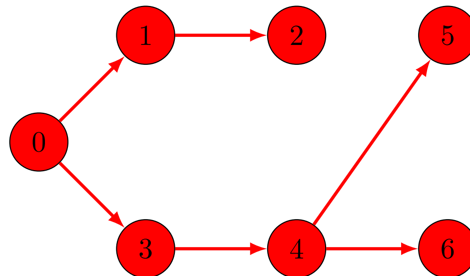


On traite ensuite le sommet 2, qui n'a aucun voisin. Il ne reste ensuite que le sommet 4 dans la file d'attente. Il a deux voisins blancs, les sommets 5 et 6, qu'on traite donc pour obtenir la situation

File : $5, 6$



Il reste ensuite les sommets 5 et 6 à traiter, dans cet ordre, mais tous les sommets étant déjà rouges, on ne fait rien de plus. Le graphe précédent est donc la situation finale sur laquelle on s'arrête. Observons qu'on a bien découvert tous les sommets du graphe qu'on pouvait atteindre via un chemin depuis le sommet 0. De plus, si on ne regarde que les arêtes colorées en rouge :



on obtient une représentation d'un chemin possible pour aller de la source $s = 0$ à n'importe quel sommet rouge : pour aller de 0 à 5 par exemple, il faut suivre le chemin $0, 3, 4, 5$. Notons qu'il s'agit d'un plus court chemin (en terme du nombre d'arcs visités) pour aller de 0 à 5. C'est une propriété toujours satisfaite par le parcours en largeur : il permet de construire les plus courts chemins issus de la source s.

Voici une façon de décrire, à l'aide d'un pseudo-code, l'algorithme de parcours en largeur, dans lequel on suppose que les n sommets du graphe sont numérotés $\{0, 1, 2, \ldots, n-1\}$ comme dans les exemples précédents :

```
fonction parcours_en_largeur(M, s) :
    n := nombre_sommets(M)
    H := graphe_vide(n)          #  graphe sans arcs avec n sommets
    F := file d'attente vide
    couleur := tableau de longueur n rempli de 'blanc'
    couleur[s] := rouge
    insérer(F, s)
    Tant que F ≠ ∅ faire
        u := extraire(F)
        Pour v de 0 à n − 1 faire
            Si (M[u,v] = 1 et couleur[v] = blanc) alors
                couleur[v] := rouge
                H[u,v] := 1
                insérer(F, v)
            FinSi
        FinPour
    FinTantQue
    retourner(H)                 #  graphe des chemins minimaux
```

Il prend en argument la matrice d'adjacence M du graphe, et le sommet source s. Cet algorithme retourne la matrice d'adjacence d'un graphe H qui ne contient que les arcs traversés en parcourant un chemin de s à chacun des autres sommets accessibles, c'est-à-dire les arcs rouges dans l'explication précédente. Pour insérer et extraire des éléments de la file d'attente, on suppose connues deux fonctions : `insérer(F,u)` qui insère le sommet `u` dans la file d'attente `F`, et `extraire(F)` qui renvoie le sommet en tête de la file d'attente `F`. Les couleurs des sommets sont stockées dans un tableau `couleur`.

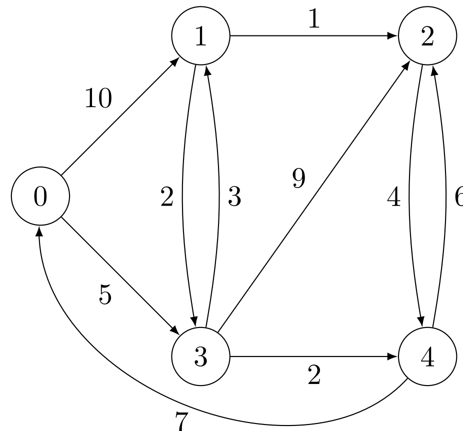
6 Plus courts chemins dans des graphes pondérés : algorithme de Dijkstra

L'algorithme de parcours en largeur permet donc de trouver des plus courts chemins dans des graphes. Ici, plus court veut dire « qui utilise le moins d'arcs/arêtes possible ». C'est suffisant pour résoudre le problème de Pagnol et de Die Hard 3, mais pas pour trouver des plus courts chemins dans un plan, puisqu'alors ce n'est pas tant le nombre d'arcs qui compte que le temps ou la distance parcourue au total le long de l'itinéraire choisi.

Pour résoudre ce problème, il faut donc incorporer dans la modélisation en graphe une information supplémentaire, que ce soit le temps pour parcourir l'arc (c'est-à-dire le temps qu'il faut pour aller en voiture d'une intersection à une autre), ou plus simplement la longueur de l'arc (c'est-à-dire la distance qui sépare les deux intersections). On utilise pour cela la notion de graphe pondéré.

Définition 3. Un graphe pondéré est la donnée d'un graphe orienté (S, A) et d'une fonction $p\colon A \to \mathbf{N}$ associant un poids (qu'on suppose entier naturel ici) à chaque arc du graphe. On le représente à l'aide d'une matrice d'adjacence M à coefficients dans $\mathbf{N} \cup \{+\infty\}$ dans laquelle le coefficient $M[u, v]$ vaut $p(u, v)$ si $(u, v) \in A$, et vaut $+\infty$ sinon.

Voici un exemple de graphe pondéré dans lequel on fait apparaître le poids de l'arc à côté de celui-ci :



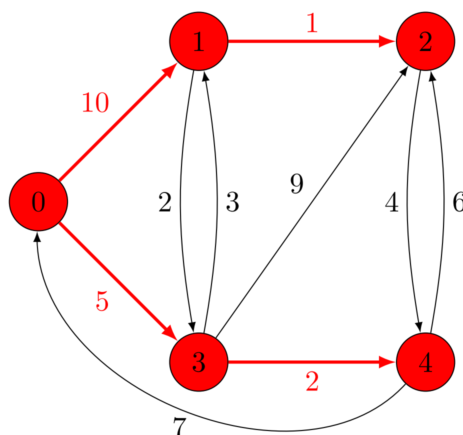
On a ainsi, par exemple, $p(0, 1) = 10$ et $p(4, 0) = 7$. On peut coder ce graphe pondéré avec la matrice d'adjacence

$$M = \begin{pmatrix} +\infty & 10 & +\infty & 5 & +\infty \\ +\infty & +\infty & 1 & 2 & +\infty \\ +\infty & +\infty & +\infty & +\infty & 4 \\ +\infty & 3 & 9 & +\infty & 2 \\ 7 & +\infty & 6 & +\infty & +\infty \end{pmatrix}$$

Pour un chemin $s_1, s_2, s_3, \dots, s_{n-1}, s_n$ dans le graphe (S, A), le *poids* du chemin est la somme des poids des arcs empruntés, c'est-à-dire $p(s_1, s_2) + p(s_2, s_3) + \cdots + p(s_{n-1}, s_n)$. Un *plus court chemin* d'un sommet u à un sommet v est un chemin de poids minimal parmi tous les chemins allant de u à v (s'il existe un tel chemin). Par exemple, le chemin $0, 1, 2$ est

un chemin de 0 à 2, mais ce n'est pas un plus court chemin car le chemin $0, 3, 1, 2$ est plus *court* en terme de poids : c'est d'ailleurs un plus court chemin pour aller de 0 à 2. On appelle *distance* de u à v le poids d'un plus court chemin de u à v. Par exemple, la distance de 0 à 2 est donc 9, alors que la distance de 0 à 4 vaut 7.

On cherche donc un algorithme qui permet de calculer la distance d'une source s à tout autre sommet, de la même manière que le parcours en largeur permettait de trouver tous les sommets qu'on peut atteindre à partir de s. En plus des distances, on aimerait aussi pouvoir trouver des plus courts chemins. Clairement l'algorithme de parcours en largeur n'est plus correct, puisqu'il renvoie le résultat suivant pour le graphe pondéré précédent, à partir de la source $s = 0$:

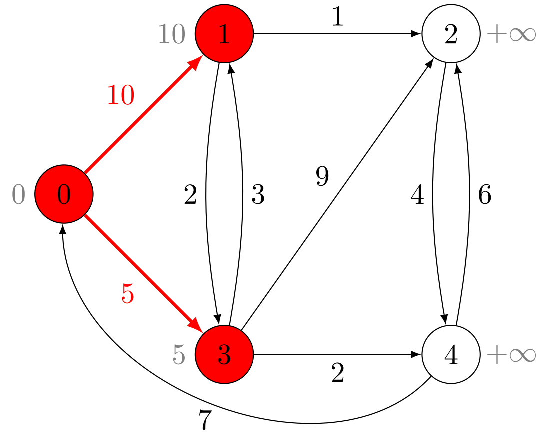


ce qui est incorrect puisqu'alors on en déduirait que la distance de 0 à 1 vaut 10, alors même qu'elle vaut 8.

On doit donc corriger l'algorithme. C'est Edsger Wybe Dijkstra (né en 1939 et mort en 2002) qui a proposé une façon élégante de corriger l'algorithme de parcours en largeur pour qu'il fonctionne dans des graphes pondérés. Son idée : plutôt que de stocker les sommets à traiter dans une file d'attente, utilisons donc plutôt une file de priorité. Il a donc utilisé le concept de file prioritaire à la caisse d'un magasin, dans le contexte des graphes pondérés... Ici, chaque sommet u aura donc une priorité qui correspond à l'estimation courante qu'on a de la distance de la source s à u. Au début, on part avec une estimation très pessimiste associant une distance $+\infty$ à tout sommet u, sauf la source à qui on associe la distance 0. À chaque étape, on traite le sommet qui a *la plus faible priorité* dans la file d'attente, c'est-à-dire celui dont on pense qu'il est le plus proche de la source possible.

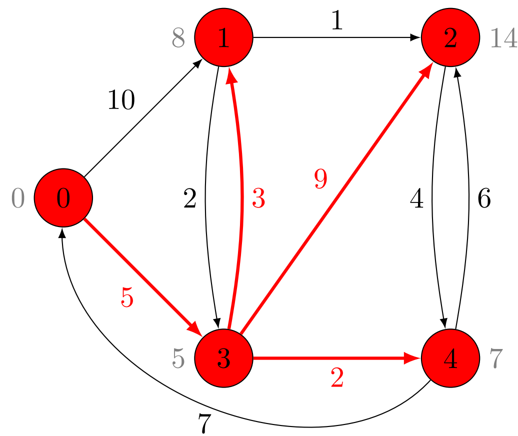
Exécutons l'algorithme de Dijkstra sur l'exemple précédent. Comme pour le parcours en largeur, on commence par colorier la source $s = 0$ en rouge et lui associer la distance 0. On traite alors ce sommet en considérant ses deux voisins, 1 et 3. On peut donc mettre à jour les distances connues de la source à ces deux sommets. On insère alors dans la file de priorité les sommets avec ces distances. On représente ci-dessous la file de priorité en rappelant pour chaque sommet de la file sa priorité. On imprime aussi à côté de chaque sommet du graphe la distance estimée courante.

File : 3 (5), 1 (10)



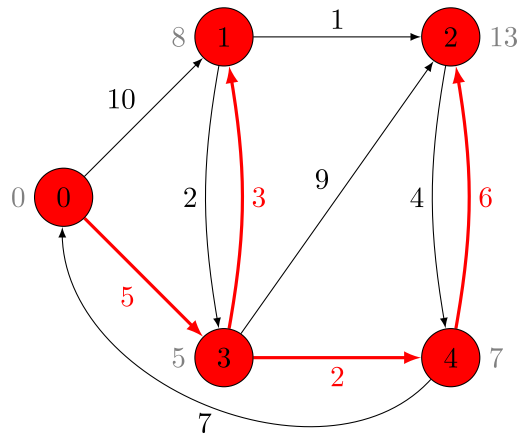
Bien qu'on ait sans doute traiter le voisin 1 avant le voisin 3, la priorité de 3 est plus faible que celle de 1 : c'est donc avec ce sommet qu'on continue. On considère donc les voisins de 3, à savoir 1, 2 et 4. C'est là que l'algorithme se distingue du parcours en largeur. Ici, même si le sommet 1 est rouge, il faut mettre à jour l'information de distance, puisqu'on a trouvé un chemin (0,3,1) de poids 8, plus petit que l'estimation courante de la distance. On doit donc mettre à jour les distances et les priorités dans la file (cela revient à dire qu'on a une file à la caisse d'un magasin dans laquelle les priorités des clients peuvent changer au cours du temps...). On met également à jour la coloration des arcs, puisqu'on souhaite que les arcs colorés en rouge représentent les plus courts chemins. On traite ensuite les voisins 2 et 4 en les insérant dans la file d'attente. On arrive donc à la situation suivante :

File : 4 (7), 1 (8), 2 (14)



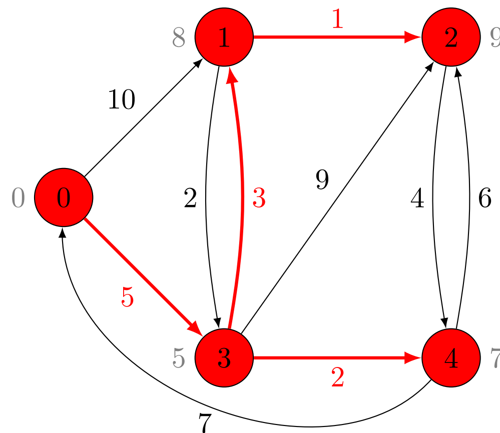
Le sommet prioritaire dans la file est désormais le sommet 4. Rien à faire avec son voisin 0, puisque l'estimée courante de la distance, 0, est imbattable. Par contre, on met à jour l'estimation de la distance pour le voisin 2 :

File : 1 (8), 2 (13)



On traite de même le sommet 1 qui remet à jour la distance estimée au sommet 2 :

File : 2 (9)



Il reste le sommet 2 à traiter qui n'induit aucun changement : le graphe précédent est donc la situation finale. Ce graphe contient les informations des distances de 0 à chacun des sommets et on peut également reconstituer des plus courts chemins, en ne considérant que les arcs rouges.

Afin d'écrire le pseudo-code de cet algorithme (rassurez-vous, ce sera l'algorithme le plus difficile de ce cours, il n'est donc pas impossible qu'il vous donne quelques sueurs froides...), il nous faut disposer d'une file de priorité. On suppose donnée une telle file F, où chacun de ses éléments est associé à une priorité (un entier naturel) ainsi que les opérations suivantes :

— on peut insérer un nouvel élément u dans la file, en précisant sa priorité $p \in \mathbf{N}$ à l'aide de la fonction `insérer_priorité(F, u, p)` ;

— l'élément prioritaire est celui de **plus petite priorité** : on peut récupérer cet élément à l'aide de la fonction `u := extraire_prioritaire(F)` ;

— on peut mettre à jour la priorité d'un élément u de la file pour lui attribuer la nouvelle priorité p dans F, à l'aide de la fonction `mettre_à_jour_priorité(F, u, p)`.

Une autre différence notable entre le parcours en largeur et l'algorithme de Dijkstra réside dans la nécessité de maintenir les estimations des distances (on utilise un tableau `distance`

pour cela) et de mettre à jour les arcs rouges : pour cela, plutôt que de maintenir un graphe `H` comme précédemment, on utilise plutôt un tableau **`prédécesseur`** qui associe à chaque sommet v rouge son prédécesseur dans le graphe `H`, c'est-à-dire l'unique sommet u tel que (u, v) est un arc rouge de `H`. Si le sommet est blanc (ou pour la source qui n'a pas de tel prédécesseur), on utilise le symbole $\perp$ pour dénoter l'absence de prédécesseur. Ces transformations permettent d'obtenir l'algorithme suivant :

```
fonction dijkstra(G, s) :
    n := nombre_sommets(G)
    distance := tableau de taille n rempli de +∞
    prédécesseur := tableau de taille n rempli de ⊥
    F := file de priorité vide
    couleur := tableau de longueur n rempli de 'blanc'
    couleur[s] := rouge
    distance[s] := 0
    insérer_priorité(F, s, 0)
    Tant que F ≠ ∅ faire
        u := extraire_prioritaire(F)
        d := distance[u]
        Pour v de 0 à n − 1 faire
            Si (couleur[v] = blanc et G[u,v] ≠ +∞) alors
                couleur[v] := rouge
                prédécesseur[v] := u
                distance[v] := d+G[u,v]
                insérer_priorité(F, v, d+G[u,v])
            Sinon Si (d+G[u,v] < distance[v]) alors
                mettre_à_jour_priorité(F, v, d+G[u,v])
                prédécesseur[v] := u
                distance[v] := d+G[u,v]
            FinSi
        FinPour
    FinTantQue
    retourner(prédécesseur)
```