Introduction aux graphes

L2 informatique – C. Renaud

christophe.renaud@univ-littoral.fr

www-lisic-univ-littoral.fr/~renaud

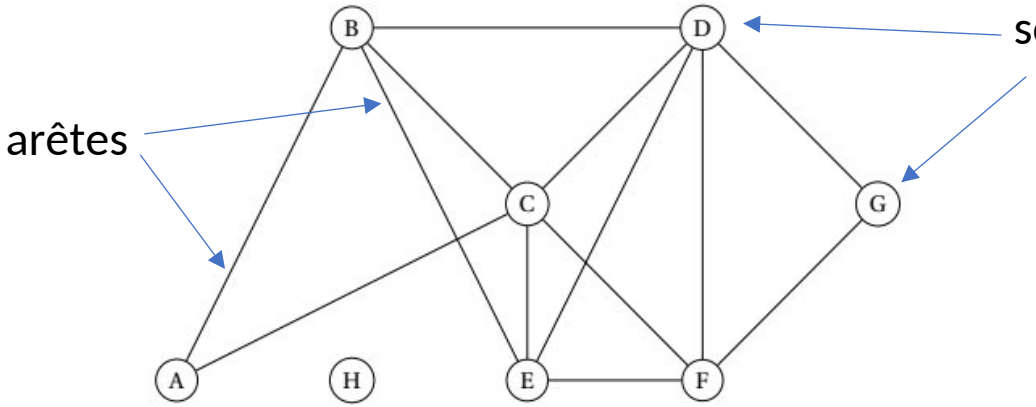
Version 1.6.1 du 02/02/2024

Introduction

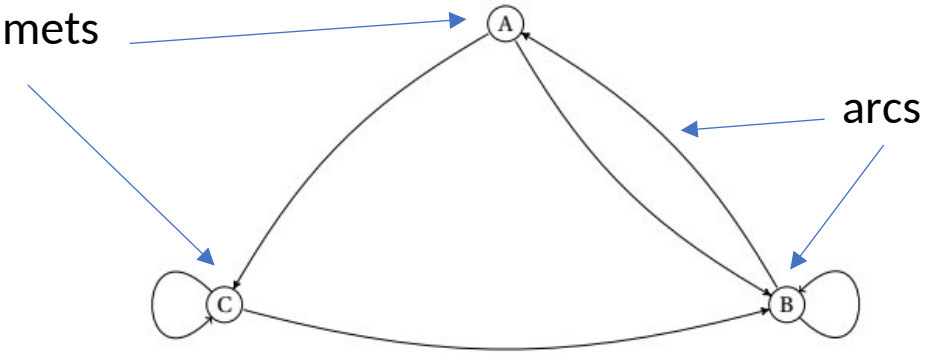
- Objectifs
  - Connaissances de base sur les graphes
  - Compréhension de l'algorithmique dédiée
  - Mise en pratique au travers du langage C
- Evaluation
  - Notes de TPs
  - Un projet à réaliser seul
  - Un examen sur machine
- Documentation
  - www-lisic.univ-littoral.fr/~renaud

Quelques définitions

- **Définition 1** : Un **graphe** est un ensemble de points, appelés **sommets**, pouvant être reliés entre eux par des **arêtes**. Il peut être :
  - **non orienté** : les arêtes ne possèdent pas de sens de parcours;
  - **orienté** : les arêtes possèdent un sens de parcours, représenté sur chacune des arêtes par une flèche. Les arêtes portent alors le nom d'**arcs**.



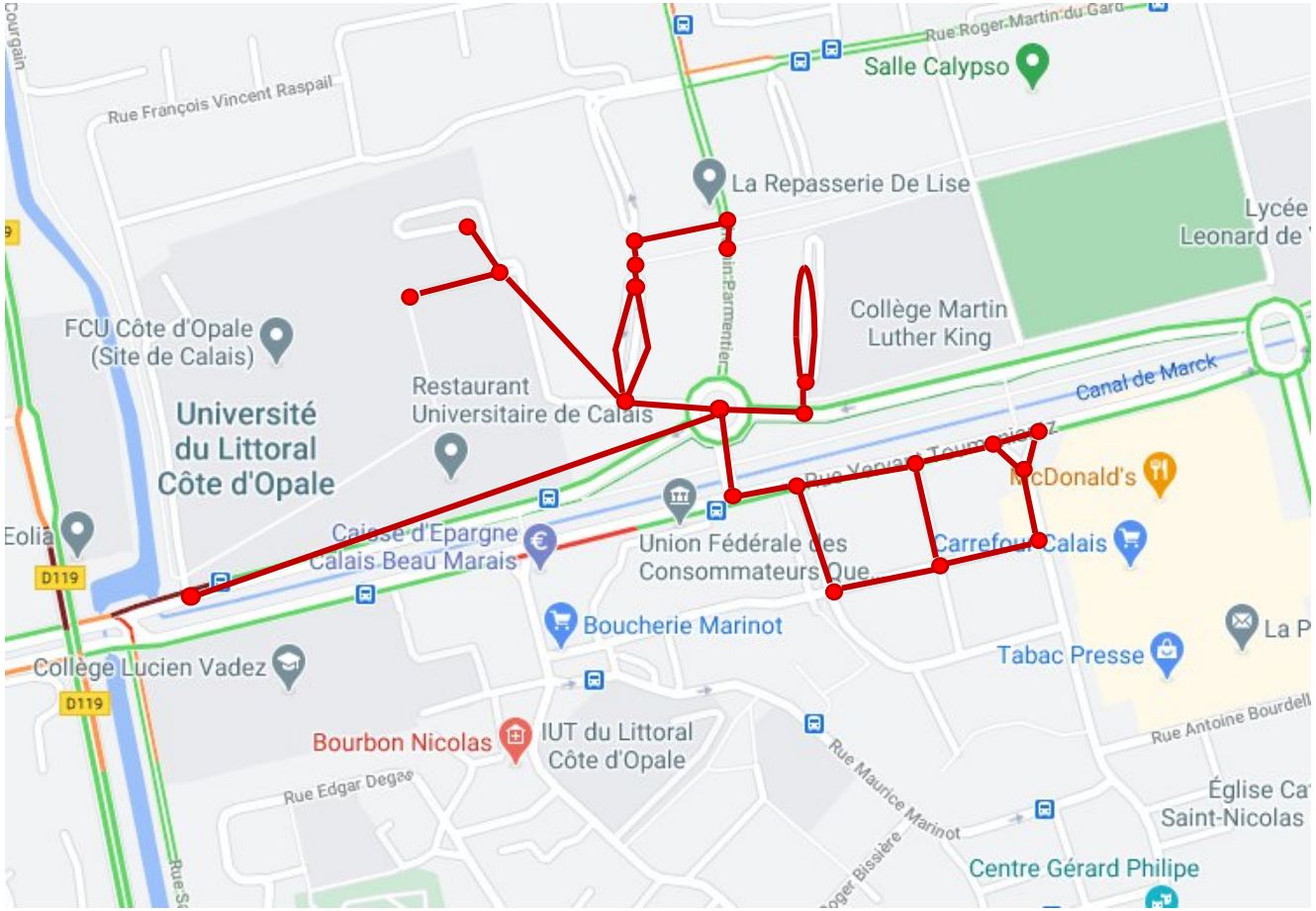
Un graphe non orienté
de sommets S={A,B,C,D,E,F,G,H}



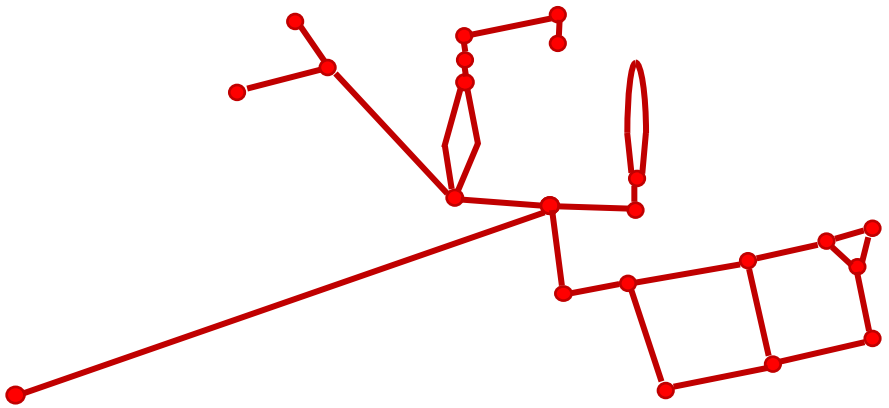
Un graphe orienté
de sommets S={A,B,C}

Quelques définitions

- Exemples : carte routière



Une arête = une voie de cheminement



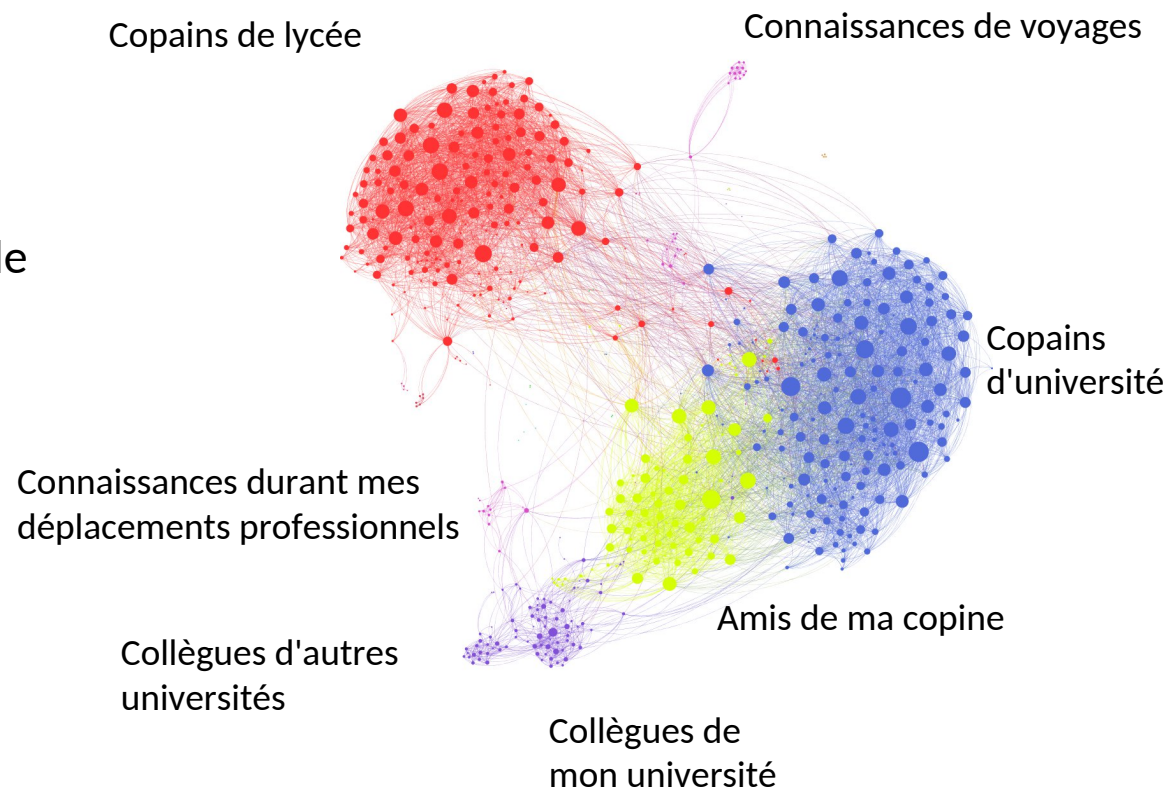
un sommet = une intersection

Quelques définitions

- Exemples : réseau social

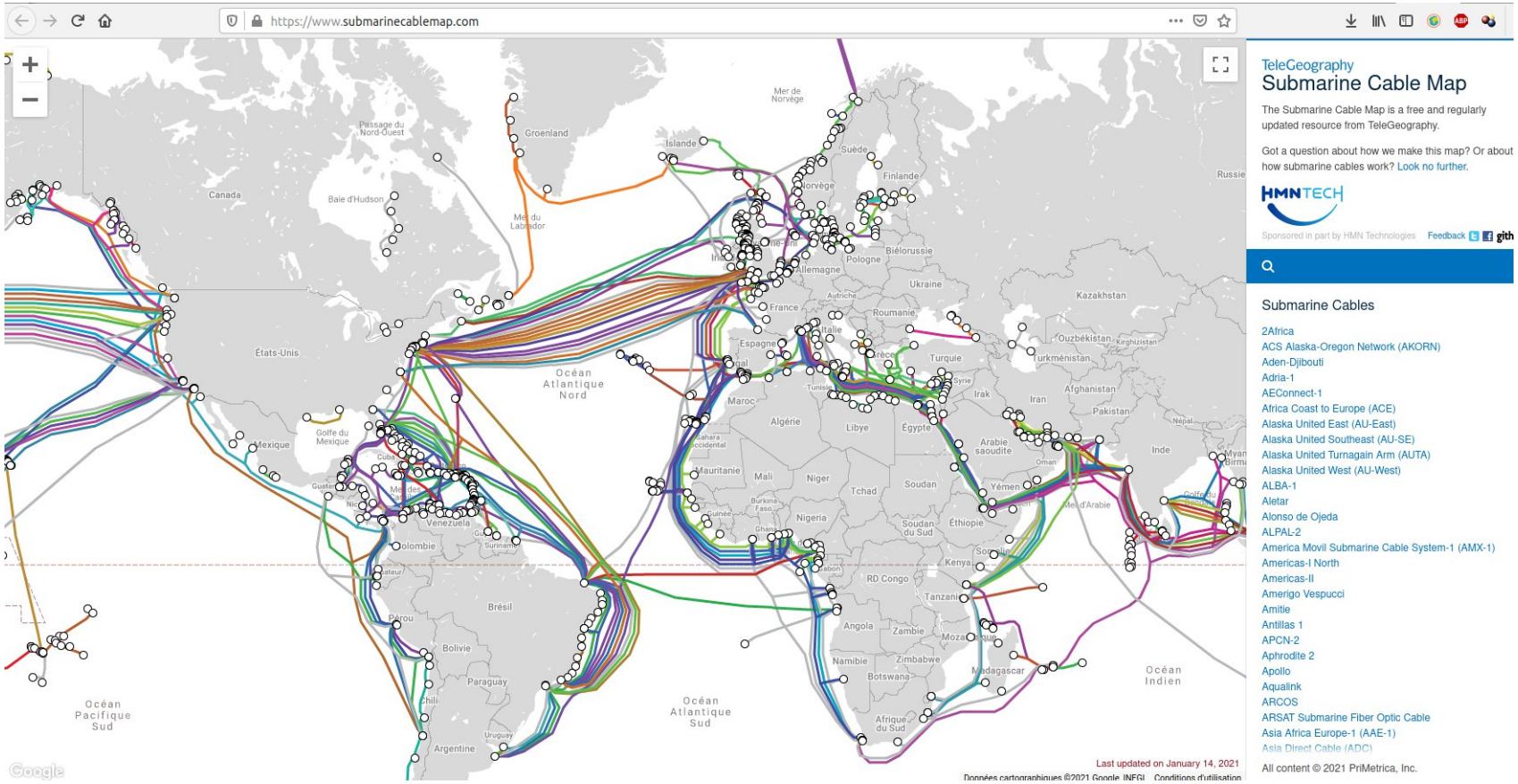
Source : https://griffsgraphs.wordpress.com/2012/07/02/a-facebook-network/

Un sommet = une personne
Une arête = une relation sociale
Une couleur = type de relation



Quelques définitions

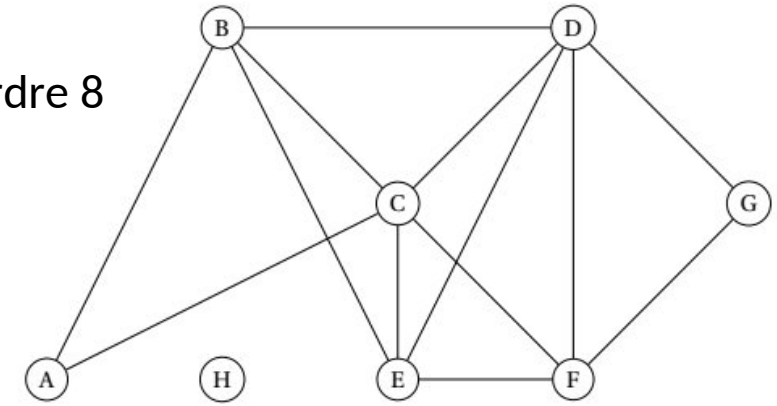
- Exemples : cables internet sous-marins



Quelques définitions

- **Définition 2 :** L'**ordre** d'un graphe est le nombre de sommets qu'il possède.

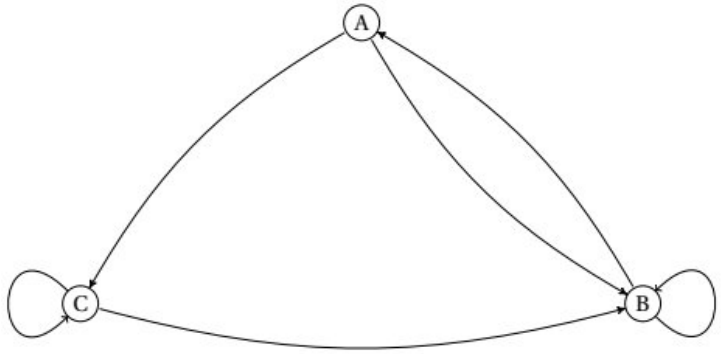
Graphe d'ordre 8



Degré de A = 2
Degré de B = 4
Degré de C = 5
Degré de D = 5
Degré de E = 4
Degré de F = 4
Degré de G = 2
Degré de H = 0

- **Définition 3 :** Le **degré** d'un sommet est le nombre d'arêtes dont ce sommet est une extrémité

Graphe d'ordre 3

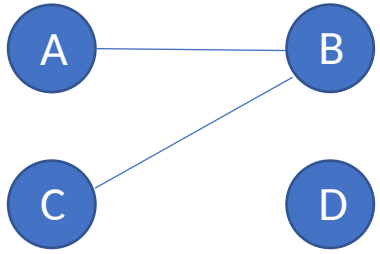


Dans un graphe orienté, on compte pour 1 les arêtes entrantes et pour 1 les arêtes sortantes. Une boucle compte donc pour 2.

Degré de A = 3
Degré de B = 5
Degré de C = 4

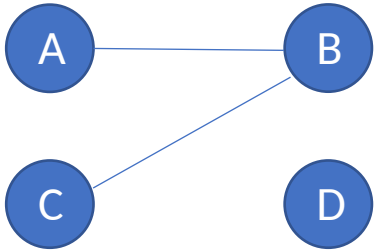
Quelques définitions

- **Définition 4** : Deux sommets distincts sont dits **adjacents** s'ils sont reliés par une arête.

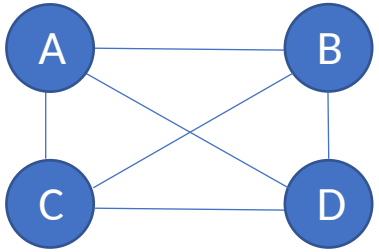


A et B sont adjacents
B et C sont adjacents

- **Définition 5** : Un graphe est dit **simple** si aucun sommet ne possède de boucle et si deux sommets distincts sont reliés par au plus une arête.

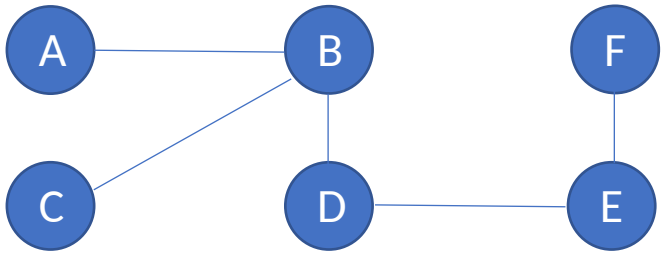


- **Définition 6** : Un graphe d'ordre n >= 1 est dit **complet** si tous ses sommets sont deux à deux adjacents.

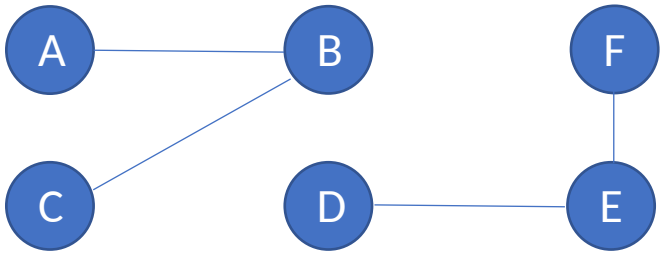


Quelques définitions

•**Définition 7** : Un graphe est dit **connexe** si il est possible, à partir de n'importe quel sommet, de rejoindre tous les autres sommets en suivant les arêtes.

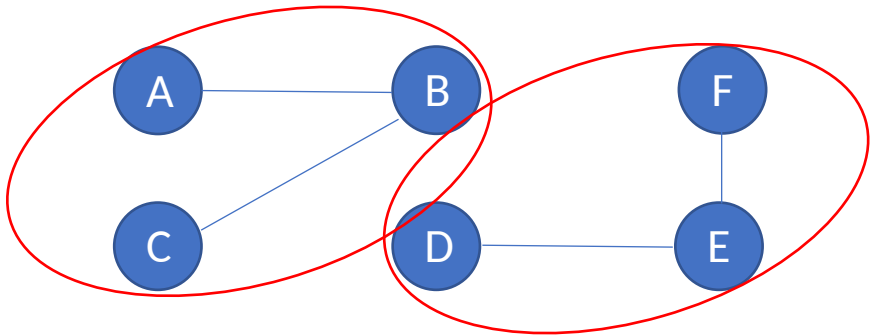


Graphe connexe



Graphe non connexe

•**Définition 8** : Un graphe non connexe se décompose en **composantes connexes**.



ici deux composantes connexes

Exemples d'applications

Exemples d'application (1)

• On a six wagons à trier. Dans une gare de triage, on doit trier 6 wagons, numérotés de 1 à 6. Les wagons entrent dans dans la gare dans l'ordre 2, 5, 3, 6, 1, 4 et doivent sortir dans l'ordre croissant. Deux wagons i et j peuvent être mis sur une même voie si et seulement s'ils entrent sur cette voie dans l'ordre dans lequel ils doivent sortir.

•De combien de voies de triage au minimum doit on disposer pour faire le tri demandé ?

- •Dessinez un graphe illustrant la situation, dans lequel un sommet représente un wagon et une arête le fait que les deux wagons (sommets) d'extrémité ne peuvent pas être sur la même voie. En déduire le nombre minimal de voies nécessaires au tri ? (*)

•Mêmes questions avec l'ordre 2, 6, 5, 3, 1, 4.

(*) source : introduction à la théorie des graphes, Didier Müller, COMMISSION ROMANDE DE MATHÉMATIQUE, 2012

Exemples d'application (2)

- Un tournoi d’échecs oppose 6 personnes. Chaque joueur doit affronter tous les autres.
  - •Construisez un graphe représentant toutes les parties possibles. Quel type de graphe obtenez-vous ?
  - •Si chaque joueur ne joue qu’un match par jour, combien de jours faudra-t-il pour terminer le tournoi ?
  - •Aidez-vous du graphe pour proposer un calendrier des matches.(*)

(*) source : introduction à la théorie des graphes, Didier Müller, COMMISSION ROMANDE DE MATHÉMATIQUE, 2012

Exemples d'application (3)

- Un jour, Sherlock Holmes reçoit la visite de son ami Watson que l'on avait chargé d'enquêter sur un assassinat mystérieux datant de plus de trois ans. À l'époque, le Duc de Densmore avait été tué par l'explosion d'une bombe, qui avait entièrement détruit le château de Densmore où il s'était retiré. Les journaux d'alors relataient que le testament, détruit lui aussi dans l'explosion, avait tout pour déplaire à l'une de ses sept ex-épouses. Or, avant de mourir, le Duc les avait toutes invitées à passer quelques jours dans sa retraite écossaise.

-Holmes : Je me souviens de cette affaire ; ce qui est étrange, c'est que la bombe avait été fabriquée spécialement pour être cachée dans l'armure de la chambre à coucher, ce qui suppose que l'assassin a nécessairement effectué plusieurs visites au château !

- Watson : Certes, et pour cette raison, j'ai interrogé chacune des femmes : Ann, Betty, Charlotte, Edith, Félicia, Georgia et Helen. Elles ont toutes juré qu'elles n'avaient été au château de Densmore qu'une seule fois dans leur vie.

- Holmes : Hum ! Leur avez-vous demandé à quelle période elles ont eu leur séjour respectif ?

- Watson : Hélas ! Aucune ne se rappelait les dates exactes, après plus de trois ans ! Néan- moins, je leur ai demandé qui elles avaient rencontré :

Ann a rencontré Betty, Charlotte, Félicia et Georgia.

- Betty a rencontré Ann, Charlotte, Edith, Félicia et Helen.
- Charlotte a rencontré Ann, Betty et Edith.
- Edith a rencontré Betty, Charlotte et Félicia.
- Félicia a rencontré Ann, Betty, Edith et Helen.
- Georgia a rencontré Ann et Helen.
- Helen a rencontré Betty, Félicia et Georgia.

Vous voyez, mon cher Holmes, les réponses sont concordantes !

C'est alors que Holmes prit un crayon et dessina un étrange petit dessin, avec des points marqué A, B, C, E, F, G, H et des lignes reliant certains de ces points.

Puis, en moins de trente secondes, Holmes déclara : "Tiens, tiens ! Ce que vous venez de me dire détermine d'une façon unique l'assassin".

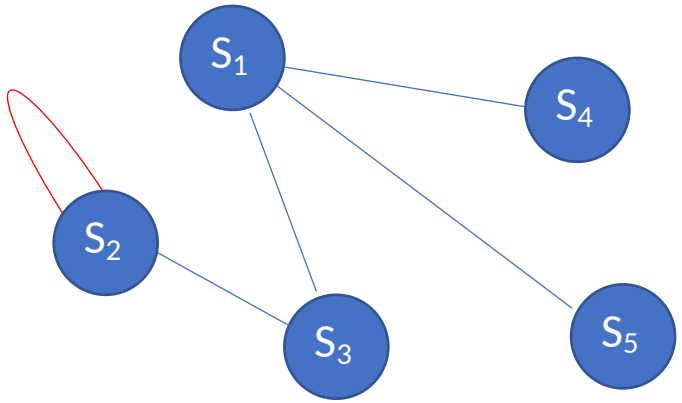
Qui est l'assassin ?(*)

(*) source : introduction à la théorie des graphes, Didier Müller, COMMISSION ROMANDE DE MATHÉMATIQUE, 2012

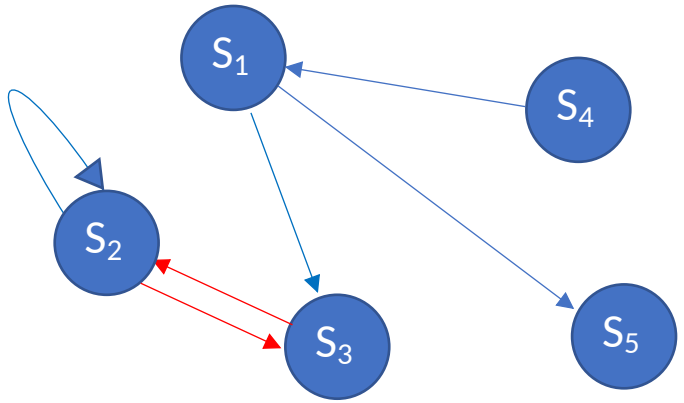
Matrice d'adjacence

Matrice d'adjacence (1)

- À tout graphe d'ordre n, de sommets S_1, S_2, …, S_n, on peut associer une matrice carrée $M=(m_{ij})$ d'ordre n, où le coefficient m_{ij} est égal au nombre d'arêtes reliant le sommet s_i au sommet s_j (en respectant le sens de parcours dans le cas d'un graphe orienté)



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

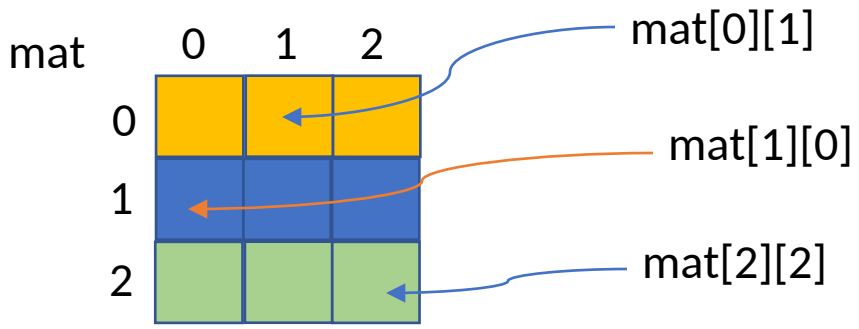
Remarque : La matrice d'adjacence d'un graphe non orienté est symétrique.

Matrice d'adjacence (2)

- Représentation générique en C
  - Dimension connue

```
// exemple : ordre 3

int mat[3][3];
```



Accès à une case mat_{ij} : mat[i][j]

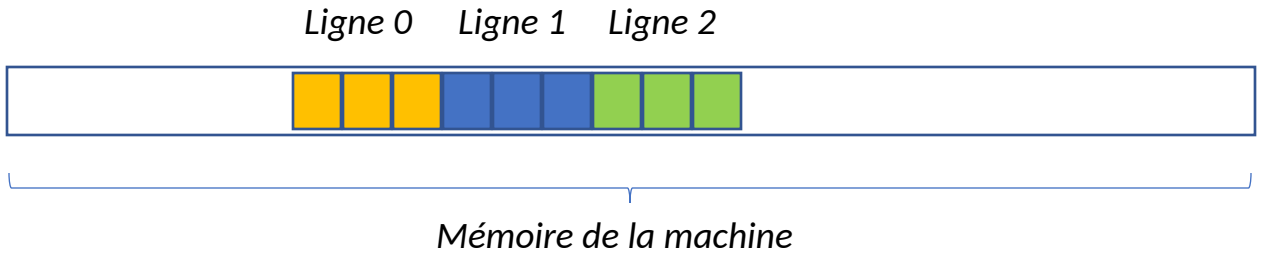
Indice de ligne

Indice de colonne

Indices dans {0, 1, …, ordre-1}

Remarque :

- les lignes sont stockées les unes après les autres en mémoire



Matrice d'adjacence (2)

- Représentation générique en C
  - Dimension inconnue

On doit stocker l'ordre et allouer dynamiquement les cases de la matrice

Accès à une case mat_{ij} : mat.coef[i *mat.ordre + j]

(j^{eme} case de la ligne i ; Début de la i^{eme} ligne)

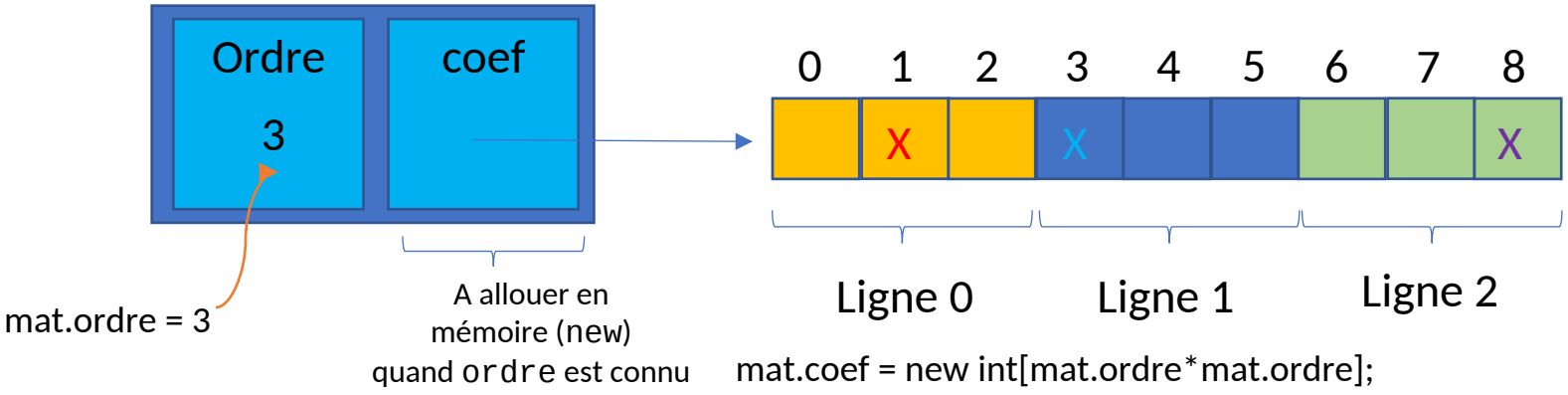
```
struct MatriceAdjacence {
  int ordre; // nombre de sommets du graphe
  int *coef; // les (ordre x ordre) coefficients
};
```

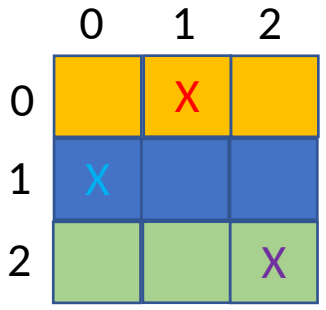
mat_{01} : mat.coef[0 * 3 + 1] = mat.coef[1]

mat_{10} : mat.coef[1 * 3 + 0] = mat.coef[3]

mat_{22} : mat.coef[2 * 3 + 2] = mat.coef[8]

MatriceAdjacence mat;

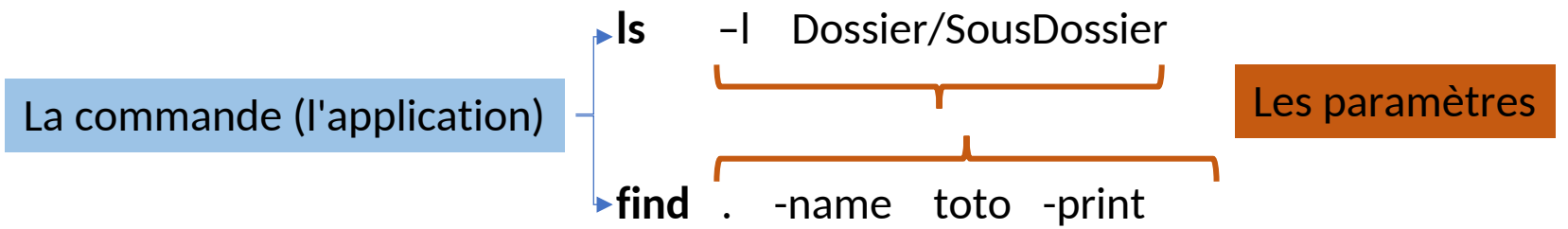




Préparation au TP

Passage de paramètres dans le main (1)

- Objectif : fournir des données à l'application sur la ligne de commande
- Exemples :



argc = 3

argv = | "ls" | "–l" | "Dossier/SousDossier" |

argc = 5

argv = | "find" | "." | "-name" | "toto" | "-print" |

- Généralisable à toute application

```
int main(int argc, char *argv[])
```

Nombre de "mots" sur la ligne de commandes

Liste des "mots"

Passage de paramètres dans le main (2)

- Exemple :
  - Affichage de la liste des paramètres

exple.cpp

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){

  for(int i=0; i<argc; i++)
    cout << "->" << argv[i] << endl;

  return 1;
}
```

Compilation

```
g++  -std=c++11  exple .cpp  -o  exple

./exple  toto titi
-> ./exple
-> toto
-> titi
```

Exécution

- argc >=1
- argv[0] est le nom de la commande

Lecture dans un fichier texte (1)

- Outils
  - Utilisation de flots (stream) dédiés au fichiers
  - Définis dans la bibliothèque <fstream>
  - Type ifstream (input file stream)
  - Utilise le même opérateur que pour la lecture clavier (cin) : >>
- Étapes d'utilisation
  1. Ouvrir le fichier en lecture
  2. Vérification de l'ouverture
  3. Lectures
  4. Fermeture du fichier

Lecture dans un fichier texte (2)

void *flot* .open(nom_fichier,
mode d'ouverture)

Modes définis par :
- ios::out // ouverture en écriture
- ios::in // ouverture en lecture

- Refermer le fichier connecté au flot après utilisation
- Après fermeture, la variable flot peut être réutilisée pour un autre fichier

void *flot*.close()

```
#include <fstream>
...
ifstream fichier;// flot d'entrée

// ouverture du fichier en mode lecture
fichier.open ("test.txt", ios::in);

// test d'ouverture du fichier
if(fichier.is_open()==false){
    cout << "erreur d'ouverture "<< endl;
 return;
}

// lecture d'un entier dans le fichier
int vEntiere
fichier >> vEntiere;

//lecture d'un réel dans le fichier
float vReelle;
fichier >> vReelle;

// fermeture du fichier
fichier.close();
```

Toujours tester l'ouverture avant d'utiliser le flot la première fois

bool *flot*.is_open()

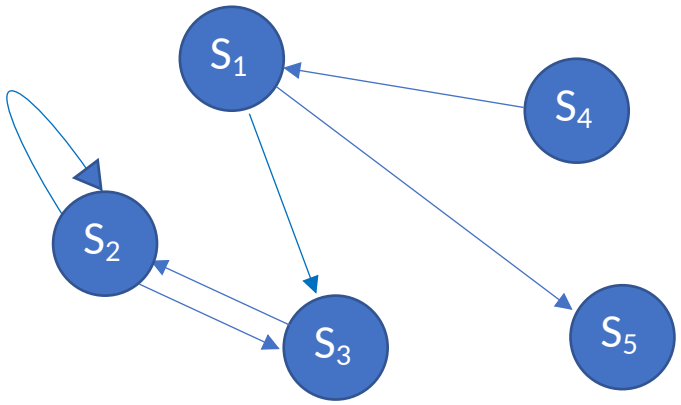
Lecture avec l'opérateur >>
(comme pour cin) :
- L'opérateur convertit les données lues dans le type attendu en partie droite

Matrice d'adjacence compacte

Matrice d'adjacence compacte (1)

- Retour sur la représentation matricielle

Avantage : simplicité



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Inconvénient : coût mémoire

Matrice de taille N x N

1 octet par coefficient

→ N^2 octets

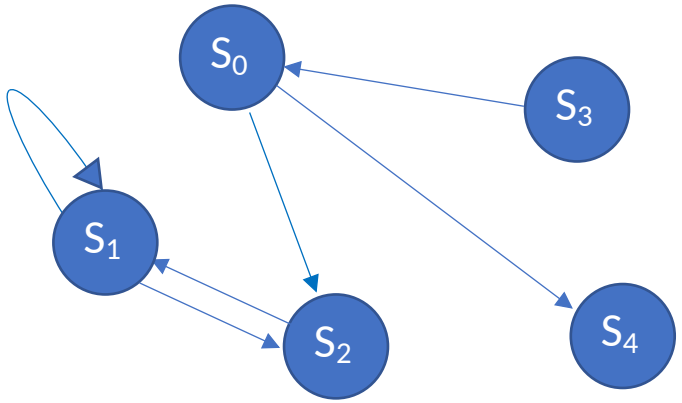
N	Taille mémoire
1000	1 Mo
10000	100 Mo
36000	1.2 Go

Nombre de communes en France
(cf guidage GPS)

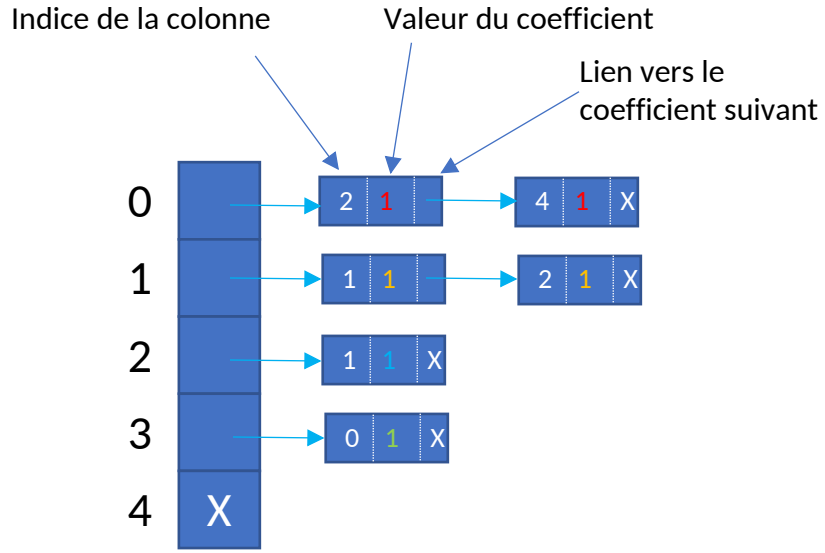
Nombreux 0

Matrice d'adjacence compacte (2)

- Représentation compacte
  - Matrice creuse
  - On ne stocke que les coefficients non nuls



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Une liste chaînée des coefficients non nuls pour chaque ligne

Matrice d'adjacence compacte (3)

- Rappels sur les pointeurs

TYPE `*var;`

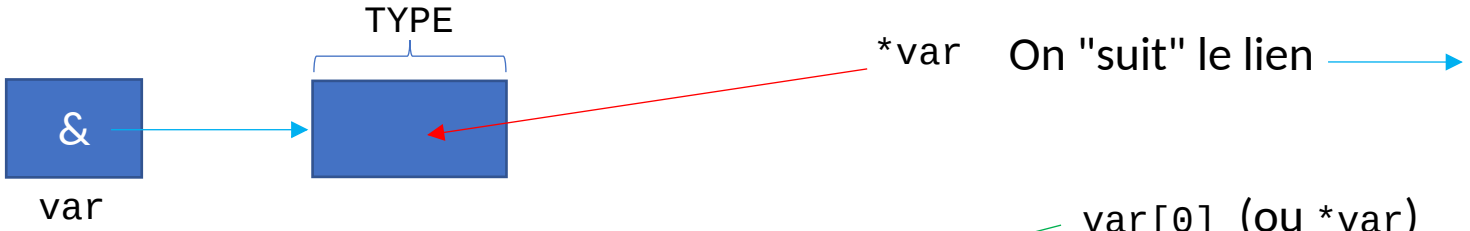
les adresses sont typées

`var` est un pointeur vers une zone mémoire → Contiendra l'adresse du début de la zone

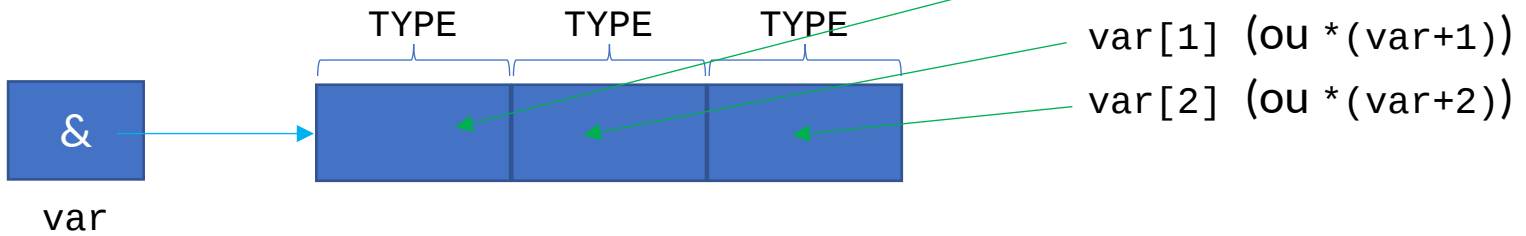
Cette zone mémoire devra contenir un (ou plusieurs) élément(s) de type TYPE.

Elle doit être allouée dynamiquement avec l'opérateur `new`

var = new TYPE;

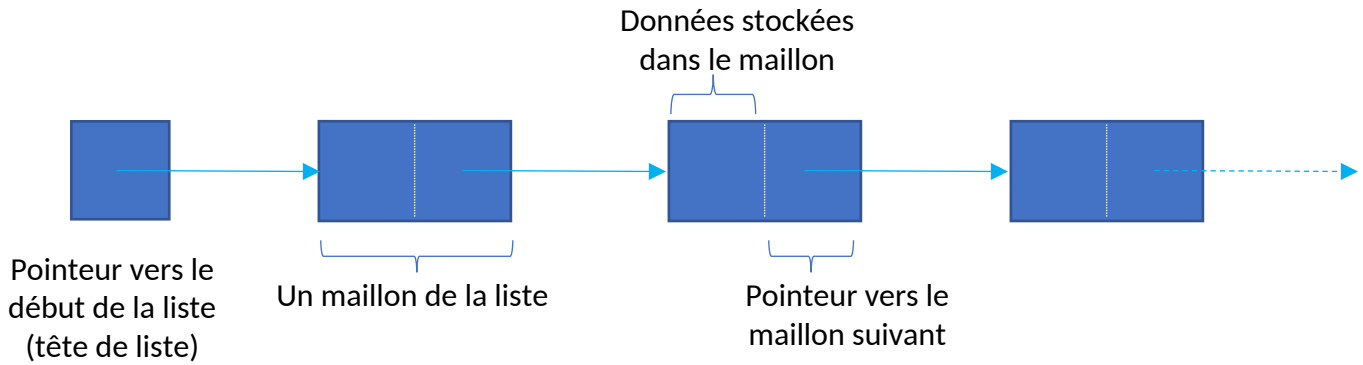


var = new TYPE[3];



Matrice d'adjacence compacte (4)

- Rappel sur les listes chaînées
  - Structure dynamique permettant de représenter des listes
  - Longueur inconnue *a priori* (donc tableaux inutilisables)




```
// maillon d'une liste chaînée

struct Maillon {
  Type donnees;   // données stockées dans le maillon
  Maillon *suiv;  // pointeur vers l'élément suivant
} ;
```

Matrice d'adjacence compacte (5)

- Rappel sur les listes chaînées

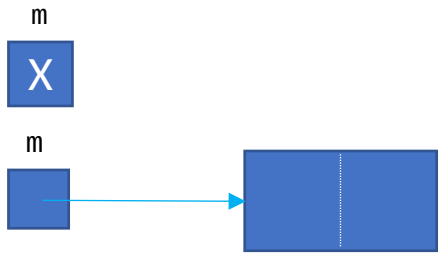
Initialisation d'une nouvelle liste :

```
Maillon *teteListe = nullptr;
```

Création d'un nouveau maillon :

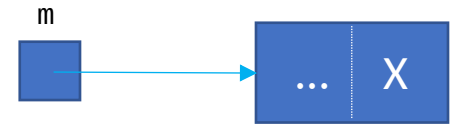
```
Maillon *m;

m = new Maillon;
```



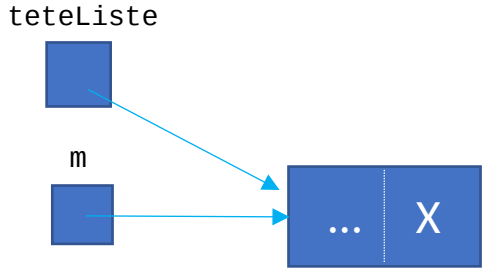
Initialisation du nouveau maillon :

```
m->donnees = …;
m->suiv = nullptr;
```



Ajout du premier maillon à la liste :

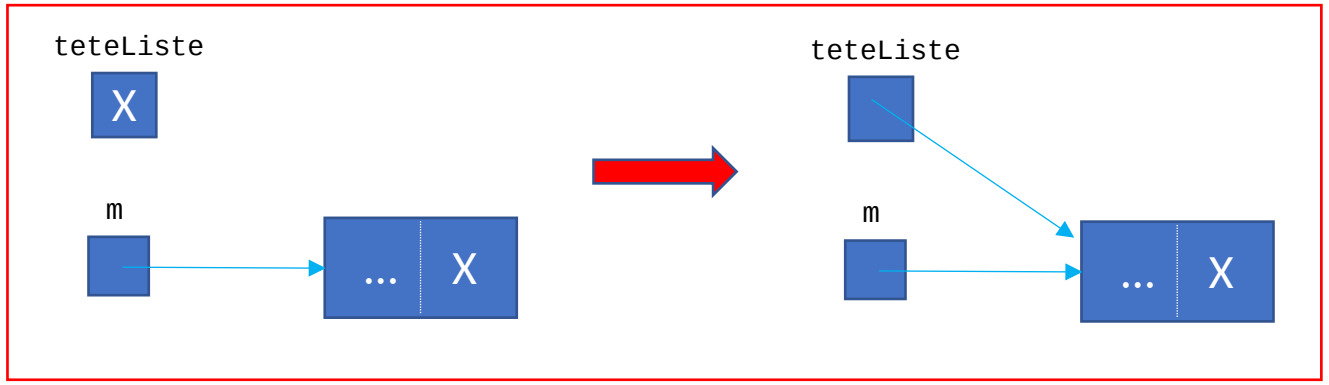
```
teteListe = m;
```

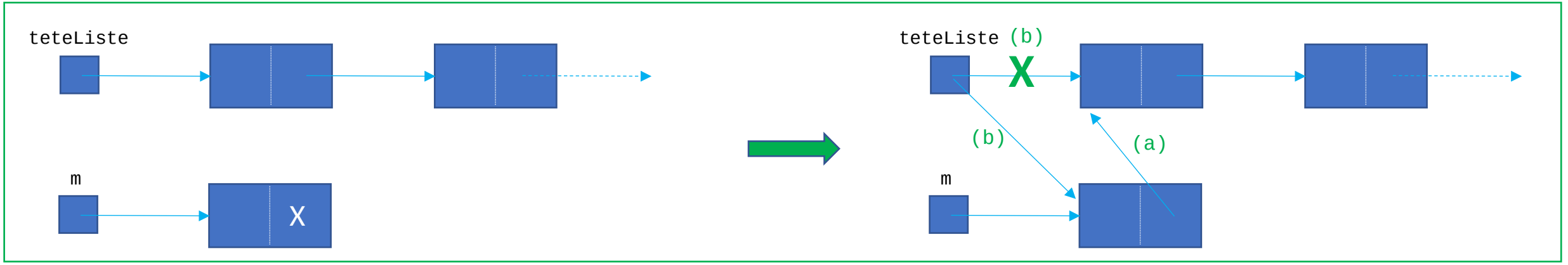


Matrice d'adjacence compacte (6)

- Rappel sur les listes chaînées
  - Insertion en tête de liste

```
if(teteListe == nullptr){ // la liste est vide
  teteListe = m;
} else { // la liste n'est pas vide
  m->suiv = teteListe; (a)
  teteListe = m; (b)
}
```

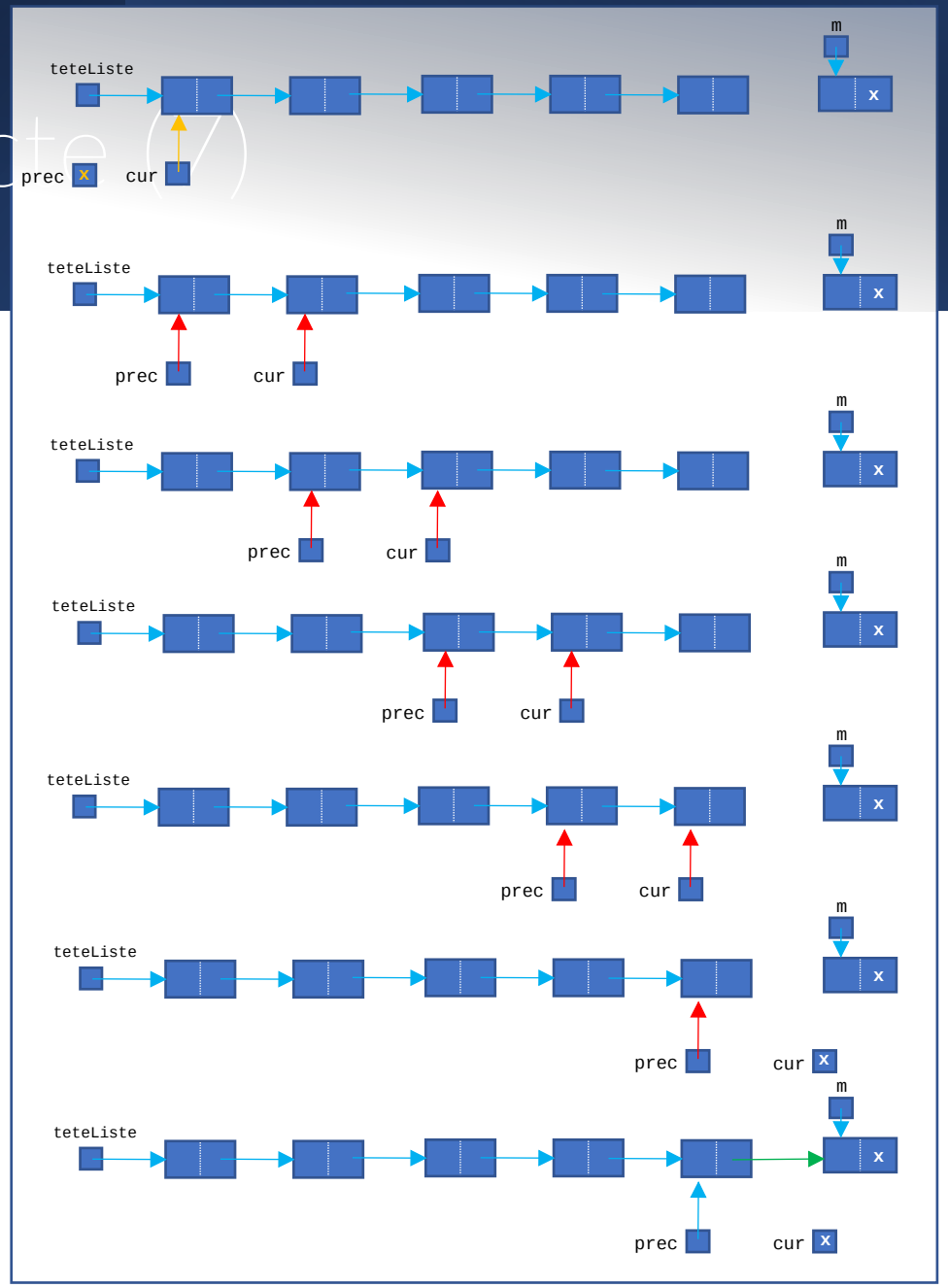




Matrice d'adjacence compacte (7)

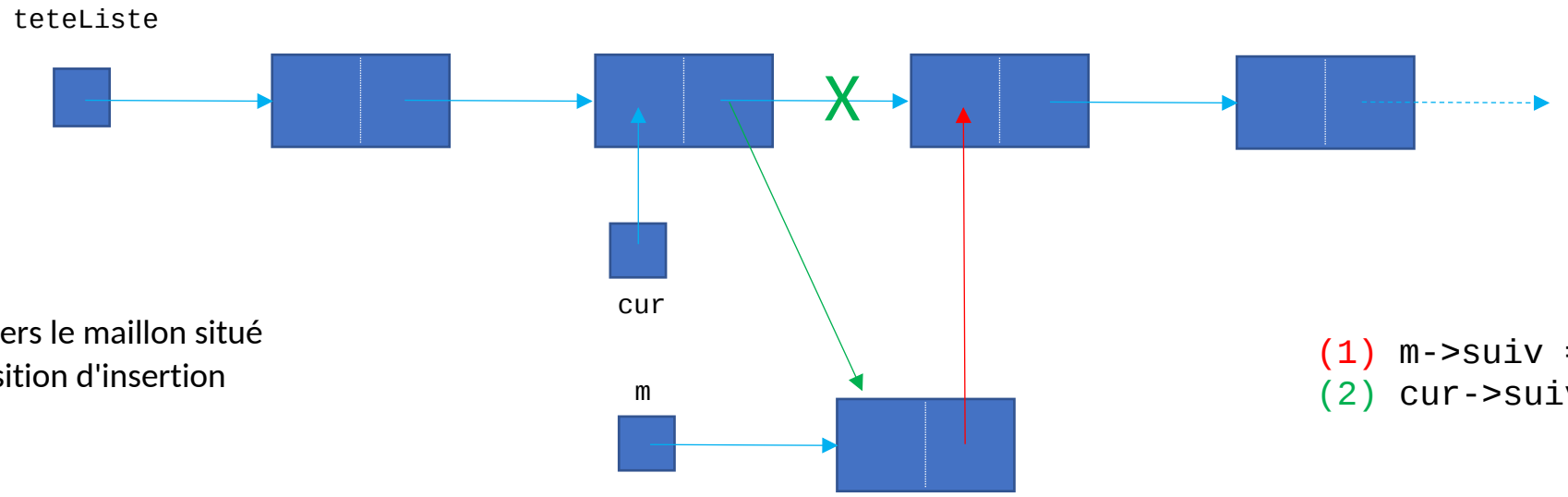
- Rappel sur les listes chaînées
  - Ajout en fin de liste

```
if(teteListe == nullptr){ // la liste est vide
  teteListe = m;
} else { // la liste n'est pas vide
  Maillon *cur = teteListe; //pointeur de parcours
  Maillon *prec = nullptr; // pointeur vers le maillon précédant cur
  while(cur!=nullptr){// il y a encore des maillons derrière cur
    prec = cur;
    cur = cur->suiv;
  }// while
  // cur pointe sur le dernier maillon
  prec->suiv = m;
}
```



Matrice d'adjacence compacte (8)

- Rappel sur les listes chaînées
  - Insérer entre deux maillons



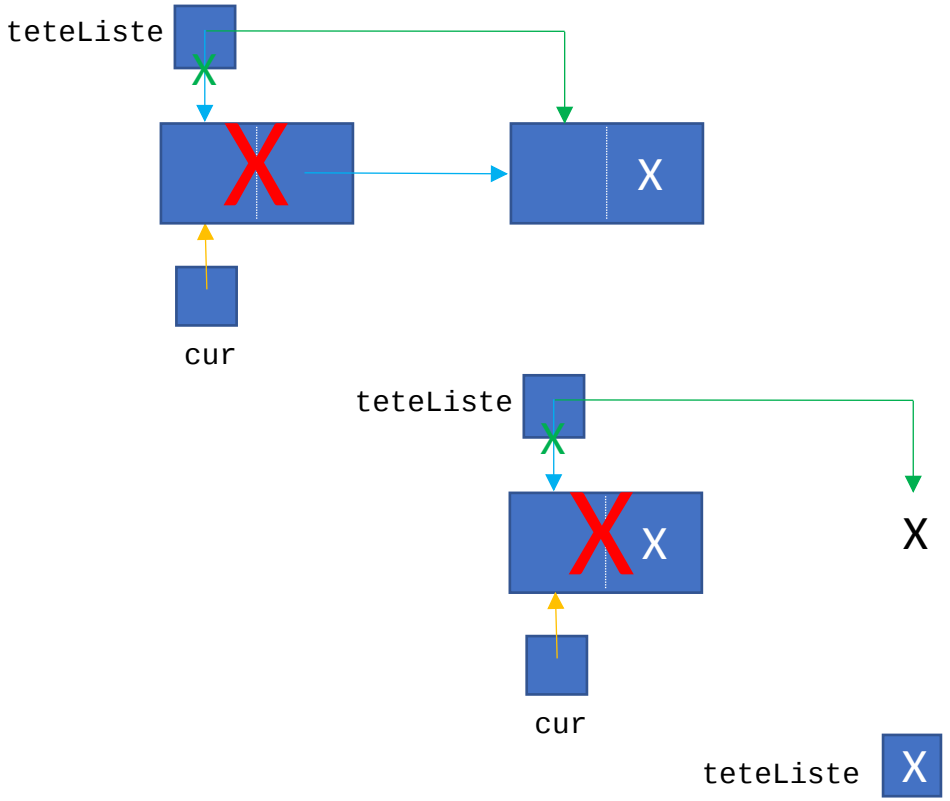
`cur` = pointeur vers le maillon situé juste avant la position d'insertion

```
(1) m->suiv = cur->suiv;
(2) cur->suiv = m;
```

Matrice d'adjacence compacte (9)

- Rappel sur les listes chaînées
  - Effacer une liste

```
Maillon *cur;
...
while(teteListe != nullptr){
  cur = teteListe;
  teteListe = teteListe->suiv;
  delete cur;
}
```



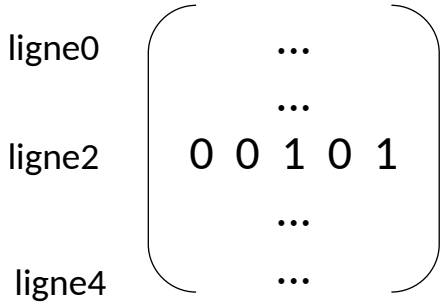
Matrice d'adjacence compacte (10)

- Structures de données pour la représentation compacte

Structure de données pour une liste chaînée

```
// maillon d'une liste chaînée

struct Maillon {
  int col;   // numéro de la colonne à laquelle correspond le coefficient
  int coef;    // coefficient de la matrice
  Maillon *suiv;  // élément suivant non nul sur la ligne
} ;
```



Matrice d'adjacence compacte (11)

- Structures de données pour la représentation compacte

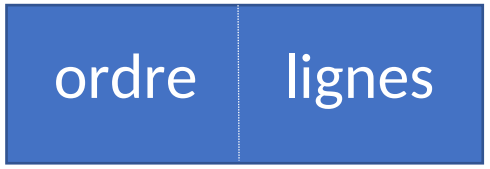
Structure de données pour une matrice d'adjacence

```
// représentation compacte d'une matrice d'adjacence

struct MatriceAdjacence {
  int ordre; // nombre de sommets du graphe
  Maillon** lignes; // tableau à allouer de taille "ordre",
                    // représentant les lignes de la matrice
};
```

Chaque case est une pointeur vers la liste chainée contenant les valeurs non nulles de la matrice.

Tableau
(1 case par ligne de la matrice)



MatriceAdjacence mat;

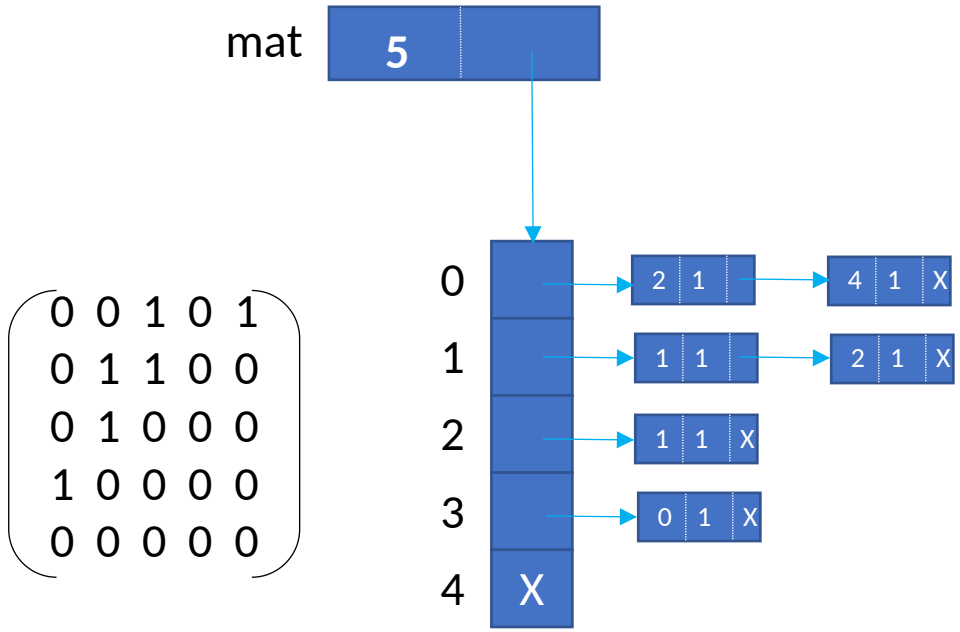


Tableau de têtes de liste chaînée
(1 par ligne)

Matrice d'adjacence (13)

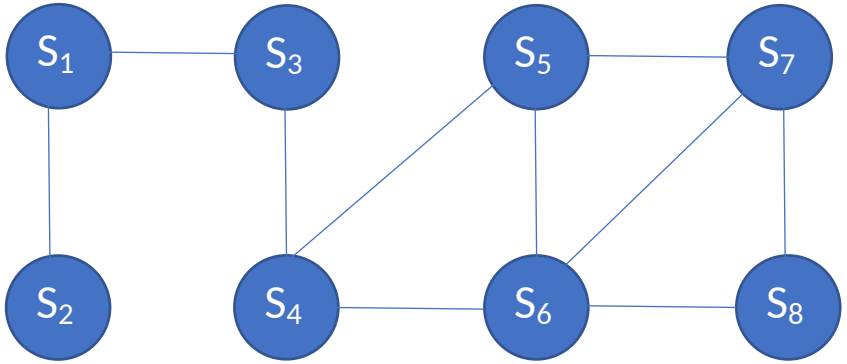
- Exercice
  - Relire et afficher le contenu d'un fichier contenant N entiers, la valeur N étant le premier entier présent dans le fichier.
  - Compléter le programme pour que les entiers soient stockés dans une liste chaînée en ordre inverse de leur apparition dans le fichier.
  - Compléter le programme pour afficher le contenu de la liste chaînée.
  - Compléter le programme pour effacer le contenu de la liste chaînée.

```
5
67 45
89 12
4
```

Parcours en largeur

Parcours en largeur (1)

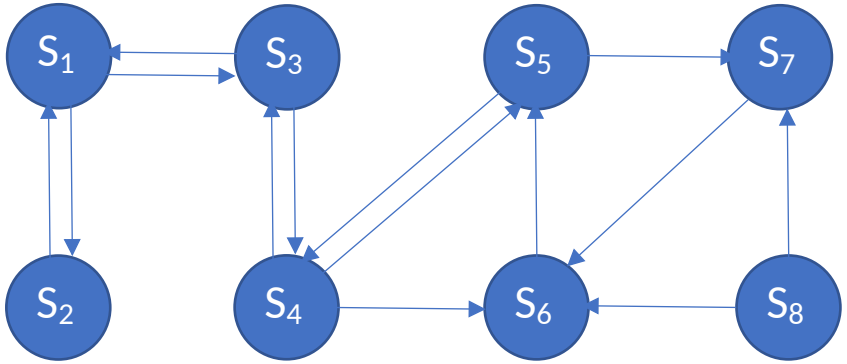
- Questions :
  - À partir d'un sommet A, peut on atteindre un sommet B ?
  - Si oui, en combien d'étapes au minimum et par quel chemin ?



S1 à S8 possible ? **oui**

En combien d'étapes au minimum ? **4**

Chemin = S1, S3, S4, S6, S8



S1 à S8 ? impossible

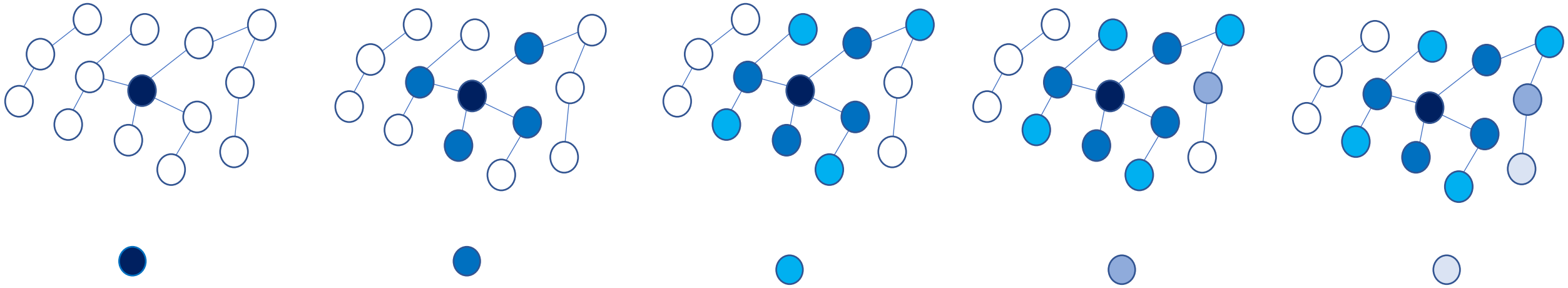
S8 à S1 ? possible en minimum 5 étapes

Chemin = S8, S6, S5, S4, S3, S1

Parcours en largeur (2)

- Idée générale
  - Explorer le graphe à partir du sommet demandé
  - Avancer par couche successive

Sommets inatteignables



Sommet de départ

Sommets de la 1ere couche

Accessibles directement depuis le sommet de départ

Sommets de la 2nde couche

Accessibles depuis la 1ere couche sans revenir en "arrière"

Sommets de la 3eme couche

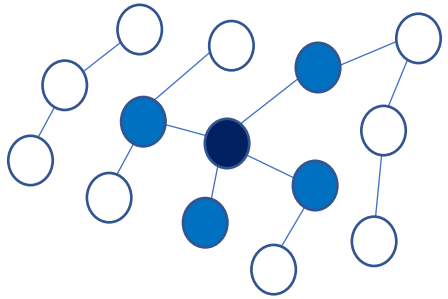
Accessibles depuis la 2nde couche sans revenir en "arrière"

Sommets de la 4eme couche

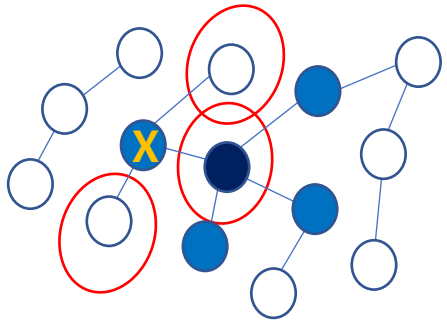
Accessibles depuis la 3eme couche sans revenir en "arrière"

Parcours en largeur (3)

- Problème 1 :
  - Comment ne pas repartir en arrière ?



On construit la 2nde couche



X Sommet en cours d'examen

Sommets adjacents à X

On repart en arrière !!!

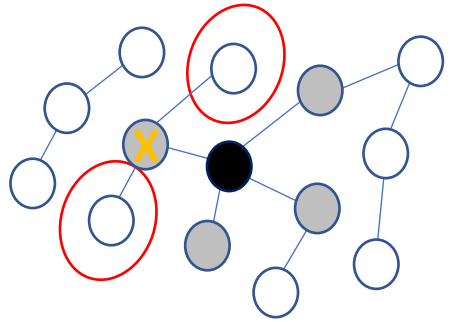
Il faut se souvenir qu'un sommet a déjà été exploré

On associe un code couleur à chaque sommet :

Sommet non exploré

Sommet traité
on a construit la couche suivante à partir de ce sommet

Sommet dans la couche courante
frontière entre ce qui a été traité et ce qui est en cours de traitement

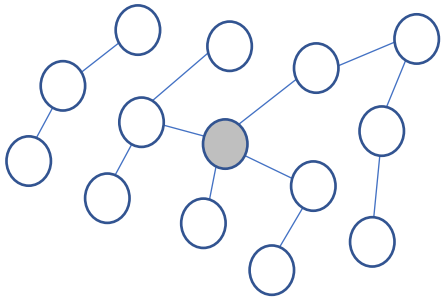


X Sommet en cours d'examen

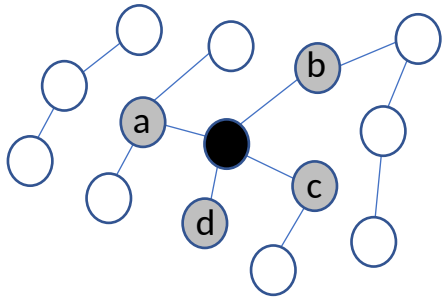
Sommets **blancs** adjacents à X

Parcours en largeur (4)

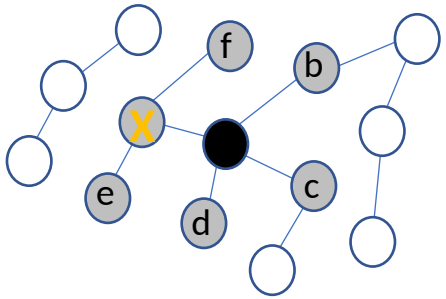
- Problème 2 :
  - On ne peut traiter qu'un seul sommet à la fois
    - Comment stocker les sommets "gris" en attente de traitement ?



Au départ : 1 seul sommet gris



1ere couche : 4 sommets gris



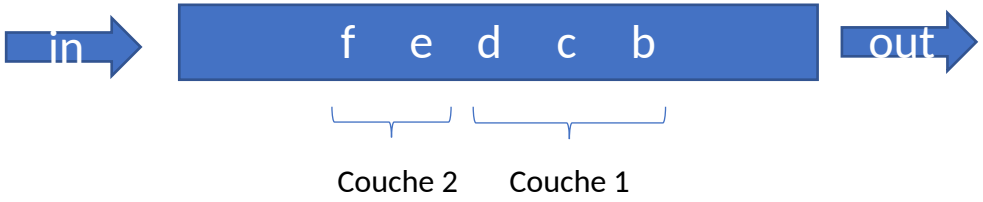
On traite X : 2 sommets gris en plus

<u>Problème supplémentaire</u> :

Quels sommets gris sont dans la couche 1 et lesquels sont dans la couche 2 ?

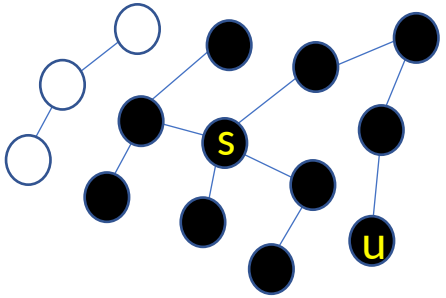
- On veut explorer le graphe couche par couche …

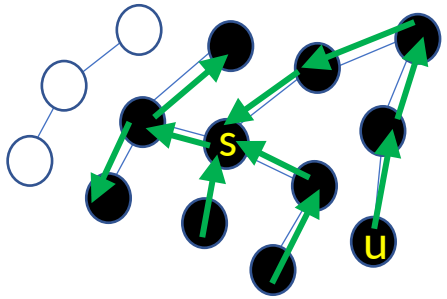
On les stocke dans une structure de file de type FIFO (*First In First Out*)



Parcours en largeur (5)

- Problème 3:
  - Lors du parcours en largeur, on ne sait plus d'où on vient
  - On ne sait pas retrouver le chemin à prendre pour aller d'un noeud **s** à un noeud **u** quand tout le graphe a été parcouru
    - On sait juste quels sommets sont accessibles

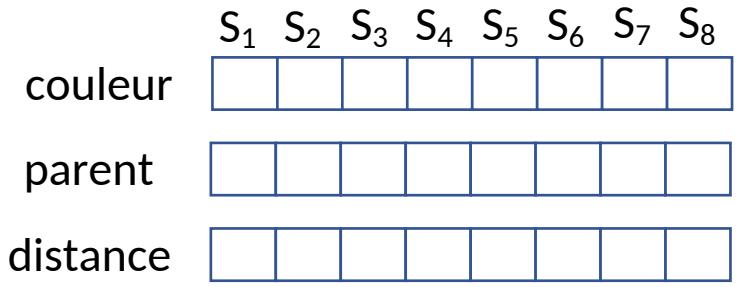




  - Lors de l'exploration, on conserve un lien vers le noeud gris parent dans la couche supérieure … (un seul choix possible …)

Parcours en largeur (6)

- Algorithme:
  - Données utilisées :
    - 3 tableaux de même taille que le nombre de sommets
      - Couleur : mémorise la couleur de chaque sommet (blanc, gris, noir)
      - Parent : mémorise l'indice du sommet parent trouvé lors de l'exploration
      - Distance : mémorise le nombre d'arcs utilisés pour aller du sommet de départ au sommet correspondant à l'indice dans le tableau



Parcours en largeur (7)

- Algorithme:
  - initialisations

Tous les sommets sont inexplorés → initialisés en "blanc"

On ne connaît par le nombre d'étapes nécessaires pour les atteindre → distance initialisée à l'infini

Aucun parent connu → parent initialisé à "null"

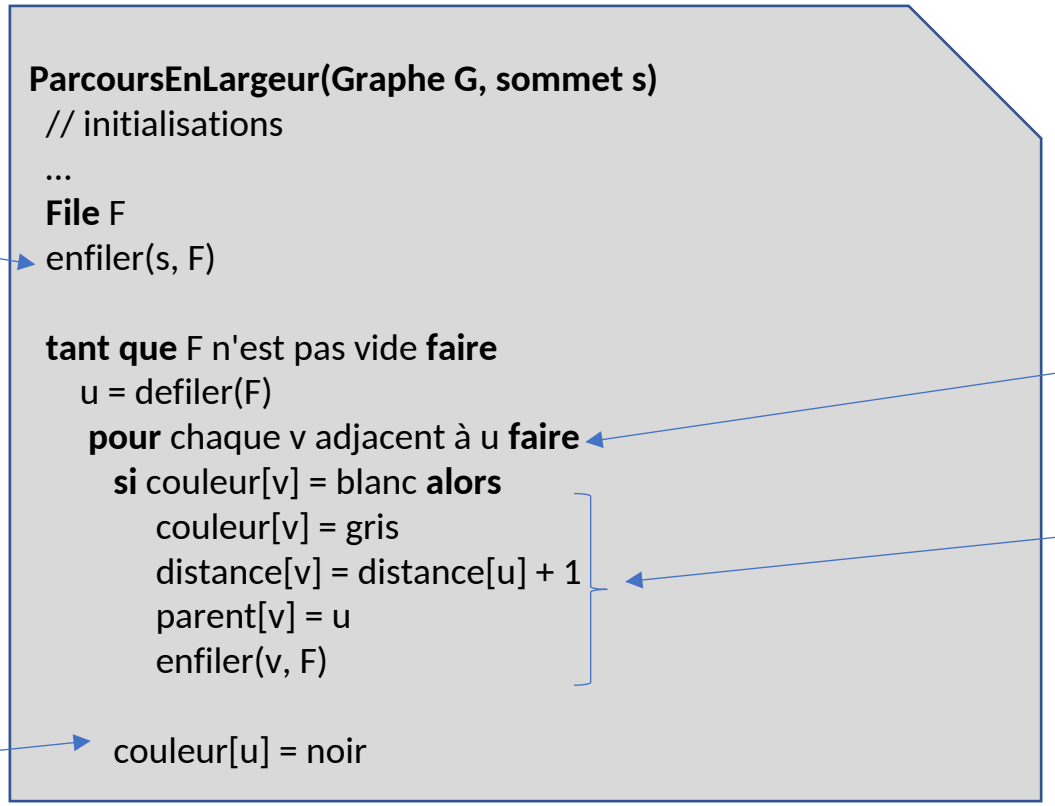
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	∞	∞	∞	∞	∞	∞	∞	∞
parent	-	-	-	-	-	-	-	-

```
ParcoursEnLargeur(Graphe G, sommet s)
  // initialisation de tous les sommets
  pour chaque sommet u de G faire
     couleur[u] = blanc
     distance[u] = infini
     parent[u] = null

  // initialiser le sommet de départ
  couleur[s] = gris
  distance [s] = 0
  parent[s] = null
```

Parcours en largeur (8)

- Algorithme:
  - La boucle d'analyse



Le sommet de départ s est le premier sommet "frontière"

On regarde tous les sommets adjacents au sommet (gris) en cours d'analyse

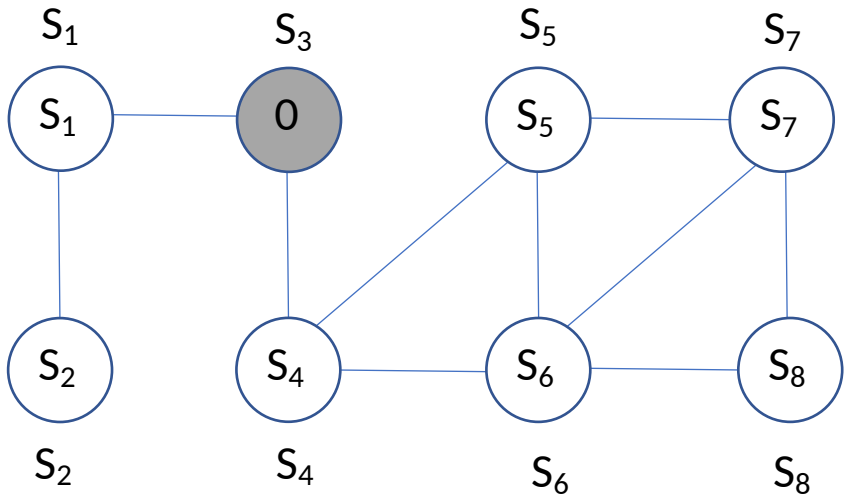
un sommet adjacent blanc devient un nouveau sommet frontière qu'il faudra analyser par la suite

- sommet adjacent gris : il est déjà dans la file
- sommet adjacent noir : on y repassera plus

L'analyse du sommet (gris) courant est terminée - il devient noir

Parcours en largeur (9)

- Exemple
  - Initialisation du sommet de départ



File

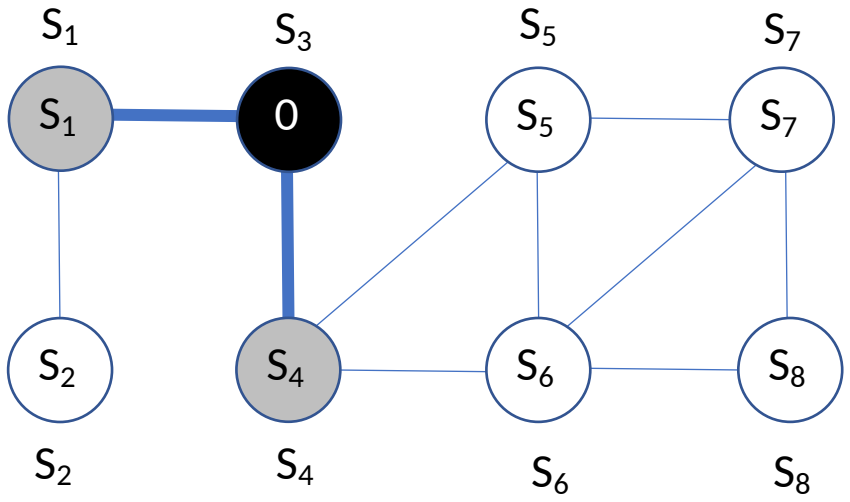
S_3

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	∞	∞	0	∞	∞	∞	∞	∞
parent	-	-	-	-	-	-	-	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (10)

- Exemple
  - Obtention de la 1ere couche



File

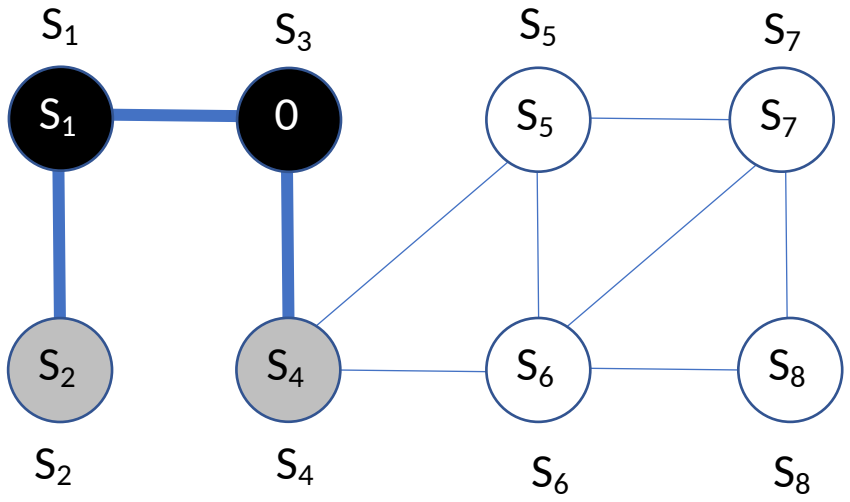
S_4	S_1

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	1	∞	0	1	∞	∞	∞	∞
parent	S_3	-	-	S_3	-	-	-	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (11)

- Exemple
  - Début de la 2nde couche



File

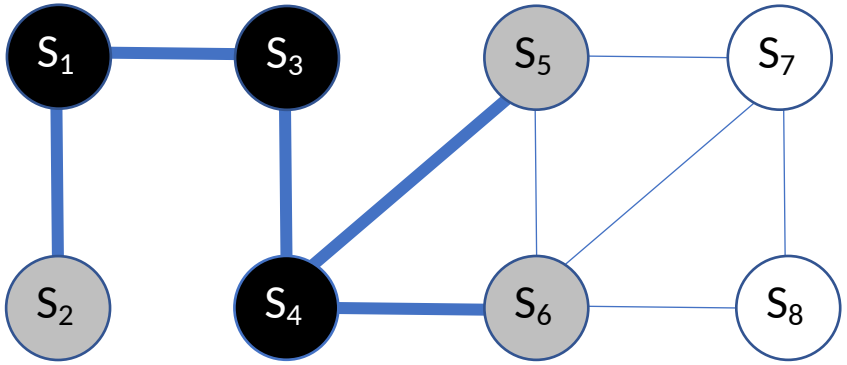
S_2	S_4

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur	■	■	■	■				
distance	1	2	0	1	∞	∞	∞	∞
parent	S_3	S_1	-	S_3	-	-	-	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (12)

- Exemple
  - Fin de la seconde couche



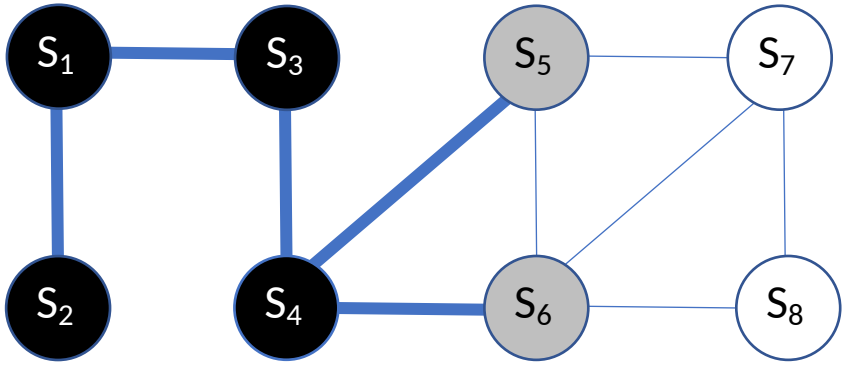
File

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur	■	■	■	■	■	■	□	□
distance	1	2	0	1	2	2	∞	∞
parent	S_3	S_1	-	S_3	S_4	S_4	-	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (13)

- Exemple
  - Début de la 3eme couche



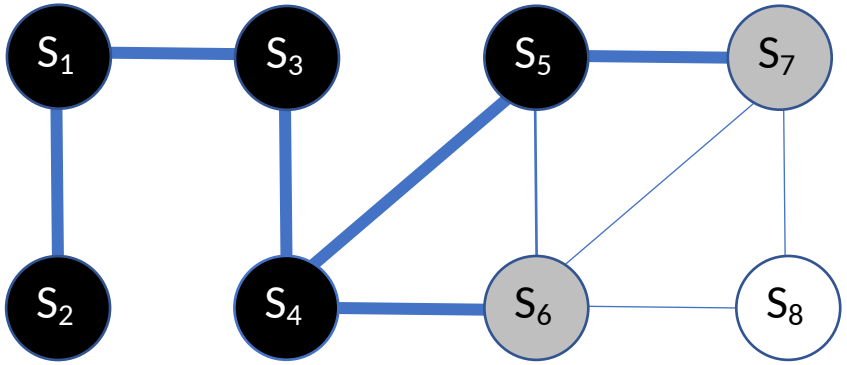
File

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur	■	■	■	■	▒	▒	□	□
distance	1	2	0	1	2	2	∞	∞
parent	S_3	S_1	-	S_3	S_4	S_4	-	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les tableaux sont mis à jour

Parcours en largeur (14)

- Exemple
  - Milieu de la 3eme couche



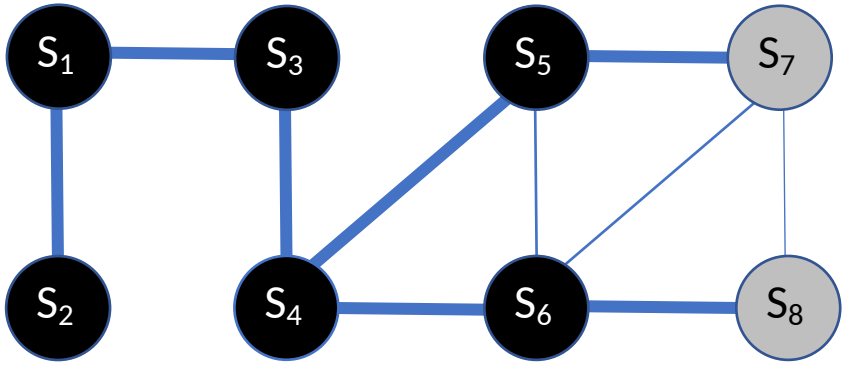
File

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	1	2	0	1	2	2	3	∞
parent	S_3	S_1	-	S_3	S_4	S_4	S_5	-

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (15)

- Exemple
  - Fin de la 3eme couche



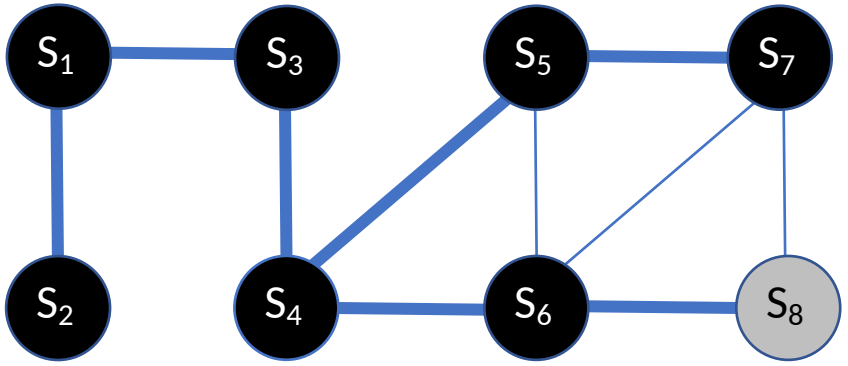
File

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur	■	■	■	■	■	■		
distance	1	2	0	1	2	2	3	3
parent	S_3	S_1	-	S_3	S_4	S_4	S_5	S_6

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (16)

- Exemple
  - Traitement de la 3eme couche (1)



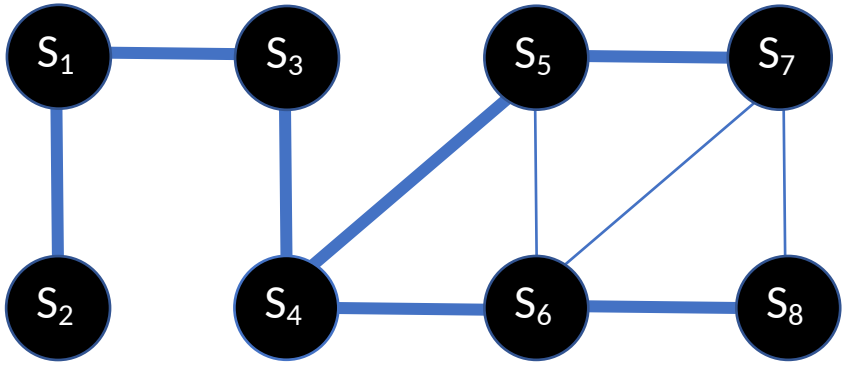
File

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur	■	■	■	■	■	■	■	
distance	1	2	0	1	2	2	3	3
parent	S_3	S_1	-	S_3	S_4	S_4	S_5	S_6

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en largeur (17)

- Exemple
  - Traitement de la 3eme couche (2)



File

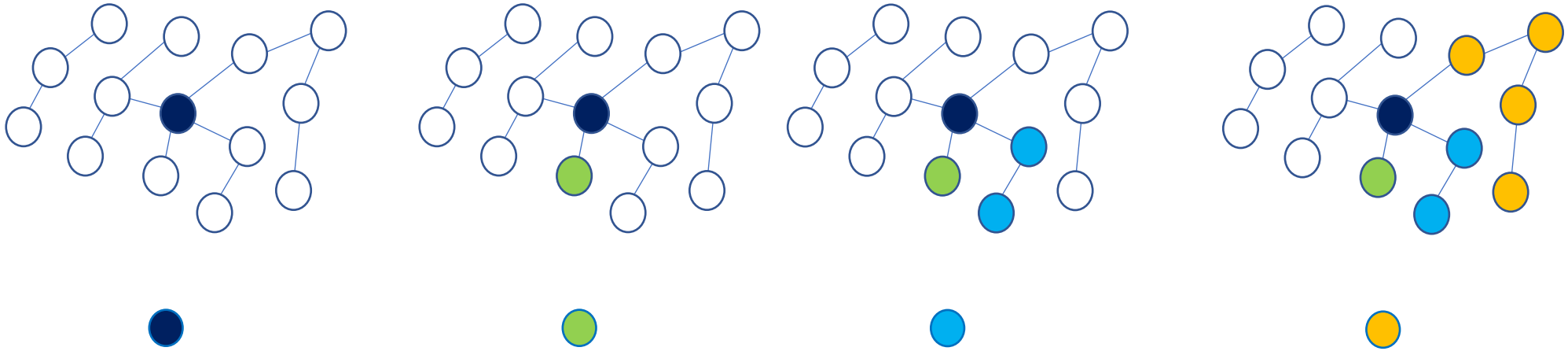
	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	1	2	0	1	2	2	3	3
parent	S_3	S_1	-	S_3	S_4	S_4	S_5	S_6

- On récupère le premier noeud de la file et on "enfile" les nœuds avec lesquels il est connecté ;
- Le noeud récupéré passe à "noir"
- Les nœuds enfilés deviennent gris
- Les tableaux sont mis à jour

Parcours en profondeur

Parcours en profondeur

- Objectifs :
  - Explorer l’ensemble des noeuds d'un graphe
  - Suivre les arêtes/arcs le plus profondément possible à chaque exploration



1er sommet considéré

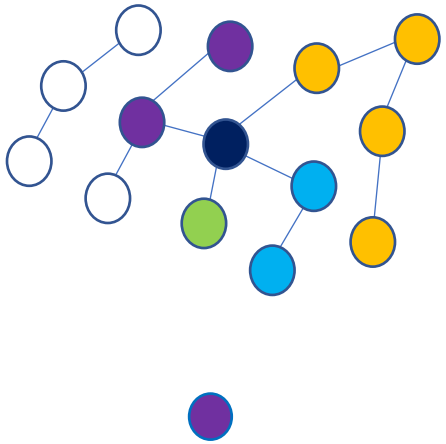
1ere branche adjacente explorée

2nde branche adjacente explorée

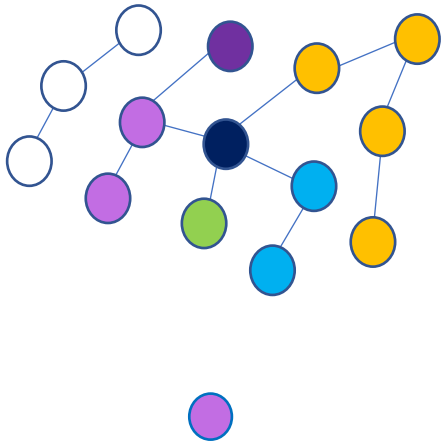
3eme branche adjacente explorée

Parcours en profondeur

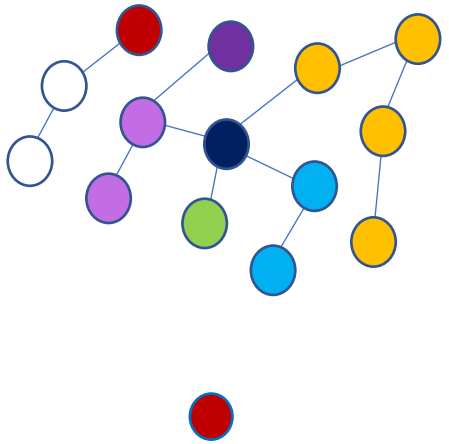
- Objectifs :
  - Explorer l’ensemble des noeuds d'un graphe
  - Suivre les arêtes/arcs le plus profondément possible à chaque exploration



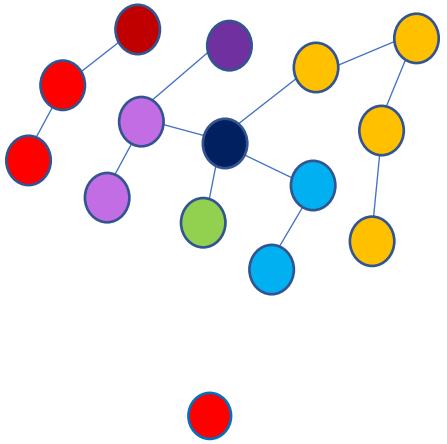
4eme branche adjacente explorée 1ere sous-branche



4eme branche adjacente explorée 2nde sous-branche



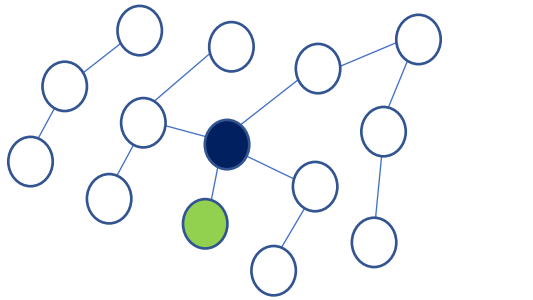
Démarrage d'un second sommet non exploré



Unique branche adjacente

Parcours en profondeur

- Problème 1 :
  - Comment ne pas repartir en arrière ?



On est parti de ● et on a trouvé ●

On explore les sommets adjacents de ●

Il n'y a que ● → On repart en arrière !!!

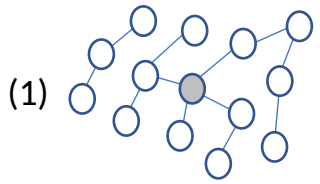
Il faut se souvenir qu'un sommet a déjà été exploré

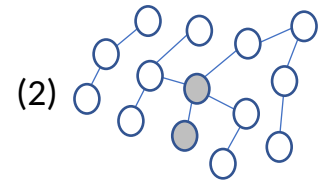
On associe un code couleur à chaque sommet :

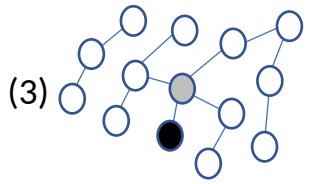
○ Sommet non exploré

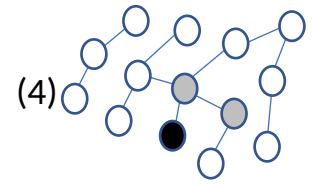
● Sommet traité
on a exploré toutes ses branches d'adjacences

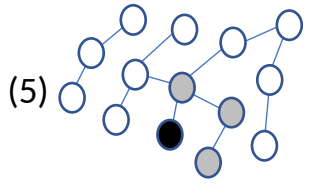
● Sommet en cours de traitement
Ses branches d'adjacence sont en cours d'exploration

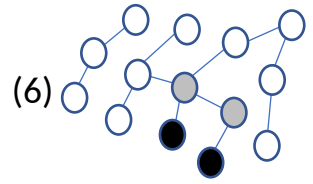


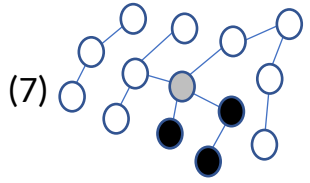


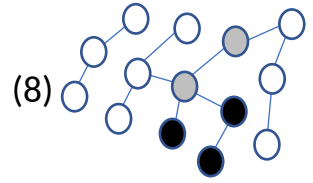






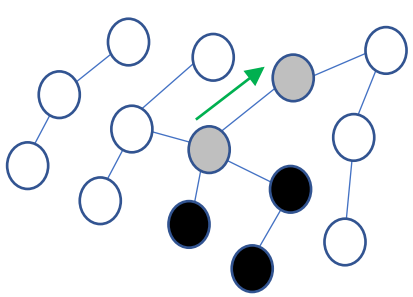




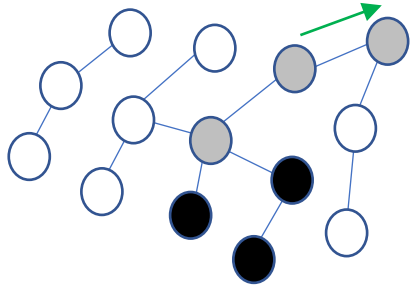


Parcours en largeur

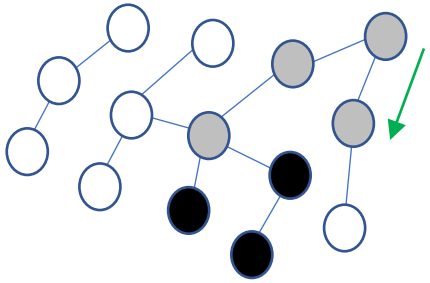
- Problème 2 :
  - On ne peut traiter qu'un seul sommet à la fois
    - Comment stocker les sommets "gris" en attente de traitement ?



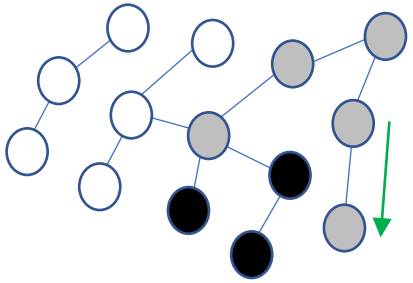
On descend d'un niveau



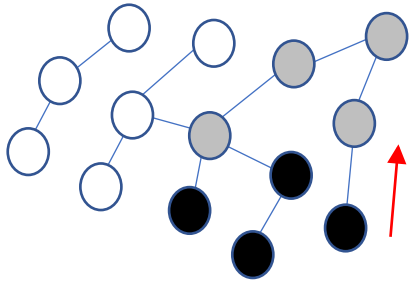
On descend d'un niveau



On descend d'un niveau



On descend d'un niveau



On doit remonter les niveaux

  - 2 possibilités :
    - Utiliser une pile (qui stocke les sommets gris successifs rencontrés dans une branche)
    - Utiliser la récursivité de la fonction de parcours

Parcours en profondeur

- Algorithme récursif:
  - Initialisations & lancement

Tous les sommets sont inexplorés ⟹ initialisés en "blanc"

Aucun parent connu ⟹ parent initialisé à "null"

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
parent	-	-	-	-	-	-	-	-

```
ParcoursEnProfondeur(Graphe G)
  // initialisation de tous les sommets
  pour chaque sommet u de G faire
     couleur[u] = blanc
     parent[u] = null

  // lancer l'exploration à partir de chaque sommet
  pour chaque sommet u de G faire
      si couleur[u] = blanc
      alors explorerDepuis(G, u)
```

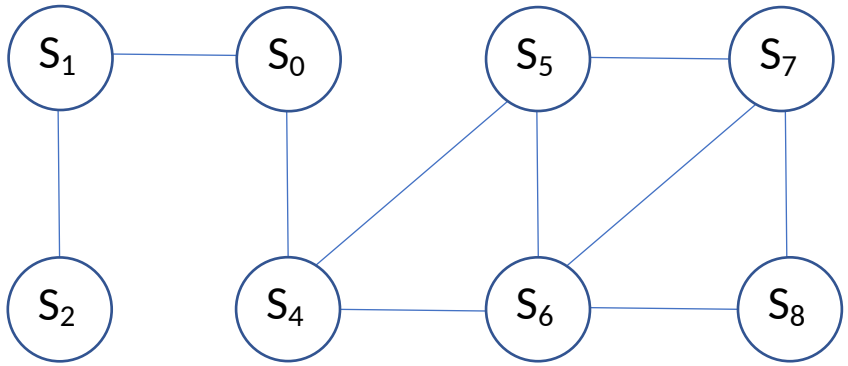
Parcours en profondeur

- Algorithme:
  - Exploration en profondeur

```
ExplorerDepuis(Graphe G, Sommet u)
  // le sommet u vient d'être découvert
  couleur[u] = gris

 // visiter les sommets adjacents
 pour chaque sommet v adjacent à u faire
    si couleur[v] = blanc
    alors
       parent[v] = u
       ExplorerDepuis(G, v) // exploration en profondeur

 // fin de traitement du sommet u
 couleur[u] = noir
```



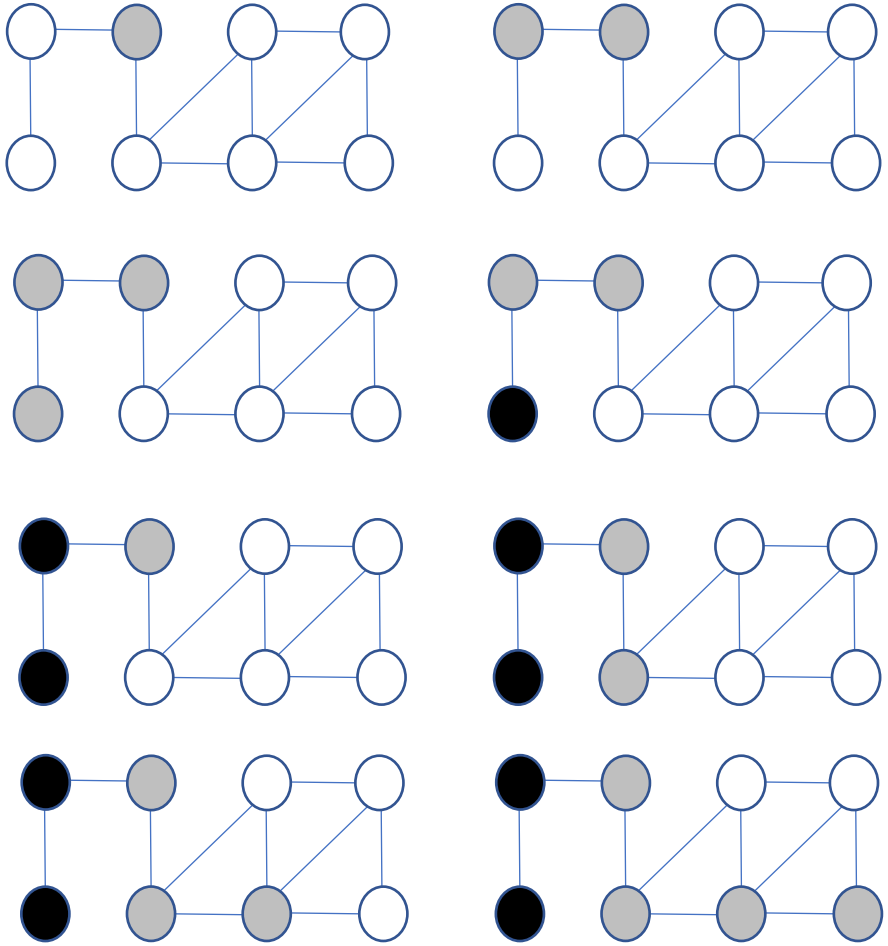
Parcours en profondeur

- Algorithme:
  - Exploration en profondeur

```
ExplorerDepuis(Graphe G, Sommet u)
  // le sommet u vient d'être découvert
  couleur[u] = gris

 // visiter les sommets adjacents
 pour chaque sommet v adjacent à u faire
    si couleur[v] = blanc
    alors
       parent[v] = u
       ExplorerDepuis(G, v) // exploration en profondeur

 // fin de traitement du sommet u
 couleur[u] = noir
```



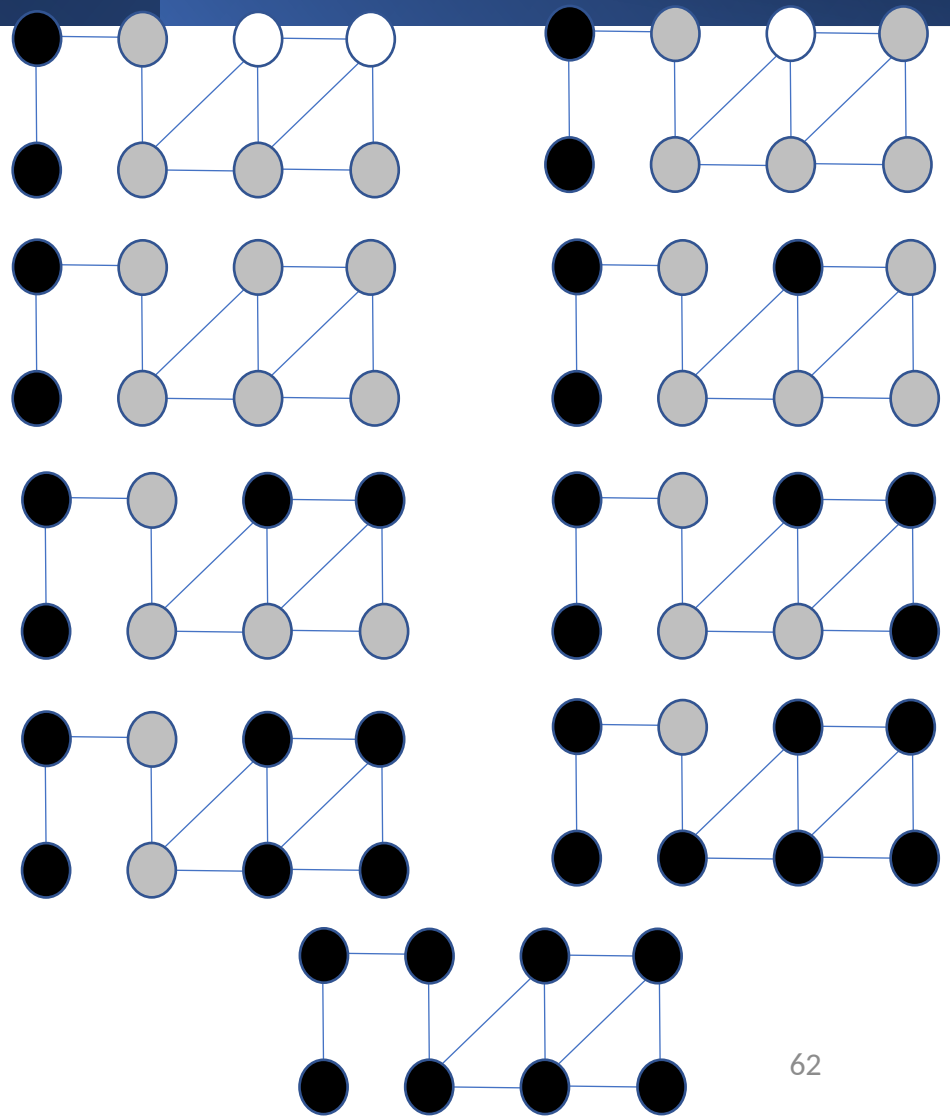
Parcours en profondeur

- Algorithme:
  - Exploration en profondeur

```
ExplorerDepuis(Graphe G, Sommet u)
  // le sommet u vient d'être découvert
  couleur[u] = gris

 // visiter les sommets adjacents
 pour chaque sommet v adjacent à u faire
    si couleur[v] = blanc
    alors
       parent[v] = u
       ExplorerDepuis(G, v) // exploration en profondeur

 // fin de traitement du sommet u
 couleur[u] = noir
```



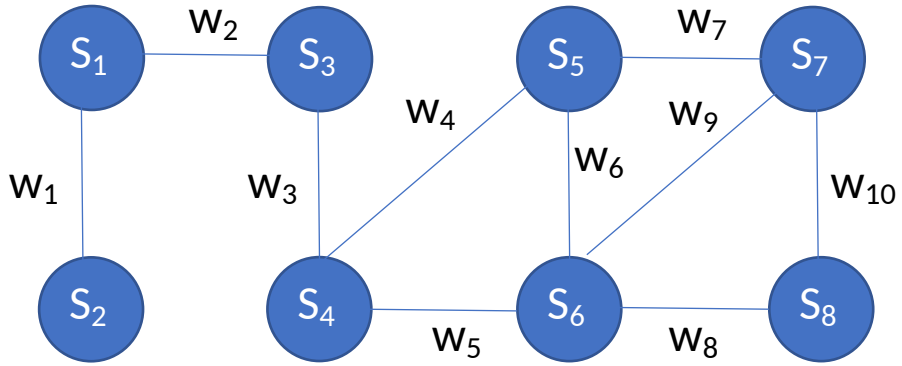
Parcours en profondeur

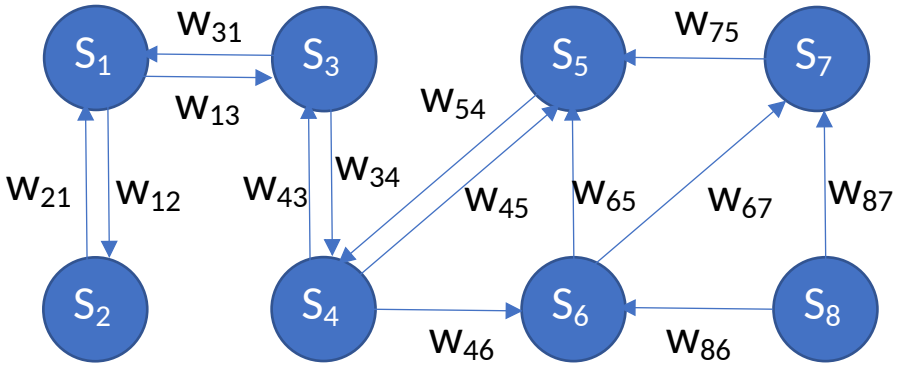
- TP

Graphes pondérés

Graphes pondérés

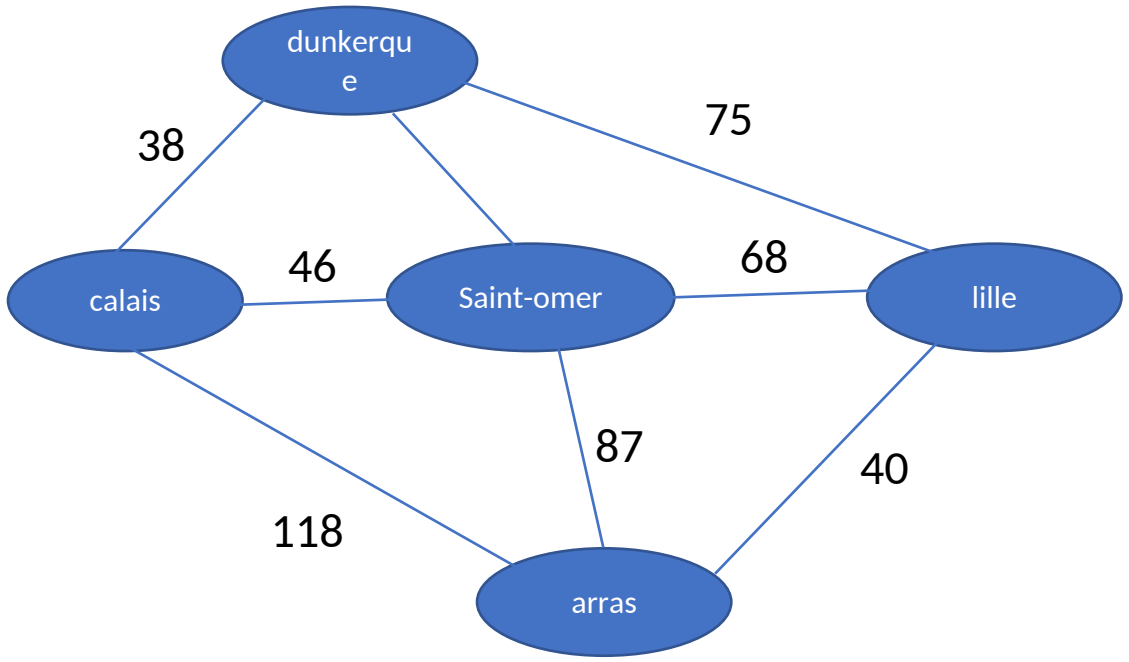
- Définition
  - Un graphe non orienté (orienté) est dit pondéré si chacune de ses arêtes (chacun de ses arcs) possède un poids réel noté w (*weight*)
    - Un graphe non pondéré peut être vu comme un graphe pondéré dont tous les poids valent 1



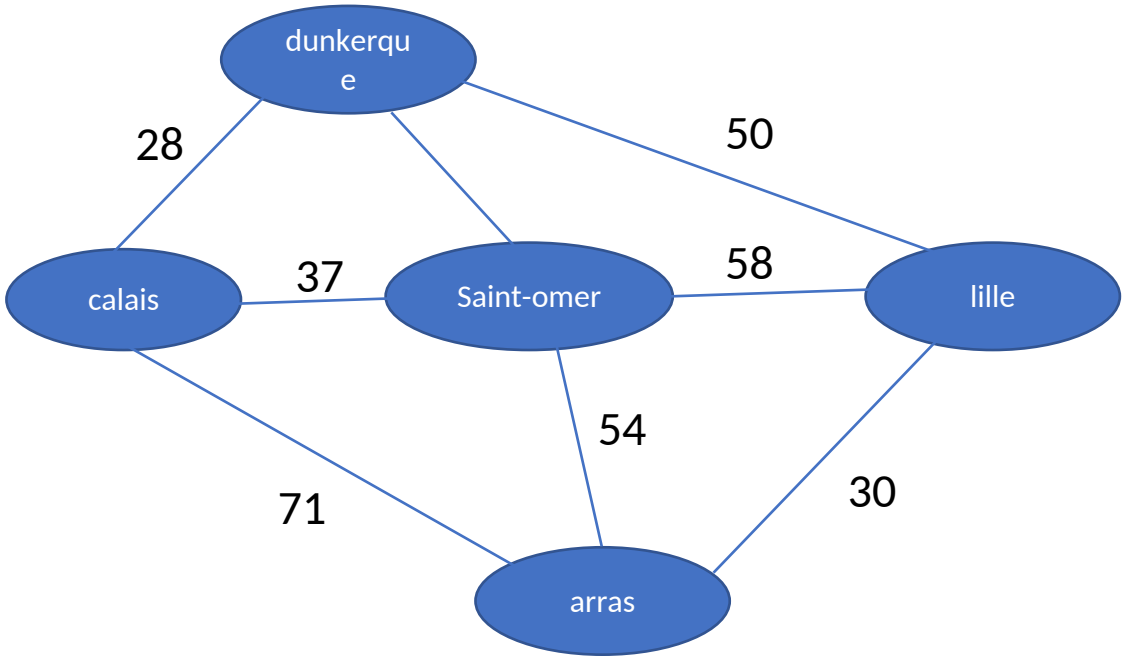


Graphes pondérés

- Exemples



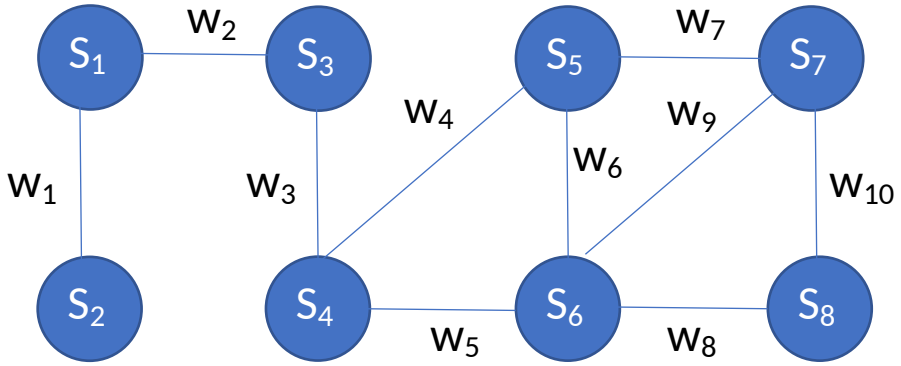
Poids = distances (en km)



Poids = temps de parcours (en mn)

Graphes pondérés

- Représentation
  - Un graphe pondéré est représenté par sa matrice d'adjacence, dans laquelles les coefficients représentent les poids des arêtes/arcs.



$$\begin{pmatrix} 0 & w_1 & w_2 & 0 & 0 & 0 & 0 & 0 \\ w_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ w_2 & 0 & 0 & w_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_3 & 0 & w_4 & w_5 & 0 & 0 \\ 0 & 0 & 0 & w_4 & 0 & w_6 & w_7 & 0 \\ 0 & 0 & 0 & w_5 & w_6 & 0 & w_9 & w_8 \\ 0 & 0 & 0 & 0 & w_7 & w_9 & 0 & w_{10} \\ 0 & 0 & 0 & 0 & w_8 & 0 & w_{10} & 0 \end{pmatrix}$$

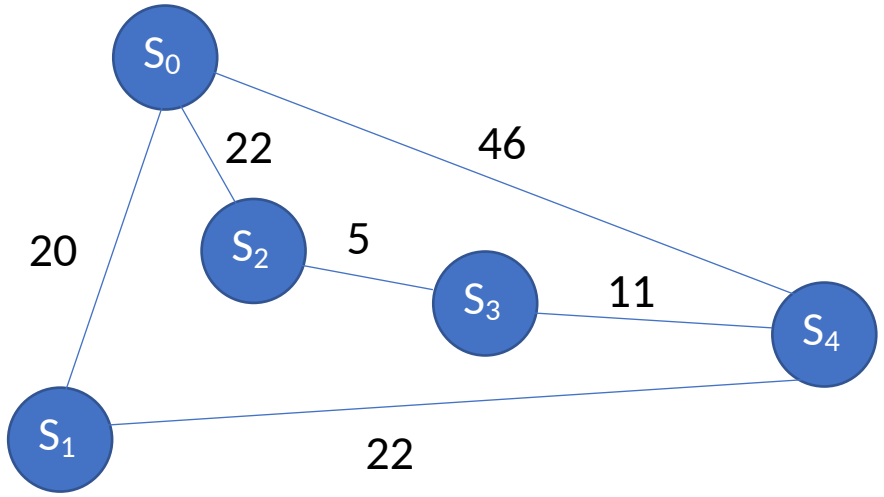
Plus court chemin dans un graphe

Plus court chemin dans un graphe

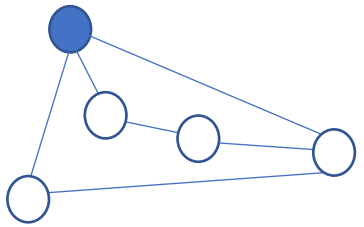
- Problématique :
  - Comment aller d'un sommet A à un sommet B en parcourant le moins de distance ?
- Graphe non ponderés:
  - Le parcours en largeur fournit les plus courts chemins
    - Distance mesurée en nombre d'arêtes/arcs
- Graphe ponderés:
  - Il faut mesurer la distance en fonction du poids de chaque arête/arc

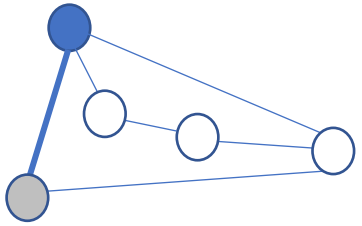
Plus court chemin dans un graphe

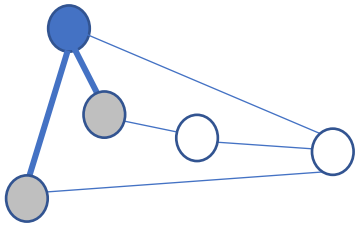
- Comment parcourir le graphe ?
  - Exemples : plus court chemin entre S_0 et S_4 ?

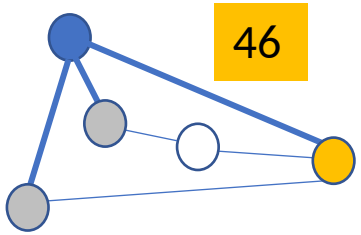


On explore les voisins directs de S_0





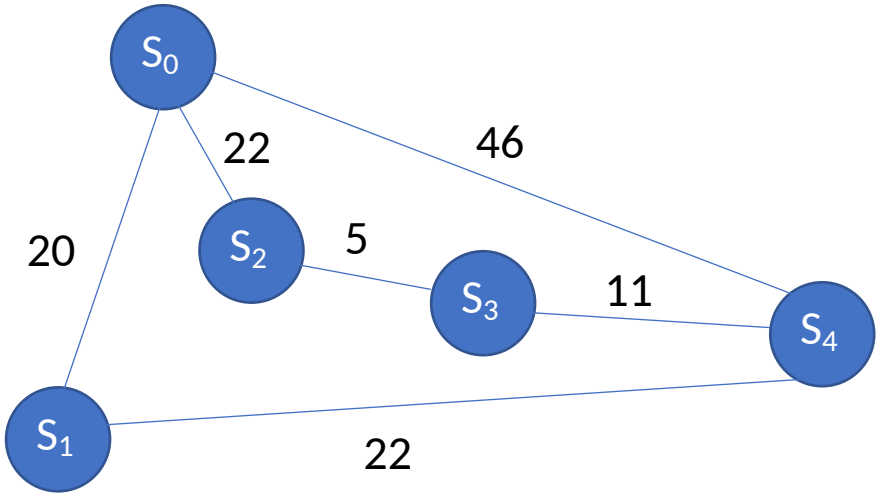




Parcours en largeur ?

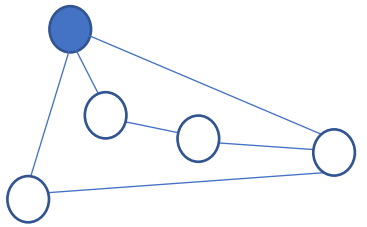
Plus court chemin dans un graphe

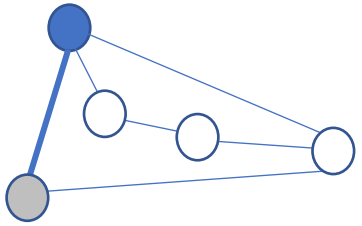
- Comment parcourir le graphe ?
  - Exemples : plus court chemin entre S_0 et S_4 ?

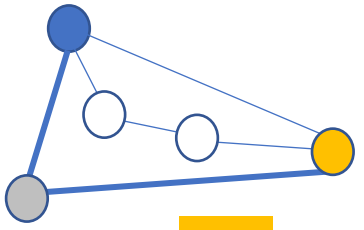


Parcours en profondeur ?

On explore chaque branche en priorité



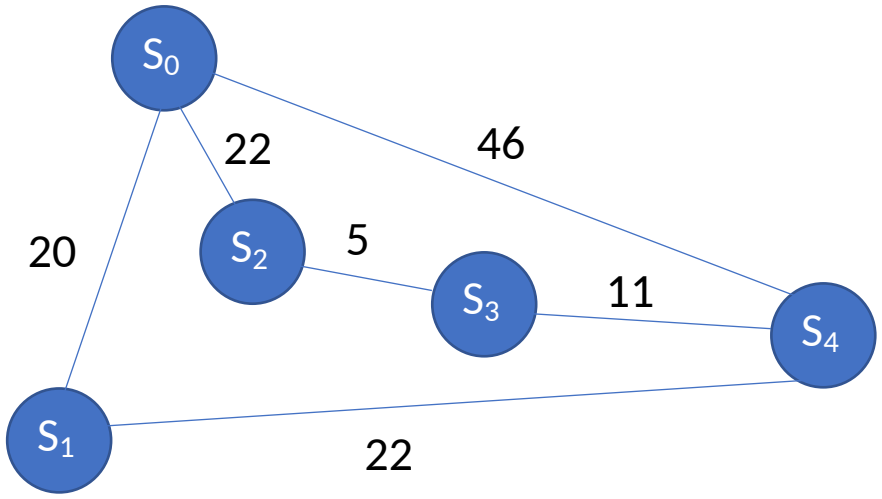


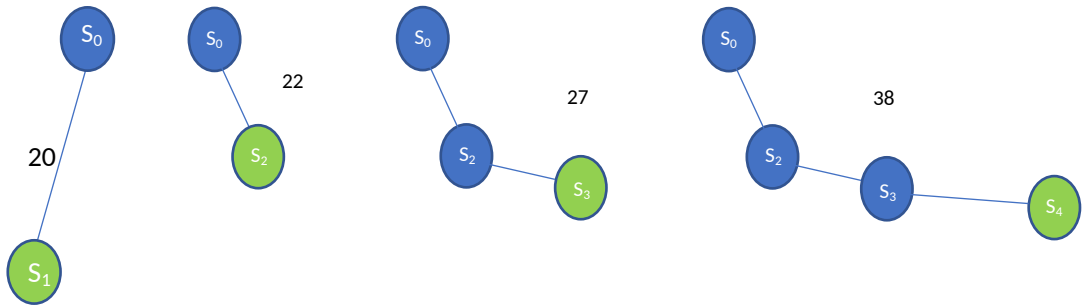


42

Plus court chemin dans un graphe

- Algorithme de Dijkstra
  - Publié en 1959
  - Recherche du plus court chemin à **origine unique** vers **chaque sommet** d'un graphe pondéré
  - Les poids sont supposés **positifs ou nuls**



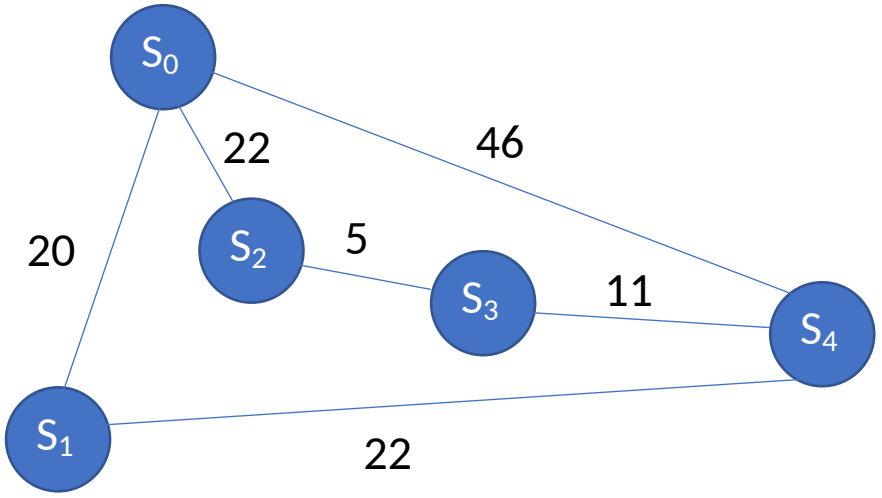


Plus court chemin dans un graphe

- Algorithme de Dijkstra : principe
  - On part du noeud de départ
  - on regarde ses voisins (*parcours en largeur*)
    - On met à jour leur distance par rapport au noeud de départ
      - Soit parce que c'est la première fois qu'on les rencontre
      - Soit parce qu'on les a déjà rencontrés et qu'il faut vérifier si le nouveau chemin trouvé n'est pas plus court
    - On sélectionne le noeud ayant la distance minimale et on réitère

Plus court chemin dans un graphe

- Algorithme de Dijkstra : principe



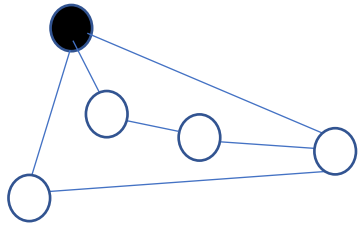
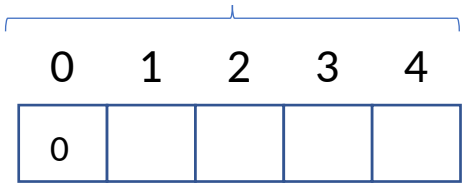


Tableau des distances minimales de chaque sommet rencontré pendant le parcours depuis le sommet de départ



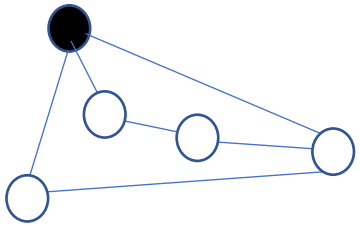
Propriété : Après traitement d'un sommet du graphe, chaque distance dans le tableau est la plus petite entre le sommet de départ et le sommet correspondant à l'entrée du tableau (pour les sommets traités)

Vraie au départ :

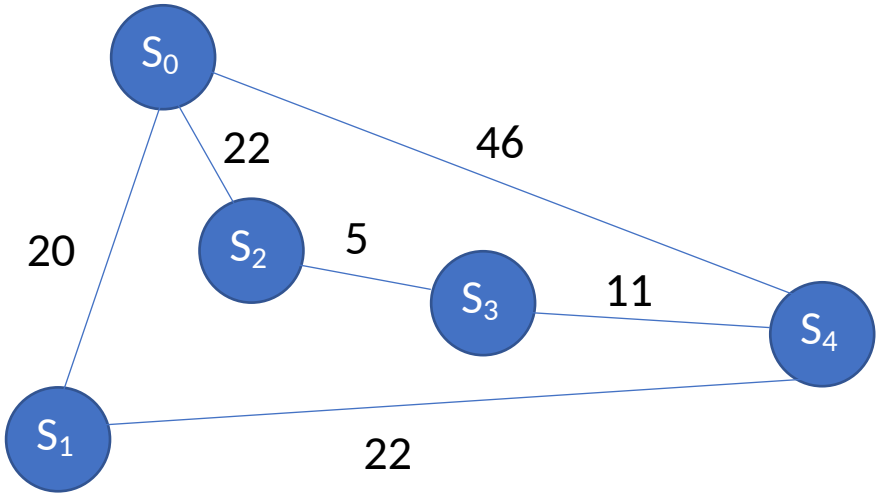
- on n'a rencontré que le sommet de départ
- Sa distance est à 0

Plus court chemin dans un graphe

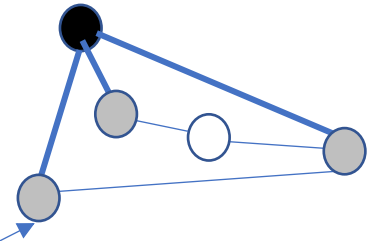
- Algorithme de Dijkstra : principe



0	1	2	3	4
0				



On met à jour la distance entre le sommet de départ et ses sommets voisins (parcours en largeur)



0	1	2	3	4
0	20	22		46

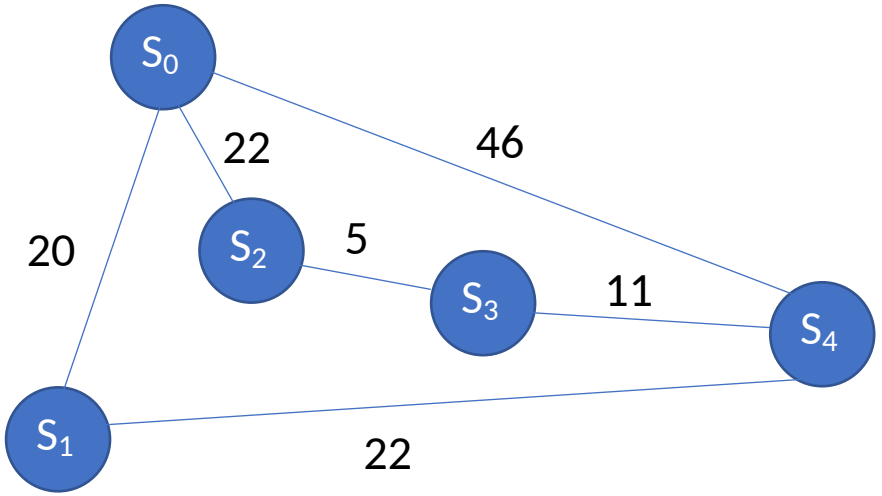
On sélectionne le sommet qui a la distance la plus faible (S_1) par rapport au sommet de départ (S_0) <u>en ne considérant que les sommets non terminés</u>

<u>Propriété encore vraie</u> :

- On n'a pas de chemins plus courts (à ce stade) que les liens directs entre le sommet de départ et les 3 sommets voisins

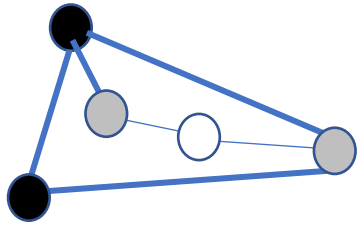
Plus court chemin dans un graphe

- Algorithme de Dijkstra : principe



On sélectionne le sommet qui a la distance la plus faible (S_2) par rapport au sommet de départ (S_0)
<u>en ne considérant que les sommets non terminés</u>

On met à jour la distance entre le sommet de départ et les sommets <u>non traités</u> voisins du sommet sélectionné (S_1)

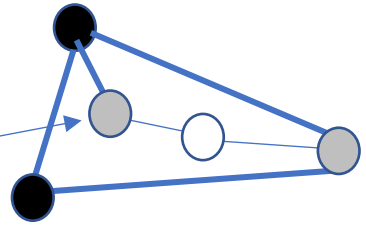


0	1	2	3	4
0	**20**	22		46

$S_0S_1S_4$ est plus court que S_0S_4 **42**

<u>Propriété encore vraie</u> (sans tenir compte de S_3) :

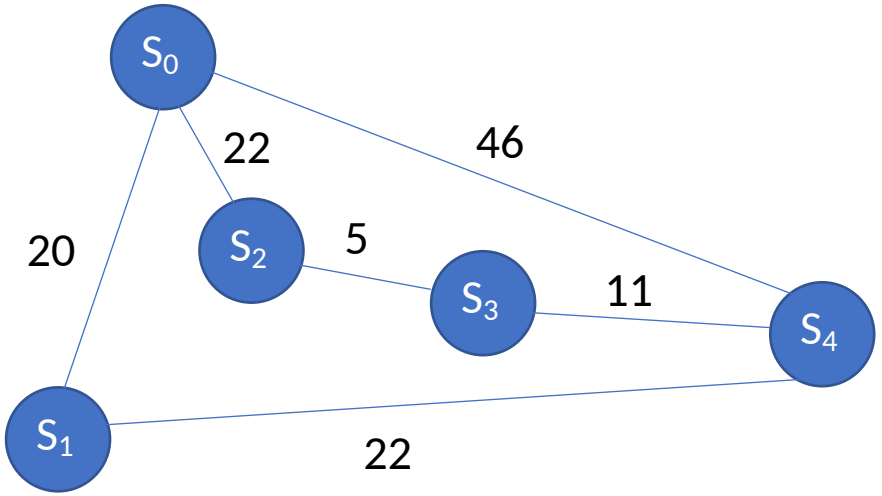
- On a le chemin le plus court entre S_0 et S_1
- On a le chemin le plus court entre S_0 et S_2
- On a le chemin le plus court entre S_0 et S_4



0	1	2	3	4
0	**20**	22		42

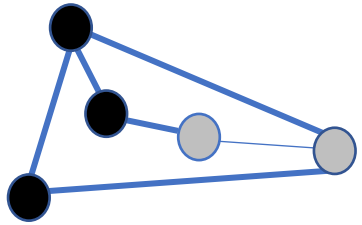
Plus court chemin dans un graphe

- Algorithme de Dijkstra : principe



On sélectionne le sommet qui a la distance la plus faible par rapport au sommet de départ
<u>en ne considérant que les sommets non terminés</u>

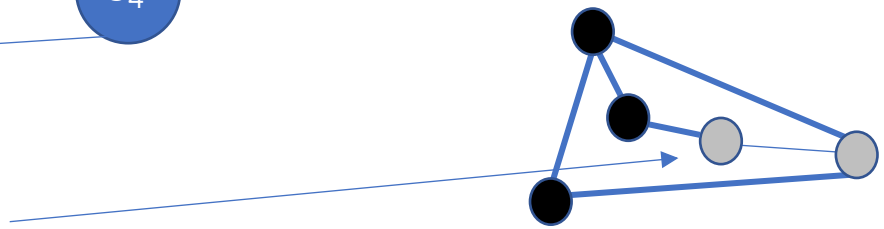
On met à jour la distance entre le sommet de départ et les sommets <u>non traités</u> voisins du sommet sélectionné (S_2)



0	1	2	3	4
0	**20**	**22**	27	42

<u>Propriété encore vraie</u> (sans tenir compte de S_3S_4) :

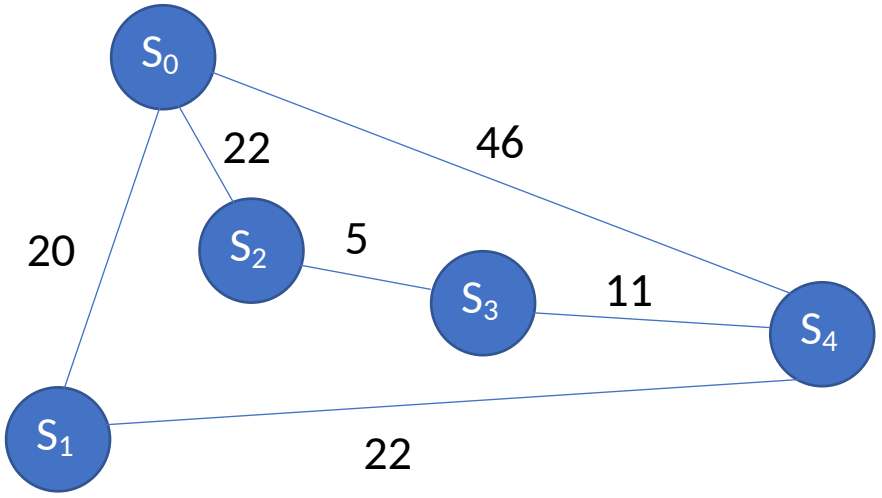
- On a le chemin le plus court entre S_0 et S_1
- On a le chemin le plus court entre S_0 et S_2
- On a le chemin le plus court entre S_0 et S_4
- On a le chemin le plus court entre S_0 et S_3



0	1	2	3	4
0	**20**	**22**	27	42

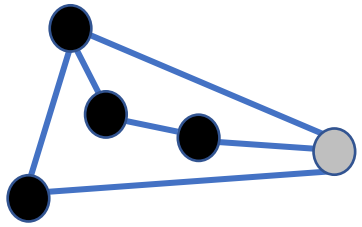
Plus court chemin dans un graphe

- Algorithme de Dijkstra : principe



On sélectionne le sommet qui a la distance la plus faible par rapport au sommet de départ
en ne considérant que les sommets non terminés

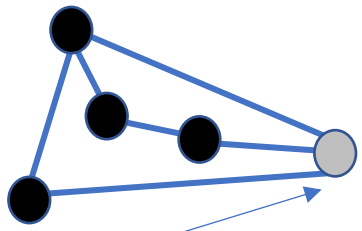
On met à jour la distance entre le sommet de départ et les sommets non traités voisins du sommet sélectionné (S_3)



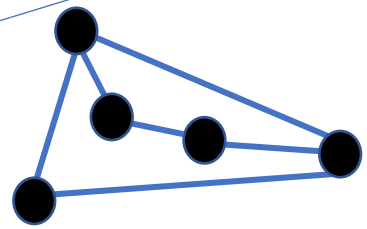
0	1	2	3	4
0	20	22	27	42

38

$S_0S_2S_3S_4$ est plus court que $S_0S_1S_4$



0	1	2	3	4
0	20	22	27	38



0	1	2	3	4
0	20	22	27	38

Plus de voisins non traités

Plus court chemin dans un graphe

- Algorithme de Dijkstra
  - initialisations

Tous les sommets sont inexplorés → initialisés en "blanc"

On ne connaît par la distance à laquelle ils se trouvent → distance initialisée à l'infini

Aucun parent connu → parent initialisé à "null"

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
couleur								
distance	∞	∞	∞	∞	∞	∞	∞	∞
parent	-	-	-	-	-	-	-	-

```
rechercheCheminMinimum(Graphe G, sommet s)
  // initialisation de tous les sommets
  pour chaque sommet u de G faire
     couleur[u] = blanc
     distance[u] = infini
     parent[u] = null

  // initialiser le sommet de départ
  distance [s] = 0
  parent[s] = null
```

Plus court chemin dans un graphe

- Algorithme:
  - La boucle d'analyse

Remarque :
- pas de couleur "gris"
- remplacée par la distance

```
rechercheCheminMinimumGraphe G, sommet s)
 // initialisations
 …

 tant que il existe des sommets blancs faire
   u = sommetBlancMinimum()
    pour chaque v adjacent à u faire
      si couleur[v] = blanc alors
          si distance[v] > distance[u] + poids(u,v) alors
             distance[v] =  distance[u] + poids(u,v)
             parent[v] = u
    fin pour

    couleur[u] = noir
fin tant que
```

On sélectionne le sommet (non traité) le plus proche du sommet de départ

On regarde tous les sommets adjacents au sommet en cours d'analyse

Si le traitement du sommet voisin n'est pas terminé (il est blanc), on vérifie que l'on a pas trouvé un chemin plus court en passant par u

L'analyse du sommet courant est terminée - il devient noir

Plus court chemin dans un graphe

- Algorithme:
  - La recherche du minimum → `sommetBlancMinimum()`
  - L'implantation dépend de la structure de données utilisée pour représenter l'état de traitement des sommets
  - Ici utilisation de tableaux
    - Couleurs
    - Distances
    - Parents

Hypothèse :
- Il reste encore au moins un sommet blanc

```
SommetBlancMinimum()
  vmin = INFINI
  pour tous les sommets u faire
    si couleur[u] = blanc alors
      si distance[u] <= vmin alors
        vmin = distance[u]
        smin = u;

  retourner smin
```

Plus court chemin dans un graphe

- Application : dérouler l'algorithme sur le graphe suivant
  - À partir du sommet S_0
  - À partir du sommet S_8