

Queste de savoir

# À la découverte des algorithmes de graphe

---

12 août 2019



# Table des matières

<b>I. Bases de la théorie des graphes</b>	<b>3</b>
<b>1. Graphes et représentation de graphe</b>	<b>4</b>
1.1. Dessine moi un graphe!	4
1.1.1. Le graphe	4
1.1.2. Une carte, un labyrinthe, un lieu... et la recherche de chemins	5
1.1.3. Des relations entre individus	5
1.1.4. L'ordonnancement de tâches : le tri topologique	5
1.1.5. Minimiser l'usage de ressources avec les arbres couvrants minimaux	6
1.1.6. Tromper votre ennui en géographie en coloriant des cartes	7
1.1.7. Poster du courrier	8
1.1.8. ...et tout le reste!	8
1.2. Un peu de vocabulaire	8
1.2.1. Quelques conventions et notations...	8
1.2.2. Graphe simple ou multigraphe	9
1.2.3. Connexité	9
1.2.4. Graphes orientés et non-orientés	10
1.2.5. Graphes pondérés	10
1.2.6. Graphes cycliques	10
1.2.7. Densité d'un graphe et degré d'un nœud	11
1.2.8. Arbre	12
1.3. Représentations et stockage en mémoire	13
1.3.1. Généralités	13
1.3.2. La liste d'adjacence	14
1.3.3. Autres représentations	15
<b>2. Parcourir un graphe</b>	<b>16</b>
2.1. Le parcours en profondeur et le labyrinthe	16
2.1.1. Les pyramides	16
2.1.2. Le parcours en profondeur	18
2.1.3. Luke, je suis ton père	21
2.2. Le parcours en largeur et le buzz	22
2.2.1. Big Buzz	22
2.2.2. Le parcours en largeur	23
2.2.3. Le prix du succès	24
2.3. L'exhaustif et Uno	25
2.3.1. L'exhaustif	27
<b>3. Les problèmes usuels de graphes</b>	<b>30</b>
3.1. Le tri topologique et make	30
3.1.1. Solution	31

Contenu masqué . . . . . 32


Ce tutoriel va vous expliquer ce qu'est un graphe, et à quoi il sert. Il détaillera les algorithmes de graphe les plus courants, en indiquant leur complexité en temps et en mémoire, avec peut-être des schémas si vous êtes sages.

Chaque algorithme sera accompagné d'un pseudo-code pour laisser au programmeur l'opportunité de le coder dans son langage favori. Le cours est ouvert aux contributions : vous pouvez m'envoyer une implémentation de l'algorithme dans le langage de votre choix et je l'y ajouterais.

Il sera appuyé d'exemples concrets pour que l'intérêt de chaque algorithme apparaisse dans une situation courante. L'objectif est d'apprendre à reconnaître des problèmes de graphe, ou à modéliser un problème sous forme de graphe, pour le résoudre avec des outils éprouvés et efficaces.

Ce document n'a pas pour ambition de faire une démonstration formelle de la complexité ou de la validité des algorithmes. Il n'a aucune vocation académique. Les concepts seront présentés de façon intuitive et didactique pour permettre à tous de les appréhender et de les appliquer aisément. Il se veut être une "boîte à outils" aussi bien qu'un moyen de découvrir ce domaine de l'algorithmique. Le formalisme et la rigueur ont été sacrifiés sur l'autel de la simplicité, mais pas l'exactitude ni la précision.

**Pour suivre ce tutoriel vous devez**

- Savoir programmer (variables, boucles, tableaux, fonctions)
- Avoir des notions de base en algorithmique : récursivité, structures de données courantes, tris...
- Comprendre ce qu'est la [complexité](#)  (facultatif mais utile)

**Première partie**  
**Bases de la théorie des graphes**

# 1. Graphes et représentation de graphe

Dans ce chapitre vous verrez la définition d'un graphe, ainsi que ses différentes caractéristiques remarquables.

Nous donnerons un premier aperçu des problèmes pouvant être identifiés comme des problèmes de graphe. Puis nous étudierons les deux principaux moyens de le stocker en mémoire.

## 1.1. Dessine moi un graphe !

### 1.1.1. Le graphe

Un graphe est un ensemble de points, dont certaines paires sont directement reliées par un (ou plusieurs) lien(s).

*Wikipédia*

Ces points sont nommés **nœuds** ou **sommets**. Ces liens sont nommés **arêtes**.



Notez bien que ces fameux points n'ont pas de position absolue dans le plan, ni de position relative les uns aux autres. De la même façon, les arêtes ne sont pas forcées d'être droites (bien que conventionnellement on les représente comme tel par souci de lisibilité) : elles peuvent se croiser, être courbées ou faire des détours, etc... Ainsi un graphe **n'est pas** une figure, bien que l'on puisse représenter un graphe par une figure. Les graphes ne se cantonnent donc pas à des considérations d'ordre purement géométrique, bien au contraire !

En effet, **un graphe sert avant tout à manipuler des concepts, et à établir un lien entre ces concepts**. N'importe quel problème comportant des objets avec des relations entre ces objets peut être modélisé par un graphe.

Il apparaît donc que les graphes sont des outils très puissants et largement répandus qui se prêtent bien à la résolution de nombreux problèmes.

En voici quelques uns.

### 1.1.2. Une carte, un labyrinthe, un lieu... et la recherche de chemins

Il s'agit d'un cas très classique, notamment dans le domaine du jeu vidéo. Chaque nœud représente une **position** (ici une case libre) et chaque arête correspond à un **chemin** entre deux positions (ici deux cases libres adjacentes).

Il est courant de chercher le chemin le plus court entre deux positions : les IA s'en servent pour déplacer les créatures aussi vite que possible dans le monde virtuel. En remplaçant les nœuds par des **adresses** et les arêtes par des **routes**, on obtient le graphe utilisé par les GPS ou Google Map par exemple.

On utilise couramment le terme anglais *pathfinding* pour désigner la recherche de chemins dans un graphe.



<http://zestedesavoir.com/media/galleries/912/0dcbf772-4c53-471a-89a5-80a64a>

### 1.1.3. Des relations entre individus

Cet outil est très utilisé en sciences sociales pour représenter des **individus** et leurs différents **liens**. On en fait usage dans le milieu de la recherche, mais aussi dans les réseaux sociaux. L'objectif va être de pouvoir identifier les communautés formées, les centres d'intérêt communs... Une exploitation intelligente de ces données est indispensable pour suggérer à l'utilisateur les choses qu'il est susceptible d'aimer, les personnes qu'il connaît peut-être, et avant tout (et surtout) pour créer des publicités ciblées adaptées à chacun.

Une anecdote à ce sujet : en moyenne seules 8 personnes vous séparent de n'importe quelle autre personne à la surface de la Terre. Si vous prenez la connaissance de la connaissance etc... d'une de vos connaissances, en moyenne 8 personnes permettront de vous connecter avec n'importe quel français, américain, indien ou japonais. Et il est rare de dépasser 11 ou 12. Il s'agit là du **diamètre moyen** du graphe social de l'humanité.



<http://zestedesavoir.com/media/galleries/912/f84449b8-1172-4f05-b133-43ec69>

### 1.1.4. L'ordonnancement de tâches : le tri topologique

Les problèmes d'ordonnancement de tâches sont toujours un sujet de recherche active aujourd'hui. On peut représenter chacune des **tâches** à effectuer par un nœud, et les **dépendances entre chacune des tâches** par les arêtes. Ainsi, l'ordre dans lequel on enfle chaussettes et pantalon importe peu, mais il faut que les deux soient mis avant de lacer ses chaussures. De la même façon, on peut installer les câblages électrique et téléphonique dans l'ordre de notre choix, mais

## I. Bases de la théorie des graphes

il faut que les murs soient construits avant !  
On résout ce problème avec un **tri topologique**.



<http://zestedesavoir.com/media/galleries/912/2215a2ef-3d28-492e-a1ca-652a95>

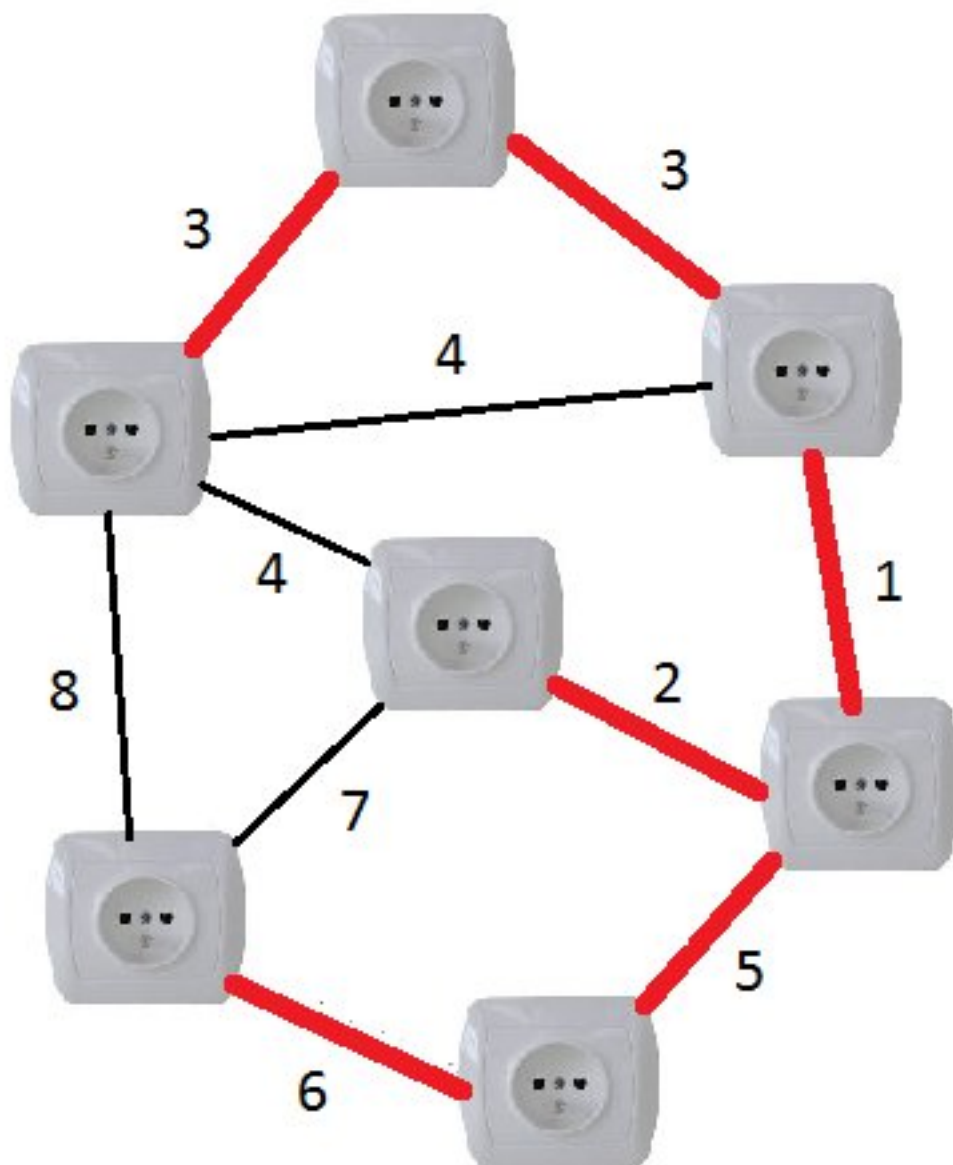
### 1.1.5. Minimiser l'usage de ressources avec les arbres couvrants minimaux

En parlant de câblage, vous souhaitez que toutes vos prises soient reliées au secteur, non ? Cela est simple ! Mais le fil électrique coûte cher au mètre et vous souhaitez minimiser le prix de votre installation.

Ainsi, vous associez un nœud à chaque **prise**, et une arête à chaque **fil** qu'il est *possible* (mais pas nécessaire) d'installer. A vous de choisir l'ensemble de câbles les plus courts pour relier toutes les prises !

Cela correspond à la recherche de l'**arbre couvrant minimal**.





Ici l'ensemble de fils les plus courts est colorié en rouge.

### 1.1.6. Tromper votre ennui en géographie en coloriant des cartes

Vous vous ennuyez à mourir en géographie. Comme vous êtes un individu froid et austère, mais aussi efficace, économe et démuné, vous souhaitez colorier cette triste carte en utilisant un minimum de couleurs, et en faisant en sorte que deux pays limitrophes ne soient jamais de la même couleur.

Il s'agit du problème de **coloration de graphe**. Pour une carte en 2D ne cherchez plus : la réponse est 4, mais ce n'est pas le cas de tout les graphes... en effet, dans un carte, chaque **pays** est un nœud, et les arêtes correspondent au **frontières communes**.



<http://zestedesavoir.com/media/galleries/912/029ad86a-cbc0-4a1a-ae7d-26aa33>

### 1.1.7. Poster du courrier

Vous êtes le facteur d'un petit village, vous distribuez le courrier à vélo, à la seule force de vos jambes. Vous devez passer devant toutes les maisons du village, ce qui implique de traverser toutes les rues. Mais soucieux de préserver vos forces et de renouveler continuellement votre découverte des paysages, vous ne voulez pas traverser deux fois la même rue. Ici chaque nœud est un **carrefour**, et chaque arête une **rue**. Vous êtes en train de chercher un **circuit Eulérien**!

Il doit son nom à Leonhard Euler, qui chercha à savoir s'il était possible de franchir les 7 ponts de Königsberg sans jamais repasser deux fois sur le même (et en ne traversant le fleuve que grâce aux ponts, bien entendu).



<http://zestedesavoir.com/media/galleries/912/f16ed77f-92ef-4a07-bd8f-e41f37>

### 1.1.8. ...et tout le reste!

Pensez à Youtube qui suggère des vidéos proches de celle que vous regardez (d'aucun prétendent qu'on finira toujours par aboutir à des vidéos de chats), pensez à wikipédia qui propose des articles **connexes** (nous reviendrons sur ce terme), pensez au cerveau humain, à ses milliards de neurones et à ses centaines de milliards de connexions entre eux! Les graphes sont partout.

Apprenez à reconnaître un problème de graphe lorsque vous en voyez un, c'est déjà la moitié du travail. Il ne vous reste alors plus qu'à trouver l'algorithme qui répond à votre problème!

## 1.2. Un peu de vocabulaire

### 1.2.1. Quelques conventions et notations...

Maintenant que nous avons une idée de ce qu'est un graphe, tâchons d'employer un vocabulaire précis pour traiter de chacune de ses caractéristiques. Certains algorithmes ne fonctionnent que sur des graphes possédant certaines particularités.

Pouvoir décrire en quelques mots les caractéristiques principales d'un graphe est donc indispensable, et vous devez toujours avoir le réflexe de le faire sitôt le graphe identifié. Cela vous guidera vers le choix de l'algorithme approprié.

## I. Bases de la théorie des graphes

Fixons d'ores et déjà quelques notations, par soucis de clarté. Ces notations sont conventionnellement utilisées dans la plupart des cours ou articles parlant de graphes.

L'ensemble des nœuds du graphe est désigné par  $N$ . L'ensemble des arêtes est désigné par  $A$ . Le graphe  $G$  est simplement défini comme  $G = (N, A)$ . Ainsi, la représentation du graphe importe peu : les deux graphes ci-dessous sont *isomorphes* (ici, on peut traduire grossièrement ce terme barbare par *équivalents*).



### 1.2.2. Graphe simple ou multigraphe

Une **boucle** est une arête qui relie un nœud à lui même.

Un **lien double** caractérise l'existence de plusieurs arêtes entre deux nœuds donnés.

Un graphe possédant l'une ou l'autre de ces caractéristiques est dit **multigraphe**. Un graphe ne possédant aucune des deux est dit **graphe simple**.

Nous travaillerons exclusivement sur des graphes simples par soucis de simplicité : ils couvrent la plupart des utilisations, et sont plus simples à traiter que les multigraphes dans le cas général.



FIGURE 1.1. – Multigraphe, avec une boucle en bleue et les liens double en rouge

### 1.2.3. Connexité

Un graphe est dit **connexe** lorsqu'il existe un chemin entre toute paire de nœuds. Une composante connexe d'un graphe est un **sous-graphe** connexe de ce graphe. Un sous-graphe est un sous-ensemble de nœuds du graphe, avec une partie de leurs arêtes associées. Ainsi, sur le dessin ci-dessous, vous ne voyez **qu'un seul et unique graphe**, comportant 3 composantes connexes.



FIGURE 1.2. – Graphe simple avec 3 composantes connexes

### 1.2.4. Graphes orientés et non-orientés

Voici une autre caractéristique fondamentale d'un graphe. Dans un **graphe orienté** les arêtes sont à *sens unique*. On les représente donc avec une flèche sur les dessins. D'ailleurs, le terme employé n'est plus arête, mais **arc**. Cette distinction est importante, car nombre d'algorithmes ne fonctionnent tout simplement pas sur des graphes orientés. Notez bien que cela n'empêche en rien que deux nœuds puissent être reliés dans les deux sens : il suffit d'utiliser deux arcs, chacun dans un sens (cela ne rompt pas la condition de graphe simple).



FIGURE 1.3. – Graphe orienté

### 1.2.5. Graphes pondérés

Dans un **graphe pondéré** les arêtes (ou les arcs) sont, ben, **pondérés**, quoi. Autrement dit, on associe une valeur à chaque arête. Elles peuvent très bien être négatives.

Cela peut-être une distance : lorsque que l'on cherche un plus court chemin entre deux nœuds, il va de soit que la somme des pondérations des arêtes doit être aussi petit que possible.

Mais ça peut aussi être un réel dans  $[0; 1]$  pour désigner des probabilités dans les [chaînes de Markov](#) par exemple. Cela peut aussi être un score, un prix, etc. N'importe quelle quantité qui peut vous passer par la tête!



FIGURE 1.4. – 3 variantes du même graphe, avec une pondération différente

### 1.2.6. Graphes cycliques

Un graphe cyclique comporte au moins un **cycle**. Un cycle est un chemin qui permet de relier un nœud à lui même, sans jamais passer deux fois par la même arête. Un graphe comportant cette propriété peut-être orienté ou non orienté (indifféremment).

La détection de cycles est d'ailleurs un problème récurrent en informatique, notamment lorsqu'on s'intéresse aux dépendances d'un fichier ou d'un programme : **A** requiert **B**, **B** requiert **C**, et **C** requiert **A** est un cas de dépendances cyclique habituel.



FIGURE 1.5. – Graphe orienté comportant un unique cycle

Les graphes ne possédant pas de cycles sont dit *acycliques*. Il existe un sous-ensemble remarquable de graphes acycliques : les **DAG** (pour *Directed Acyclic Graph*) ; ce sont des graphes acycliques orientés. Les algorithmes dynamiques (nous verrons ça plus tard) ne peuvent travailler que sur des DAG.

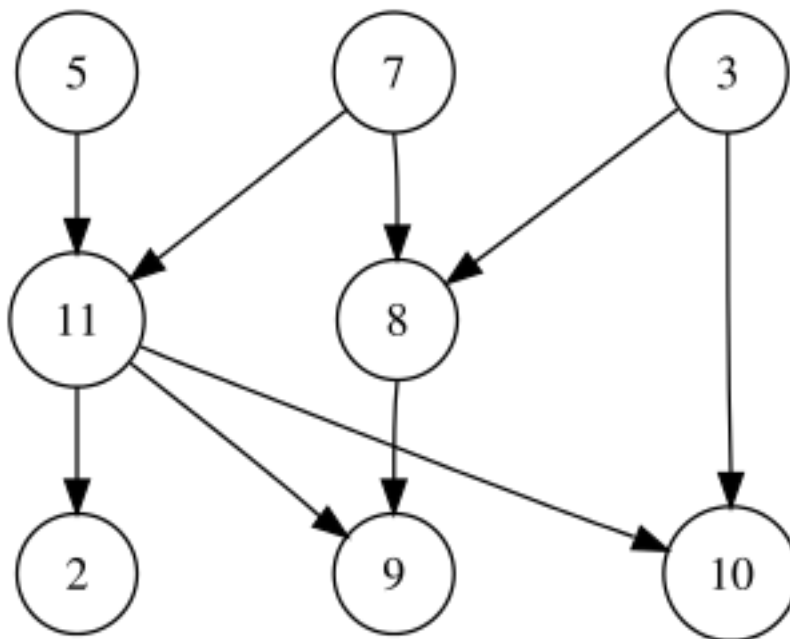


FIGURE 1.6. – DAG

### 1.2.7. Densité d'un graphe et degré d'un nœud

La **densité** d'un graphe correspond au **rapport du nombre d'arêtes sur le nombre total d'arêtes possibles**. C'est donc un réel compris entre 0 et 1. Cette caractéristique influe sur le choix de sa représentation.

Une densité de 0 correspond à un graphe sans arêtes où tout les sommets sont isolés.

Une densité de 1 correspond à un **graphe complet** : chaque nœud est relié à chaque autre nœud.



FIGURE 1.7. – Ci-dessus le graphe complet à 9 nœuds.

## I. Bases de la théorie des graphes

Dans un graphe simple orienté de  $N$  nœuds, chaque nœud ne peut être relié qu'à ses  $N - 1$  voisins au maximum, soit un total de  $N(N - 1) = N^2 - N$  arcs.

Et  $\frac{N^2 - N}{2}$  arêtes dans le cas d'un graphe non orienté.

Le **degré** d'un nœud correspond au nombre d'arêtes reliées à ce nœud. Dans le cas d'un graphe orienté, le **degré entrant** d'un nœud est le nombre d'arcs qui aboutissent à ce nœud, et le **degré sortant** le nombre d'arcs qui partent de ce nœud.

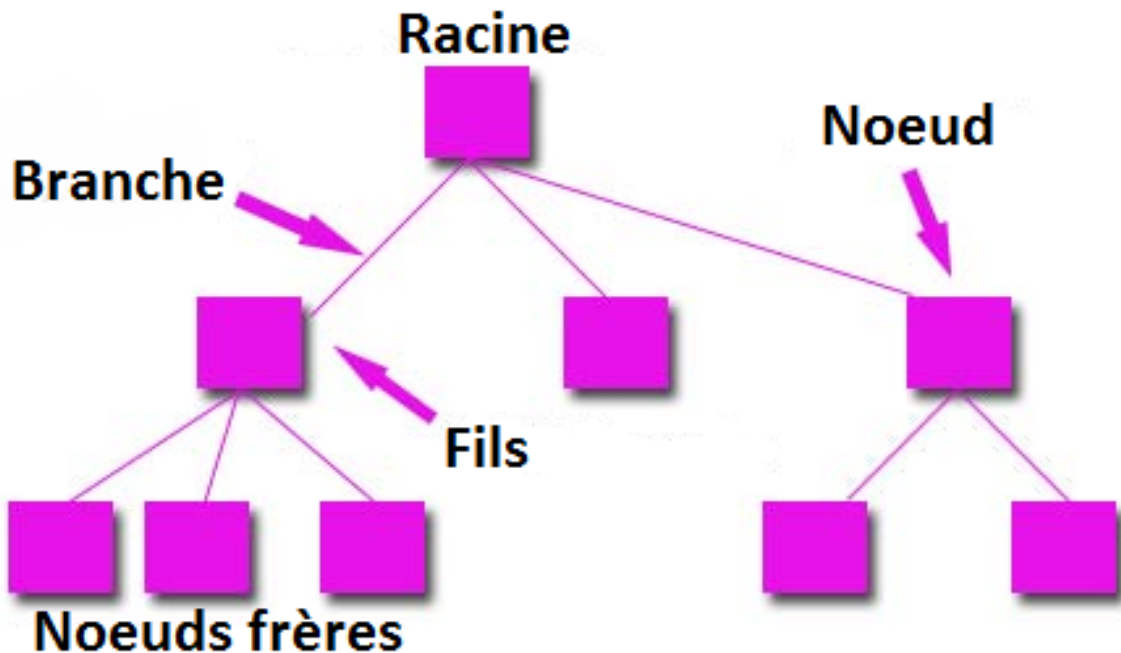
De façon générale, lorsque le nombre d'arêtes d'un graphe est de l'ordre de  $N$ , le graphe est dit **creux**. Le degré moyen de ses nœuds est une constante. A l'inverse, lorsque le nombre d'arêtes est de l'ordre de  $N^2$ , le graphe est dit **dense** : le degré moyen de ses nœuds est de l'ordre de  $N$ .

### 1.2.8. Arbre

Un graphe est un **arbre** s'il vérifie les propriétés suivantes : il est *acyclique*, *non orienté*, et *connexe*. De cela il découle qu'il n'existe qu'un et un seul chemin entre deux nœuds donnés (c'est vraiment très pratique, je vous l'assure!).

Attardons sur quelques évidences. Un arbre possède une seule et unique *racine*. Il est relié à d'autres nœuds - ses *filles* - par des *branches* (le terme arête demeure correct toutefois). Tous les nœuds peuvent posséder 0, 1 ou plusieurs fils. En revanche, ils possèdent tous un seul et unique *père*, à l'exception de la racine qui n'en a pas. Les *feuilles* sont caractérisées par l'absence de fils.

Un dessin pour mieux comprendre tout ce fouillis :



Comme vous pouvez le voir, les nœuds correspondent à des embranchements. Si on part d'un nœud et qu'on descend de fils en fils, on aboutira nécessairement à une feuille. Et si on remonte de père en père, on aboutit à la racine (et dire qu'il y a encore des gens qui doutent du créationnisme!).

## I. Bases de la théorie des graphes

Si le graphe n'est pas connexe mais qu'il possède les propriétés cités ci-dessus, alors chacun des sous-graphes connexes est un arbre, et l'ensemble forme une **forêt** (ben oui, un ensemble d'arbres c'est une forêt).



FIGURE 1.8. – Forêt d'arbres

### #### Graphe biparti

Un graphe est **biparti** s'il est possible de former deux partitions (comprendre, de le couper en deux morceaux distincts) de ses sommets de façon à ce que chaque arête passe d'une partition à l'autre, sans que jamais une arête ne relie deux nœuds dans la même partition.

Une fois encore, un joli dessin vaut tout les discours du monde.



FIGURE 1.9. – Graphe biparti

J'imagine que l'intérêt doit vous paraître limité pour le moment. Mais les graphes bipartis servent dans plein de situations, et à pleins de choses différentes, comme représenter une [relation binaire](#) [↗](#) par exemple.

Remarquez qu'un arbre est forcément un graphe biparti. En effet, les nœuds à profondeur paire sont "pris en sandwich" entre des nœuds à profondeur impaire (son père et ses fils), et vice-versa. Ainsi deux nœuds à une profondeur de même parité ne seront jamais reliés par une arête, on peut donc former une partition contenant les nœuds de même parité.

## 1.3. Représentations et stockage en mémoire

### 1.3.1. Généralités

Maintenant, cher lecteur, nous avons une idée bien plus précise de ce qu'est un graphe. Mais il n'est pas destiné à rester un outil purement théorique! Il faut pouvoir résoudre des problèmes avec, et donc implémenter des algorithmes qui travaillent dessus.

Nous devons donc trouver une structure de données adaptée pour le stocker, qui soit économe en mémoire, et qui permette aux algorithmes de l'exploiter rapidement.

## I. Bases de la théorie des graphes

Chaque nœud est stocké dans un tableau, une liste, ou n'importe quelle autre structure de données plus complexe (aucune contrainte particulière à ce propos, à vous de choisir la plus adaptée). Il est composé de **propriétés**, comme par exemple une **position**, une **lettre**, une **valeur** ou n'importe quelle entité plus complexe, qui dépend du problème (au cas par cas).

La difficulté consiste donc à lister intelligemment les arêtes entre les nœuds.

On résout ce problème avec une **liste d'adjacence** ou une **matrice d'adjacence**. La quasi totalité des implémentations d'un graphe sont des variantes de ces deux structures de données.

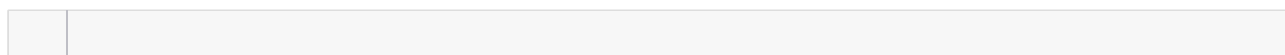
Attention ! La liste et la matrice ne sont pas seulement des structures de données, **elles sont aussi une représentation du graphe**.

La matrice et la liste contiennent suffisamment d'informations pour définir le graphe, au même titre que la représentation géométrique à base de sommets et d'arêtes.

Mais contrairement à cette dernière, qui est plus intuitive et plus adaptés aux êtres humains, la matrice et la liste peuvent être aisément implémentées sous forme de structure de données et utilisées par un programme.

### 1.3.2. La liste d'adjacence

La **liste d'adjacence** est le moyen le plus répandu pour stocker un graphe en mémoire : elle correspond à la représentation intuitive que l'on s'en fait. La liste d'adjacence d'un nœud est la liste de ses voisins (ou la liste des arêtes qui le relie à ses voisins).



Fin de la théorie.

Passons à la pratique : vous pouvez avoir, selon ce que vous propose votre langage, plusieurs moyens de stocker la liste des voisins.

Opérations	Liste d'adjacence	Matrice d'adjacence
Retirer une arête	$O(d)$ avec $d$ le degré du nœud	$O(1)$
Ajouter une arête	$O(1)$	$O(1)$
Itérer sur les voisins d'un nœud	$O(d)$ avec $d$ le degré du nœud	$O(N)$
Tester si deux nœuds sont voisins	$O(d)$ avec $d$ le degré du nœud	$O(1)$
Complexité mémoire	$O(N + A)$	$O(N^2)$



### 1.3.3. Autres représentations

Il existe bien d'autres représentations d'un graphe.

Citons particulièrement la **matrice d'incidence**.

C'est une matrice de dimensions  $N \times A$  qui indique quels liens arrivent sur quels sommets.

A l'intersection d'une ligne correspondant à nœud et d'une colonne correspondant à une arête on trouve un nombre.

Il vaut 0 si cette arête n'est pas reliée à ce nœud.

En revanche, si ce nœud est une extrémité de l'arête, ce coefficient diffère selon si le graphe considéré est orienté ou non.

Pour un graphe non orienté, cette valeur vaudra alors 1 dans le cas général, et 2 si cette arête réalise une boucle sur ce nœud (car ce nœud est deux fois l'extrémité de cette arête). Ainsi la somme des coefficients sur une colonne vaudra toujours 2.

Pour un graphe orienté ce coefficient vaut  $-1$  si l'arc sort du nœud considéré, et 1 lorsque l'arc entre dans le nœud.

La raison pour laquelle je ne vous l'ai pas proposée en structure de données est simple : ses performances sont en tout point inférieures à celles de la liste ou de la matrice d'adjacence, quelque soit l'opération considérée. La consommation en mémoire, du  $O(N \times A)$ , soit un  $O(N^3)$  en pire des cas, est aberrante si le graphe est dense.

Alors, pourquoi cette représentation a été inventé, me direz-vous ?

Eh bien il se trouve que les graphes sont des objets mathématiques comme les autres, et leurs différentes représentations ont nécessairement des liens entre elles. Des relations relient [la matrice d'incidence](#)  $\square$  avec [la matrice Laplacienne](#)  $\square$ , la matrice d'adjacence, [la matrice des degrés](#)  $\square$  ou encore [le line graph](#)  $\square$ .

Cela sert dans des calculs, et occasionnellement dans certains algorithmes.

Il y a cependant trop de choses à dire sur chacun de ces sujets pour que je m'y risque ici. Les curieux pourront approfondir leurs recherches.

## 2. Parcourir un graphe

C'est bon ? Vous avez un graphe joli tout plein ? Il attend sagement dans la RAM que vous vous occupiez de lui ?

Vous êtes ici pour pouvoir - enfin ! - programmer vos premiers algorithmes de graphe.

Vous allez apprendre à explorer la bête, avec le **DFS**, le **BFS** et l'exhaustif. Et avec la foudroyante multitude d'exemples que je vous donnerai, je vous promets que vous ne tarderez pas à découvrir le nombre incroyable de problèmes que ces simples algorithmes résolvent.

Préparez vous à plonger *profondément*<sup>1</sup> dans l'univers des graphes.

### 2.1. Le parcours en profondeur et le labyrinthe

#### 2.1.1. Les pyramides

Aaaah l'Égypte ! Le soleil, le sable, le Nil ! Le Sphinx et son nez ! Les pyramides, leurs labyrinthes, et leurs... euh... labyrinthes.

Fidèle à la tradition familiale vous êtes devenu un pilleur de tombeaux. Vous n'êtes pas sans ignorer que la chambre funéraire recèle maints trésors, que vous avez hâte de vous approprier afin d'ouvrir le magasin de guimauve dont vous rêvez depuis votre enfance.

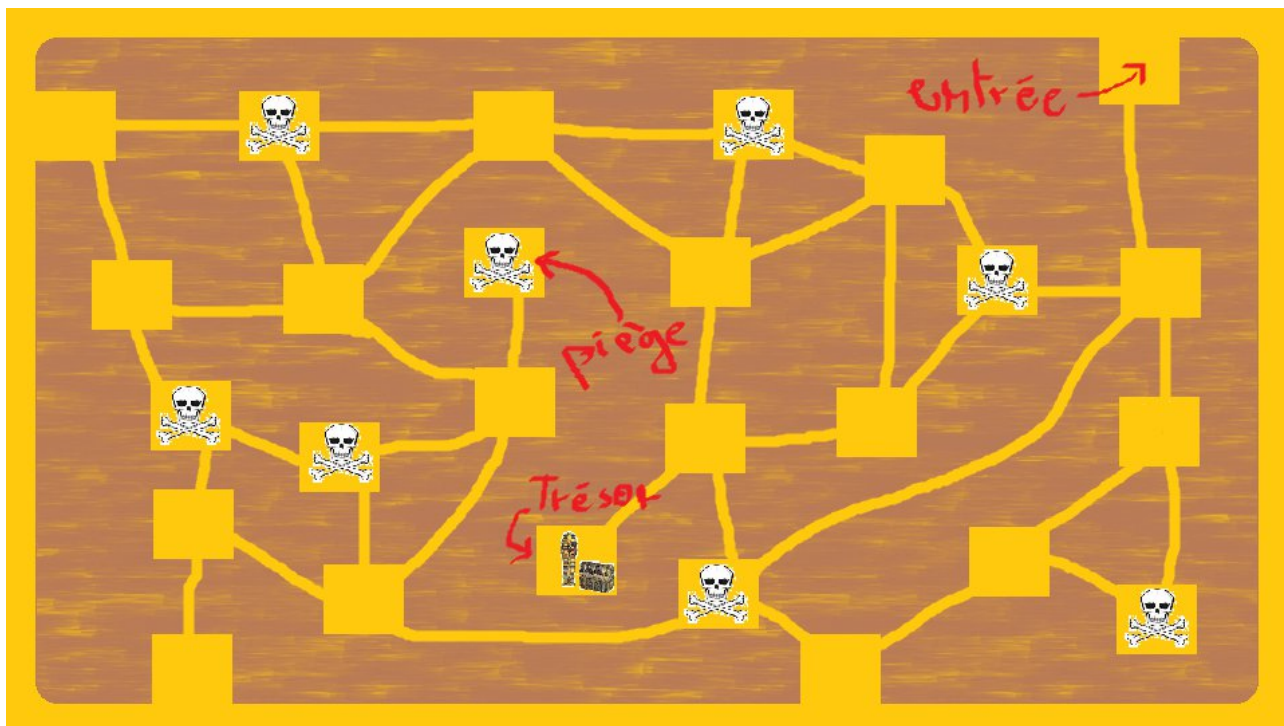
Sauf qu'aujourd'hui vous êtes tombé sur une pyramide très particulière, avec de nombreux carrefours... mais certains mènent à des pièges abominables, tellement abominables que je n'ose pas les décrire ici !

Vous disposez d'une carte du labyrinthe (remettant ainsi en cause l'utilité de ce dernier). Soucieux de ne pas prendre de risques superflus, vous confiez à votre ordinateur la dure tâche de trouver un itinéraire fiable.

Voici le plan :

---

1. Vous comprendrez plus tard.



**Exercice** : trouvez le graphe associé à ce problème. Trouvez ses caractéristiques principales. Trouvez quelle question on se pose sur ce graphe. Puis choisissez la structure de données qui vous semble la plus appropriée à la résolution du problème !

**Commençons par le graphe.**

Il apparaît clairement que chaque *couloir* est une arête.  
Par conséquent les nœuds seront des *intersections*.

Certains nœuds sont particuliers.

Les *entrées* du labyrinthe : c'est de l'une d'entre elles que commence le chemin.

La *chambre funéraire* : c'est l'aboutissement d'un chemin (s'il existe).

Les salles piégées posent problème : elles existent mais nous ne pouvons pas les traverser. Deux solutions.

- soit on ajoute ces nœuds particuliers au graphe, en précisant bien qu'ils sont piégés pour que l'algorithme les ignore (compliqué et désagréable à implémenter, plein de cas particuliers à gérer).
- soit on les ignore purement et simplement lors de la construction du graphe, comme s'ils n'en faisaient pas parti (plus simple, plus propre).

Nous choisirons donc la seconde solution.

Et retenez de cela qu'il ne faut pas s'encombrer de superflu (un sérieux coup de pied dans les fesses de la société de consommation, pas vrai ?), la simplicité, l'expressivité et la concision sont les mots d'ordre du développeur. Cela vous évitera de tristes après-midis de débogage.

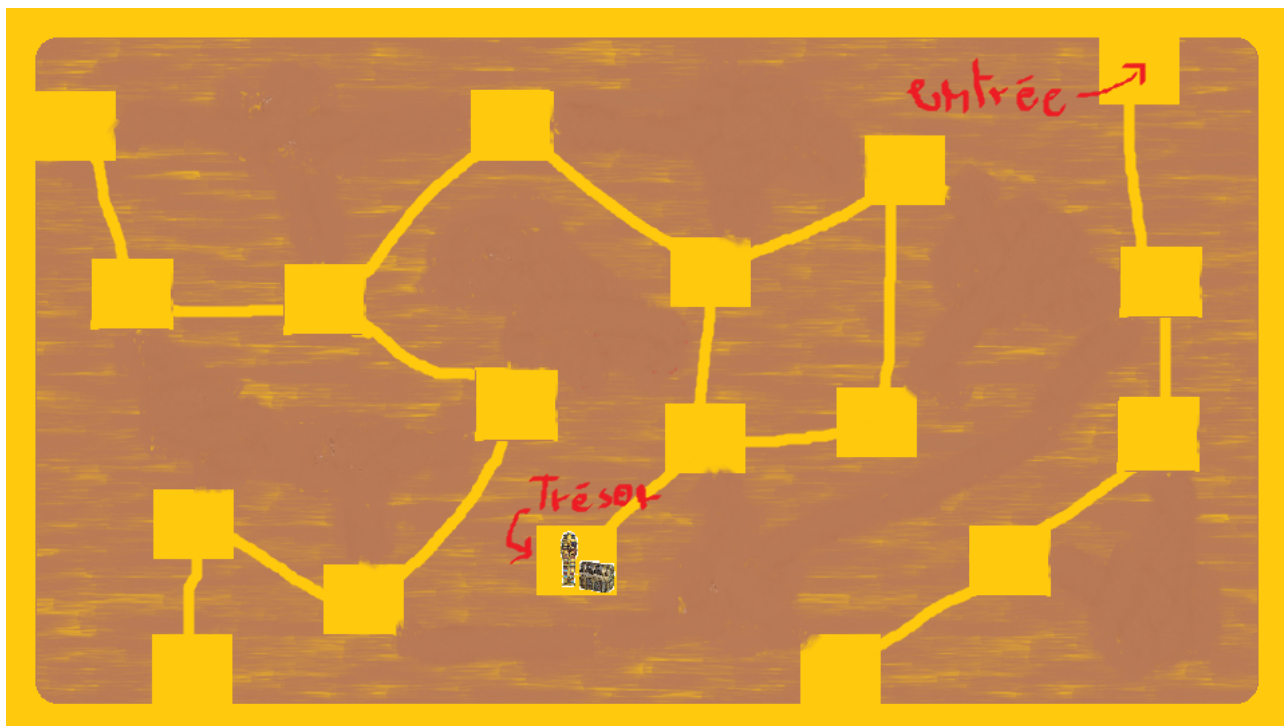


FIGURE 2.1. – Labyrinthe simplifié

Passons aux caractéristiques du graphe.

1. Ce graphe est **non orienté** : vous pouvez traverser un couloir dans les deux sens, et même sur les mains si ça vous plaît.
2. Ce graphe est **cyclique**, car certains itinéraires tournent en rond (*ah les fourbes !*).
3. Ce graphe n'est **pas pondéré**. Certains objecteront qu'on peut associer à chaque couloir sa longueur, le coefficient de dureté du sol, l'âge de sa construction ou que sais-je encore. C'est vrai. Sauf qu'on s'en fout. On est pas ici pour trouver le chemin le plus respectueux pour vos pieds, donc ne nous encombrons pas de ces broutilles. *Pas de superflu.*
4. Ce graphe n'est **pas connexe** : vous ne pouvez pas accéder à n'importe quel endroit depuis n'importe quel autre endroit. On dénombre 2 composantes connexes d'ailleurs.
5. Ce graphe est **creux** : on a seulement 17 arêtes pour 18 nœuds (après suppression des nœuds piégés et des couloirs qui leur sont associés).

On choisira donc la *liste d'adjacence* pour stocker ce graphe !

**Pour finir voici la question que l'on se pose.**

Existe-t-il un chemin entre une entrée et la chambre funéraire ?

Autrement dit, la chambre funéraire est-elle connexe à au moins une entrée ?

Quel est ce chemin (ou l'un de ces chemins) ?

### 2.1.2. Le parcours en profondeur

Le **DFS** est la méthode la plus simple pour parcourir un graphe. Elle fonctionne sur tout type de graphe, cyclique ou non, orienté ou non, etc.


## I. Bases de la théorie des graphes

En langage naturel ça donne : *je pars d'un endroit que je ne connais pas, je me dirige vers d'autres endroits que je ne connais pas, et quand je suis bloqué je fais demi-tour jusqu'à retrouver un chemin que je n'ai pas encore parcouru. Assez instinctif pas vrai ?*

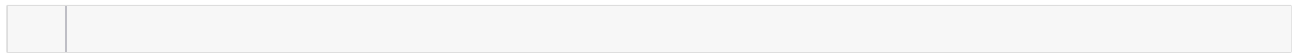
C'est la méthode que l'on utilise lorsqu'on est perdu, ou lorsqu'on visite un bâtiment : on essaie plusieurs chemins, et lorsqu'on est coincés on fait demi-tour jusqu'à la précédente intersection.

Voici sa description sous forme d'algorithme :

- 1) Je cherche un nœud non visité.
- 2) Pour visiter ce nœud, je marque le nœud comme visité.
- 3) Je prends l'un de ses voisins :
  - si le voisin a déjà été visité je l'ignore et je cherche un autre voisin
  - si le voisin n'a pas encore été visité, je le visite
  - si tout les voisins ont été visité, je reviens au nœud précédent, et je ré-applique le **3)**
- 4) Je reprend le **1)** tant qu'il reste des nœuds non visités

Vous avez remarqué ? Une boucle (de **4** à **1**) englobe la totalité de l'algorithme. Cette condition d'arrêt (*tant qu'il reste des nœuds non visités*) nous assure que l'entièreté du graphe sera exploré. Il n'y a pas que ça : pour visiter un nœud, il faut aussi visiter l'un de ses voisins non visité. Un concept qui se renvoie à lui même pour se définir, ça ne vous fait penser à rien ? Mais si, [la récursivité](#)  bien sûr !

Passons au pseudo-code.



Remarquez une chose : la fonction `explorer` fera autant d'appels à **DFS** qu'il existe de composantes connexes distinctes, dans un graphe non orienté. Le **DFS** est donc un bon moyen de retrouver les composantes connexes d'un graphe non orienté.

Si nous ne marquons pas les nœuds comme étant visités, nous nous mettrions à tourner en rond, dans le cas d'un graphe cyclique. Cela entraînerait donc des appels récursifs infinis, et notre programme ne se terminerait jamais ! Pour cette raison, le **DFS** est l'algorithme le plus adapté à la détection de cycles.

Vous comprenez maintenant l'origine du nom **DFS** : l'algorithme descend en profondeur dans le graphe, avant de faire demi-tour.

Sur le graphe ci-dessous l'ordre d'exploration pourra donc être : **A - B - D - F - E - C - G**



Voyons le problème sous un autre angle, et dessinons les arêtes empruntées par le **DFS** :

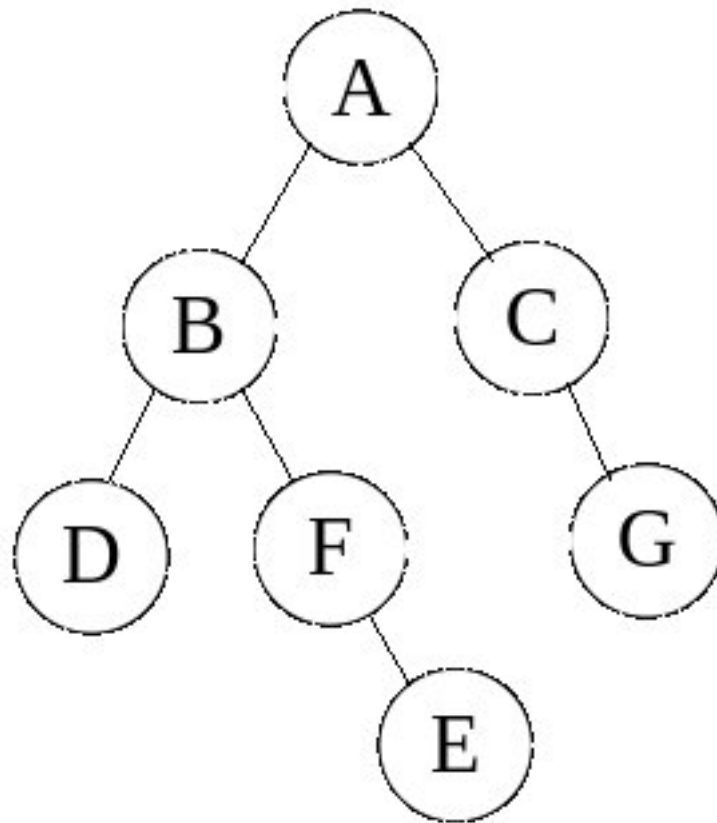


FIGURE 2.2. – Arbre

Cela ne vous fait penser à rien ? C'est un arbre !

En effet, ce graphe ne comporte pas de cycles (puis que le **DFS** ne doit pas boucler).

Et on s'aperçoit que le **DFS**, de nature récursive, se prête bien au parcours de cette structure de données récursive : pour explorer un arbre, on explore sa racine puis les sous-arbres qui le composent.

Jetons un petit coup d'œil à la complexité en temps de cet algorithme.

On remarque que chaque nœud ne sera traité qu'une seule fois, et on a tôt fait de conclure (à tort) qu'il est en  $O(N)$ .

Sauf que... pour chaque nœud, on itère sur tout ses voisins (qu'un appel récursif soit fait dessus ou non).

D'où une complexité en temps de  $O(N + A)$ . On voit donc ici que le nombre de nœuds est fonction - de façon assez évidente - de la vitesse d'exécution, mais que la densité du graphe a elle aussi un rôle très important.

La complexité en mémoire dépend du nombre d'appels récursifs (qui font grossir la pile), et il peut y en avoir autant que de nœuds dans le graphe. La complexité en mémoire est donc  $O(N)$ . Le pire des cas correspond à un graphe sous la forme d'une liste de nœuds chaînés les uns à la suite des autres, là où la profondeur d'appel est maximale.

Le **DFS** tel que je vous l'ai présenté ici est assez "nu" mais il va sans dire qu'avec quelques modifications il est en mesure de résoudre un grand nombre de problèmes (ici déterminer si deux nœuds sont sur la même composante connexe). Nous verrons cela dans les chapitres suivants.

Que le **DFS** soit de nature récursive est autant un avantage qu'un inconvénient. Un avantage car beaucoup plus simple à lire, à écrire et à déboguer (lorsqu'on est à l'aise avec la récursivité). Mais un inconvénient car chaque appel récursif fait grossir la pile d'appel, qui possède une taille limitée dépendant de votre langage, de votre OS et de votre compilateur/interpréteur (souvent 1000 mais n'en faites pas une généralité). Si votre graphe est trop gros vous pourriez rencontrer d'importants problèmes de mémoire. La solution consiste à ré-implémenter le **DFS** en itératif en simulant la pile d'appel grâce à une pile **LIFO**.

### 2.1.3. Luke, je suis ton père

Normalement, vous disposez d'assez d'informations pour résoudre le problème à présent. Il suffit de modifier un peu la fonction **DFS**.

Ne regardez pas la solution avant d'avoir bien cherché !

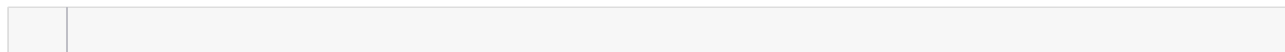
Bon, récapitulons : si ce chemin existe, après l'avoir parcouru, il faut savoir le retrouver.

Remarquons deux choses : ce chemin, quel qu'il soit, reste le même quelque soit le sens dans le quel on l'emprunte ; en outre, chaque nœud n'est visité qu'une et une seule fois, donc il n'existe qu'un seul moyen de parvenir à lui dans le **DFS**.

C'est directement en lien avec l'observation de tout à l'heure : le **DFS** explore un arbre, donc chaque nœud n'a qu'un seul père.

Il suffit donc de remonter dans l'arbre, depuis la chambre funéraire jusqu'à la racine (c'est à dire le premier appel à la fonction **DFS** sur cette composante connexe).

Et pour cela rien de plus simple : il faut que chaque nœud retienne qui est son père.



Si ce chemin existe il sera trouvé. Sinon, la fonction `trouverChemin` renverra la liste singleton `[NUL]` pour signifier l'absence de chemin.

Ci-dessous, un exemple de solution (pas forcément la plus courte).



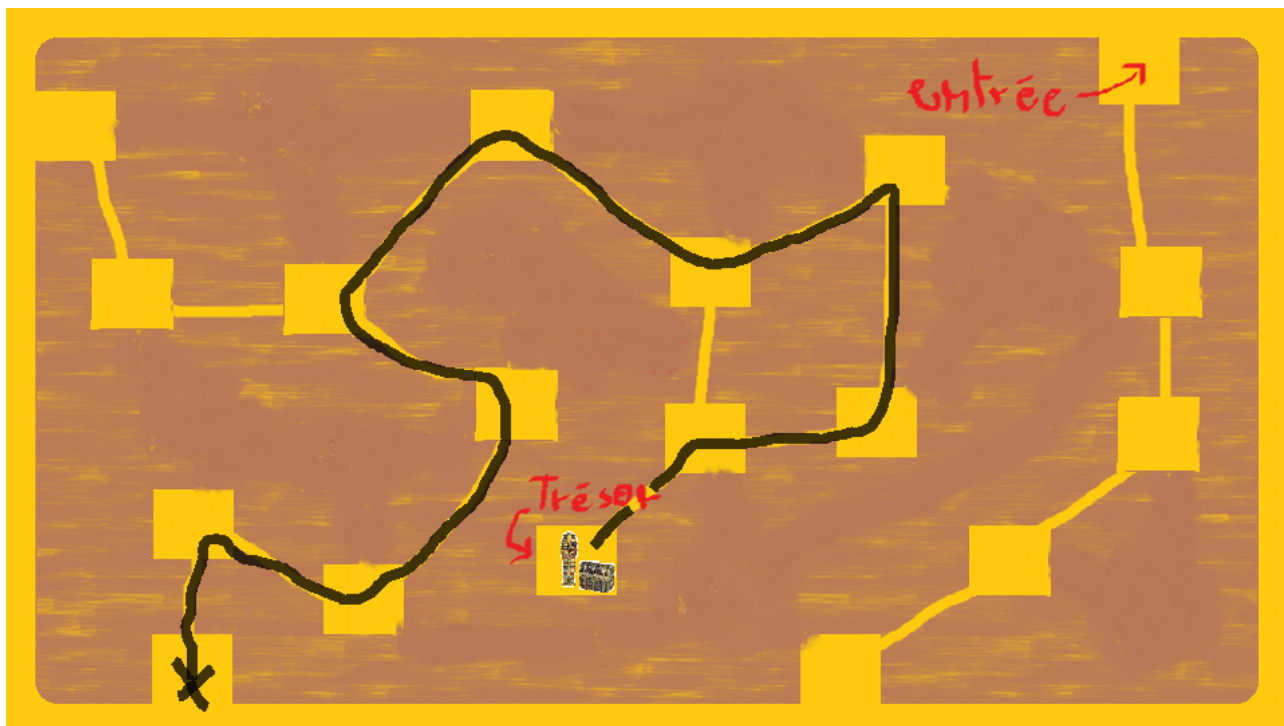


FIGURE 2.3. – Labyrinthe résolu

Et voilà ! Vous avez trouvé votre chemin. Mais n'oubliez pas : *il y a une différence entre connaître le chemin et arpenter le chemin.*

## 2.2. Le parcours en largeur et le buzz

\*[BFS] : Algorithme de parcours en largeur

### 2.2.1. Big Buzz

Vous venez d'achever l'œuvre de votre vie : une vidéo si stupide qu'elle va rencontrer un succès incroyable.

Elle va faire le *buzz*, vous le savez : quiconque la voit ne pourra pas s'empêcher de la partager juste après. Personne n'y échappera, subjugué par tant de bêtise humaine.

Vous voulez suivre la progression de votre création sur le net, et en particulier vous souhaitez savoir combien de personnes en tout ont visionné votre vidéo au bout d'un certain nombre d'heures.

Vous êtes ami avec un employé de la NSA, ce qui vous permet d'obtenir une carte très détaillée de votre réseau social préféré, où vous voyez les liens entre chaque individu. Sitôt qu'un individu voit la vidéo, il la partage. Tous ses contacts la voient très exactement une heure après, puis la partagent immédiatement à leur tour, etc.

Vous êtes le point d'émission de la vidéo à l'heure 0, qui ira vers vos amis, puis les amis de vos amis... A partir du réseau social, vous souhaitez obtenir la liste triée des gens l'ayant vue, en fonction de l'heure.



## Quel est le graphe ?

Bon là c'est assez explicite, un nœud pour chaque *individu*, et une arête pour chaque *relation entre deux individus*.

Ce graphe est :

1. **cyclique**
2. **non orienté** (l'amitié marche dans les deux sens normalement, mais si vous préférez un système avec des *followers* ce sera un graphe orienté)
3. **non pondéré**
4. **pas forcément connexe** car il est possible de trouver des communautés ou individus parfaitement isolés
5. **creux** (sauf quand tout le monde connaît tout le monde, mais c'est rare).

On va donc, une fois de plus, utiliser une liste d'adjacence.

On veut obtenir la liste des nœuds en fonction de leur distance à un nœud particulier, *le vôtre*.

### 2.2.2. Le parcours en largeur

Le **BFS** est l'algorithme qui permet de parcourir tout les nœuds en fonction de leur distance à l'origine. Il explore les cartes par cercles concentriques de plus en plus grands. Il fonctionne sur tout type de graphe, cyclique ou non, orienté ou non, etc.

Le **BFS** procède de la façon suivante : il prend le premier nœud (distance 0), puis traite tout les nœuds qui sont à une distance de 1, puis tout les nœuds à une distance de 2, puis tout les nœuds à une distance de 3... et ainsi de suite.

On remarque très vite que les nœuds à une distance 1 de l'origine sont ses voisins, à une distance de 2 ce sont les voisins de ses voisins, à une distance 3 les voisins des voisins de ses voisins... de manière générale, un nœud à distance  $n$  est : soit relié à des nœuds de distance  $n - 1$ , soit des nœuds de distance  $n$ , soit des nœuds de distance  $n + 1$ . Certainement pas plus : sinon il existerait un moyen plus rapide de rejoindre le nœud suivant ; certainement pas moins, sinon le nœud courant aurait pu être atteint plus rapidement.

Si nous disposons de tout les nœuds à distance  $n$ , nous avons accès à tout les nœuds de distance  $n + 1$ .

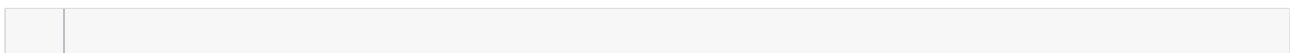
Dès le début on a accès au nœud de distance 0, donc en appliquant itérativement ce procédé on aura accès à tout les nœuds en fonction de leur distance à l'origine (s'ils sont connexes à elle).

Si vous ignorez ce qu'est une file FIFO, [il n'est pas trop tard](#) !

Voici l'algorithme :

- 1) Je prend le premier nœud de la file d'attente des nœuds à traiter.
- 2) Pour visiter ce nœud je le marque comme visité.
- 3) Je prend chacun de ses voisins non visités et je les ajoute à la fin de la file d'attente des nœuds à visiter
- 4) Je reprends le 1) tant qu'il reste des nœuds à traiter dans la file d'attente

Voici le pseudo code correspondant :



## I. Bases de la théorie des graphes

Cet algorithme fonctionne, je vous le promet !

Il commence par enfiler l'origine à distance 0, puis tout les nœuds 1, puis pour chaque nœud à distance 1 il va enfiler d'autres nœuds à la distance 2 (à la suite des nœuds à distance 1 donc), puis il défilera les nœuds à distance 2 pour enfiler des nœuds à distance 3 à leur suite.

Ainsi les nœuds ne sont jamais mélangés : ils sont présents dans la file par "paquets" consécutifs qui correspondent à leur distance à l'origine.

Sur un arbre, il explore les nœuds en fonction de leur hauteur (distance à la racine) contrairement au **DFS** qui va effectuer toutes les descentes possibles de la racine à une feuille. Ainsi, l'ordre d'exploration des nœuds du graphe ci-dessous pourra être : **A - B - C - E - D - F - G**. Comme vous pouvez le constater, tout les nœuds frères sont parcourus les uns à la suite des autres, la racine étant l'origine du **BFS**.



Et que se passera-t-il si d'aventure nous remplacions la file par une pile? On retrouve la version itérative du **DFS** présentée à la section précédente!

Il apparaît donc qu'il y a un lien important entre algorithme et structure de données. L'écriture sous forme itérative ou récursive dépend essentiellement de facteurs comme la lisibilité, la manière de laquelle l'algorithme s'explique le mieux, et enfin les performances.

Tout comme pour le **DFS**, la complexité en temps est en  $O(N + A)$ . Une fois encore la densité du graphe influe beaucoup sur les performances de l'algorithme.

La file peut contenir autant de nœuds que le graphe en possède, d'où une complexité en mémoire de  $O(N)$ . Le pire des cas concerne un graphe de diamètre très faible : la file est encombrée par la présence des nombreux nœuds à distance égale de l'origine.

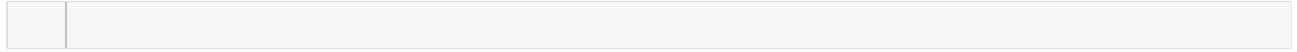
Le **BFS** peut servir à détecter les composantes connexes d'un graphe (tout comme le **DFS**), si plusieurs appels à la fonction **BFS()** sont réalisés avec des origines différentes. Mais le **BFS** a un avantage sur le **DFS** lors de la recherche de composantes connexes : il trouve le plus court chemin entre ces deux nœuds (s'il existe) dans le cas d'un graphe non pondéré.

Dans le cas d'un graphe implicite de taille infini (ou de très très grande taille) le **DFS** risque de s'engager dans une mauvaise voie dans le début et de s'y enfoncer trop profondément, voir à l'infini! Impossible de déterminer en un temps raisonnable (ou un temps fini) un chemin entre deux nœuds connexes, alors que le **BFS** va explorer plusieurs chemins possibles à la fois, ce qui lui donne l'assurance de trouver un jour ce chemin. Pour cette raison les graphes implicites de grande taille sont souvent parcouru par un **BFS** plutôt qu'un **DFS**.

Mais le **BFS** peut aussi servir à marquer les nœuds en fonction de leur distance à l'origine. C'est ce que nous allons voir tout de suite!

### 2.2.3. Le prix du succès

Une petite modification suffit à la résolution du problème.




Ci-dessous, vous pouvez voir l'effet de cet algorithme sur un graphe simple. Les nœuds sont marqués en fonction de leur ordre de visite par le **BFS** : le nœud n°1 correspond donc à l'origine. Les nœuds en cours de visite (c'est à dire présents dans la file) sont coloriés en bleu. Les nœuds déjà visités et extraits de la file sont coloriés en vert. Ici, tous les nœuds à distance 2 ou moins ont été explorés. Les nœuds à distance 3 sont en cours d'exploration. Le seul nœud à distance 4 n'a pas encore été traité.



Et voilà. Vous pouvez également indiquer le nombre total de gens ayant vu la vidéo à chaque heure, je vous laisse le coder vous même, ce n'est pas très difficile.

## 2.3. L'exhaustif et Uno

Connaissez-vous *Uno*<sup>TM</sup> ?

Comme nous l'explique si bien [Wikipédia](#) , c'est un jeu de carte américain créé en 1971 par Merle Robbins. Il est pourvu de règles subtiles et de cartes agressives pour faire rager les petits comme les grands.

Aujourd'hui, je ne vous propose pas de programmer ce célèbre jeu. A la place je vous propose plutôt de résoudre ce petit problème.

Dans Uno, chaque joueur possède un certain nombre de cartes en main, qui peuvent être de 4 couleurs : *bleues*, *vertes*, *rouges* ou *jaunes*.

Elles sont numérotées de 0 à 9.

Il existe également certaines cartes spéciales ("Joker", "Inversion", "Super Joker"...) mais nous ne y intéresserons pas ici, par soucis de simplicité.

Le jeu comporte un talon. On ne peut poser une carte sur le sommet de ce talon que si la carte au sommet du talon est :

- De même couleur que la carte qu'on joue
- De même valeur faciale que la carte qu'on joue

Un petit exemple :

- poser un 7 rouge sur un 9 rouge est autorisé
- poser un 4 vert sur un 4 bleu est autorisé
- poser un 2 jaune sur un 2 jaune est autorisé
- poser un 6 bleu sur un 3 vert est interdit

## I. Bases de la théorie des graphes

Il est ainsi possible d'empiler ces cartes de diverses façons en suivant ces règles, pour obtenir un talon plus ou moins haut.

Comme vous n'avez rien de mieux à faire, vous voulez savoir, à partir d'un ensemble de cartes donné, quels sont tout les talons qu'il est possible de réaliser avec. Les cartes qui ne pourront pas être ajoutées au talon seront éventuellement laissées sur le côté.

Vous pouvez empiler les cartes de votre choix dans n'importe quel ordre, tant que vous respectez les règles.



**Exercice :** vous commencez à le connaître par cœur normalement.

Quel est le graphe ? Quels sont ses caractéristiques ? Quelle question se pose-t-on sur lui ?

**Correction :**

Rappelez-vous : un graphe représente des objets et des relations entre ces objets.

Ici chaque objet, chaque nœud, est donc une *carte*. Les arêtes sont définies par *une valeur faciale ou une couleur commune*. C'est donc un bon exemple de graphe dans lequel les arêtes peuvent être déduites du nœud à partir de certaines règles de construction simples.

1. Ce graphe est **cyclique**. Certes, on ne peut pas utiliser une même carte plus d'une fois, mais il y a plus d'une manière de poser une carte, qui aboutira à d'autres situations où il serait ensuite théoriquement valide de poser cette carte si elle n'avait pas déjà été jouée.
2. Ce graphe est **non orienté**. L'ordre dans lequel sont empilés les cartes n'a pas d'importance, puisque les règles ne s'intéressent qu'à la relation d'adjacence de deux cartes, indépendamment de leur ordre.
3. Ce graphe est **non pondéré** (saprستي, un de plus!).
4. Une fois encore, ce graphe n'est pas nécessairement connexe. Il est parfois possible de former deux groupes de cartes n'ayant aucune couleur ou aucun numéro en commun.

Ce graphe est **dense**, selon moi. En effet il n'y a pas de caractérisation formelle entre un graphe dense et un graphe qui ne l'est pas, c'est laissé à l'appréciation de chacun. Laissez moi vous expliquer ma démarche.

En supposant que l'on ait beaucoup de cartes, disons  $N$ , et qu'elles soient toutes tirées au hasard : en moyenne nous aurons  $\frac{N}{4}$  cartes de chaque couleur (car il y a 4 couleurs).

Toutes les cartes de même couleur seront au moins reliées entre elles. Donc, en moyenne, chaque carte sera reliées à au moins  $\frac{N}{4}$  autres cartes.

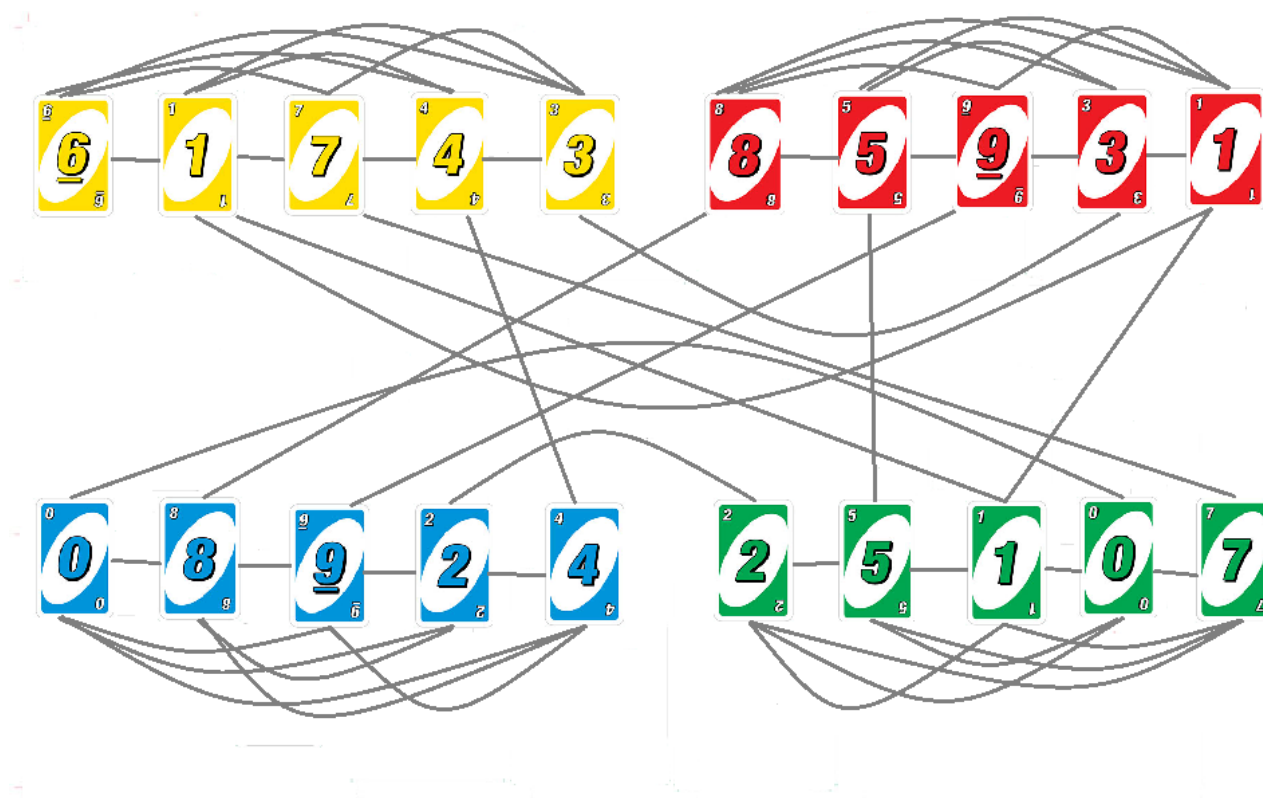
Chaque carte est aussi reliée à un  $\frac{1}{10}$  (car 10 chiffres) des cartes des 3 autres couleurs, soit  $\frac{3 \times N}{40}$  autres cartes en moyenne.

Cela fait une moyenne de  $0,325 \times N$  arêtes par nœud. Le degré de chaque nœud n'est pas constant : il dépend de  $N$ .

Ce qui porte le total à environ  $0,162 \times N^2$  arêtes, car c'est un graphe non orienté (il ne faut pas compter deux fois chaque arête).

Le nombre d'arêtes du graphe est fonction de  $N^2$ . On peut donc qualifier ce graphe de dense. Et il est suffisamment dense pour rendre la matrice d'adjacence plus intéressante que la liste.

Voici le graphe associé à un ensemble de 20 cartes (5 de chaque couleur).



En théorie chaque carte devrait être reliée à elle même. Mais comme on ne peut pas utiliser deux fois la même carte (ce qui n'empêche pas celle-ci d'être présente en plusieurs exemplaires dans le jeu de départ), on sait d'avance que ces arêtes (qu'on nomme *boucle*) ne nous serviront pas. Autant les supprimer maintenant.

On souhaite connaître quels sont tous les talons possibles. Un talon est caractérisé par un ensemble de cartes dans un certain ordre. Et les contraintes d'adjacence de deux cartes sont celles des arêtes du graphe!

Par conséquent, chaque talon peut être représenté par un chemin du graphe : l'ordre de visite des nœuds et la taille du chemin suffit à définir le talon.

Pour générer tous les talons possibles, il faut donc générer tous les chemins du graphe!

### 2.3.1. L'exhaustif

**Exhaustif** : qui inclut tous les éléments possibles d'une liste, qui traite totalement un sujet.

*Wiktionnaire*

Nous cherchons une liste de *tout* les talons possibles, sans en oublier aucun. Il nous faut donc une liste *exhaustive* des talons possibles.

Cela nécessite un algorithme capable d'effectuer une **énumération exhaustive** de tous les chemins du graphe. D'où le nom "exhaustif".

Il faut trouver une méthode systématique pour dénombrer les chemins du graphe.

Comment procéderions-nous "à la main" ?

## I. Bases de la théorie des graphes

Par exemple, nous pourrions prendre une carte et l'ajouter au talon. Puis prendre une autre carte et l'ajouter au talon (en respectant les règles). Puis continuer à ajouter des cartes qu'il est possible d'en ajouter, sans réutiliser deux fois la même (évidemment).

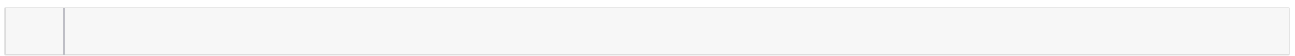
Et lorsqu'on est bloqué? Une retire une carte au sommet du talon et on met une autre à la place, et c'est reparti!

On continue d'ajouter des cartes jusqu'à ce qu'on soit forcé d'en retirer une au sommet pour la remplacer par une autre qui n'a pas déjà été posée à cette hauteur et avec ce talon spécifique en dessous.

Ainsi, à la fin, tout les talons auront été construits.

Vous avez remarqué? Un talon est une pile, donc nous aurons sûrement affaire à une pile dans cet algorithme!

En voici le pseudo-code :



Sans grande surprise, ici, la pile est la pile d'appel. Mais une fois encore, cet algorithme peut être passé sous forme itérative si le besoin s'en fait sentir.

Cette fonction ressemble beaucoup à celle du **DFS**, à deux exceptions près.

Premièrement, elle retourne quelque chose : la liste des chemins.

Secondement, et c'est là le point le plus important : l'état d'un nœud n'est pas définitivement fixé. A l'entrée de la fonction le nœud est marqué comme "utilisé"; en revanche, à la sortie de la fonction, il est de nouveau marqué comme "inutilisé".

Ainsi, la fonction **exhaustif** va pouvoir traiter plusieurs fois le même nœud, ce qui est somme-toute logique puisque qu'un même nœud peut faire parti de plusieurs chemins distincts. L'importance de marquer un nœud comme utilisé durant son utilisation (comme les toilettes publiques à verrou coloré) est la même que celle du **DFS** : empêcher les appels récursifs infinis en bouclant dans un cycle.

Sur le graphe ci-dessous, après un appel sur le nœud A, l'ordre d'exploration des nœuds pourrait être : **A - B - D - F - E - C - G - E - F - B - D**.



<http://upload.wikimedia.org/wikipedia/commons/thumb/6/61/Graph.traversal.ex>

Cela générerait les chemins suivants :

- A
- A - B
- A - B - D
- A - B - F
- A - B - F - E
- A - C
- A - C - G
- A - E
- A - E - F

## I. Bases de la théorie des graphes

- A - E - F - B
- A - E - F - B - D

Et cela juste avec le nœud A comme origine ! Je vous épargne les autres.

Comme vous pouvez le constater le nombre de chemins différents augmente très vite et atteint un nombre important, même sur un graphe de petite taille.

Ce qui nous amène tout de suite à une question épineuse : quelle est la complexité en temps et en mémoire de l'algorithme ?

Commençons par la complexité en temps.

Quel est le pire des cas ?

Le pire des cas serait un jeu de carte où toutes les cartes sont reliées entre elles (typiquement : toutes de même couleur), ce qui revient à travailler sur un **graphe complet**.

Ainsi, je dispose de  $N$  choix pour le premier élément du talon. Puis de  $N - 1$  pour le second. Puis de  $N - 2$  pour le troisième, et ainsi de suite. Le nombre de talons possible est donc égal à  $N \times (N - 1) \times (N - 2) \times \dots \times 2 \times 1 = N!$

En fait, toutes les permutations de cartes sont possibles (puisque toutes les cartes peuvent être adjacentes). Et le nombre de [permutations](#) de  $N$  éléments est bien égal à  $N!$ .

Cet algorithme est donc en  $O(N!)$  en temps.

La fonction factorielle croît asymptotiquement plus vite que la fonction exponentielle. Pour cette raison, même sur des instances de petite taille, certains problèmes sont insolubles en un temps raisonnable.

Pour vous faire une idée,  $20! = 2\,432\,902\,008\,176\,640\,000$ . Donc si le graphe ci-dessus était complet, il faudrait plusieurs dizaines de milliers d'années de calcul. Heureusement il ne l'est pas ! De mon côté j'ai trouvé 437 672 443 chemins différents (vous pouvez vérifier si l'envie vous en prend).

Quant à la complexité en mémoire, elle est nécessairement  $O(N!)$  elle aussi, puisqu'il s'agit là du nombre de solutions.

Cependant, si vous n'aviez qu'à dénombrer les solutions (sans les produire), vous auriez alors un algorithme en  $O(N)$ , car la profondeur de récursion peut atteindre  $N$  nœuds.

La recherche exhaustive, bien que donnant des résultats exacts de par son exhaustivité, ne peut pas être utilisée en pratique pour tous les problèmes, à cause de sa lenteur.

Pour cette raison, de nombreux problèmes sont résolus avec des algorithmes donnant une réponse approchée (ou exacte mais dont on ne peut pas prouver qu'elle est exacte) qui ont une durée d'exécution plus courte. Ces algorithmes sont des sujets de recherche encore actifs qui mériteraient un tutoriel entier à eux seuls, donc nous n'en parlerons pas plus ici.

## 3. Les problèmes usuels de graphes

Bon, maintenant que vous avez dans votre boîte à outil quelques algorithmes, il est temps d'éprouver leur efficacité et leur polyvalence.

Vous allez voir ici les problèmes de graphe les plus classiques : nous allons les résoudre de façon élégante en adaptant ce que nous avons vu précédemment.

Et puis, comme d'habitude, je ne saurais que trop vous conseiller de chercher la solution avant d'aller regarder la correction ! C'est ainsi que vous développerez une plus grande autonomie face à ce type de problèmes.

### 3.1. Le tri topologique et make

[make](#) est un utilitaire mis au point en 1977 par Stuart Feldman permettant d'automatiser la génération de fichiers à partir de règles simples (et éventuellement en faisant appel à d'autres programmes ou à la ligne de commande). Il est souvent utilisé pour fabriquer des fichiers exécutables ou des bibliothèques à partir des fichiers sources et d'un [compilateur](#) .

Les projets de grande envergure qui contiennent un nombre important de fichiers ont rendu indispensable l'utilisation de ce type d'outils.

Le (ou les) fichiers que l'on cherche à produire (appelés **fichiers cibles**) dépendent de la fabrication de plusieurs autres fichiers "intermédiaires", qui eux même peuvent dépendre de la création de plusieurs autres fichiers, et ainsi de suite jusqu'aux fichiers qui étaient déjà présents au début du processus (les **fichiers source**).

Les dépendances entre les différents fichiers, et la liste de commande permettant de créer chacun d'eux, sont spécifiées par le programmeur dans un fichier appelé **Makefile** que le programme make va lire.



Vous avez pour mission de programmer l'un des composants du logiciel présenté ci-dessus. Vous devez créer un algorithme qui prend en entrée une liste de dépendances entre différents fichiers, et retourne un ordre de construction valide des fichiers.

Commencez par formaliser les caractéristiques du graphe.





Ne lisez pas tout de suite la correction !

A présent, vous disposez normalement d'un bagage suffisant pour trouver l'algorithme approprié en vous appuyant sur ceux que nous avons vu précédemment.

Forcez-vous à chercher, c'est ainsi que vous progresserez.

### 3.1.1. Solution

Ici, comme souvent, formaliser ce qu'on fait naturellement à la main suffit à obtenir l'algorithme. Une méthode naturelle consiste à prendre un fichier qui ne dépend d'aucun d'autre, et l'ajouter à la liste des fichiers à traiter. Par la suite, les fichiers qui en dépendaient n'auront plus cette contrainte pour exister, puisque le fichier requis aura été considéré comme traité.

Puis prendre un nouveau fichier, qui ne dépend d'aucun autre, ou qui ne dépend que de celui que l'on vient de traiter, et l'ajouter (en respectant les ordres d'ajout successifs) à la liste des fichiers à traiter.

Et répéter ainsi l'opération qui consiste à ne traiter que les nœuds pour lesquels on a résolu toutes les dépendances. Cet algorithme est nommé **Tri Topologique**.

Si à la fin il reste encore des nœuds non traités mais qu'ils sont tous dépendants, alors cela signifie qu'il y a un cycle. Autrement dit, le problème n'est pas soluble.

Et comment déterminer efficacement si un nœud dépend d'un autre ?

Eh bien en regardant son **degré entrant** pardi !

On peut ainsi modifier dynamiquement le graphe de façon très simple : pour supprimer un nœud on supprime **toutes ses arêtes sortantes**, et par conséquent on réduit donc de 1 le degré entrant des nœuds de destination.

```
1 explorer(Graphe G, Noeud N)
2 {
3     Liste ordreTotal = {N}
4     Pour chaque voisin V de N
5         V.degreEntrant -= 1
6         Si V.degreEntrant == 0
7             noeudsTries = explorer(G, V)
8             ordreTotal.concatener(noeudsTries)
9     Renvoyer ordreTotal
10 }
11
12 TriTopologique(Graphe G)
13 {
14     Liste ordreTotal
15     Pour chaque noeud N dans G
16         Si N.degreEntrant == 0
17             noeudsTries = explorer(N)
18             ordreTotal.concatener(noeudsTries)
19
20     Si ordreTotal.taille < G.nbNoeuds
```

```
21         Afficher "Cycle detecte, resolution des dependances
           impossible"
22     Sinon
23         Afficher ordreTotal
24 }
```

Remarquez la fonction `explorer` : elle est construite sous la forme d'une fonction exploration récursive fortement semblable à celle du `DFS`.

Ainsi, comme je vous l'avais précédemment dit, les algorithmes classiques (moyennant quelques modifications mineures) permettent de résoudre un nombre très varié de problèmes. Gardez cela en tête!

---

Ce tutoriel n'est pas terminé!

D'autres chapitres viendront :

### III] Problèmes usuels de graphes

- Circuit Eulérien
- Nœuds essentiels
- Arêtes essentielles
- Algorithmes dynamiques pour les DAG (Directed Acyclic Graph)

### IV] Pathfinding

- `BFS` (on en reparle)
- Ford-Bellman
- Dijkstra
- Floyd-Warshall
- A\* et autres heuristiques du même type

Restez au courant si ça vous intéresse.

## Contenu masqué

### Contenu masqué n°1

1. Ce graphe *peut* être **cyclique**! Nous ne sommes pas à l'abri d'un programmeur négligeant qui aurait rendu certains fichiers mutuellement dépendants. Nous devons donc détecter ces cas particuliers et afficher une erreur le cas échéant.
2. Ce graphe est clairement orienté. Une dépendance est (normalement) à sens unique.
3. Ce graphe n'est **pas pondéré**.
4. Le graphe peut ne pas être connexe si le programmeur spécifie plusieurs cibles totalement indépendantes dans son makefile. Toutefois, on peut se ramener à un graphe connexe grâce à une petite astuce! On peut créer une cible "globale" abstraite (liée à aucun fichier physique) qui aurait pour dépendance toutes les autres cibles concrètes. C'est d'ailleurs ainsi que procède make.

[Retourner au texte.](#)

# Liste des abréviations

**BFS** Breath First Search. 16, 22–25, 32

**DFS** Depth First Search. 16, 18–21, 24, 28, 32

**IA** Intelligence Artificielle. 5

**LIFO** Last In First Out. 21