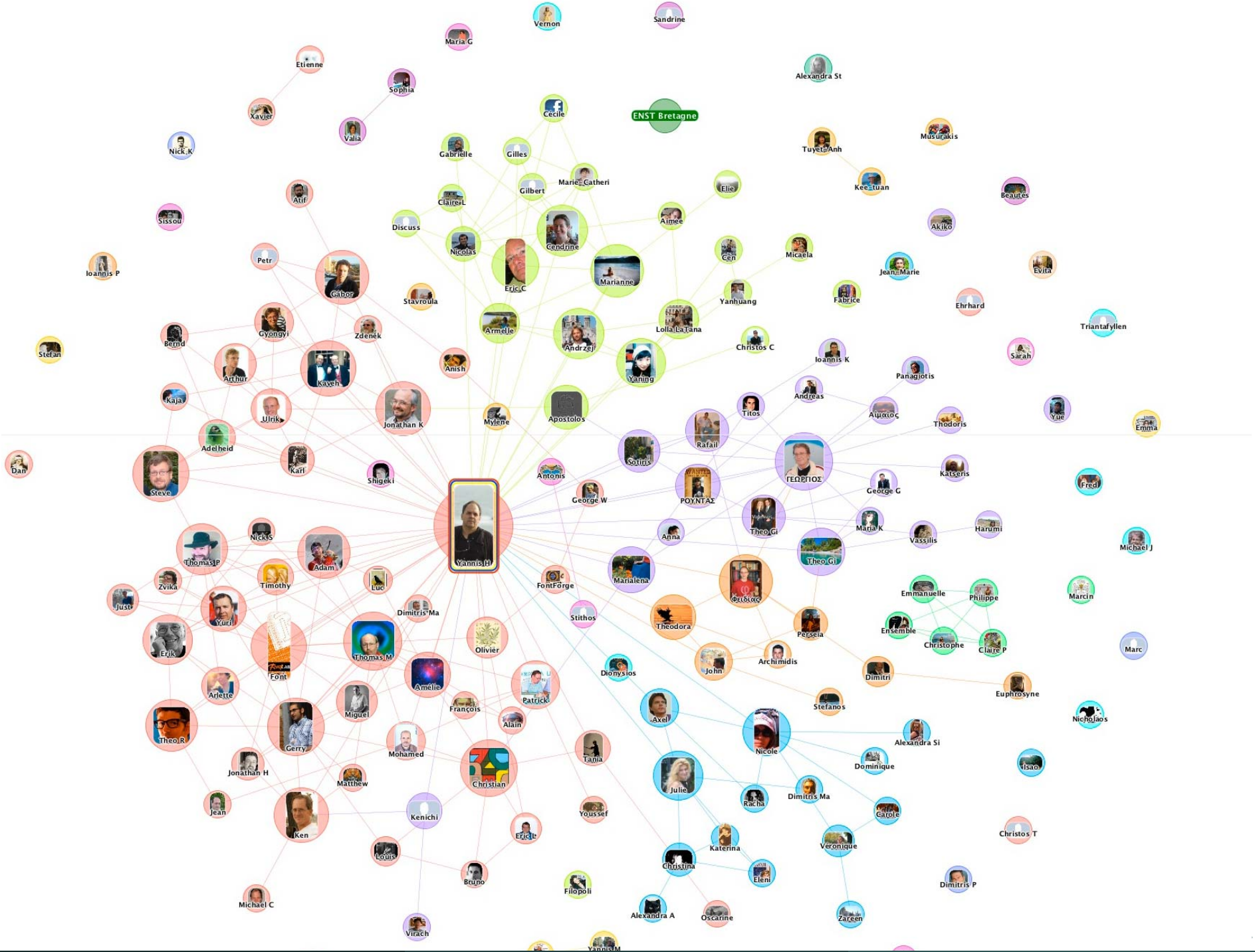
Formation LIESSE 2021
Théorie des graphes sous Python

Yannis Haralambous (IMT Atlantique)

10 mai 2021

Plan du cours


Vernon
Sandrine
Maria G
Etienne
Alexandra St
Sophia
Xavier
Cecile
ENST Bretagne
Valia
Musurakis
Nick K
Gabrielle
Gilles
Tuyet-Anh
Marie-Catheri
Gilbert
Kee-tuan
Elie
Claire L
Atif
Beautes
Sissou
Discuss
Aimee
Akiko
Nicolas
Cendrine
Cen
Micaela
Petr
Eric C
Marianne
Jean-Marie
Evita
Ioannis P
Gabor
Stavroula
Yanhuang
Fabrice
Ehrhard
Gyongyi
Armelle
Andrzej
Lolla LaTana
Triantafyllen
Zdenek
Bernd
Christos C
Stefan
Ioannis K
Sarah
Arthur
Anish
Yaning
Panagiotis
Kaveh
Andreas
Kajal
Titos
Yue
Ulrik
Jonathan K
Mylene
Apostolos
Thodoris
Emma
Adelheid
Rafail
Dan
Karl
Sotiris
Katseris
Steve
Shigeki
Antonis
ΓΕΩΡΓΙΟΣ
Fred
George W
ΡΟΥΝΤΑΣ
George G
Nicki S
Theo Gi
Maria K
Harumi
Thomas P
Anna
Vassilis
Michael J
Adam
Yannis H
Timothy
Luc
FontForge
Marialena
Emmanuelle
Philippe
Marcin
Zvika
Dimitris Ma
Stithos
Theodora
Perseia
Ensemble
Christophe
Claire P
Marc
Yuri
Erik
Thomas M
Olivier
Font
Archimidis
John
Dionysios
Dimitri
Arlette
Amélie
Patrick
Euphrosyne
Miguel
François
Stefanos
Axel
Theo R
Alain
Nicholaos
Gerry
Mohamed
Nicole
Alexandra Si
Jonathan H
Tania
Dominique
Matthew
Christian
Julie
Dimitris Ma
Jean
Racha
Carole
Ken
Kenichi
Youssef
Christos T
Eric L
Katerina
Veronique
Louis
Christina
Bruno
Eleni
Dimitris P
Filopoli
Michael C
Alexandra A
Oscarine
Zareen
Virach

partie I

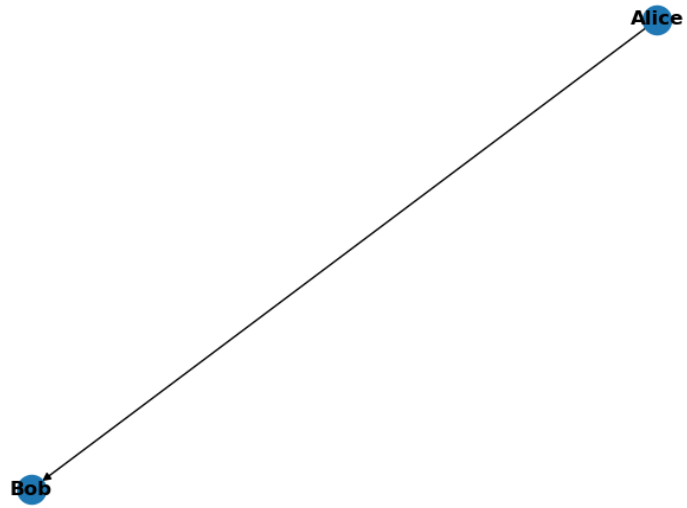
Notions fondamentales

Approche informatique

- Dit simplement, un graphe est un ensemble de sommets et d'arêtes, les arêtes étant des paires de sommets.
- Il existe des nombreux packages Python pour gérer les graphes, parmi les plus connus :
  - graph-tool
  - networkit
  - networkx
  - igraph
- Ainsi que des outils pour les représenter :
  - LaTeX avec package tikz
  - Gephi
  - graphviz
  - yEd, etc.

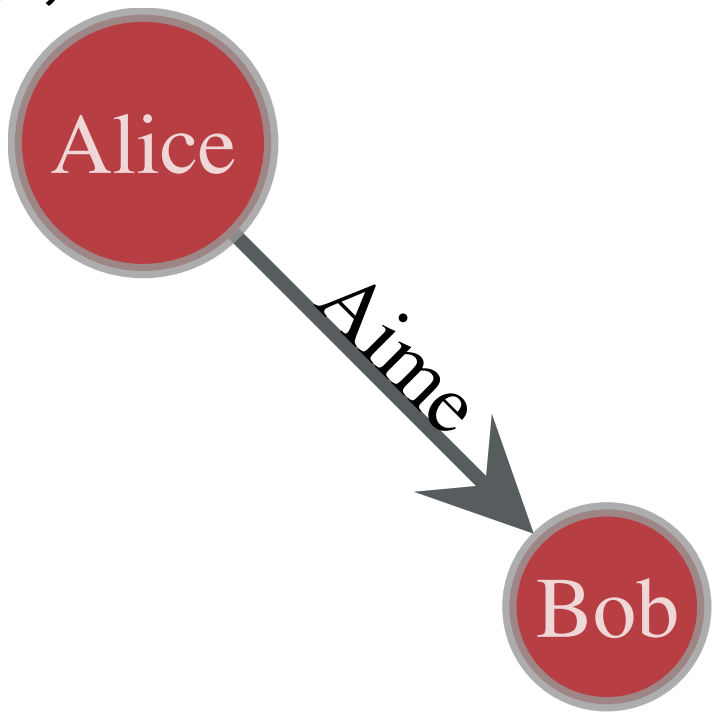
Approche informatique : networkx

```
import networkx as nx
import matplotlib.pyplot as plt
g = nx.DiGraph()
g.add_node("Alice")
g.add_node("Bob")
g.add_edge("Alice","Bob")
nx.draw(g, with_labels=True, \
    font_weight='bold')
plt.show()
```



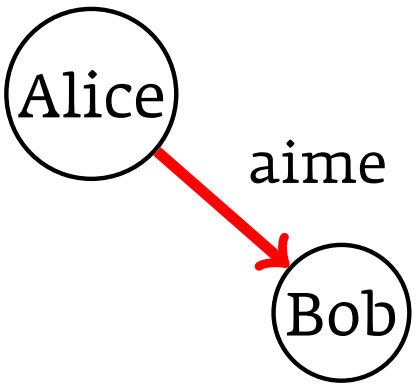
Approche informatique : networkx

```
from graph_tool.all import *
g = Graph(directed=True)
v1=g.add_vertex()
v2=g.add_vertex()
e=g.add_edge(v1,v2)
nom = g.new_vertex_property("string")
nom[v1]="Alice"
nom[v2]="Bob"
rel = g.new_edge_property("string")
rel[e]="Aime"
graph_draw(g, vertex_text=nom, \
    edge_text=rel, \
    output="test-graph-tool.pdf")
```



Approche informatique : LaTeX, tikz

```
\begin{tikzpicture}[->,x=.8cm,y=.7cm,auto,
node distance=3cm, thick,main node/.style={circle,
inner sep=1pt,fill=blue!20,draw}]
\node[main node] (1) at (0,2) [fill=white]  {Alice};
\node[main node] (2) at (2,0) [fill=white]  {Bob};
\path[every node/.style={font=\sffamily\small}]
(1) edge [draw=red, line width=2pt] node {aime} (2)
;
\end{tikzpicture}
```

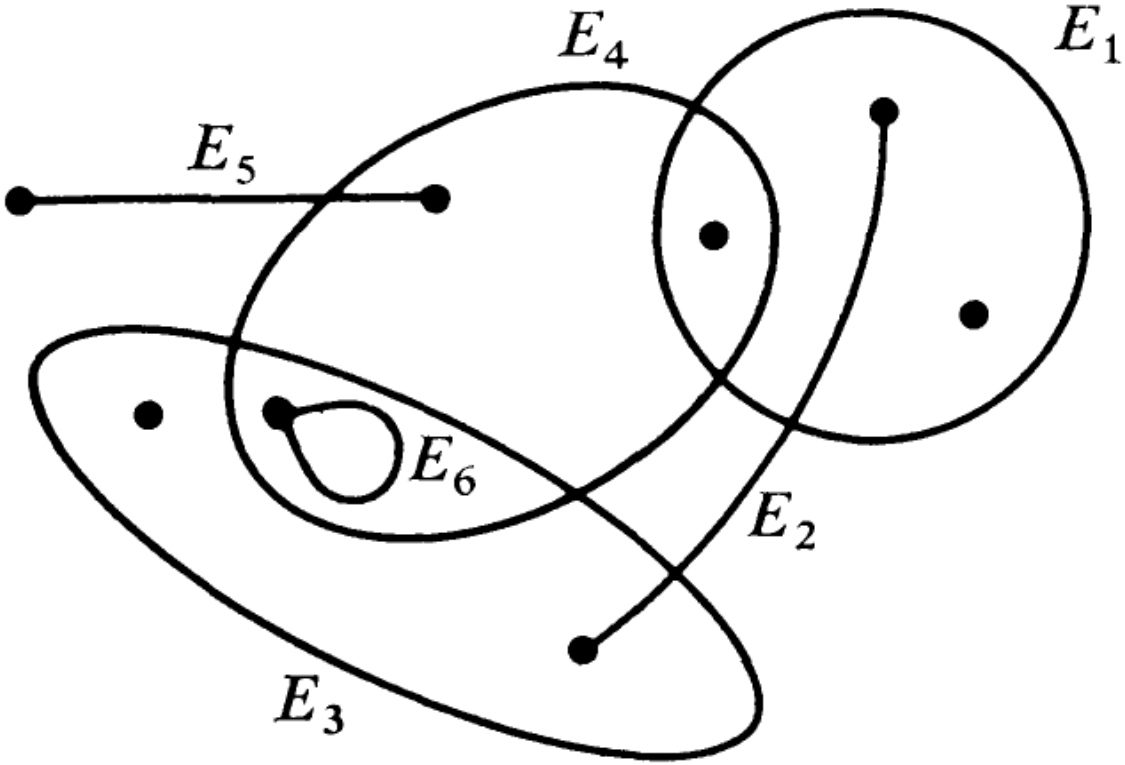


Approche mathématique : définitions

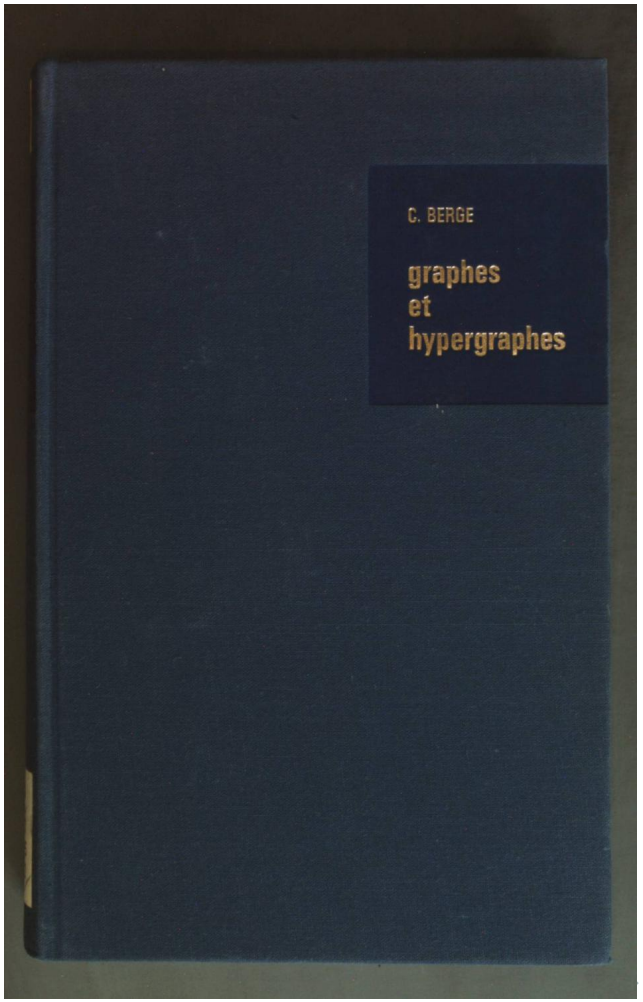
- Soit V un ensemble *quelconque*, dont nous appelons les éléments *sommets*, et E un autre ensemble, celui des *arêtes*.
- Un *hypergraphe (orienté)* est une application $g : E \to \mathcal{P}_d(V)$, où $\mathcal{P}_d(V)$ est l'*ensemble des parties* V et où chaque $X \in \mathcal{P}_d(V)$ a une partition $X_{\text{tête}} \cup X_{\text{queue}}$.
- Un *hypergraphe non-orienté* est une application $g : E \to \mathcal{P}(V)$, où $\mathcal{P}(V)$ est l'*ensemble des parties* V.

Exemple d'hypergraphe

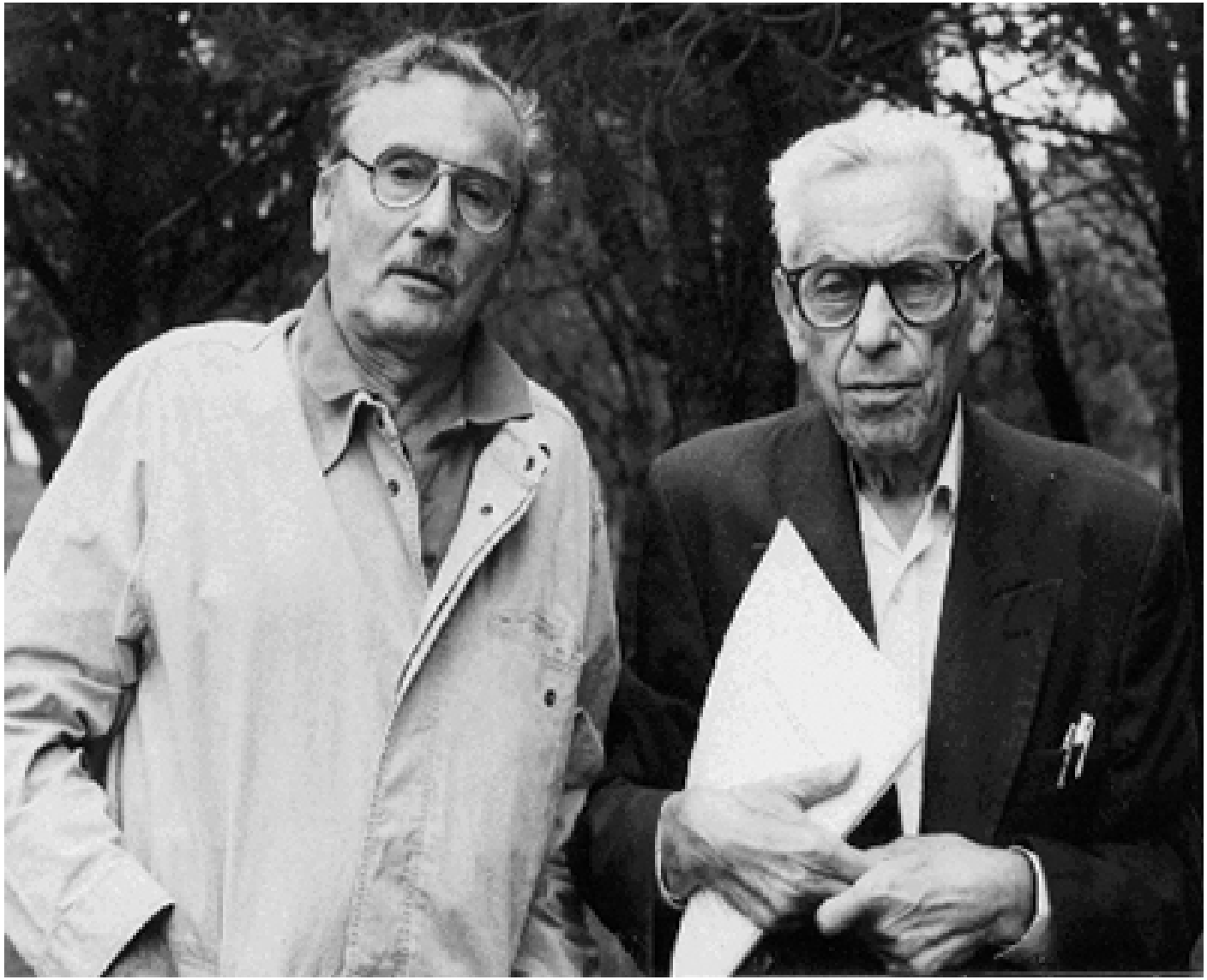
- Exemple d'hypergraphe :



tiré de Claude Berge, *Graphes et hypergraphes*, Dunod, 1970, p. 375. (Trois types de notations, pour $|\#(E_i)| = 1, 2$ ou $|\#(E_i)| \geq 3$.)



Claude Berge



Claude Berge (1926–2002) et Paul Erdős (1913–1996)

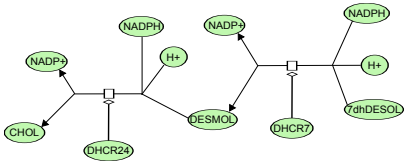
Hypergraphs: Algorithms, Implementations, and Applications

Brendan Avent[1], **Anna Ritz**[1], **and T. M. Murali** [1,2]

[1]Department of Computer Science,Virginia Tech [2]ICTAS Center for Systems Biology of Engineered Tissues,Virginia Tech

Motivation

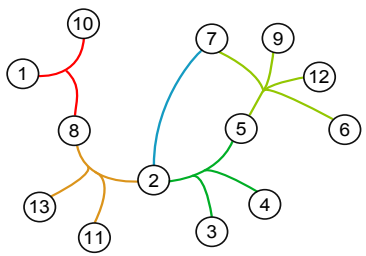
- Networks are ubiquitous in modern science, engineering, and the humanities
- Networks modeled with graphs can only represent pairwise interactions, not higher-order relationships



- Many software libraries for graphs exist: NetworkX, Boost, JUNG, etc.
- **Challenge: no libraries for hypergraphs exist that have both data structures and algorithms. This leaves a lot of work to do!**

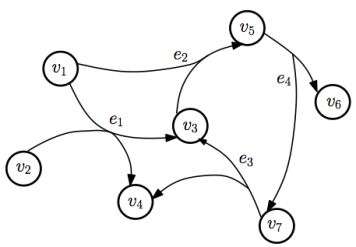
What Are Hypergraphs?

Undirected Hypergraphs



Undirected hyperedges connect groups of nodes.

Directed Hypergraphs



Directed hyperedges connect "tail" groups of nodes to "head" groups of nodes.

halp: Hypergraph Algorithms Package

Features

Open Source: Thoroughly tested Python package publicly-available on GitHub [1]

Data Structures: Directed and undirected hypergraph data structures to easily model complex networks

Usable Algorithms: Implementations of important and canonical hypergraph algorithms

Utilities: Quick extraction of hypergraph properties and statistics + conversion to other formats/structures

Algorithms

The algorithms currently implemented in *halp* span:

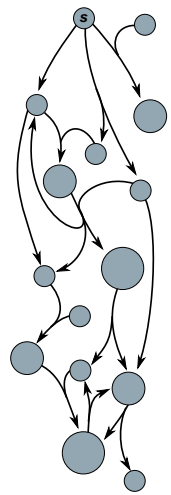
- **Connectivity** [2]
- **Hyperpaths** [2] [3] [6]
- **Hypertrees** [2]
- **Random Walks and Partitioning** [4] [5]

These algorithms are illustrated to the right:

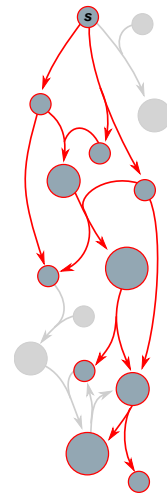
- *B-Visit* algorithm, for computing *B-connectivity*
- *s-t B-hyperpath* algorithm, for computing a minimal B-connected hyperpath

Example Algorithms

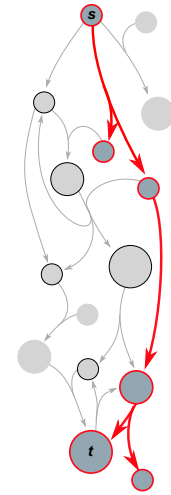
Initial Hypergraph



B-Visit from s



s-t B-Hyperpath



New Algorithm with Application to Biological Networks

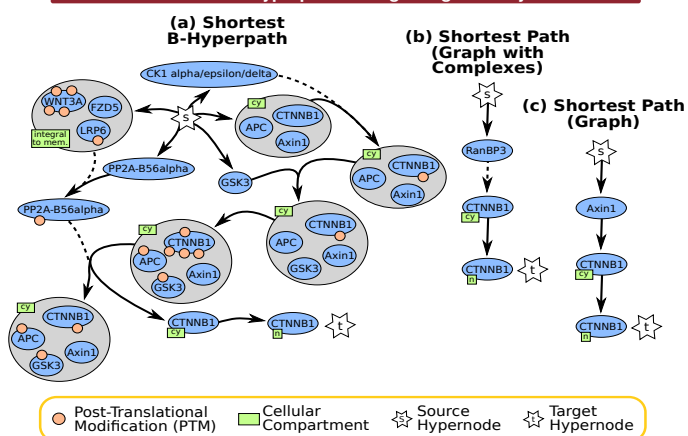
Cellular Signaling Pathways

- Cells respond to environmental signals through "signaling pathways"
- Many types of reactions can occur along these paths
- Graphs cannot model these interactions adequately...but hypergraphs can!

Shortest B-Hyperpath Algorithm

- The biological interpretation of a B-hyperpath is a path from node s to node t that contains all intermediate reactants and products needed to reach t from s
- We developed an algorithm using mixed integer linear programming to find the *shortest* acyclic B-hyperpath of all possible B-hyperpaths in a directed hypergraph [6]

Shortest B-Hyperpaths in Signaling Pathways



(a) Shortest B-hyperpath in the Wnt signaling pathway when represented as a directed hypergraph. Nodes represent complexes (in grey) and standalone proteins (in blue outside of any complexes). (b, c) Shortest paths in the Wnt signaling pathway when represented as a graph with complexes (b) or as a graph (c). We see that the hyperpath is much more informative than the path in the graphs.

References

1. B. Avent, A. Ritz, T. Murali (2014). halp: hypergraph algorithms package [Online]. *Available: https://tmmurali.github.io/halp*
2. G. Ausiello, R. Giaccio, G. F. Italiano, and U. Nanni (1992). Optimal traversal of directed hypergraphs. *Tech. Rep.*
3. Nielsen, L. R., Andersen, K. A., and Pretolani, D. (2005). Finding the k shortest hyperpaths. *Computers and Operations Research.*
4. A. Ducournau, A. Bretto (2014). Random walks in directed hypergraphs and application to semi-supervised image segmentation. *Computer Vision and Image Understanding.*
5. D. Zhou, J. Huang, B. Scholkopf (2006). Learning with hypergraphs: Clustering, classification, and embedding. *Advances in neural information processing systems.*
6. A. Ritz, B. Avent, T. Murali (2015). Pathway analysis with signaling hypergraphs. *IEEE Transactions on Computational Biology and Bioinformatics, under review*

Funding

R01-GM095955 DBI01-62380 EPA-RD-83499801

Brendan Avent - Dept. of Computer Science - Virginia Tech Mail: bavent@vt.edu WWW: https://tmmurali.github.io/halp/

Moins général que les hypergraphes

- Un *multigraphe orienté* est une application (pas forcément injective) $g : E \to V \times V$.
- Un *multigraphe non-orienté* est une application (pas forcément injective) $g : E \to V^{(\leq 2)}$, où

$$V^{(\leq 2)} = \{\{x_1, x_2\} \mid x_1, x_2 \in V\} \cup \{\{x\} \mid x \in V\}.$$

- Les multigraphes sont implémentés dans networkx (classe spécifique) :

```
import networkx as nx
G = nx.MultiGraph()
keys = G.add_edges_from([(4, 5, {"route": 28}), \
    (4, 5, {"route": 37})])
```

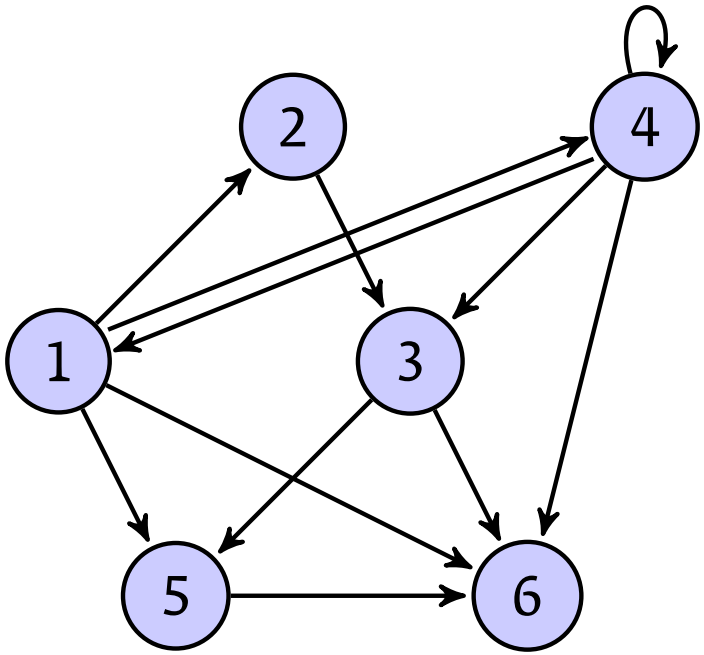
- La classe `Graph` de graph-tool couvre les multigraphes par défaut.

Le cas ordinaire

- Un *graphe orienté* est une application injective $g : E \to V \times V$.
- Nous allons noter x_1x_2 l'arête (x_1, x_2), x_1 est le *prédécesseur* et x_2 le *successeur*.
- Un *graphe non-orienté* est une application injective $g : E \to V^{(\leq 2)}$.
- Nous allons noter par x_1x_2 l'arête $\{x_1, x_2\}$, x_1 et x_2 sont alors appelés *extrémités*.
- Un graphe est dit *simple* si $\mathrm{Im}(g) \subset V^{(2)}$ (autrement dit : il n'a pas de « lacets »).
- Un graphe (orienté ou non) (G, ρ) est dit *ordonné* quand il existe un ordre ρ des sommets (c'est-à-dire une application injective $\rho : V \to \mathbb{N}$).

Matrice d'adjacence

- Une *matrice d'adjacence* A est une matrice à valeurs $a_{i,j} = n$, s'il existe, dans le graphe, n arêtes allant de x_i vers x_j. (i-ième ligne, j-ième colonne.)



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

Matrice d'adjacence sous graph-tool

```
from graph_tool.all import *
g = Graph(directed=True)
vlist = g.add_vertex(6)
for (a,b) in [(1,2),(1,4),(1,5),(1,6),(2,3),(3,5),\
    (3,6),(4,1),(4,3),(4,4),(4,6),(5,6)]:
    g.add_edge(g.vertex(a-1),g.vertex(b-1))
print(adjacency(g).toarray())
```

donne

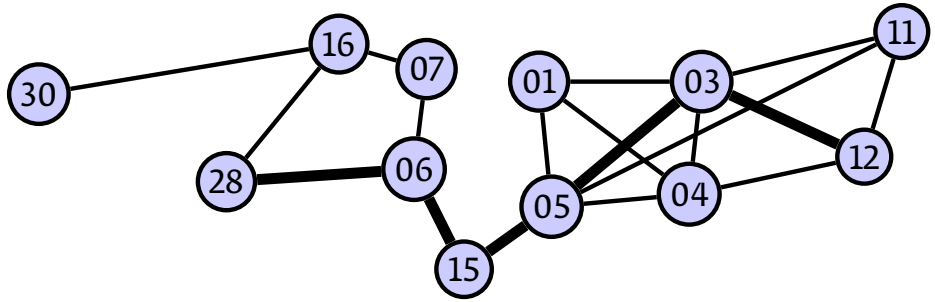
```
[[0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 1. 0. 0.]
 [1. 0. 0. 1. 0. 0.]
 [1. 0. 1. 0. 0. 0.]
 [1. 0. 1. 1. 1. 0.]]
```

Matrice d'adjacence

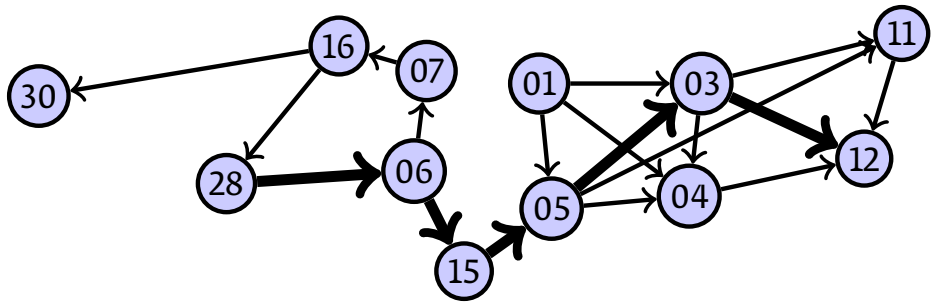
- Si le graphe est simple, la diagonale de la matrice d'adjacence est nulle et $a_{i,j} \in \mathbb{Z}/2\mathbb{Z}$, pour tout i, j.
- Si le graphe est non orienté, la matrice d'adjacence est symétrique ($a_{i,j} = a_{j,i}$, pour tout i, j).
- Un *graphe pondéré* est un graphe dont les arêtes ou les sommets (ou les deux) sont munis de *poids* (de toute sorte).

Chemins

- Dans un graphe non orienté, un *chemin* est une suite d'arêtes e_i consécutives (une extrémité de e_{i-1} est aussi extrémité de e_i, pour tout i).



- Dans un graphe orienté, un *chemin* est une suite d'arêtes orientées e_i consécutives (le successeur de e_{i-1} est prédécesseur de e_i, pour tout i).



- En l'absence de pondération, la longueur d'un chemin est le nombre d'arêtes qu'il contient.

Graphes orientés acycliques

- Dans un graphe orienté, un *cycle* est un chemin d'arêtes $x_0x_1, x_1x_2, \ldots x_{n-1}x_n, x_nx_0$.
- Un graphe orienté G est dit *acyclique* s'il ne contient aucun cycle.
- Un sommet est appelé *source* s'il n'a pas de prédécesseur.
- Un sommet est appelé *puits* s'il n'a pas de successeur.

Proposition

Un graphe orienté acyclique possède toujours au moins une source et un puits.

Tri topologique

- On appelle *tri topologique* d'un graphe orienté $G = (X, V)$ une bijection $f : X \to \{1, \ldots, |G|\}$ telle que si x est prédécesseur de y, alors $f(x) < f(y)$.

Proposition

Un graphe est orienté acyclique ss'il possède un tri topologique.

- La matrice d'adjacence d'un graphe orienté acyclique devient *strictement triangulaire* (triangulaire avec diagonale nulle) si l'on ordonne les sommets d'après leur tri topologique.

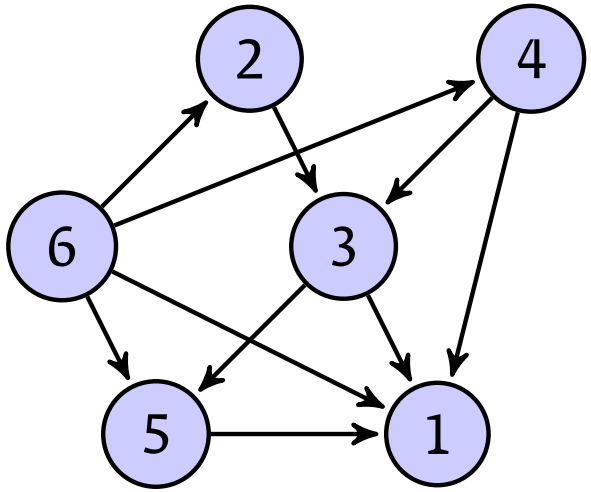
Proposition

Les valeurs propres de la matrice d'adjacence d'un graphe acyclique sont toutes nulles (= la matrice est *nilpotente*).

Nous verrons plus loin la réciproque de cette proposition.

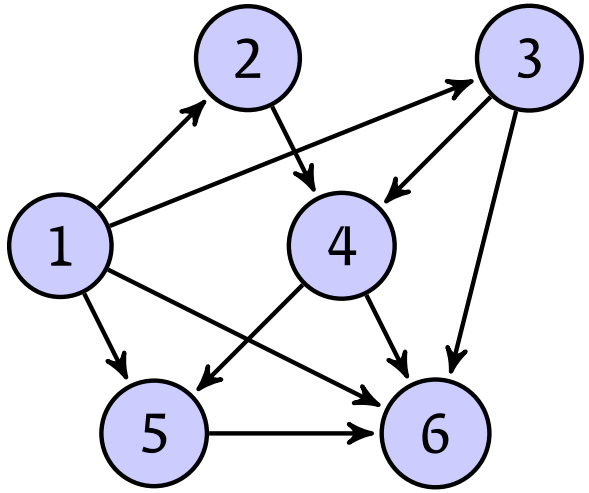
Exemple de tri topologique

Avant le tri :



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

Après le tri :

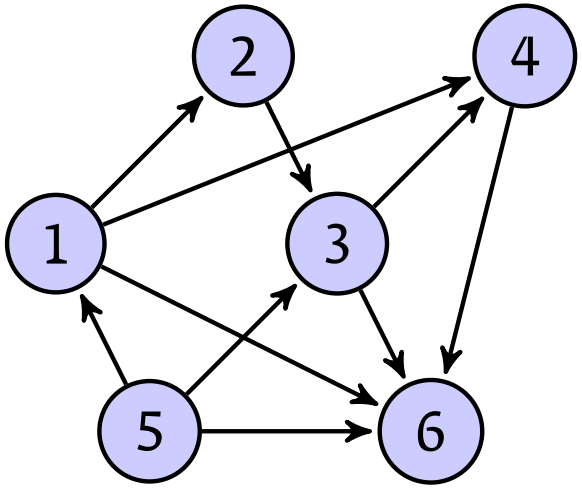


$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Lien entre cycles et valeurs propres

Exemple :

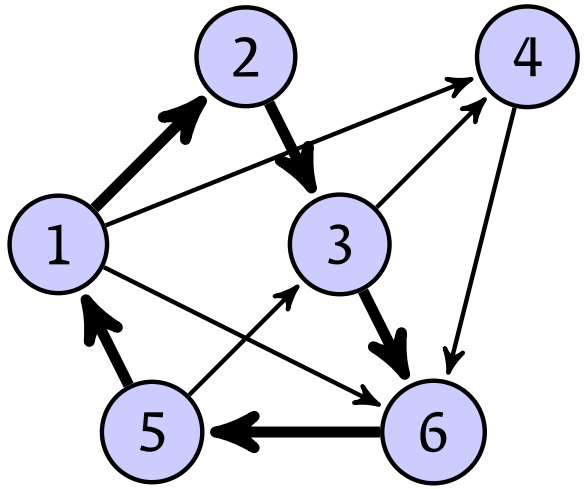
Sans cycle :



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

$vp = 0, 0, 0, 0, 0, 0$

Avec cycle :



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

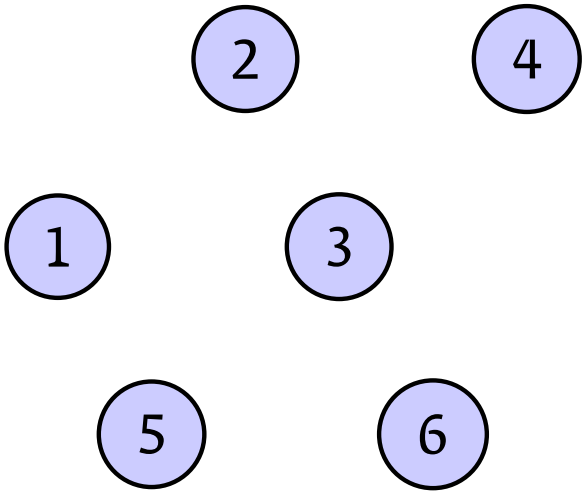
$vp = 1.5, -0.59 \pm 1.07i, 0.18 \pm 0.78i, -0.7$

Propriétés mésurables

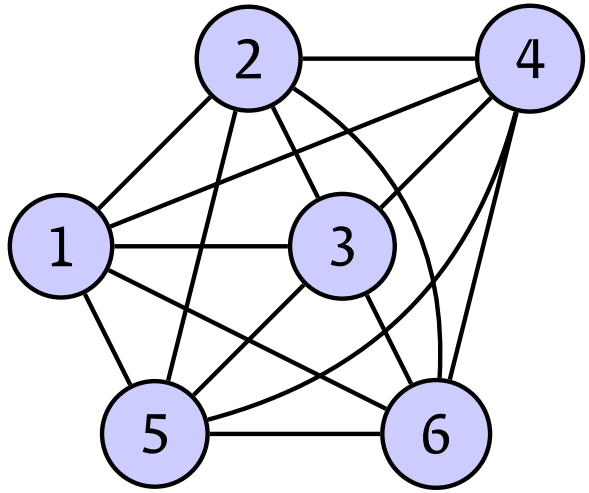
- Notons $|.|$ l'*ordre* (nb de sommets) et $\|.\|$ la *taille* (nb d'arêtes).
- Un graphe G est *discret* si $\|G\|=0$.
- Le *degré* d'un sommet est le nombre d'arêtes qui lui sont adjacentes.
- Si tous les sommets ont le même degré k, on dira que G est *k-régulier*.

Discrétion, régularité, complétude

- Un graphe sans arête est *discret*.
- Un graphe *complet* possède toutes les arêtes possibles, c'est-à-dire : $\|G\| = \frac{1}{2}|G|(|G|-1)$.
- On écrit K_n le graphe complet d'ordre n, il est $(n-1)$-régulier.



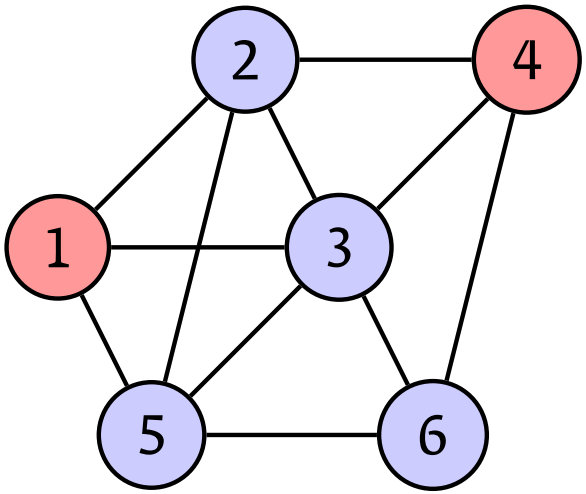
Graphe discret



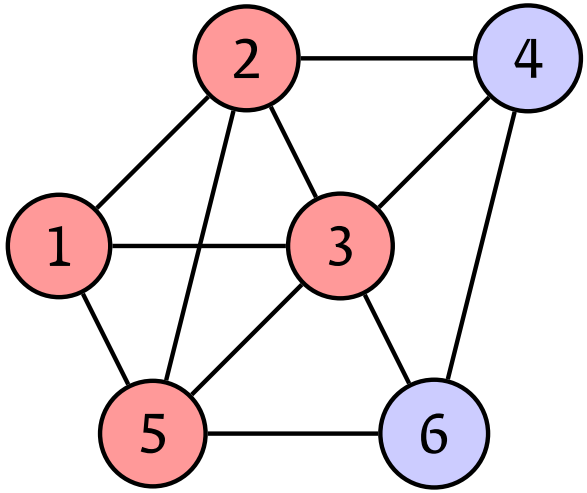
Graphe complet (K_6)

Cliques et stables

- Un graphe peut être complet ou discret, en partie seulement.
- Une *clique* est un ensemble de sommets qui induit un sous-graphe complet (un maximum d'arêtes).
- Un *stable* est un ensemble de sommets qui induit un sous-graphe discret (aucune arête).



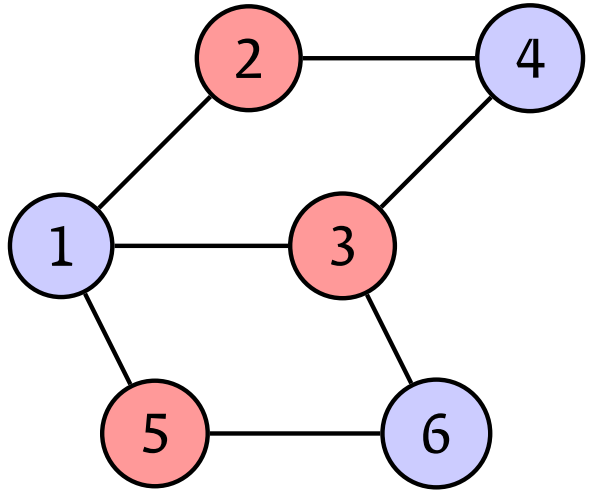
Stable (en rouge)



Clique (en rouge)

Graphes bipartis

- G est *biparti* si $X = X_1 \cup X_2$ avec X_1, X_2 stables.

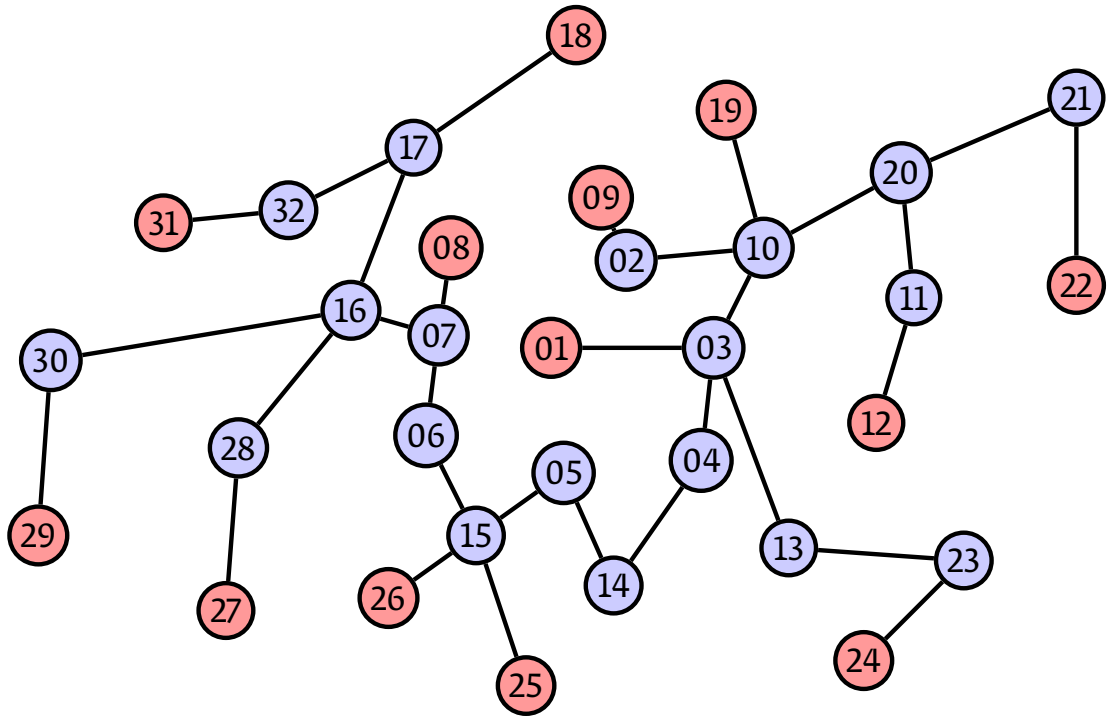


Proposition

Un graphe est biparti ss'il ne possède aucun cycle de longueur impaire.

Arbres : définitions

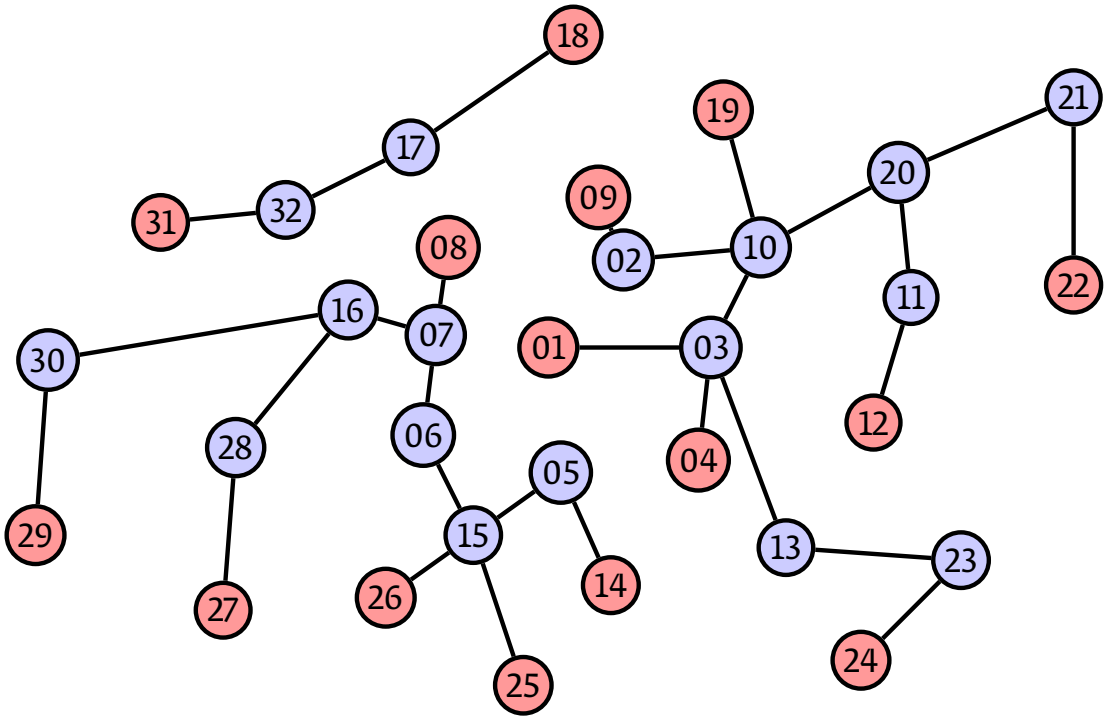
- Un graphe est *connexe* si on peut joindre n'importe quelle paire de sommets par un *chemin*.
- Un *arbre* est un graphe connexe, sans cycle.
- Les sommets extérieurs (de degré 1) sont appelés *feuilles*, les autres, *nœuds*.



(Les feuilles en rouge)

Forêts

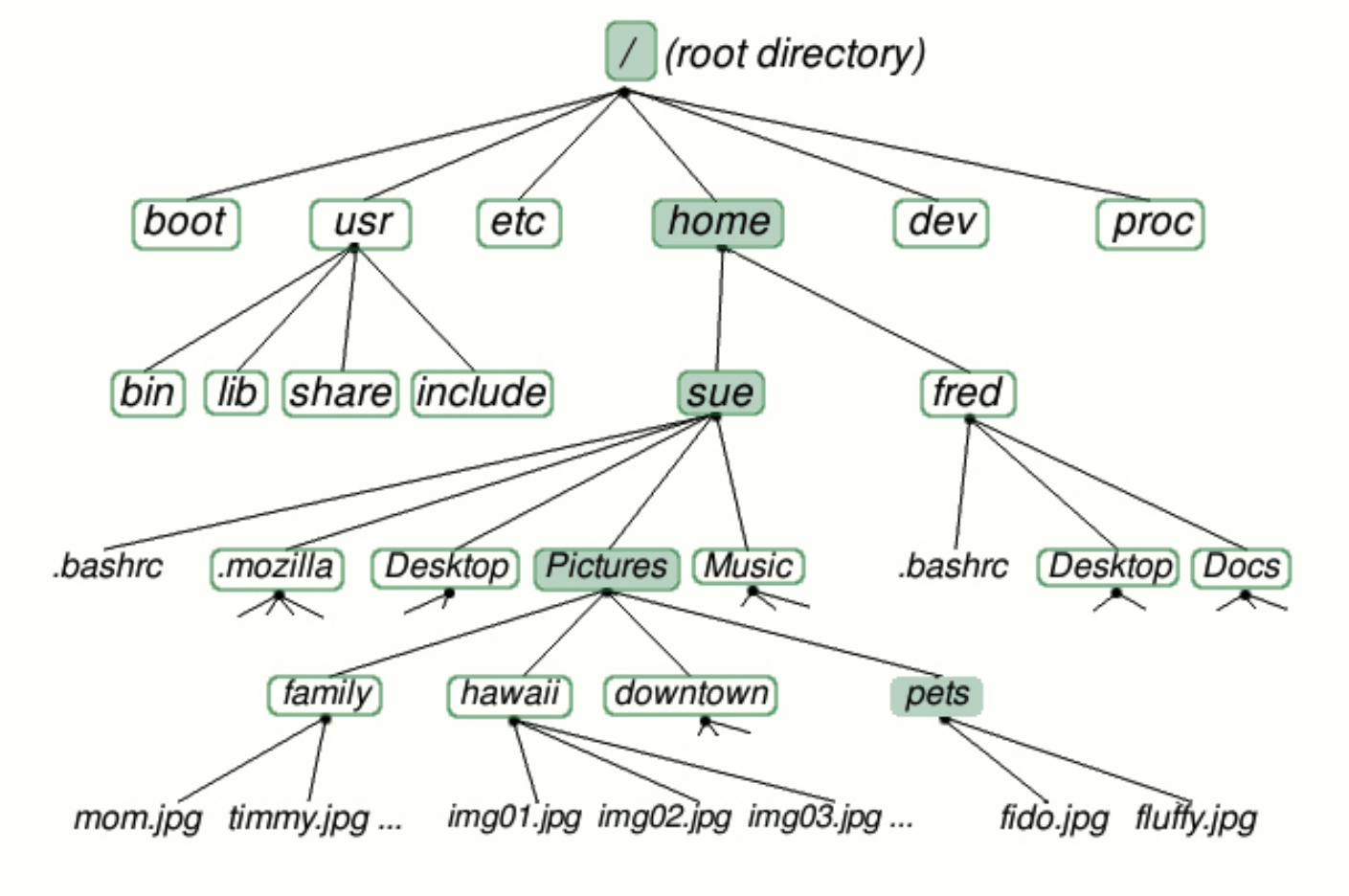
- Une *composante connexe* d'un graphe non-orienté est un sous-graphe connexe.
- Une *forêt* est un graphe dont les composantes connexes sont des arbres.



(Les feuilles en rouge)

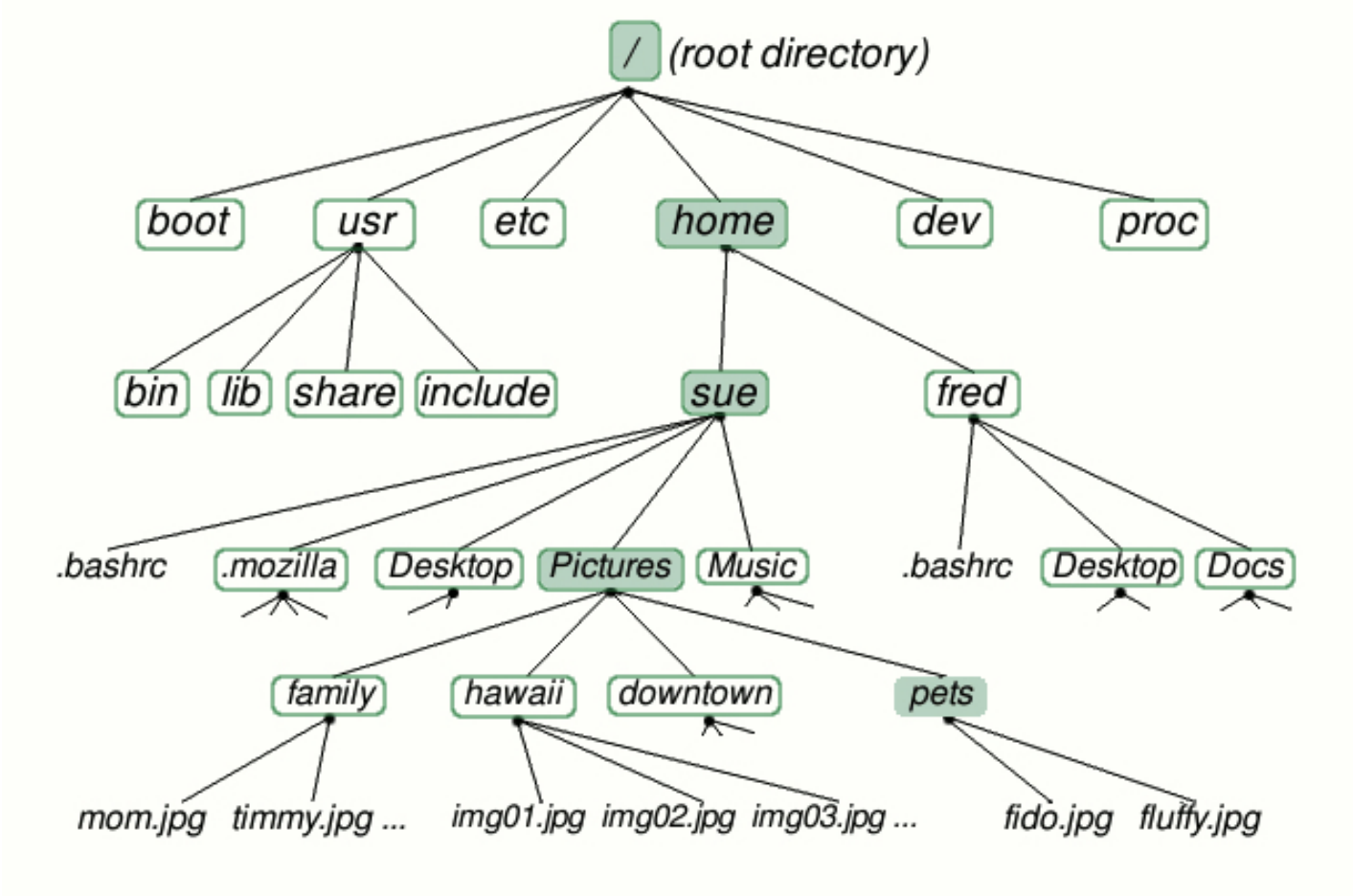
Colle, à poser aux hackers

- Est-ce qu'un système de fichiers Unix est un arbre ? Est-il orienté ?



Colle, à poser aux hackers

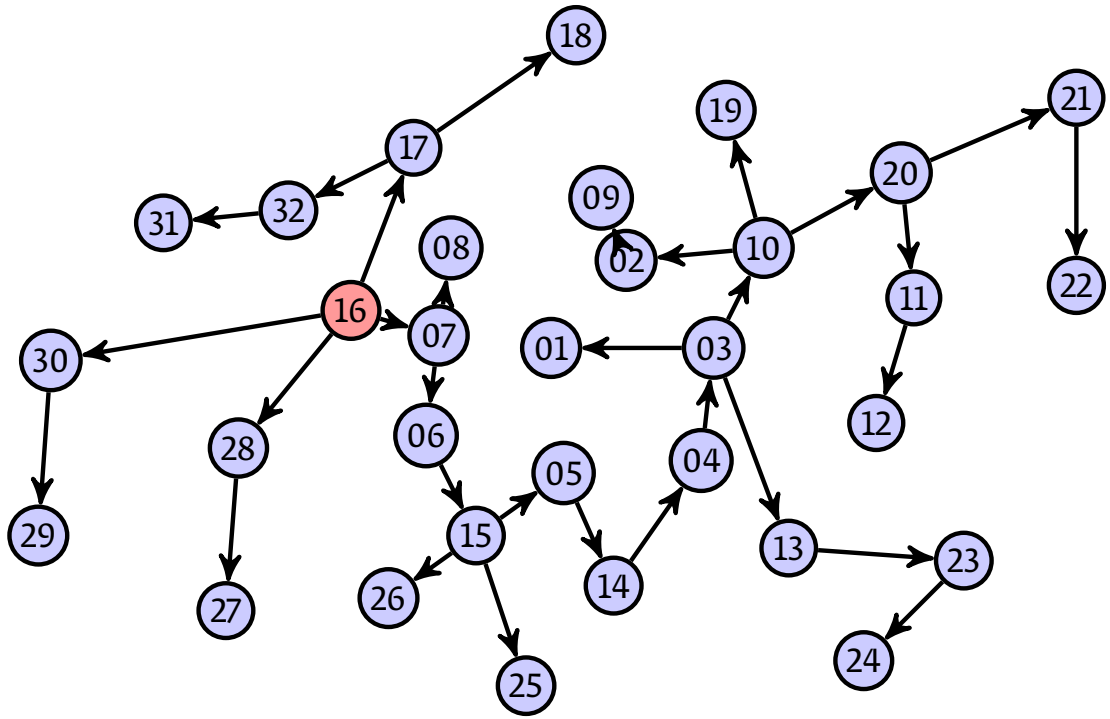
- Est-ce qu'un système de fichiers Unix est un arbre ? Est-il orienté ?



- Pas toujours ($\rightarrow$ *soft links*). Oui, par l'inclusion.

Clarification, enracinement

- *A priori*, un arbre n'est pas orienté.
- En choisissant un sommet quelconque (que l'on appelle *racine* de l'arbre), on obtient une orientation unique qui en fait un *graphe enraciné*.



(Ici on a choisi le nœud 16 en tant que racine)

Ordre et taille d'un arbre

- Un arbre est un « *état d'équilibre* » entre deux « forces » contradictoires : la connexité et l'acyclicité, dont les extrêmes sont les graphes complets et les graphes discrets.
- Dans un arbre on a toujours $\|T\| = |T| - 1$, et c'est la plus petite taille qui maintienne la connexité.
- C'est également la plus grande taille qui maintienne l'acyclicité (une arête de plus et on a un cycle) et la plus petite taille qui maintienne la connexité (une arête de moins et on a deux composantes connexes).

Proposition (démonstration ludique)

Tout arbre fini d'ordre ≥ 2 a au moins deux feuilles.

Ouverture : laplacien et composantes connexes

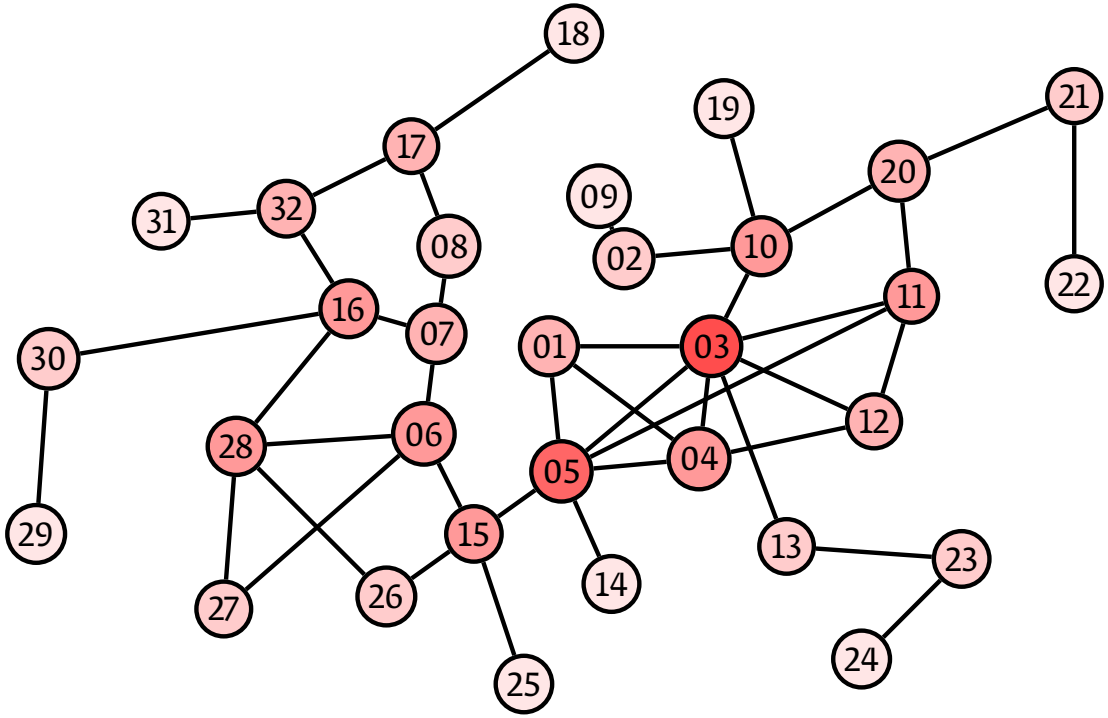
- On n'a pas le temps de développer ici, mais à partir de la matrice d'adjacence A on définit le *laplacien* L d'un graphe par
$$L := D - A,$$
où D est la matrice diagonale des degrés des sommets.
- On montre que le laplacien d'un graphe a autant de valeurs propres nulles que le graphe a des composantes connexes.
- Sous graph-tool, le laplacien peut-être calculé par `laplacian(g).array()`

Degré moyen, densité

- Retour aux graphes.
- On a vu la définition du degré $d(v_i)$. Si $A = (a_{i,j})$ est la matrice d'adjacence, on a $d(v_i) = \sum_{j=1}^{n} a_{i,j}$.
- Résultat important : *la somme des degrés de tous les sommets d'un graphe est égale à deux fois sa taille* ($\sum d(v_i) = 2\|G\|$).
- Donc, le degré moyen c du graphe est égal à $\frac{2\|G\|}{|G|}$.
- Si $|G| = n$, la taille maximum du graphe est $\frac{n(n-1)}{2}$.
- La *densité* ρ du graphe est la taille divisée par la taille maximum :

$$\rho = \frac{\|G\|}{\frac{n(n-1)}{2}} = \frac{c}{n-1}.$$

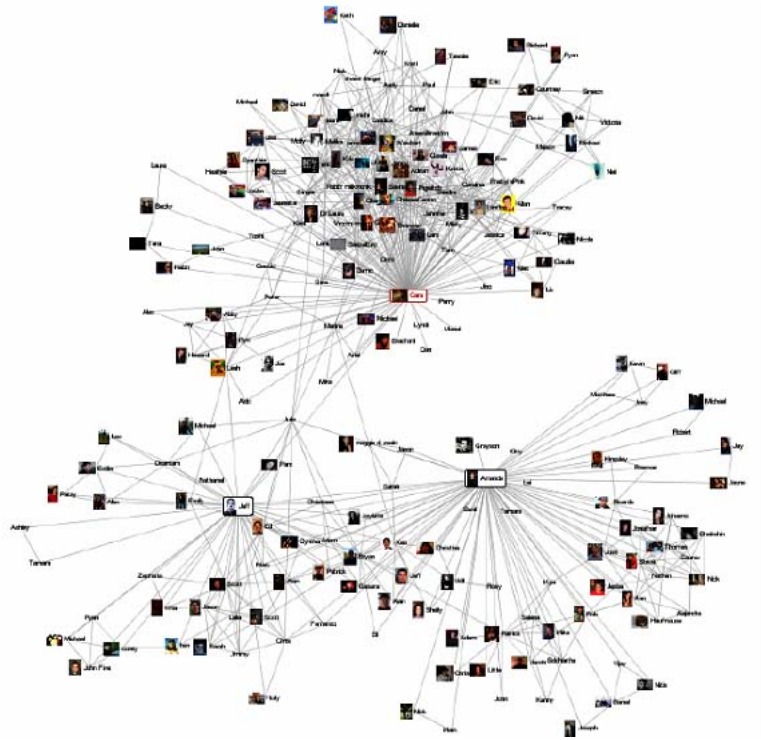
Exemple



- La profondeur du rouge représente le degré.
- Pour ce G, $|G| = 32$, $\|G\| = 42$, degré min $= 1$, degré max $= 7$, degré moyen $c = 2.625$, densité $\rho = 0.0847$.

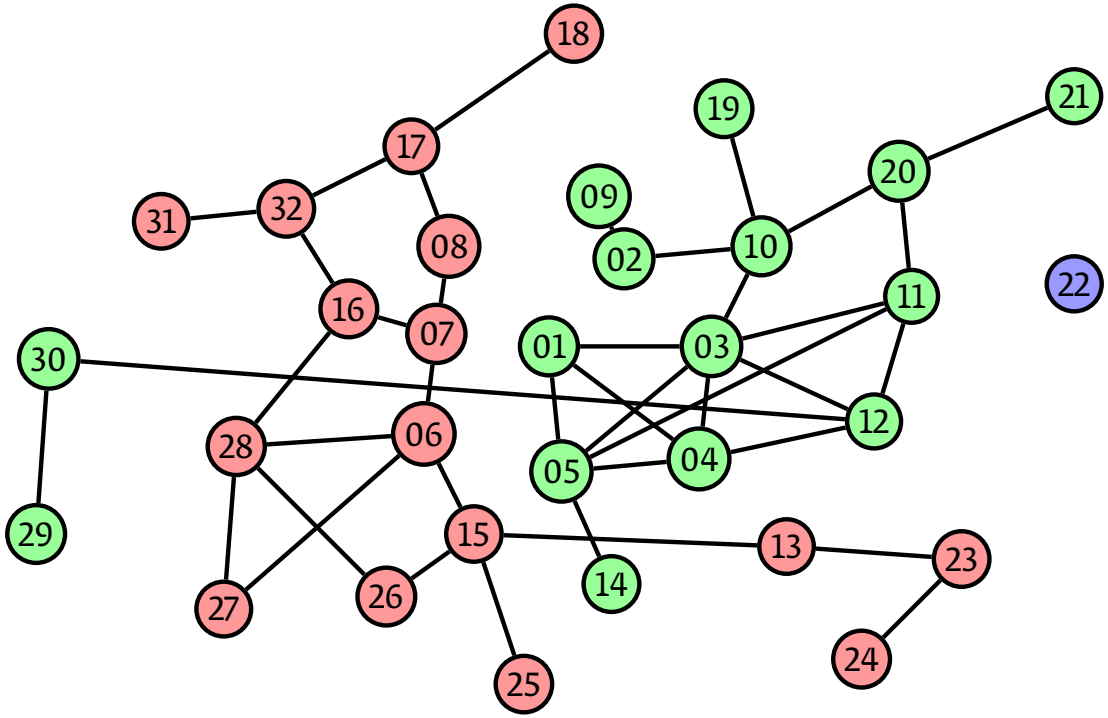
Grands réseaux, densité

- Certains graphes (comme le Web) peuvent croître indéfiniment, on les appelle des *grands réseaux*.
- On dit qu'un grand réseau G est *dense* quand $\lim_{|G|\to\infty} \rho > 0$, sinon il est *épars*.



- Exemple : un réseau social est épars car le nombre d'amis d'une personne n'augmente pas obligatoirement proportionnellement à la taille du réseau.

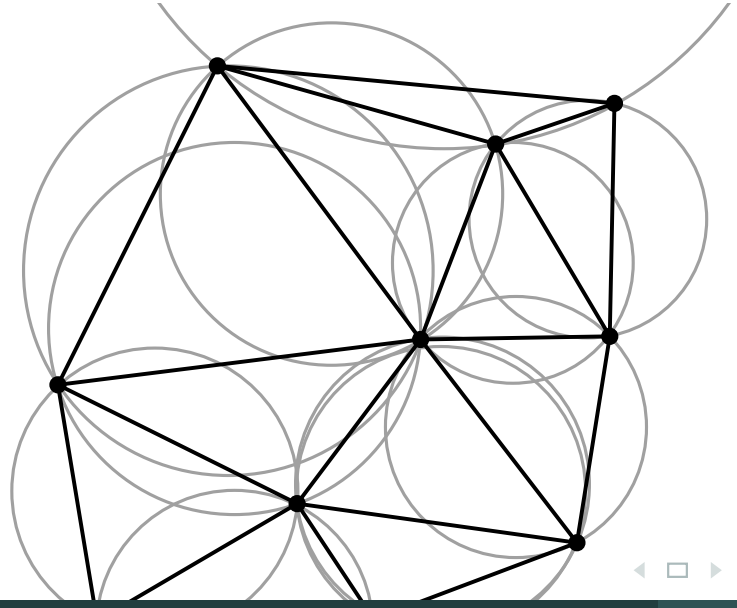
Composantes connexes



- Modulo une permutation des numéros de sommet, la matrice d'adjacence d'un graphe non connexe peut être écrite en tant que *matrice à blocs*, ayant autant de blocs que de composantes connexes.

Triangulation

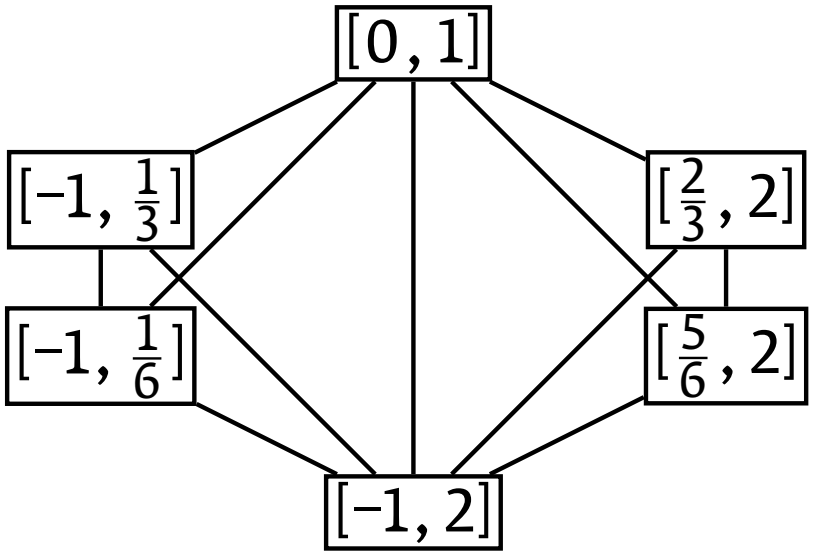
- Un graphe est *triangulé* (ou *cordal*) si tout cycle de longueur ≥ 4 possède une corde (c'est-à-dire, une arête entre deux sommets du cycle).
- Les graphes triangulés sont utilisés abondamment par les *systèmes d'information géographique* (SIG). Pour modéliser la surface de la Terre, on utilise la *triangulation de Delaunay* qui consiste à trianguler un ensemble de points de manière à ce qu'aucun point ne soit à l'intérieur du cercle circonscrit d'un des triangles de la triangulation.



Graphes d'intervalles

- Un *graphe d'intervalles* est un graphe non orienté dont les sommets représentent des intervalles (par exemple, de $\mathbb{R}$) et les arêtes représentent l'intersection non-vide entre deux intervalles.

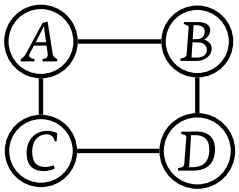
  Exemple :



Graphes d'intervalles

Théorème

Un graphe d'intervalles est un graphe triangulé.

Preuve : Soit A, B, C, D quatre intervalles de $\mathbb{R}$ qui forment un graphe 

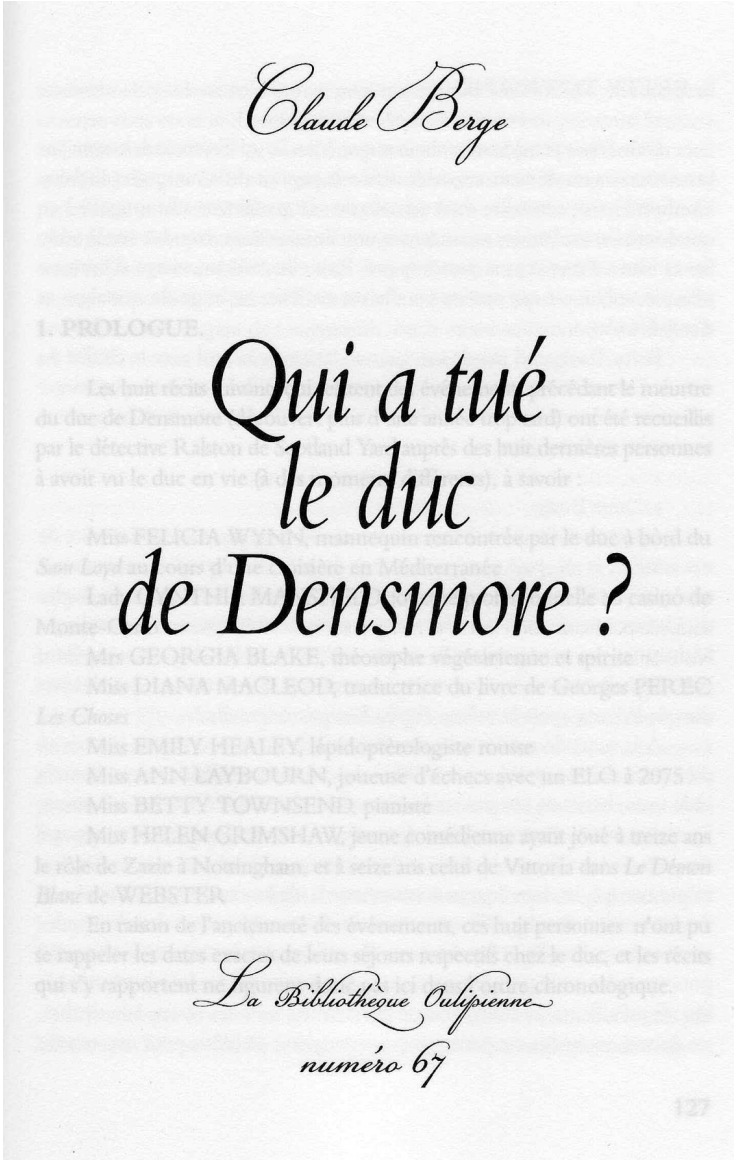
. Montrons qu'on a nécessairement une arête AD ou BC. Notons $B \succ A$ lorsque $\exists a \in A, \forall b \in B, b \geq a$. On suppose donc $A \cap B \neq \emptyset$, $A \cap C \neq \emptyset$, $B \cap D \neq \emptyset$, $C \cap D \neq \emptyset$, $A \cap D = \emptyset$ et $B \cap C = \emptyset$. Supposons $B \succ A$, alors forcément $D \succ B$ et $A \succ C$. Mais alors $D \cap C = \emptyset$ aussi, ce qui contredit la supposition $C \cap D \neq \emptyset$. Il en va de même pour $A \succ B$. □

Théorème (Hajós)

Un graphe d'intervalles est un graphe triangulé qui ne contient aucun triangle inscrit dans un hexagone.

Parenthèse : Qui a tué le duc de Densmore ?

Il s'agit du titre d'une nouvelle policière (« Qui a tué le duc de Densmore ? », *La Bibliothèque oulipienne*, n° 67, Paris, 1994, pages 75–90), écrite par Claude Berge qui était, entre autres, membre fondateur de l'Oulipo.



Parenthèse : Qui a tué le duc de Densmore ?

- L'intrigue est la suivante :
- Le Duc de Densmore est retrouvé carbonisé après l'explosion de son vieux château de l'île privée de White.
- L'explosion est causée par une bombe placée dans la cave du château.
- Dans la période qui précéda l'explosion, le Duc avait invité ses 8 ex-femmes.
- Le capitaine du bateau qui faisait la navette entre l'île et le continent se souvient d'avoir transporté chacune des 8 femmes pour un aller-retour, mais sans se souvenir des dates et heures.
- Les femmes ne se souviennent pas non plus des dates et heures, mais se rappellent de s'être croisées. Ceci étant, l'enquête a beaucoup tardé et les souvenirs des femmes ne sont qu'approximatifs.

Parenthèse : Qui a tué le duc de Densmore ?

Ainsi, elles se rappellent plus ou moins de s'être croisées comme suit :

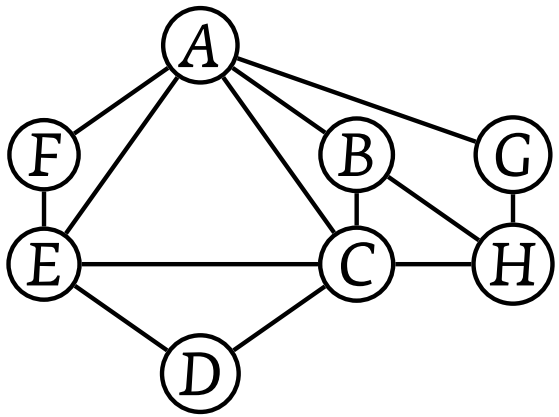
- Anne a rencontré Félicie, Cynthia, Georgia, Emilie et Betty,
- Betty a rencontré Cynthia, Anne et Hélène,
- Cynthia a rencontré Anne, Emilie, Diane, Betty et Hélène,
- Diane a rencontré Cynthia et Emilie,
- Émilie a rencontré Félicie, Cynthia, Diane et Anne,
- Félicie a rencontré Émilie et Anne,
- Georgia a rencontré Anne et Hélène
- Hélène a rencontré Cynthia, Georgia et Betty.

Parenthèse : Qui a tué le duc de Densmore ?

- On sait d'autre part que l'une de ces femmes était désavantagée par le testament et donc avait un mobile pour poser la bombe. Le testament a malheureusement disparu dans l'incendie. Comment trouver la coupable ?
- L'inspecteur Ralston de Scotland Yard suppose que la personne qui a posé la bombe n'a pas pris la navette et a donc menti. Comment trouver la menteuse parmi ces huit femmes ?

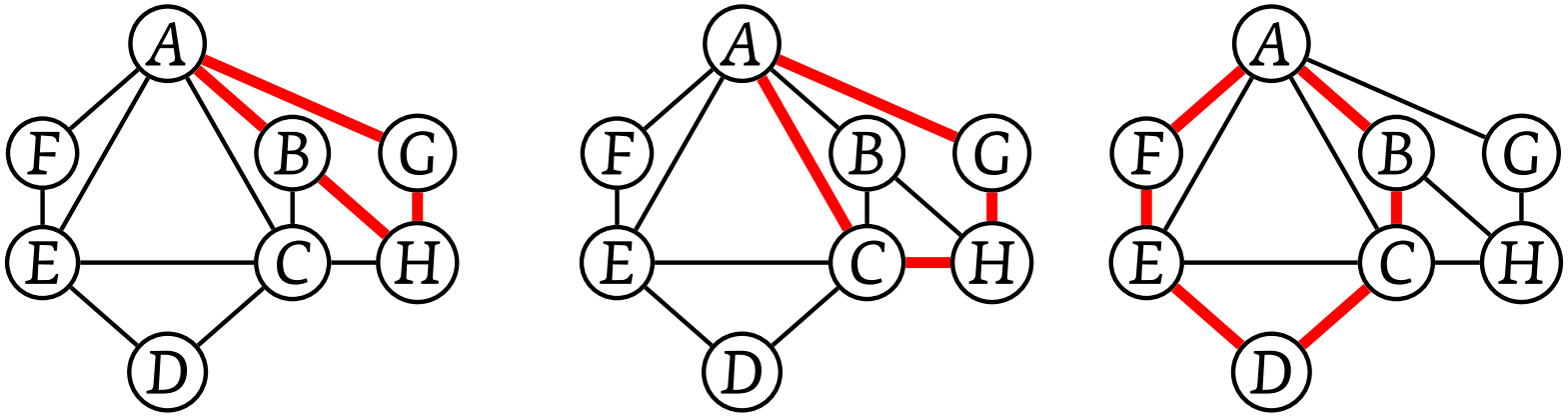
Parenthèse : Qui a tué le duc de Densmore ?

- Méthode de solution du problème :
- la présence de chaque femme dans la navette peut être représentée par un intervalle de $\mathbb{R}$,
- on peut donc former le graphe des intervalles et vérifier s'il est triangulé et s'il ne contient aucun triangle inscrit dans un hexagone.
- Si jamais ce n'est pas le cas, on peut vérifier si le fait de supprimer un sommet le rend triangulé/sans triangle inscrit dans un hexagone.



Parenthèse : Qui a tué le duc de Densmore ?

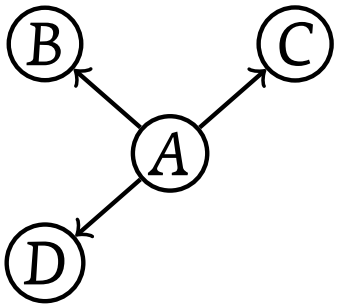
Voilà déjà deux cycles non-triangulés et un hexagone avec un triangle inscrit :



Qui est donc le coupable ? (Vérifier que si on l'enlève du graphe, celui-ci devient triangulé et n'a plus de triangle inscrit dans un hexagone et que si on enlève n'importe quel autre sommet du graphe, soit il n'est pas triangulé, soit il contient un triangle inscrit dans un hexagone.)

Représentation informatique des graphes

- Jusqu'ici on a représenté les graphes par leur *matrice d'adjacence* A.
- Cela consomme n^2 emplacements (ou $n^2/2$ pour un graphe non orienté). C'est beaucoup, surtout pour un graphe épars.
- Autre possibilité : les *listes d'adjacence*. On attache à chaque nœud v_i ses voisins $v_{i,1}$, $v_{i,2}$, ... par des liens $v_i \to v_{i,1} \to v_{i,2} \to \cdots$
- Attention PIÈGE : on attache au nœud un de ses voisins, et ensuite les voisins entre eux. Ces liaisons ne sont pas les arêtes du graphe d'origine :



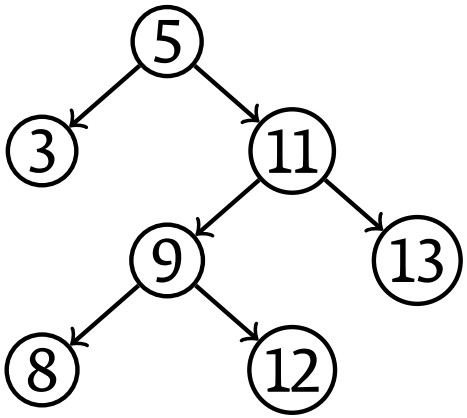
$$A \to B \to D \to C.$$

Exemple : liste d'adjacence

- 1 : 2 → 3 → 4 → 5 → 6
- 2 : 9 → 10 → 3 → 1
- 3 : 1 → 2 → 10 → 11 → 12 → 4 → 5
- 4 : 5 → 1 → 3 → 12
- 5 : 1 → 4 → 14 → 15 → 7
- 6 : 7 → 1 → 15 → 28
- 7 : 8 → 5 → 6 → 16
- 8 : 17 → 7 → 27 → 16
- 9 : 2
- 10 : 19 → 20 → 3 → 2
- 11 : 20 → 12 → 5 → 3
- 12 : 11 → 3 → 4
- 13 : 3 → 23
- 14 : 5
- 15 : 6 → 5 → 25 → 26
- 16 : 32 → 8 → 7 → 28 → 30
- 17 : 18 → 8 → 32
- 18 : 17
- 19 : 10
- 20 : 10 → 11 → 21
- 21 : 20 → 22
- 22 : 21
- 23 : 13 → 24
- 24 : 23
- 25 : 15
- 26 : 15 → 28
- 27 : 28 → 8 → 6
- 28 : 16 → 6 → 26 → 27
- 29 : 30
- 30 : 29 → 16
- 31 : 32
- 32 : 31 → 16 → 17

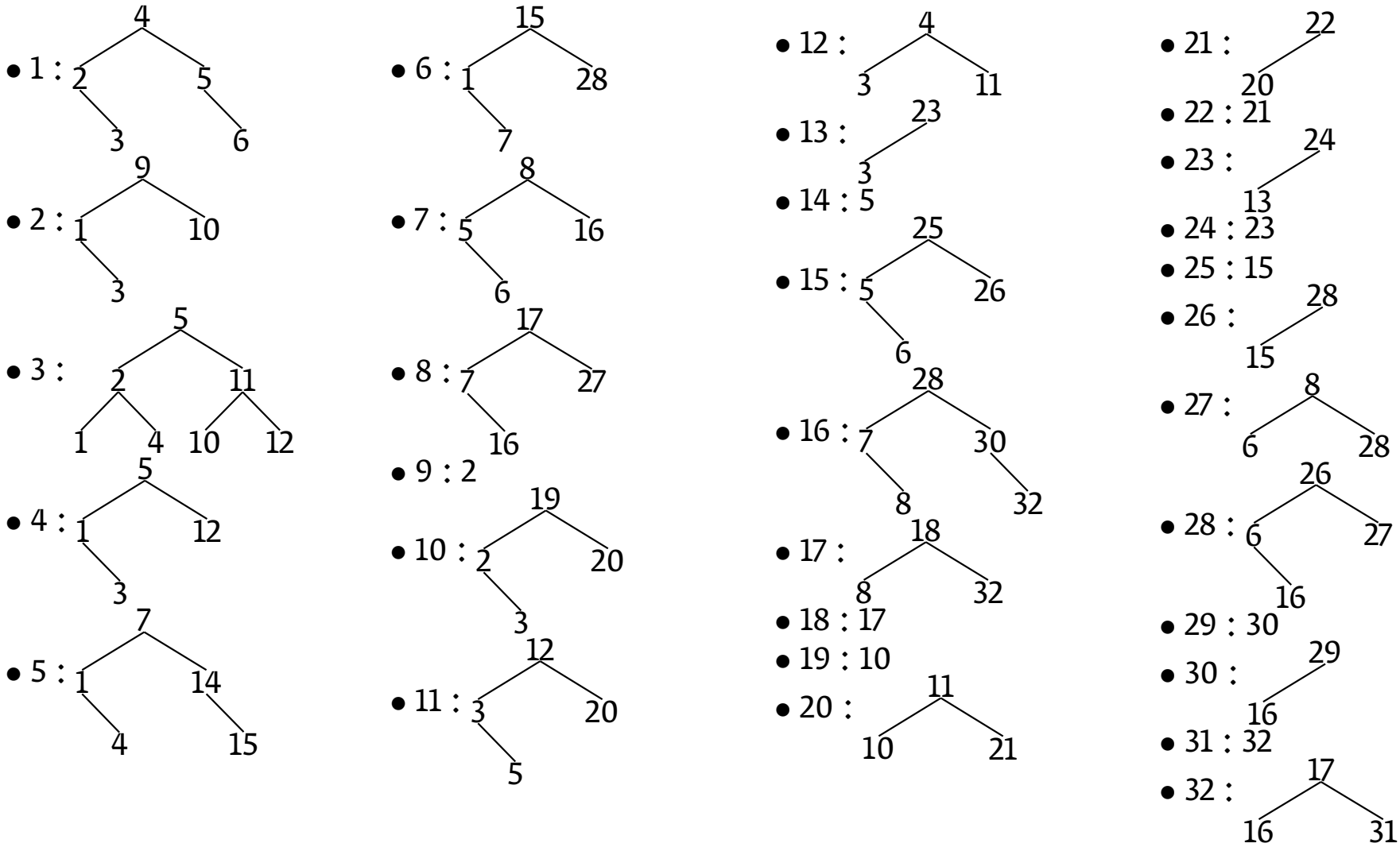
Arbres binaires de recherche

- Un *arbre binaire de recherche* est un arbre orienté tel que :
  - chaque sommet est de degré sortant ≤ 2,
  - on a attaché des valeurs numériques à tous les sommets,
  - ainsi que des étiquettes « gauche », « droit » de manière à ce que tout nœud ait au plus un enfant gauche et un enfant droit,
  - la valeur numérique de l'enfant gauche est inférieure à celle du parent, qui est inférieure à celle de l'enfant droit.



- Troisième possibilité de représentation des graphes : les *forêts d'adjacence*. On attache à chaque nœud ses voisins sous forme d'arbre binaire de recherche.

Exemple : forêt d'adjacence



Représentation des graphes

- Tableau comparatif :

Opération	Matrice	Liste	Forêt d'arbres binaires
Insertion	$O(1)$	$O(1)$	$O(\log(m/n))$
Suppression	$O(1)$	$O(m/n)$	$O(\log(m/n))$
Recherche	$O(1)$	$O(m/n)$	$O(\log(m/n))$
Énumération	$O(n)$	$O(m/n)$	$O(m/n)$

partie II

Algorithmes de base

Parcourir les sommets d'un graphe

- Avant toute chose, il faut être capable de parcourir tous les sommets d'un graphe, en partant d'un sommet donné.
- Nous verrons deux manières de le faire :
- par un *parcours en profondeur d'abord* (DFS = *depth-first search*) : on va aussi loin que possible en faisant des choix lors des branchements, et ensuite on remonte aussi près que possible pour faire les choix restant ;
- ou alors par un *parcours en largeur d'abord* (BFS = *breadth-first search*) : on procède par niveaux en considérant d'abord tous les sommets à une distance (géodésique) donnée, avant de traiter ceux du niveau suivant.
- La complexité des deux approches est la même.
- Particularités : DFS utilise une procédure récurrente, il peut être utile pour calculer le nb de composantes connexes. Avec BFS on obtient l'arbre des plus courts chemins (géodésiques).

Parcours en profondeur d'abord (DFS)

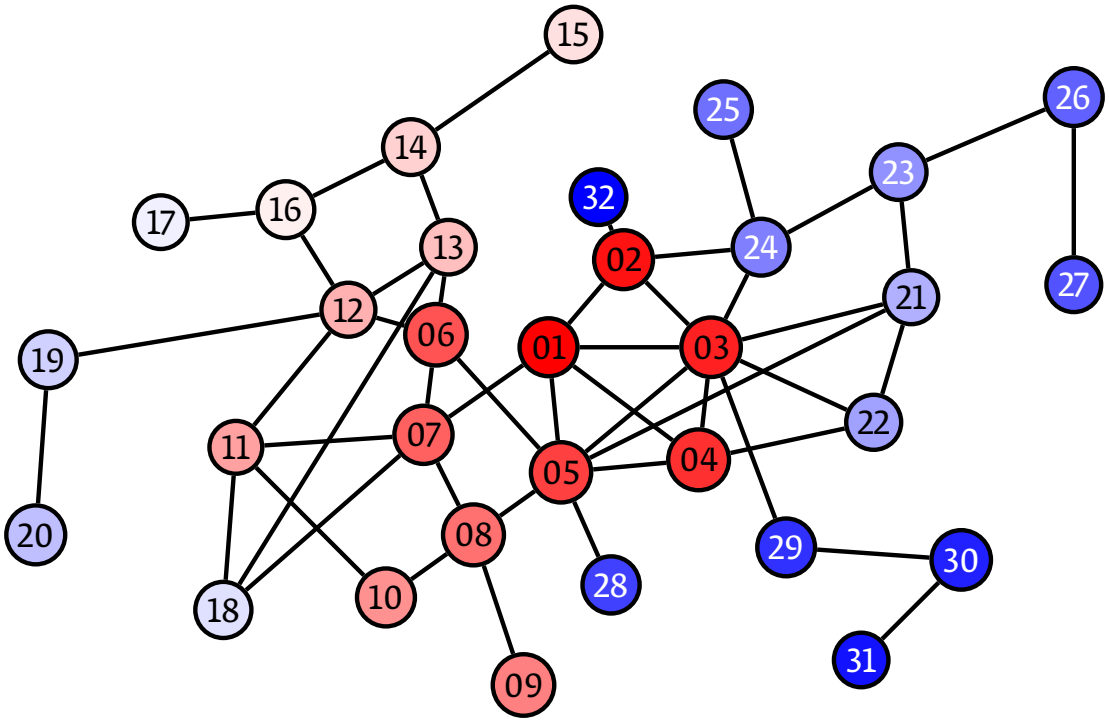
Soit G un graphe. Algorithme du *parcours en profondeur d'abord* :

DFS(G) :
  for chaque sommet $u \in G$:
    do COULEUR(u) $\leftarrow$ BLANC
  for chaque sommet $u \in G$:
    do if COULEUR(u) = BLANC :
      then DFS-VISIT(u)

DFS-VISIT(u) :
  COULEUR(u) $\leftarrow$ GRIS
  for chaque sommet $v \in$ VOISINS(u) :
    do if COULEUR(v) = BLANC :
      then DFS-VISIT(v)
  COULEUR(u) $\leftarrow$ NOIR

Temps de calcul : $O(V + E)$.

Exemple de DFS

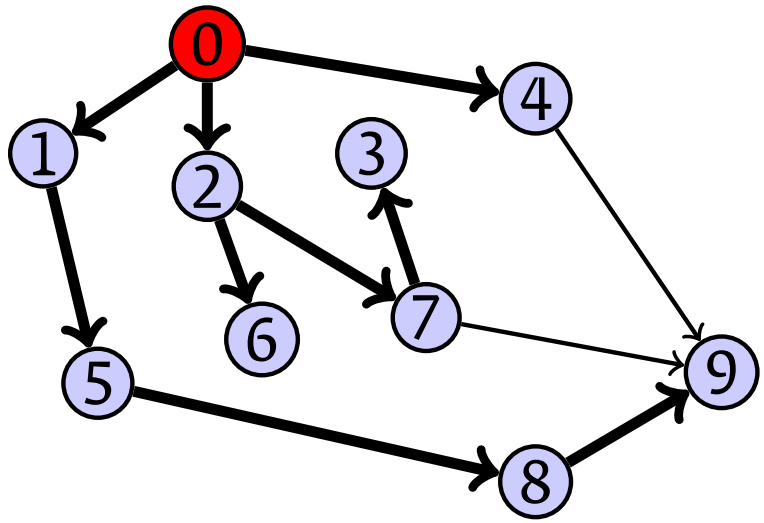


On a re-numéroté les sommets dans l'ordre du DFS en partant du sommet 1. Les couleurs vont graduellement du rouge au bleu. Noter à quel point le sommet 32 est proche du sommet 1.

DFS sous graph-tool

```
g = Graph(directed=True)
g.add_vertices(10)
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),\
    (4,9),(5,8),(7,9),(8,9)]
for (a,b) in my_edges:
    e = g.add_edge(a,b)
for e in dfs_iterator(g,g.vertex(0)):
    print(e)
```

Résultat : (0, 1), (1, 5), (5, 8), (8, 9), (0, 2), (2, 6), (2, 7), (7, 3), (0, 4)



La notion de file d'attente (FIFO)

La notion de file d'attente (FIFO)

La notion de file d'attente (FIFO)

La notion de file d'attente (FIFO)

La notion de file d'attente (FIFO)

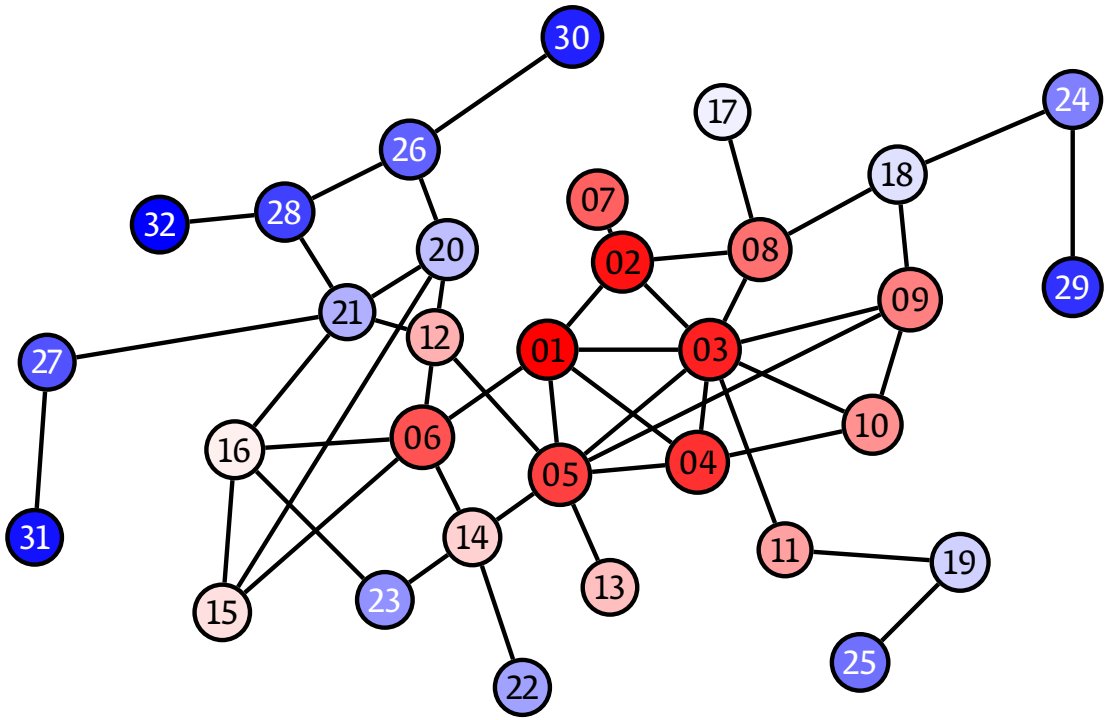
Parcours en largeur d'abord (BFS)

Soit G un graphe *connexe* et s un sommet. Algorithme du *parcours en largeur d'abord* à partir de s :

BFS(G,s) :
  for chaque sommet $u \in G \backslash \{s\}$
    do COULEUR(u) $\leftarrow$ BLANC
      $d[u] \leftarrow \infty$, $\pi[u] \leftarrow$ NIL
  COULEUR[s] $\leftarrow$ GRIS, $d[s] \leftarrow 0$, $\pi[s] \leftarrow$ NIL, $Q \leftarrow \{s\}$
  while $Q \neq \emptyset$:
    do $u \leftarrow$ DÉFILER([Q])
      for chaque $v \in$ VOISINS(u) :
        do if COULEUR[v] = BLANC :
          then COULEUR[v] $\leftarrow$ GRIS
            $d[v] \leftarrow d[u] + 1$, $\pi[v] \leftarrow u$, ENFILER(Q, v)
      COULEUR(u) $\leftarrow$ NOIR

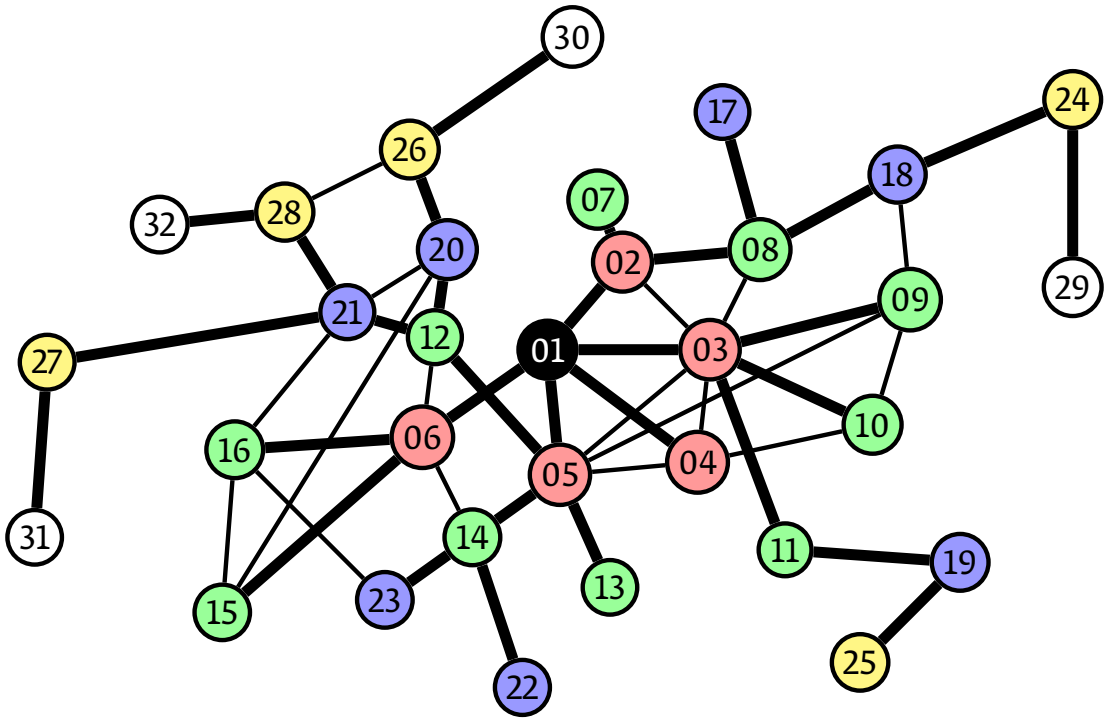
Temps de calcul : $O(V + E)$.

Exemple de BFS



On a re-numéroté les sommets dans l'ordre du DFS en partant du sommet 1. Les couleurs vont graduellement du rouge au bleu. Cette fois-ci, 1 est bien éloigné de 32.

Exemple de BFS

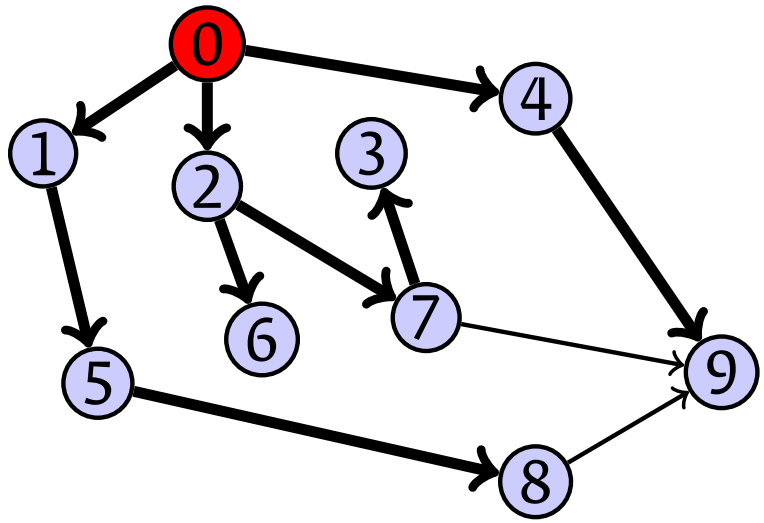


Les couleurs rouge, vert, bleu, jaune, blanc indiquent les 5 niveaux de distance. En traits épais : un arbre couvrant géodésique.

BFS sous graph-tool

```
g = Graph(directed=True)
g.add_vertices(10)
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),\
    (4,9),(5,8),(7,9),(8,9)]
for (a,b) in my_edges:
    e = g.add_edge(a,b)
for e in bfs_iterator(g,g.vertex(0)):
    print(e)
```

Résultat : (0, 1), (0, 2), (0, 4), (1, 5), (2, 6), (2, 7), (4, 9), (5, 8), (7, 3)



Recherche de plus court chemin

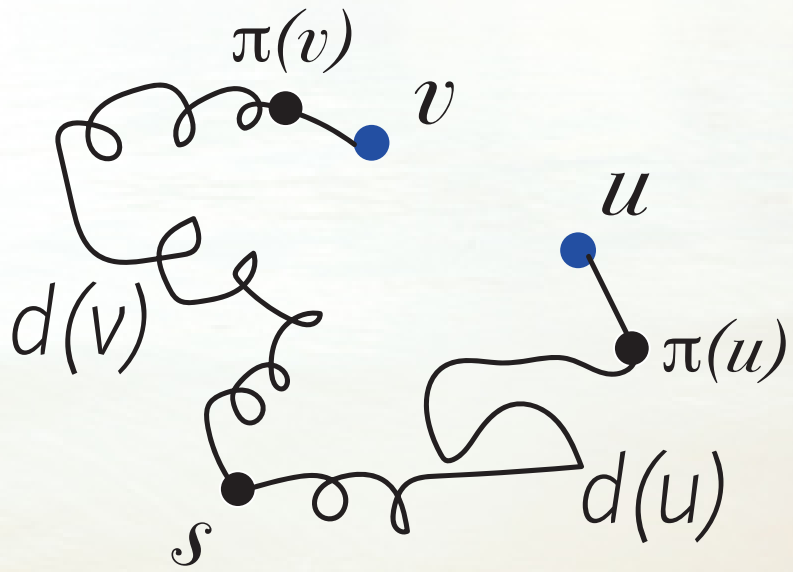
- Limitons-nous aux graphes orientés.
- Le BFS nous a permis de trouver les chemins géodésiques. Mais dans plusieurs situations, les arêtes disposent d'un poids (par exemple, une distance), et on cherche les plus courts chemins par rapport à ce poids.
- Le poids peut être négatif, mais il vaut mieux ne pas avoir de cycle de poids total négatif.
- De toute façon, un plus court chemin ne peut pas contenir de cycle de poids total strictement positif.
- On va considérer trois algorithmes de recherche de plus courts chemins : Bellman-Ford, Dijkstra et $A^\star$. Dans les trois cas, on se sert d'une méthode générale, appelée relaxation.

Exercices de relaxation

- Pour chaque sommet u du graphe, on mésure une quantité positive $d(u)$ qui va finir par être la distance de u de l'origine s. Pour stocker l'information du plus court chemin on utilise la notion de *sommet précédent* $\pi(v)$.
- La *relaxation* d'une arête (u, v) consiste à changer $d(v)$ si on peut passer par u pour arriver à v à moindre coût :

$\textsc{Relax}_d(u, v, w)$:
  if $d(v) > d(u) + w(u, v)$:
    $d(v) \leftarrow d(u) + w(u, v)$
    $\pi(v) \leftarrow u$

où $w(u, v)$ est le poids de l'arête (u, v).

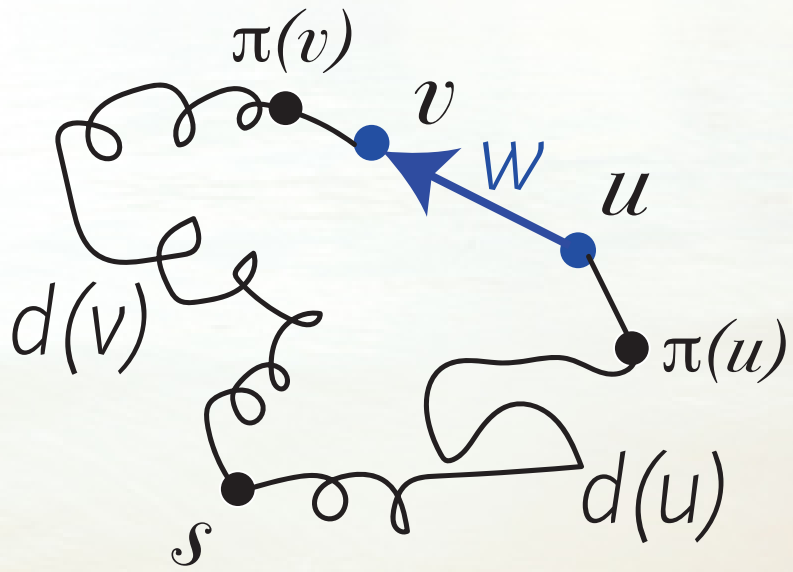


Exercices de relaxation

- Pour chaque sommet u du graphe, on mésure une quantité positive $d(u)$ qui va finir par être la distance de u de l'origine s. Pour stocker l'information du plus court chemin on utilise la notion de *sommet précédent* $\pi(v)$.
- La *relaxation* d'une arête (u, v) consiste à changer $d(v)$ si on peut passer par u pour arriver à v à moindre coût :

$\textsc{Relax}_d(u, v, w)$:
  if $d(v) > d(u) + w(u, v)$:
    $d(v) \leftarrow d(u) + w(u, v)$
    $\pi(v) \leftarrow u$

où $w(u, v)$ est le poids de l'arête (u, v).

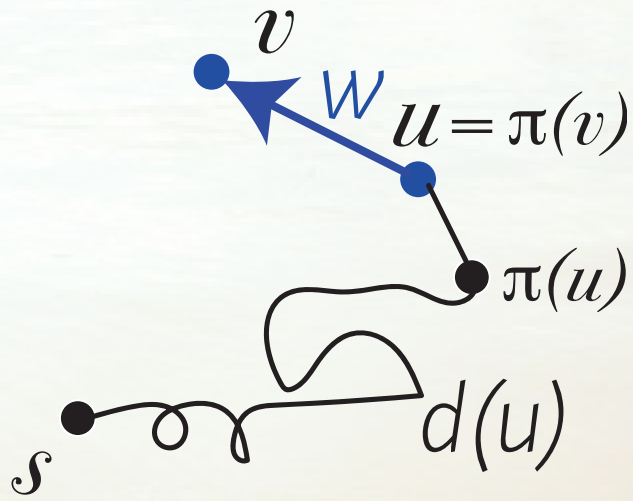


Exercices de relaxation

- Pour chaque sommet u du graphe, on mésure une quantité positive $d(u)$ qui va finir par être la distance de u de l'origine s. Pour stocker l'information du plus court chemin on utilise la notion de *sommet précédent* $\pi(v)$.
- La *relaxation* d'une arête (u, v) consiste à changer $d(v)$ si on peut passer par u pour arriver à v à moindre coût :

$\textsc{Relax}_d(u, v, w)$:
  if $d(v) > d(u) + w(u, v)$:
    $d(v) \leftarrow d(u) + w(u, v)$
    $\pi(v) \leftarrow u$

où $w(u, v)$ est le poids de l'arête (u, v).



Exercices de relaxation

- On initialise le graphe de la manière suivante :

  INITIALIZE-SINGLE-SOURCE$_d(G, s)$:
  for chaque $v \in V$:
  $d(v) \leftarrow \infty$, $\pi(v) \leftarrow \emptyset$
  $d(s) \leftarrow 0$

- Tous les algorithmes que l'on va étudier *initialisent* et *relaxent*. Ce qui diffère c'est l'ordre et la quantité de relaxations.

Bellman-Ford

- L'*algorithme de Bellman-Ford* trouve un plus court chemin pour des poids positifs ou négatifs.

  BELLMAN-FORD(G, w, s) :
    INITIALIZE-SINGLE-SOURCE$_d(G, s)$
    for $i \leftarrow 1$ à $|V| - 1$:
      for chaque arête (u, v) :
        RELAX$_d(u, v, w)$
    for chaque arête (u, v) :
      if $d(v) > d(u) + w(u, v)$:
        return FALSE
    return TRUE

  À noter l'absence de i dans le reste du code…

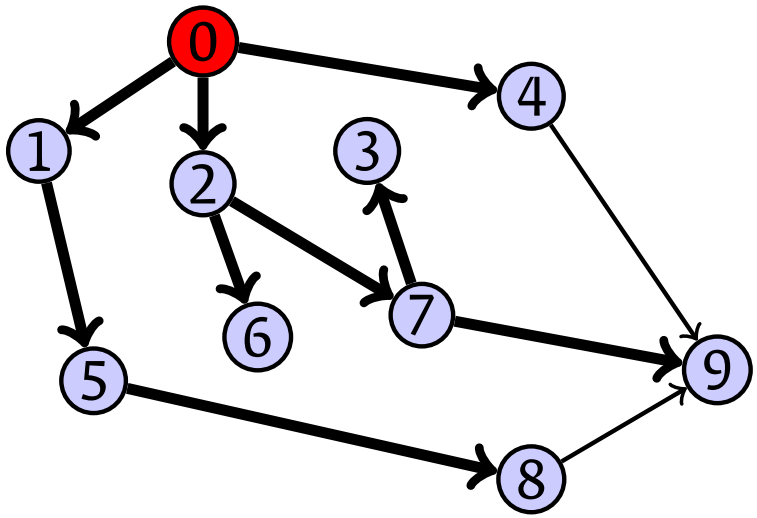
- Cet algorithme est de complexité $O(VE)$: si on a beaucoup d'arêtes, c'est-à-dire $O(n^2)$, l'algorithme est en $O(n^3)$, c'est énorme !

Bellman-Ford sous graph-tool 1/2

```
from graph_tool.all import *
g = Graph(directed=True)
weight = g.new_edge_property("int16_t")
g.add_vertex(10)
weights=[85,217,173,80,186,103,183,502,250,167,84]
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),(4,9),\
    (5,8),(7,9),(8,9)]
for ((a,b),w) in zip(my_edges,weights):
    e = g.add_edge(a,b)
    weight[e]=w
(res,dist,pred)=bellman_ford_search(g,g.vertex(0),weight)
print(res)
for v in g.vertices():
    print("d(",v,")=",dist[v],"pi(",v,")=",pred[v])
```

Bellman-Ford sous graph-tool 2/2

```
True
d( 0 )= 0 pi( 0 )= 0
d( 1 )= 85 pi( 1 )= 0
d( 2 )= 217 pi( 2 )= 0
d( 3 )= 503 pi( 3 )= 7
d( 4 )= 173 pi( 4 )= 0
d( 5 )= 165 pi( 5 )= 1
d( 6 )= 403 pi( 6 )= 2
d( 7 )= 320 pi( 7 )= 2
d( 8 )= 415 pi( 8 )= 5
d( 9 )= 487 pi( 9 )= 7
```



Files de priorité min

- Pour améliorer Bellman-Ford, on utilise une structure de données spécifique : les *files de priorité min*.
- Une *file de priorité min* est une structure de données S qui sert à stocker des éléments x selon la valeur d'une fonction $d(x)$. Elle nécessite les trois opérations suivantes :
  - $\textsc{Insert}_d(S, x)$: insère un élément dans S.
  - $\textsc{Minimum}_d(S)$: retourne le plus petit élément de S.
  - $\textsc{Extract-Min}_d(S)$: idem que $\textsc{Minimum}$ mais, en plus, supprime l'élément en question de l'ensemble.

- ATTENTION : les poids changent à chaque itération, donc on refait le calcul du minimum à chaque fois !

Algorithme de Dijkstra

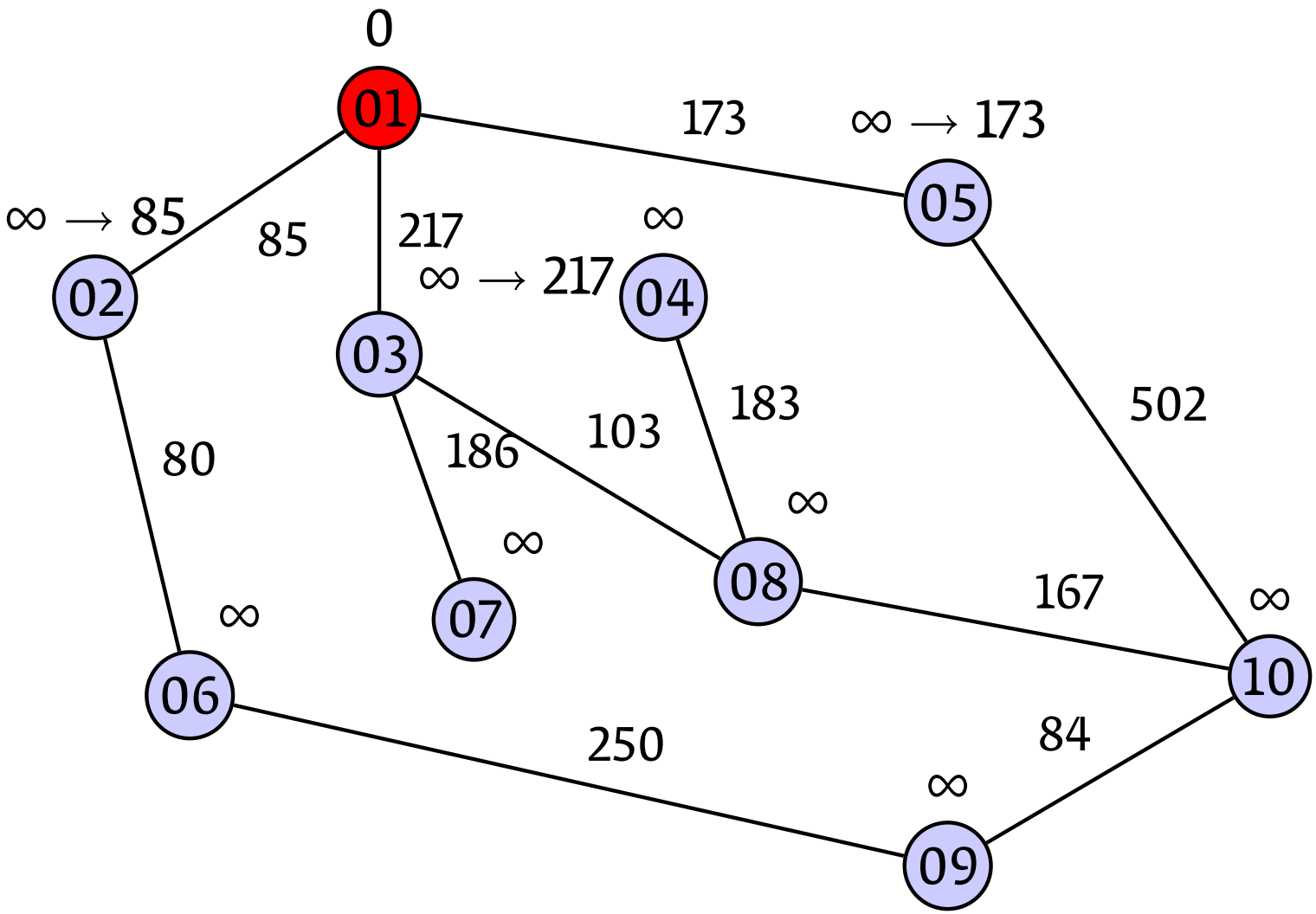
- Dijkstra est un nom néerlandais qui se prononce *Daïk-stra* !
- L'*algorithme de Dijkstra* est beaucoup plus rapide que Bellman-Ford, mais ne fonctionne que quand les poids sont positifs. On utilise une file de priorité min Q et un ensemble S.

DIJKSTRA(G, w, s) :
  INITIALIZE-SINGLE-SOURCE$_d(G, s)$
  $S \leftarrow \emptyset$, $Q \leftarrow \{s\}$
  while $Q \neq \emptyset$:
    $u \leftarrow$ EXTRACT-MIN$_d(Q)$
    $S \leftarrow S \cup \{u\}$
    for chaque $v \in$ VOISINS(u) :
      if $v \notin S$:
        RELAX$_d(u, v, w)$
        $Q \leftarrow Q \cup \{v\}$

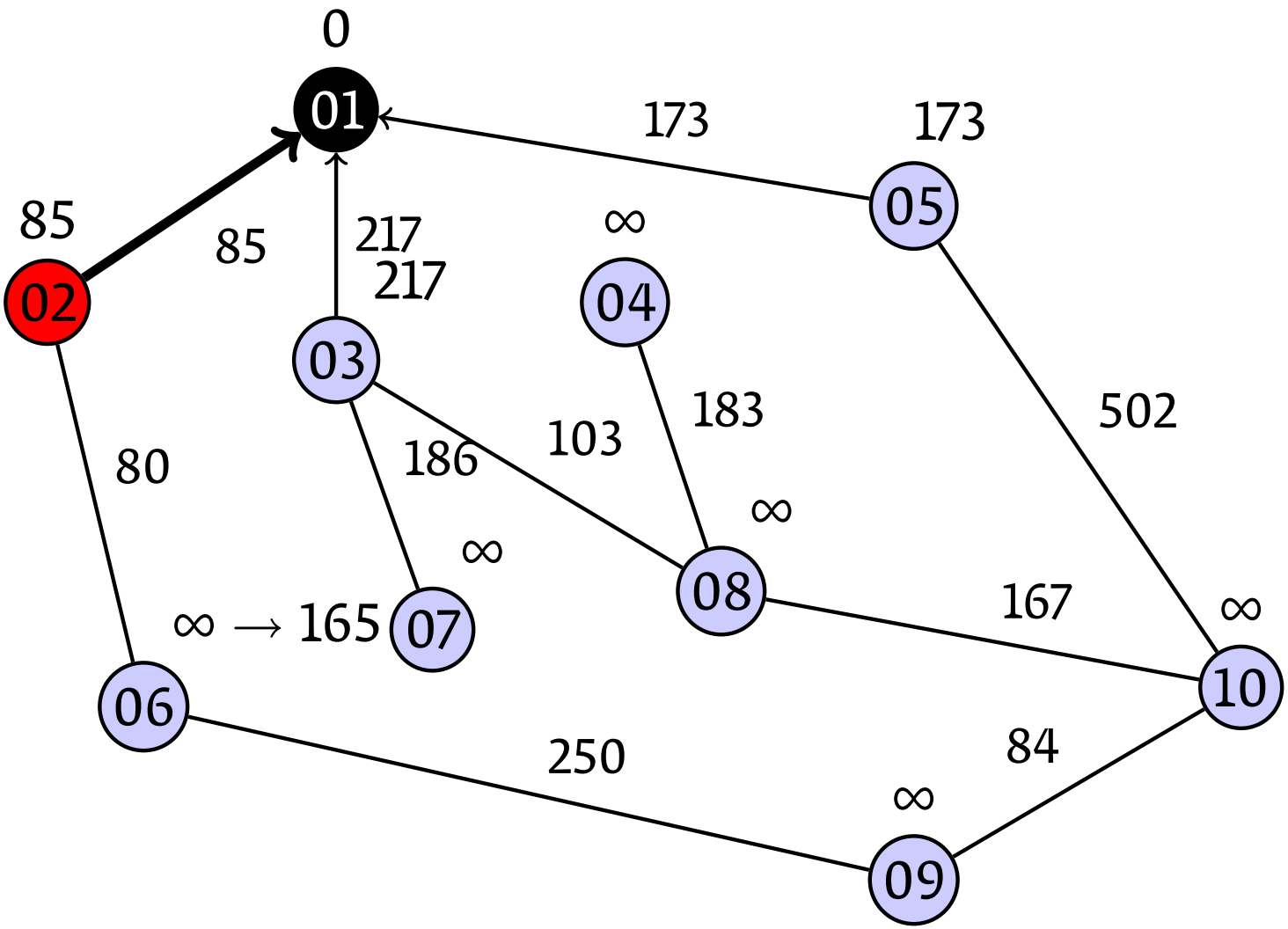
Lecture de l'exemple qui suit

- Dans l'exemple qui suit, les nombres attachés aux arêtes désignent le poids, et ceux attachés aux sommets, les valeurs de la fonction d, qui va évoluer et finir par être la distance à partir du sommet 1.
- Nous avons représenté en noir les sommets de l'ensemble S et en rouge le sommet obtenu à chaque étape par l'opération $\textsc{Extract-Min}_d(Q)$.
- L'image représente l'état du graphe à l'instant qui suit la sélection d'un élément et les relaxations de ses voisins.
- Les flèches épaisses dénotent la fonction π du sommet précédent. Elles permettent de retrouver le plus court chemin entre 1 et un sommet quelconque.

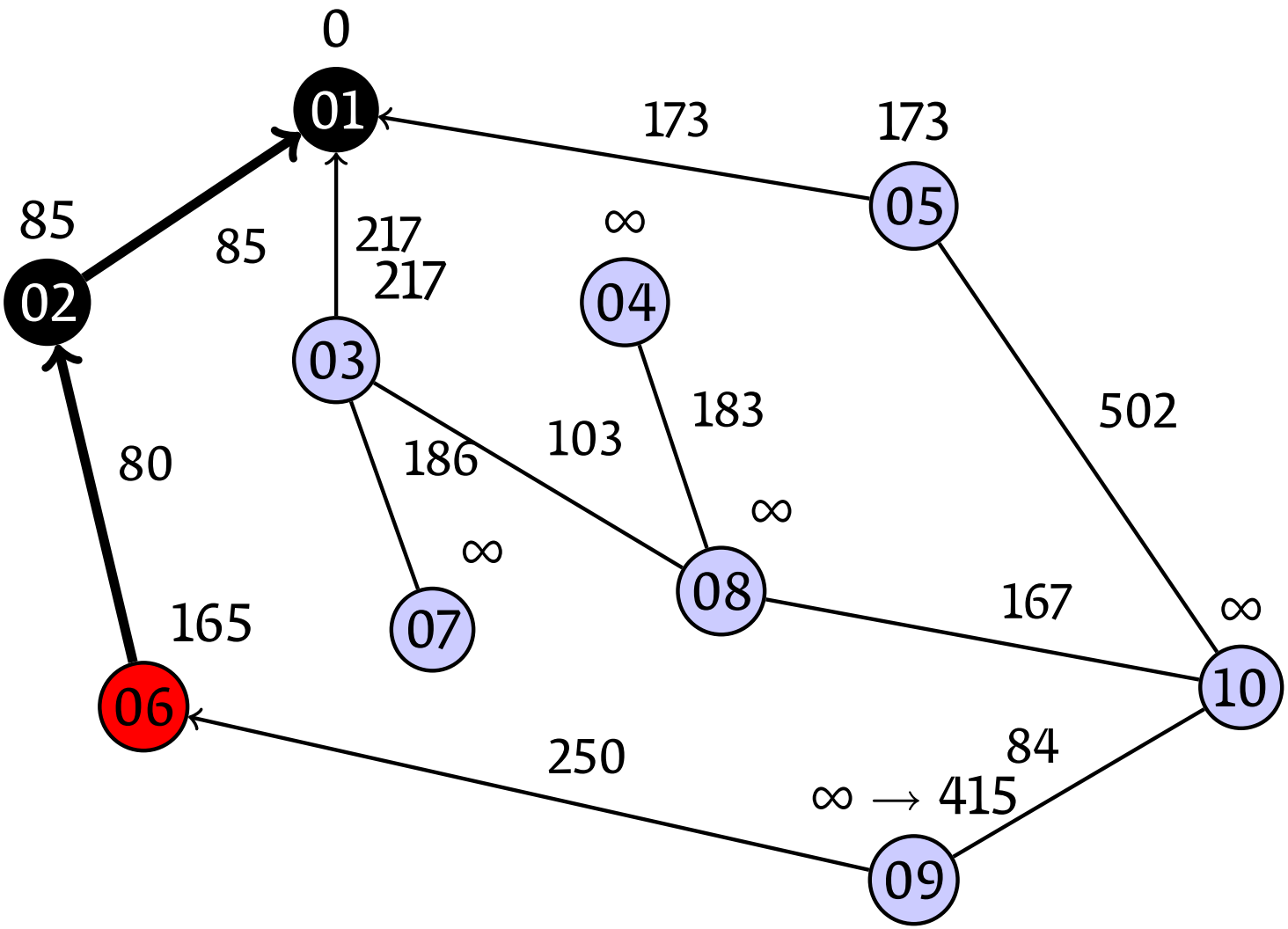
Exemple d'application de Dijkstra 1/10



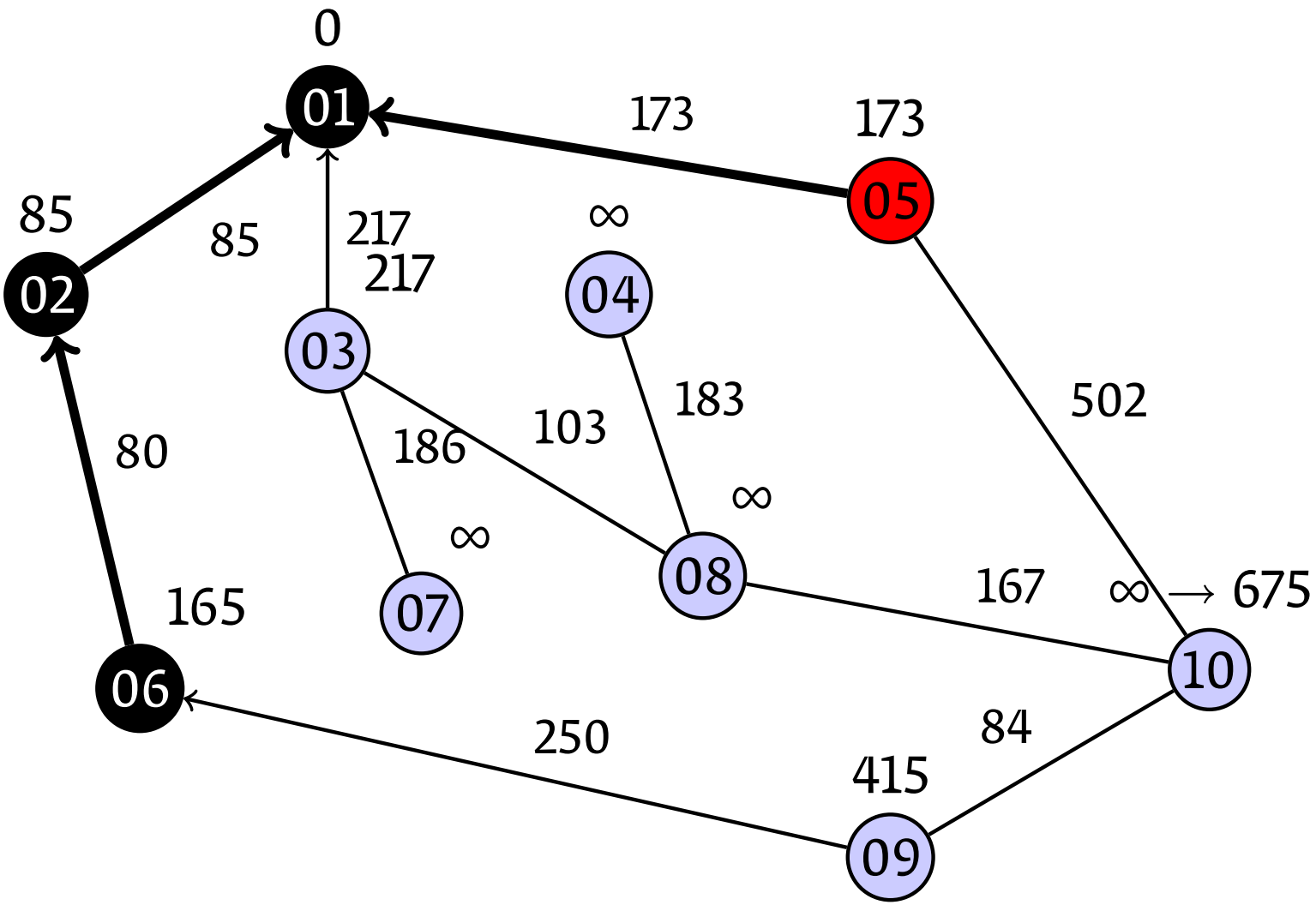
Exemple d'application de Dijkstra 2/10



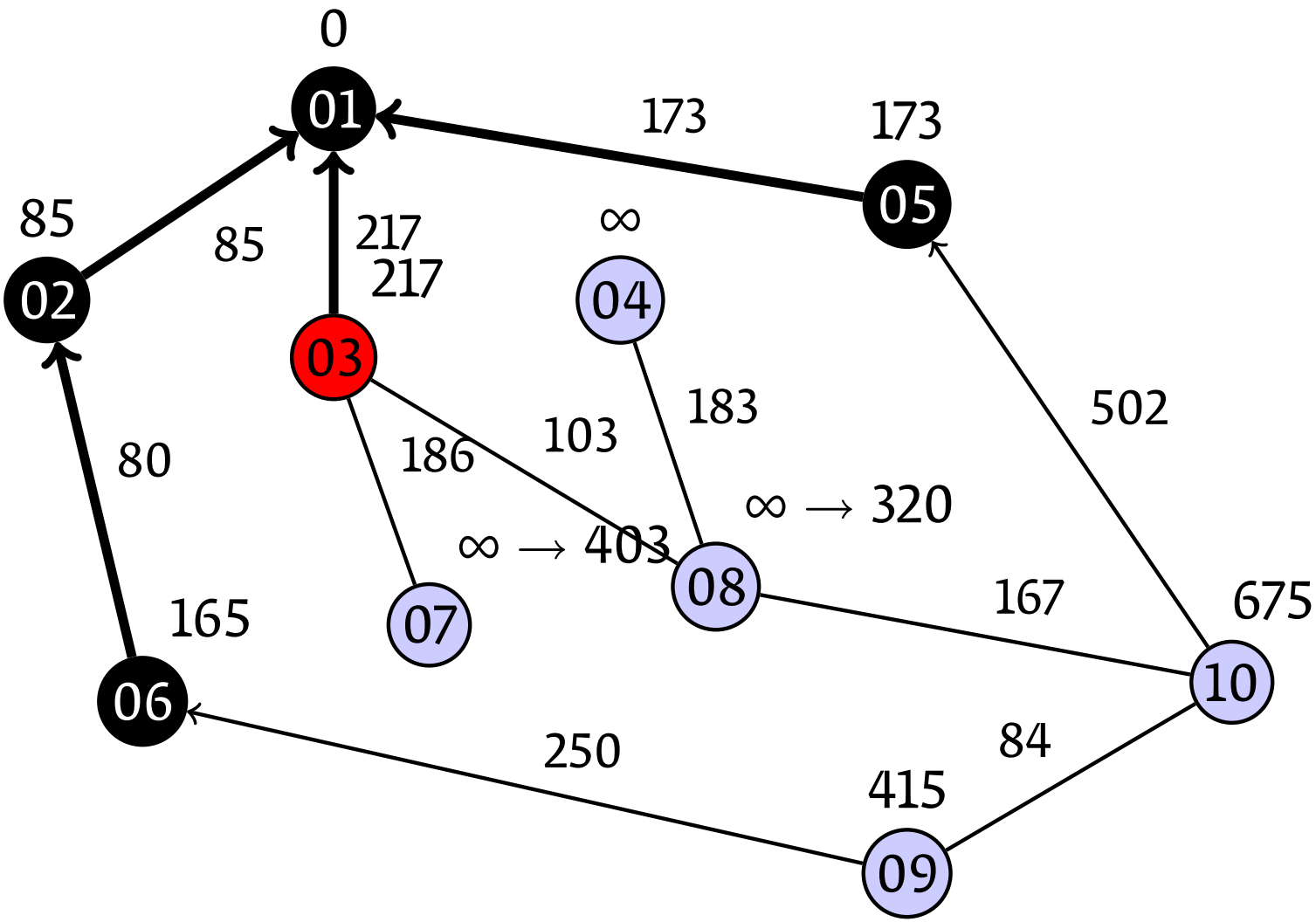
Exemple d'application de Dijkstra 3/10



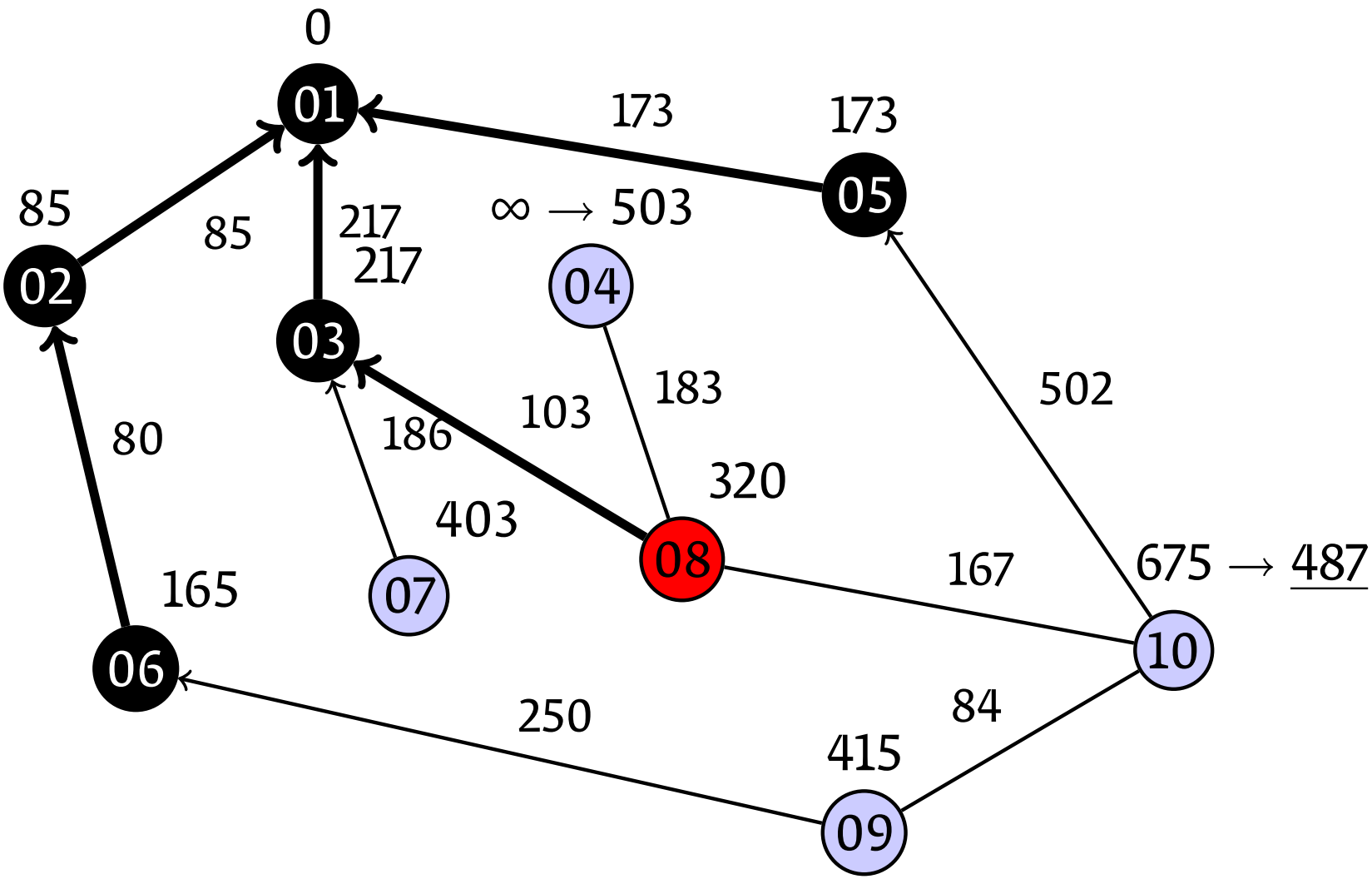
Exemple d'application de Dijkstra 4/10



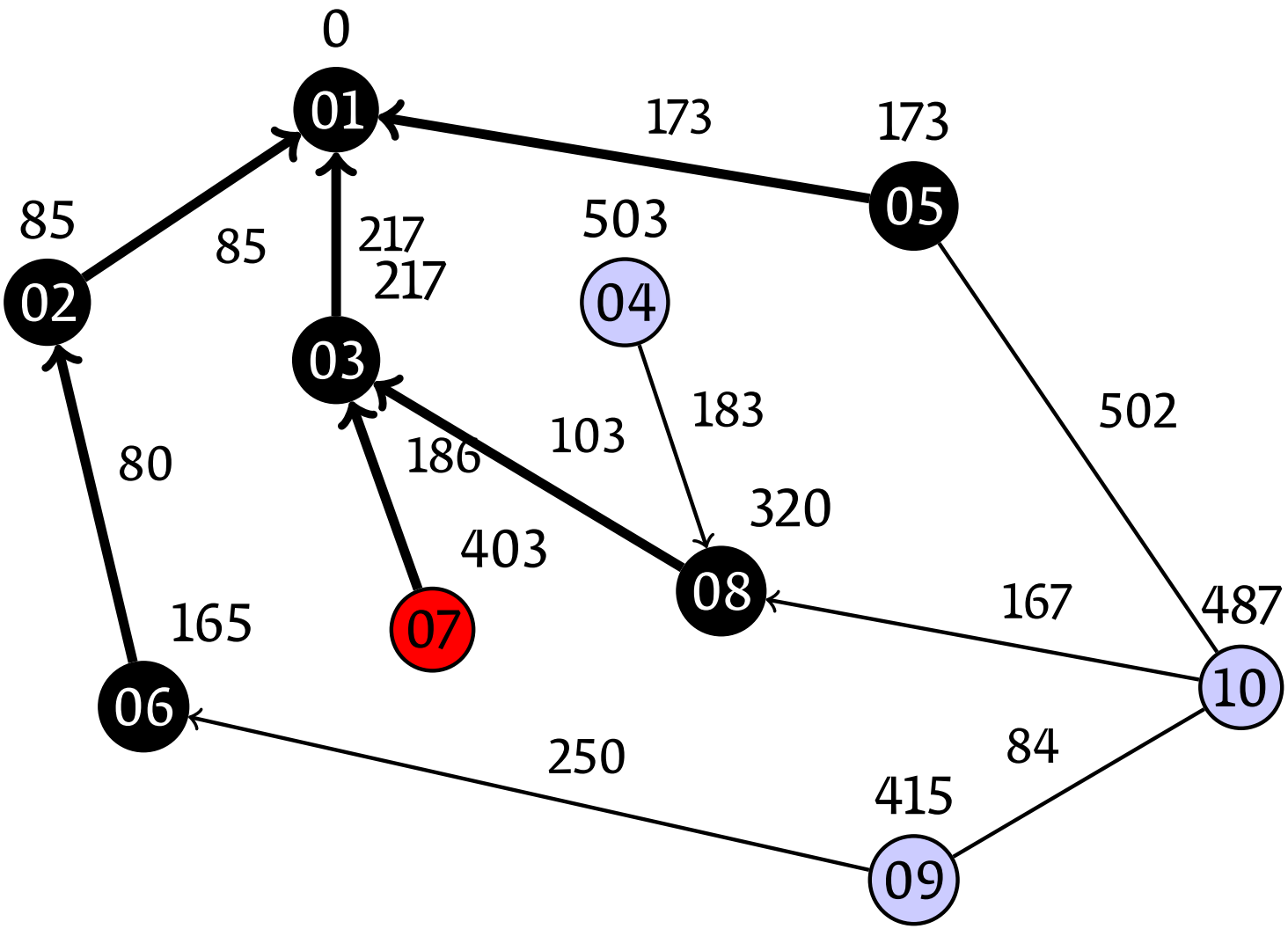
Exemple d'application de Dijkstra 5/10



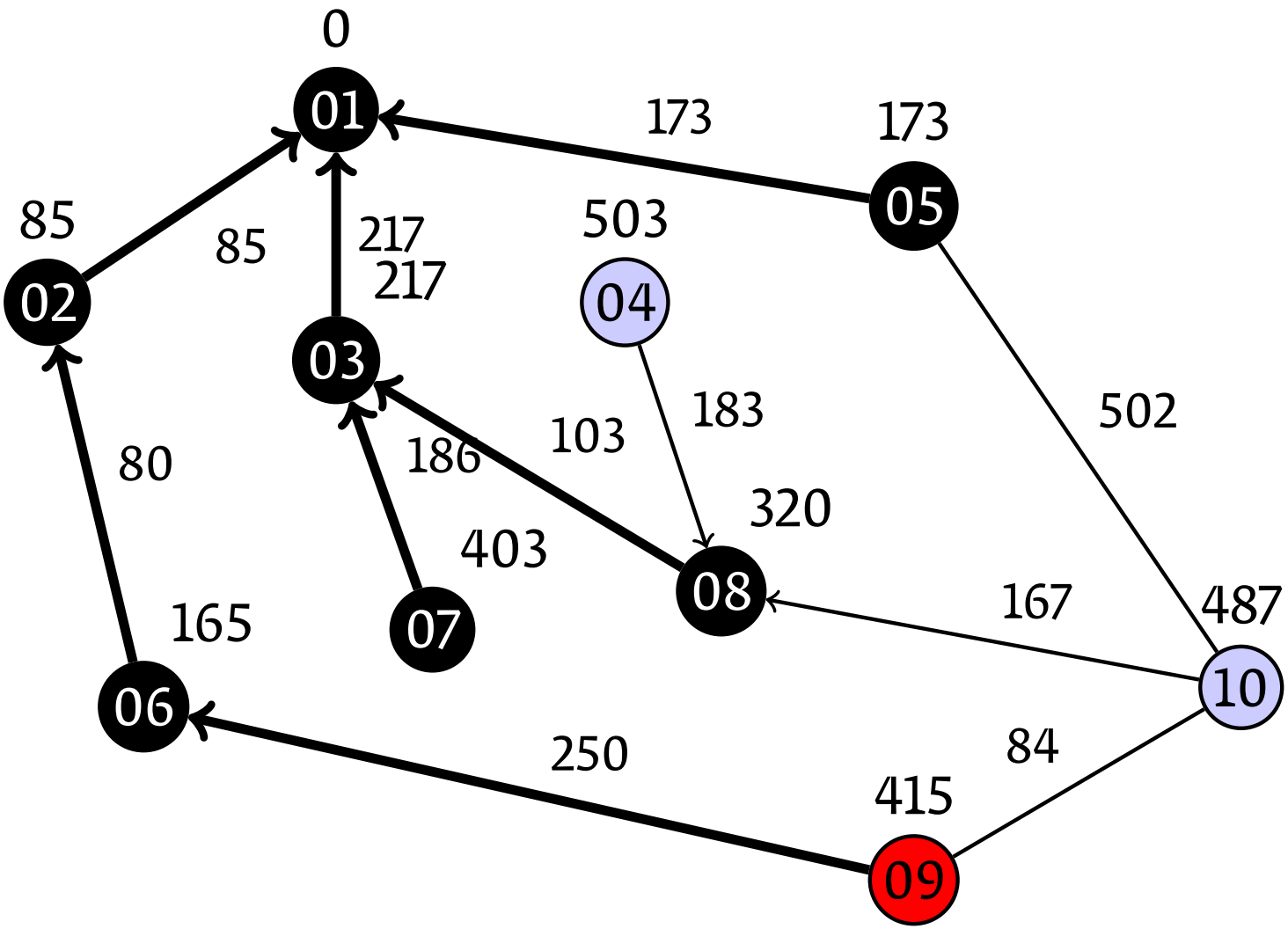
Exemple d'application de Dijkstra 6/10



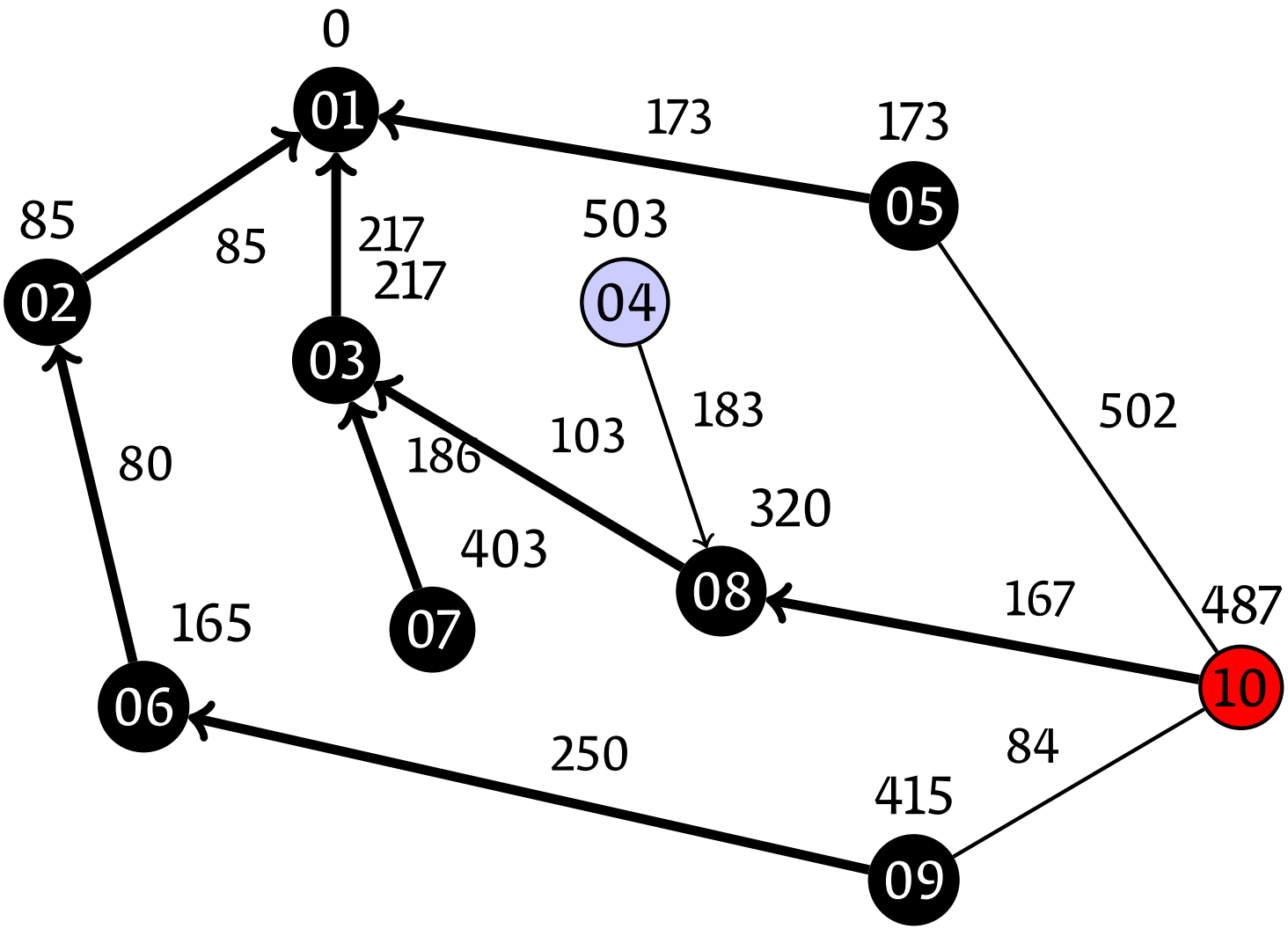
Exemple d'application de Dijkstra 7/10



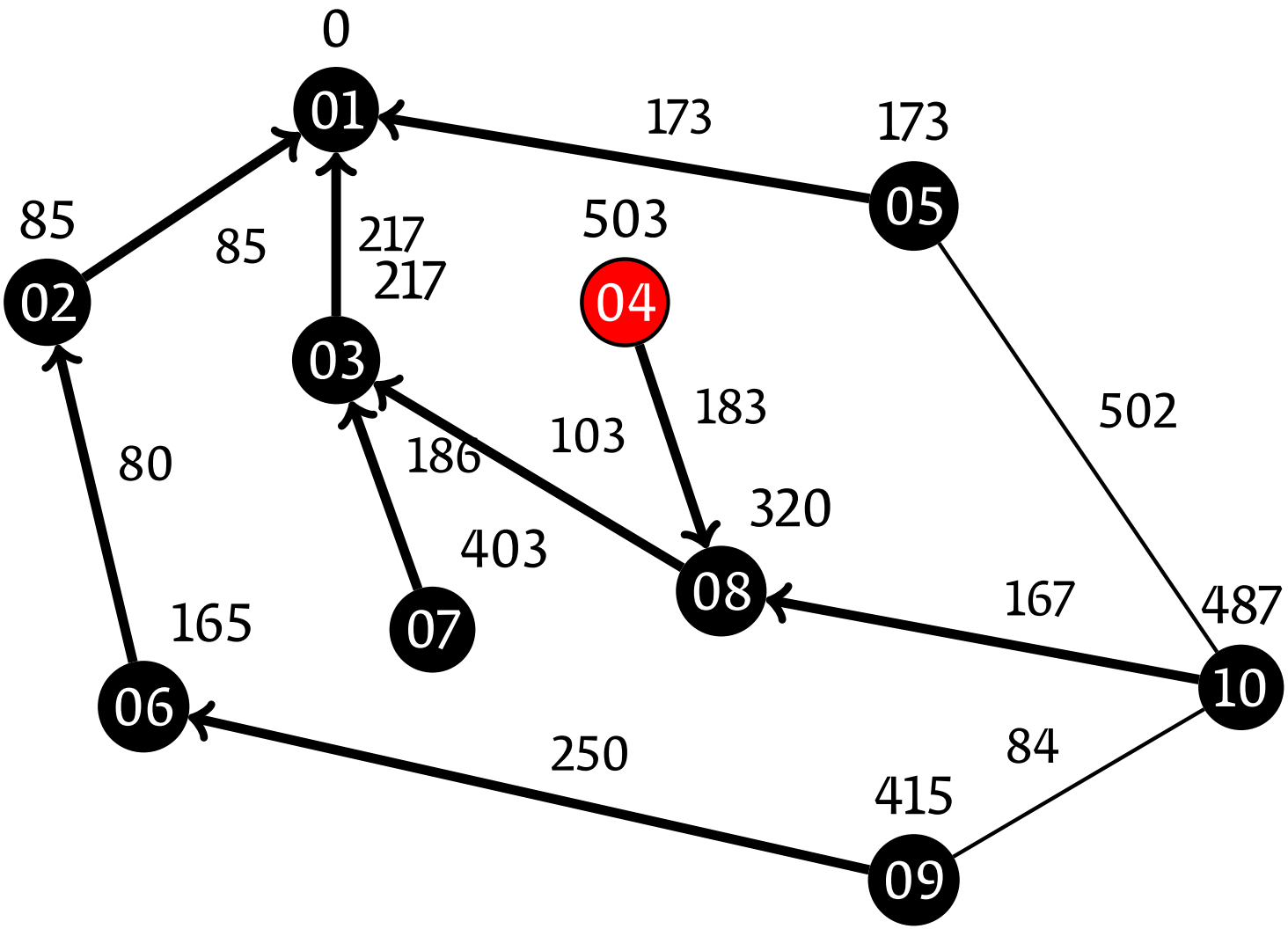
Exemple d'application de Dijkstra 8/10



Exemple d'application de Dijkstra 9/10



Exemple d'application de Dijkstra 10/10



Dijkstra sous graph-tool 1/2

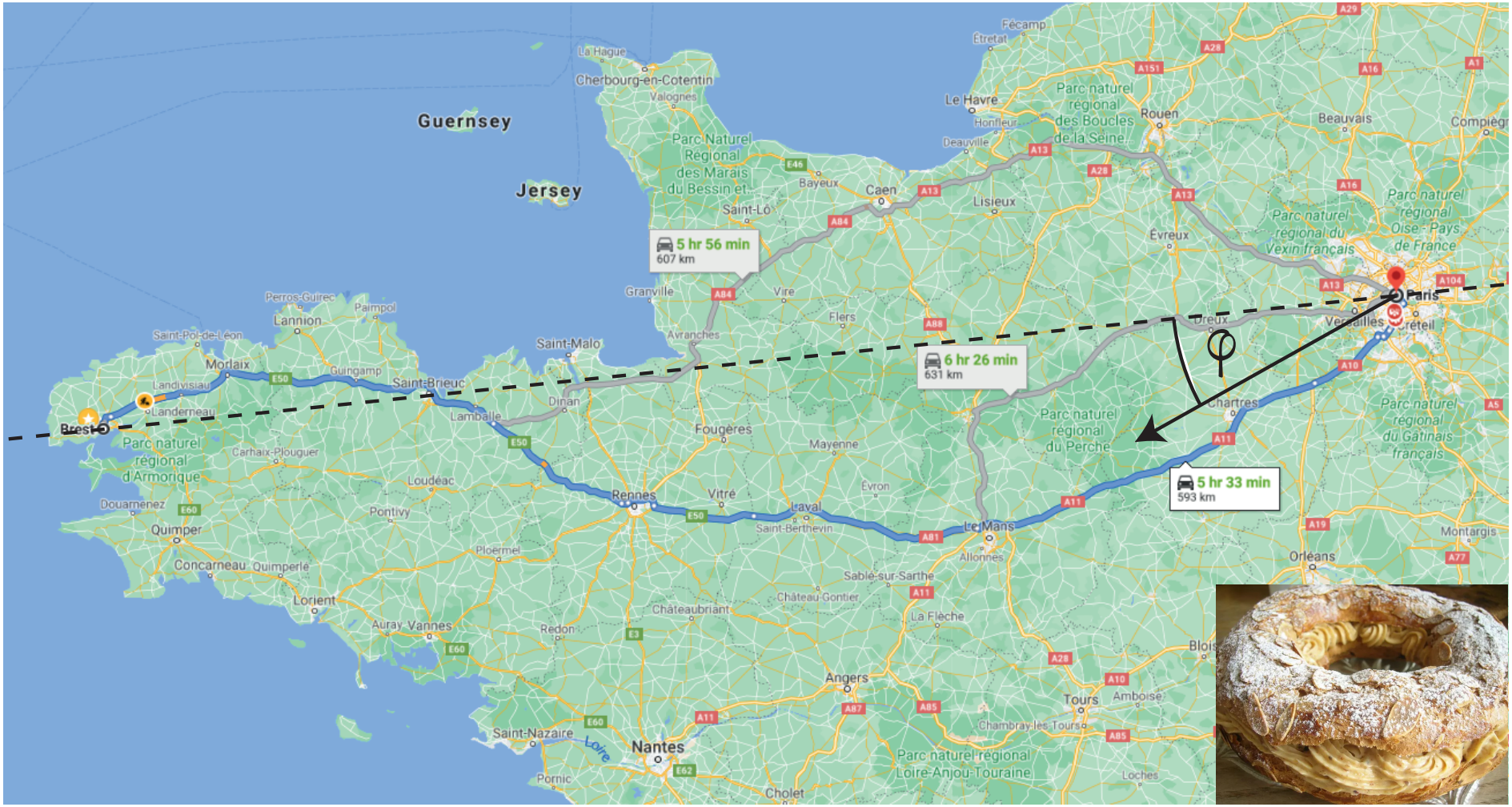
```
from graph_tool.all import *
g = Graph(directed=True)
weight = g.new_edge_property("int16_t")
g.add_vertex(10)
weights=[85,217,173,80,186,103,183,502,250,167,84]
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),(4,9),\
    (5,8),(7,9),(8,9)]
for ((a,b),w) in zip(my_edges,weights):
    e = g.add_edge(a,b)
    weight[e]=w
(dist,pred)=dijkstra_search(g,weight,source=g.vertex(0))
for v in g.vertices():
    print("d(",v,")=",dist[v],"pi(",v,")=",pred[v])
```

Le plus court chemin avec heuristique

- Bellman-Ford et Dijkstra nous ont permis de trouver les plus courts chemins entre un sommet et *tous* les autres sommets du graphe. Il peut y en avoir beaucoup…
- On se pose donc la question : comment faire pour ne parcourir qu'une partie du graphe et en dégager le plus court chemin entre deux sommets ?
- Une *heuristique* (du mot grec εὑρίσκειν = trouver, d'où le fameux « (h)eurêka »[1]) est une méthode qui permet de trouver rapidement un résultat raisonnable mais pas forcément optimal. Dans notre cas, la solution sera de toute façon optimale puisque Dijkstra fournit des optimaux *globaux* — l'heuristique nous permettra de trouver la réponse plus rapidement.

[1] Wikipédia : εὕρηκα est l'instant euphorique de soudaine compréhension et de certitude après une phase de tension…

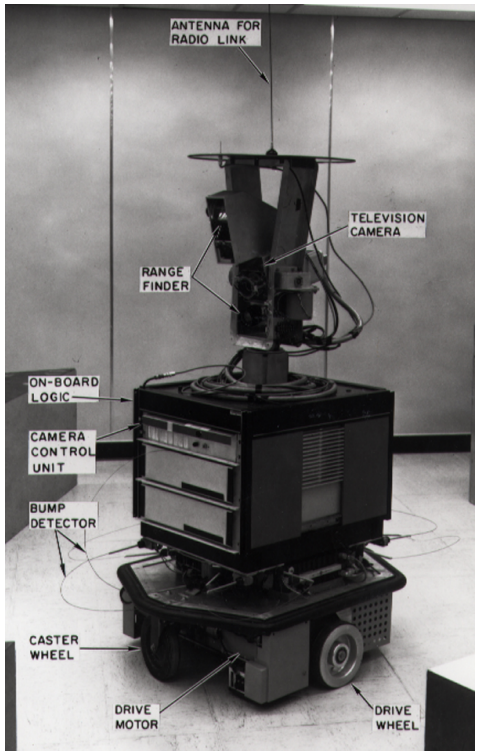
L'heuristique du Paris-Brest



$A^\star$ is born

- L'*algorithme $A^\star$* (prononcé « A-star ») calcule le plus court chemin pour aller d'un sommet s à un sommet t.
- Il généralise l'algorithme de Dijkstra en se servant d'une fonction heuristique h. Cette fonction doit être positive et nulle uniquement en t.

$A^\star(G, w, s, t)$:
  INITIALIZE-SINGLE-SOURCE$_d(G, s)$
  $S \leftarrow \emptyset$, $Q \leftarrow \{s\}$
  while $Q \neq \emptyset$:
    $u \leftarrow$ EXTRACT-MIN$_{d+h}(Q)$
    if $u = t$: **break**
    $S \leftarrow S \cup \{u\}$
    for chaque $v \in$ VOISINS(u) :
      if $v \notin S$:
        RELAX$_d(u, v, w)$
        $Q \leftarrow Q \cup \{v\}$

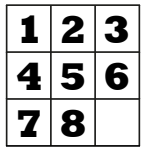
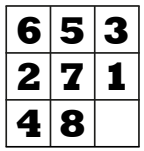


Le robot SHAKEY (1967) (ce n'est pas un dalek !)

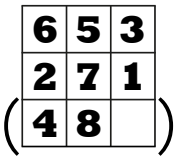
Heuristique admissible

- $A^\star$ va nous fournir un plus court chemin pour aller d'un sommet u à un sommet v si l'heuristique h est *admissible* et *monotone*.
- Une heuristique h est *admissible* quand elle ne surévalue pas la distance : c'est-à-dire si à partir de u on peut, en suivant les arêtes du graphe et en faisant au mieux, atteindre v en parcourant une distance $d(u, v)$ alors on doit avoir
$$h(u, v) \leq d(u, v).$$
- Une heuristique h est *monotone* si, pour un but v et des sommets u, u' on a toujours
$$h(u, v) \leq d(u, u') + h(u', v).$$
- Exemple typique : la distance à vol d'oiseau (et en l'absence de vent, rapace, avion, drone, usine polluante, chasseur, oiseau du sexe opposé, etc.).

Un exemple : le taquin 3×3

- Le *taquin* 3×3 est un cadre carré subdivisé en 9 cases carrées, où glissent 8 carreaux numérotés de 1 jusqu'à 8. Le jeu consiste à rétablir la configuration d'origine  à partir d'une configuration donnée, par exemple  . Exemple : `http://taquin.net/`
- Chaque configuration du taquin correspond à une permutation de l'ensemble $\{0, 1, \ldots, 8\}$ (où 0 correspond à la case vide).
- Il y a $9! = 362\,880$ telles permutations (ceci dit, toutes ne sont pas accessibles à partir de la configuration d'origine).

Taquinons !

- On peut considérer les configurations du taquin comme des sommets d'un graphe, dont les arêtes correspondent à des opérations de déplacement de carreau d'une case. Le poids de chaque arête est 1 (déplacement d'une case, horizontalement ou verticalement).
- Une heuristique possible est la *distance de Manhattan* entre une configuration donnée et la configuration d'origine : *on y compte pour chaque case (sauf la case vide) le nombre minimum de déplacements nécessaires pour que le carreau arrive à la case d'origine, et on prend la somme de ces déplacements*. (Par exemple, $h($  $) = 12$.)
- Autre heuristique : la *distance de Hamming* : *le nombre de carreaux mal placés*. On peut comparer les performances entre les deux heuristiques.

Taquinons !

- Nous avons effectué le calcul avec et sans heuristique (code Python disponible), voici les performances obtenues :

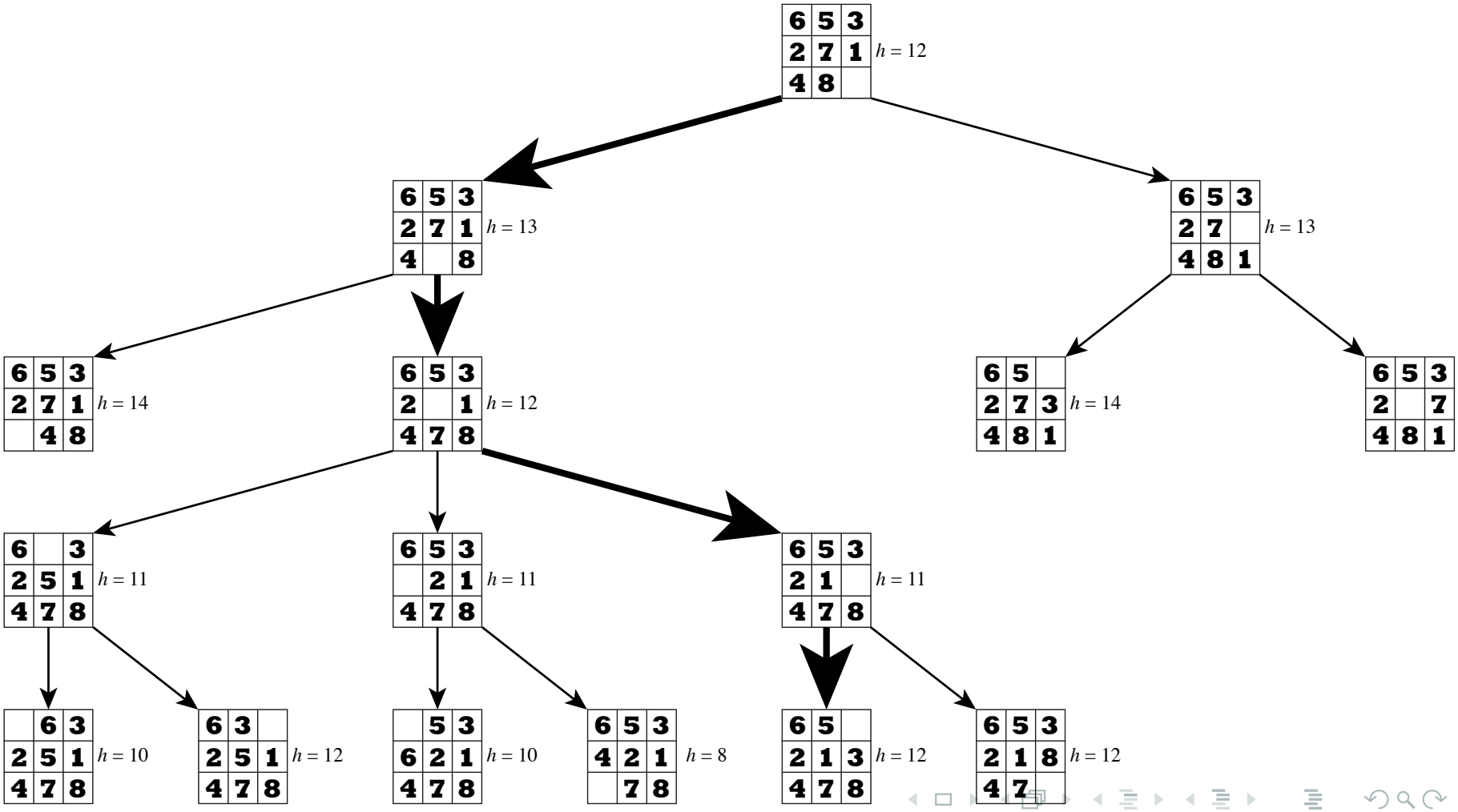
	Taille de S	Temps
Sans heuristique (Dijkstra)	18 715	31″
Avec heuristique h ($A^\star$)	167	2″

- Résultat :

$(6,5,3,2,7,1,4,8,0)$ $h=12$
$(6,5,3,2,7,1,4,0,8)$ $h=13$
$(6,5,3,2,0,1,4,7,8)$ $h=12$
$(6,5,3,2,1,0,4,7,8)$ $h=11$
$(6,5,0,2,1,3,4,7,8)$ $h=12$
$(6,0,5,2,1,3,4,7,8)$ $h=13$
$(0,6,5,2,1,3,4,7,8)$ $h=12$
$(2,6,5,0,1,3,4,7,8)$ $h=11$
$(2,6,5,1,0,3,4,7,8)$ $h=10$
$(2,0,5,1,6,3,4,7,8)$ $h=9$
$(2,5,0,1,6,3,4,7,8)$ $h=8$
$(2,5,3,1,6,0,4,7,8)$ $h=7$
$(2,5,3,1,0,6,4,7,8)$ $h=6$
$(2,0,3,1,5,6,4,7,8)$ $h=5$
$(0,2,3,1,5,6,4,7,8)$ $h=4$
$(1,2,3,0,5,6,4,7,8)$ $h=3$
$(1,2,3,4,5,6,0,7,8)$ $h=2$
$(1,2,3,4,5,6,7,0,8)$ $h=1$
$(1,2,3,4,5,6,7,8,0)$ $h=0$.

Taquinons !

- Voici le début du graphe de toutes les opérations possibles à partir de la configuration de départ (en gras, le début du plus court chemin) :



Arbres couvrants minimaux

- Dans plusieurs situations on a envie de simplifier un graphe, en limitant le nombre d'arêtes (et sans toucher aux sommets). Une manière élégante de filtrer les arêtes est de garder celles qui forment un arbre. On dira alors que l'on a un *arbre couvrant* du graphe.
- Parmi la pléthore d'arbres couvrants on s'intéresse parfois à celui dont le poids total est minimal. Exemple : un réseau de distribution d'eau, où le poids est donné par le coût des tuyaux.
- On cherche alors un *arbre couvrant minimal*.
- Vu le nombre exponentiel d'arbre couvrants, la force brute n'est pas la meilleure méthode pour en trouver un qui soit minimal. Nous allons considérer deux algorithmes qui permettent de le faire efficacement : ceux de Kruskal et de Prim.

Algorithme de Kruskal

- Soit G un graphe connexe, pondéré par un poids positif w. Voici l'*algorithme de Kruskal* :

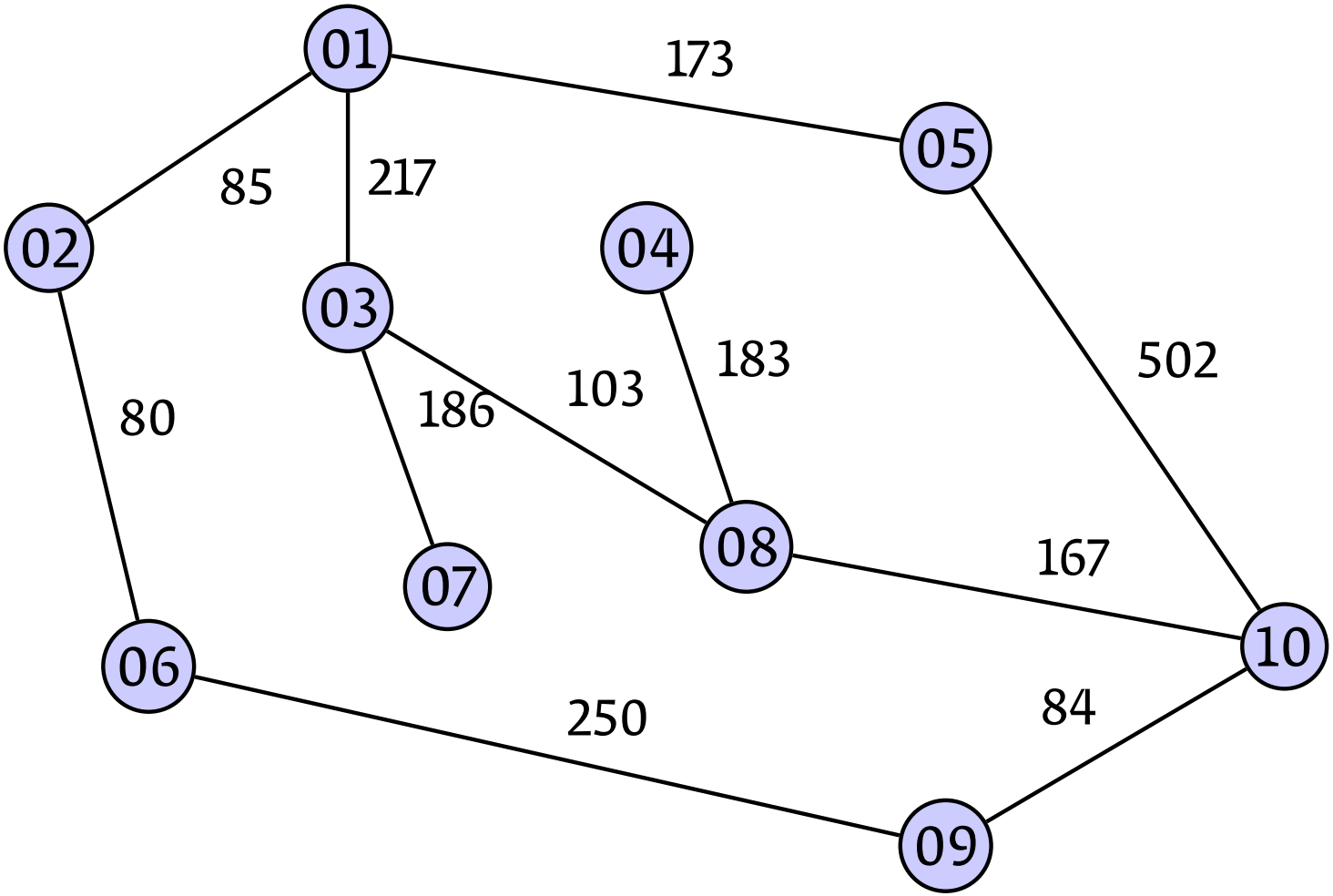
  KRUSKAL-MST(G, w) :
  $A \leftarrow \emptyset$
  trier les arêtes de G par ordre croissant de w
  for chaque arête uv prise dans cet ordre :
  if not MÊMECOMPOSANTECONNEXE(u, v) :
  $A \leftarrow A \cup \{uv\}$
  return A

- La fonction MÊMECOMPOSANTECONNEXE vérifie si les sommets u et v appartiennent à la même composante connexe du graphe A qui est en cours de construction.
- À la sortie de l'algorithme, A est un arbre couvrant minimal de G.

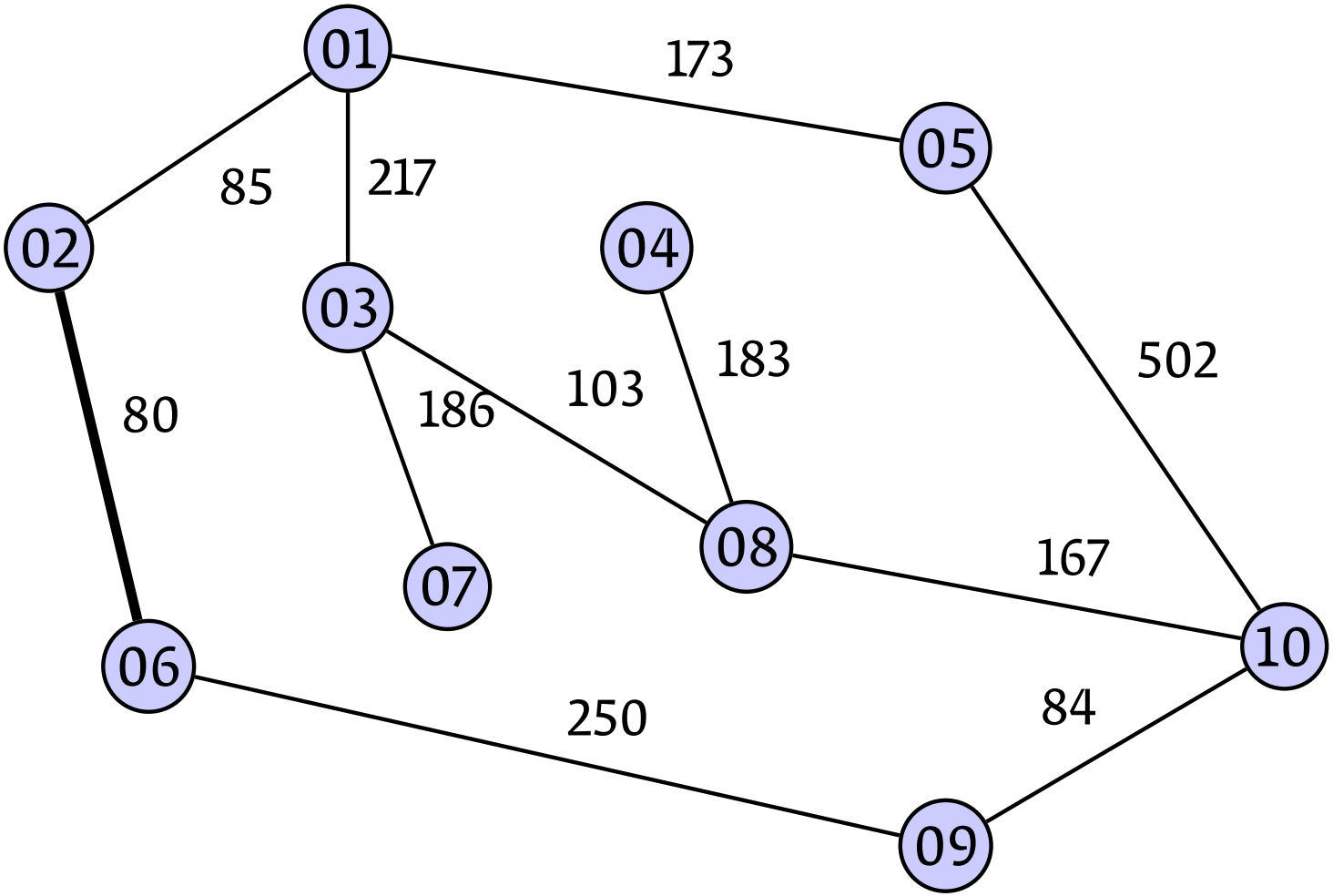
Algorithme de Kruskal

- Autrement dit : on ajoute des arêtes par ordre de coût, et seulement si elles ne forment pas de cycle (c'est-à-dire : si les deux sommets ne sont pas déjà sur la même composante connexe).
- `http://students.ceid.upatras.gr/~papagel/english/java_docs/minKrusk.htm`
- Complexité : on montre qu'en utilisant les forêts d'arbres enracinés, on arrive à une complexité de $O(E \log V)$.
- Avantage : facile à décrire et à implémenter.
- Désavantage : *pas de connexité garantie* avant la dernière itération.

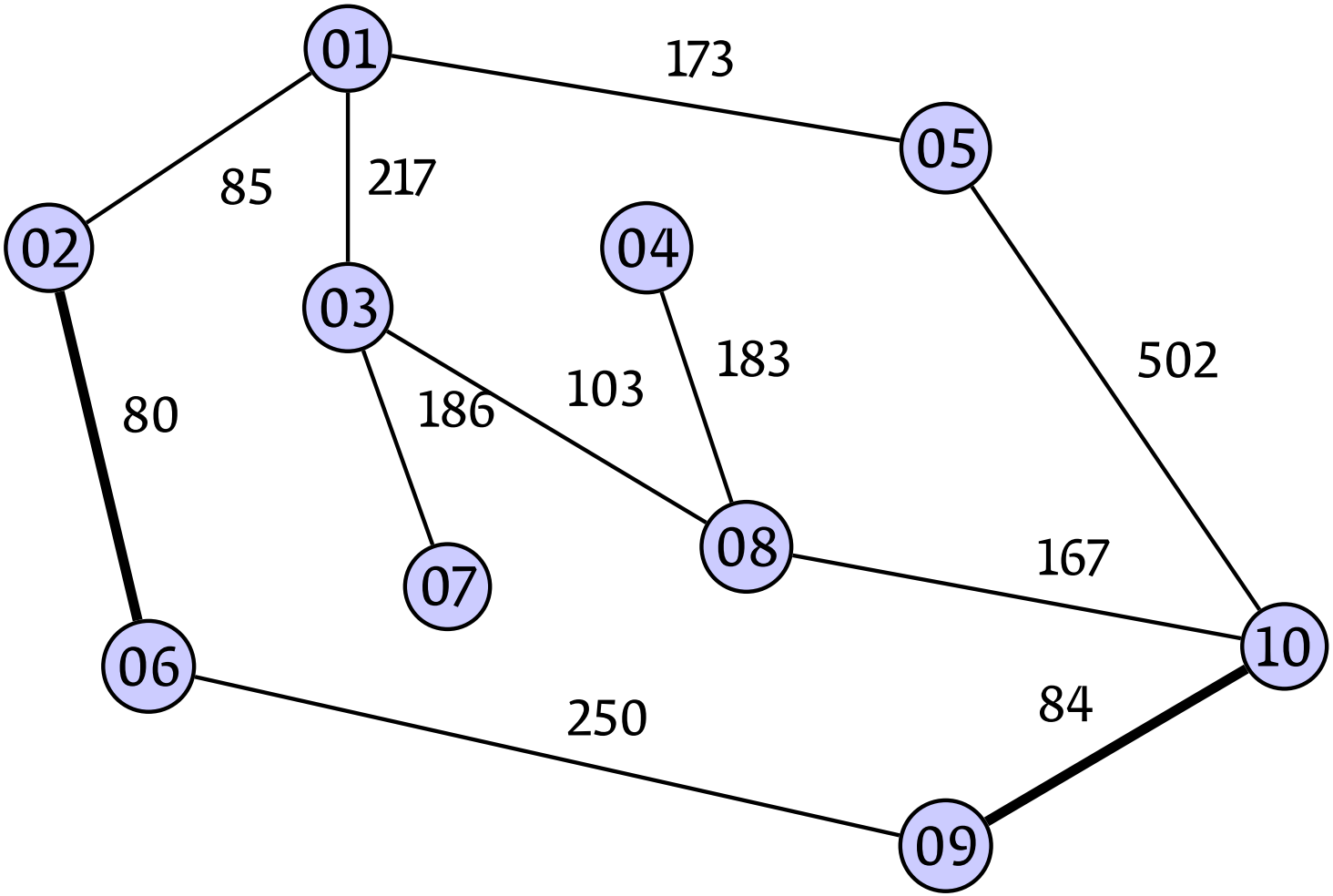
Exemple d'application de Kruskal 1/10



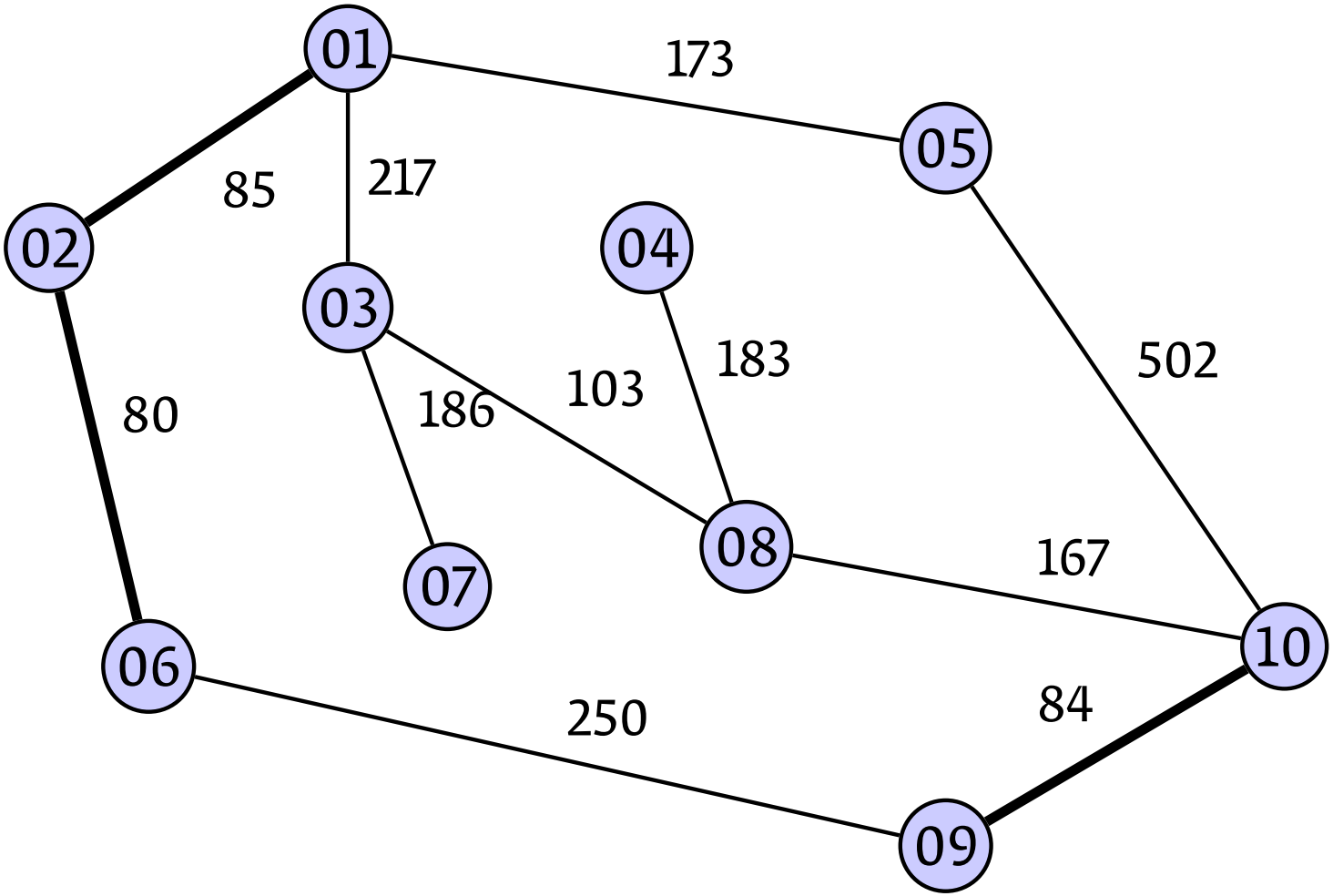
Exemple d'application de Kruskal 2/10



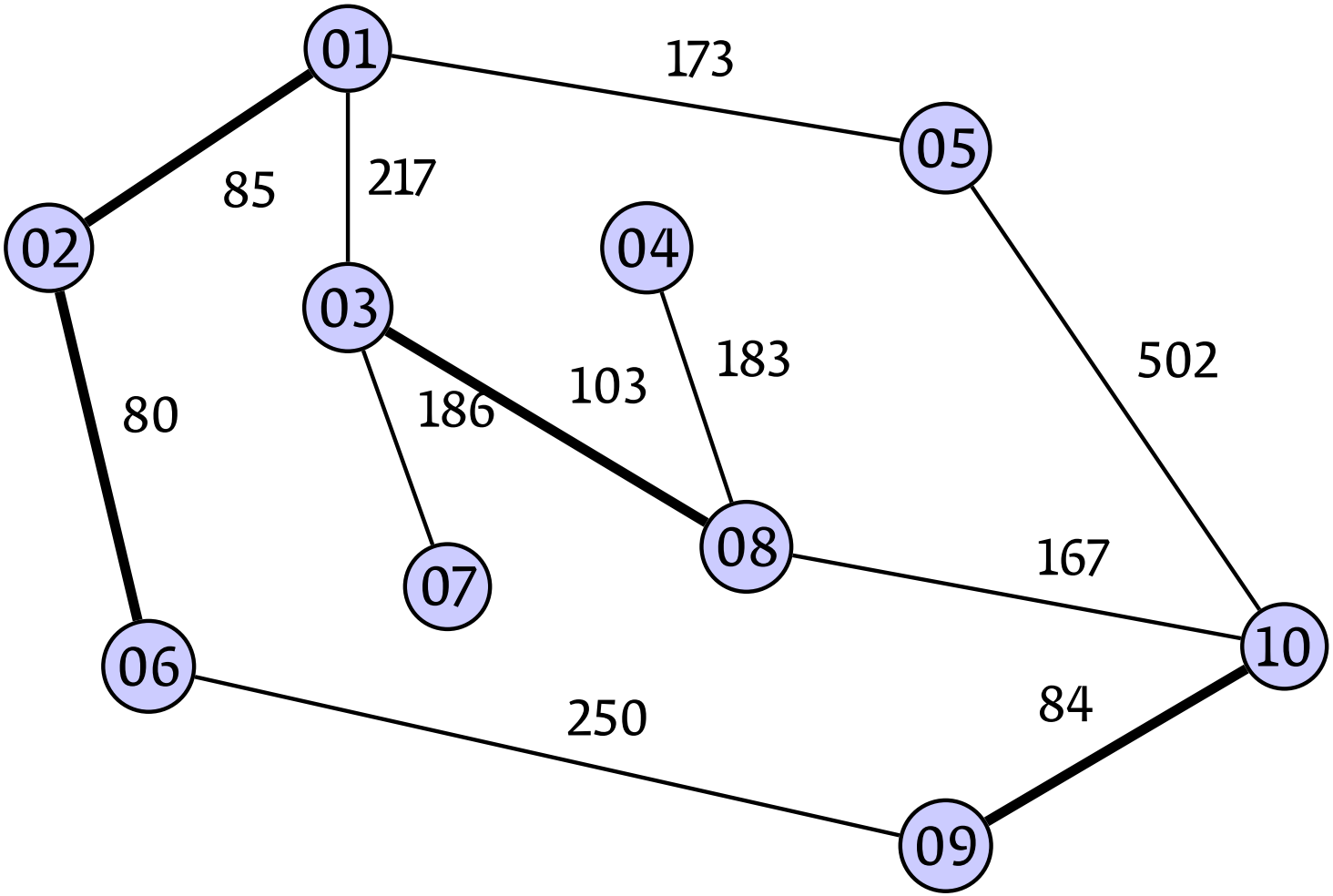
Exemple d'application de Kruskal 3/10



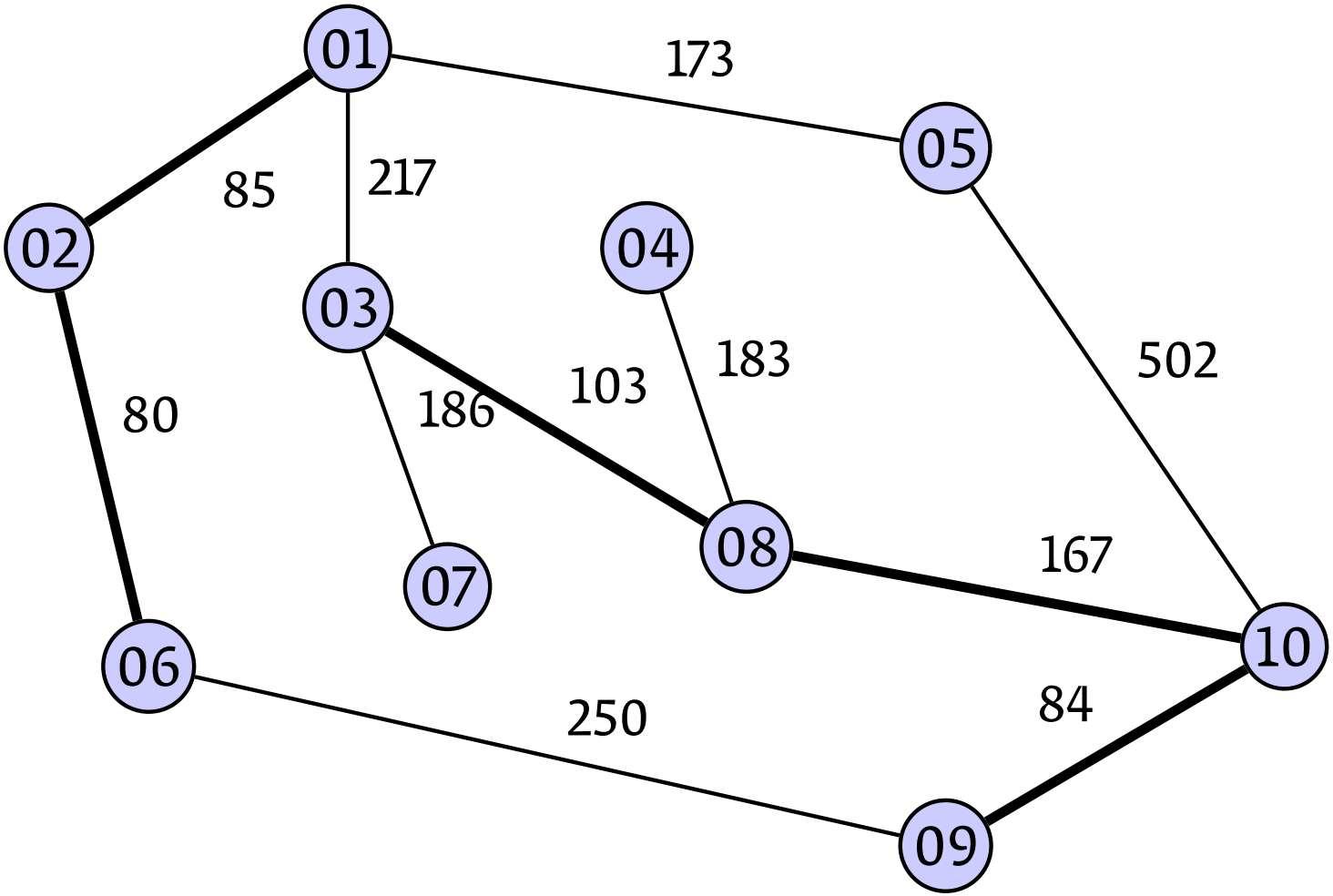
Exemple d'application de Kruskal 4/10



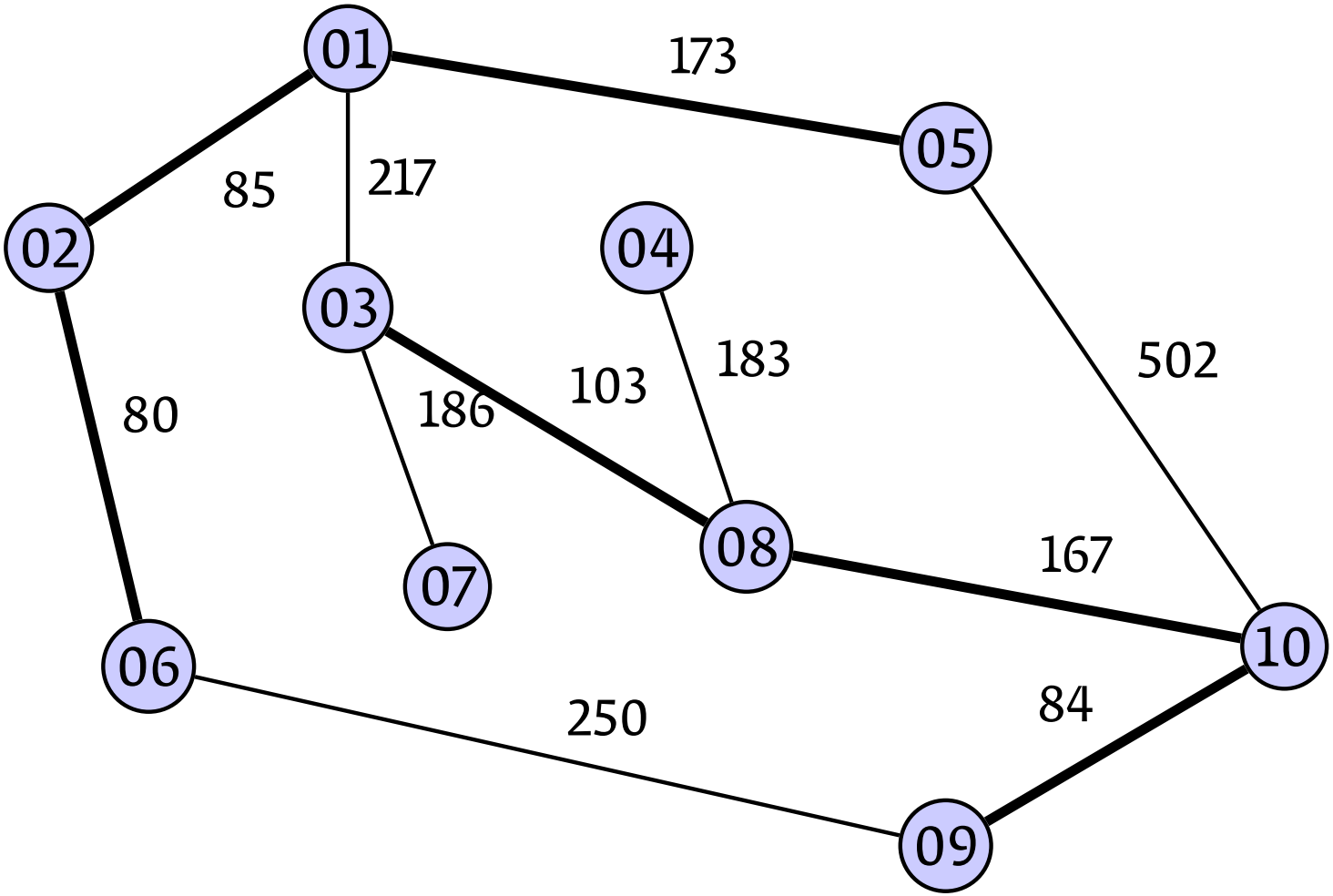
Exemple d'application de Kruskal 5/10



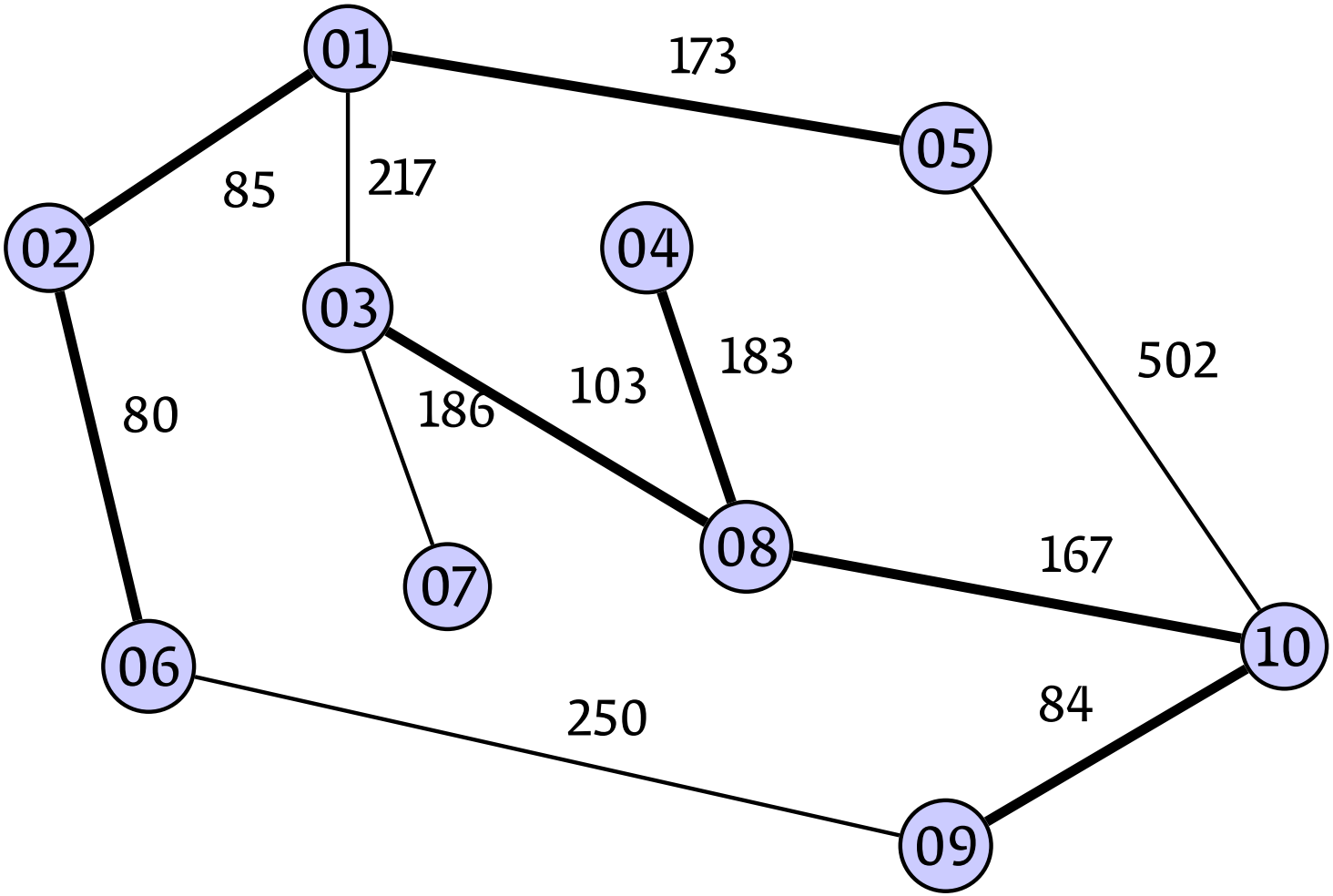
Exemple d'application de Kruskal 6/10



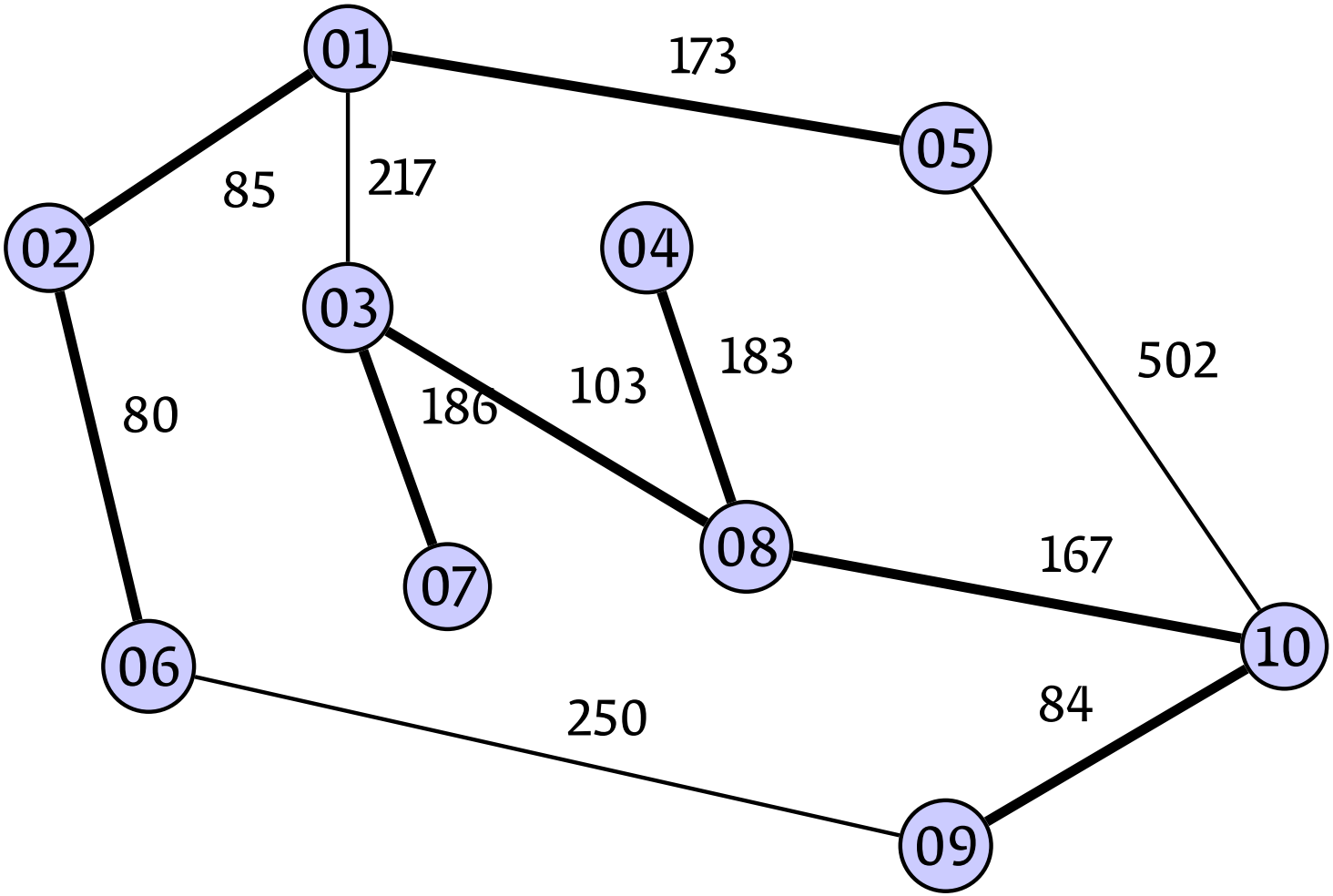
Exemple d'application de Kruskal 7/10



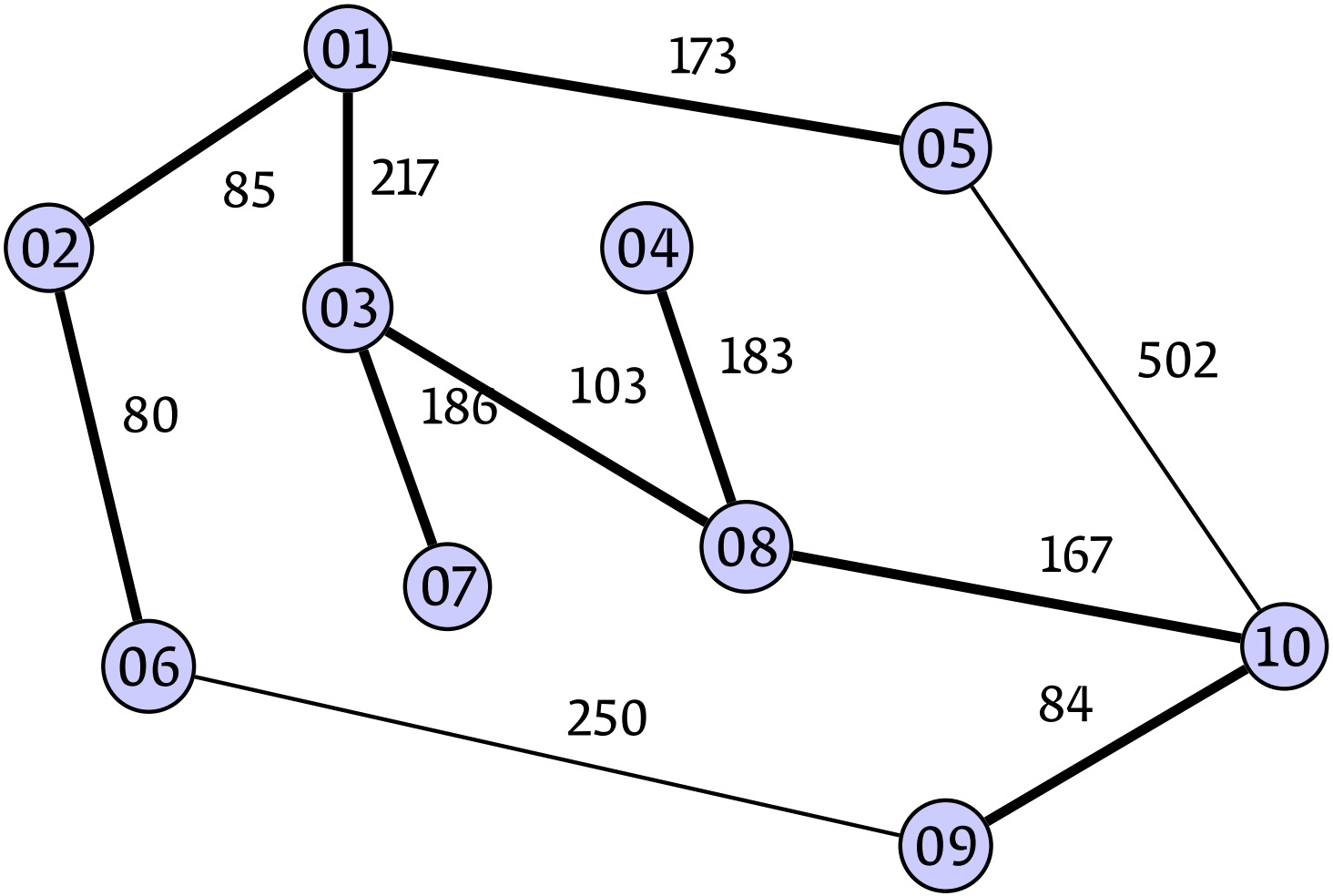
Exemple d'application de Kruskal 8/10



Exemple d'application de Kruskal 9/10



Exemple d'application de Kruskal 10/10



Kruskal sous graph-tool

```
from graph_tool.all import *
g = Graph(directed=True)
weight = g.new_edge_property("int16_t")
g.add_vertex(10)
weights=[85,217,173,80,186,103,183,502,250,167,84]
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),\
    (4,9),(5,8),(7,9),(8,9)]
for ((a,b),w) in zip(my_edges,weights):
    e = g.add_edge(a,b)
    weight[e]=w
pr=min_spanning_tree(g,weights=weight)
for e in g.edges():
    if (pr[e]==1):
        print(e)
```

qui donne :
(0, 1), (0, 2), (0, 4), (1, 5), (2, 6), (2, 7), (7, 3), (7, 9), (8, 9)

Algorithme de Prim

- Cette fois, on choisit un sommet r, on attache à tous les sommets des valeurs appelées *clés*, et on construit l'arbre A en ajoutant des arêtes (sans former de cycle) en choisissant toujours celle qui relie A avec le sommet de clé minimum.
- À chaque choix de nouveau sommet, on procède à des (pseudo-)relaxations des clés des sommets voisins.
- `http://students.ceid.upatras.gr/~papagel/project/prim.htm`
- On a volontairement écrit le pseudo-code de Prim de manière à montrer le lien de parenté avec l'algorithme de Dijkstra :

Algorithme de Prim

- Soit Q une file de priorité min.

PRIM-MST(G, w, r) :
  INITIALIZE-SINGLE-SOURCE$_\phi(G, r)$:
  $A \leftarrow \{r\}$, $Q \leftarrow V$
  while $Q \neq \emptyset$:
    $u \leftarrow$ EXTRACT-MIN$_\phi(Q)$
    $A \leftarrow A \cup \{u\}$
    for chaque $v \in$ VOISINS$(u) \cap Q$:
      PSEUDORELAX$_\phi(u, v, w)$

où PSEUDORELAX$_\phi$ est définie de la manière suivante :

PSEUDORELAX$_\phi(u, v, w)$:
  if $\phi(v) > w(u, v)$:
    $\phi(v) \leftarrow w(u, v)$
    $\pi(v) \leftarrow u$

Comparaison entre Dijkstra et Prim

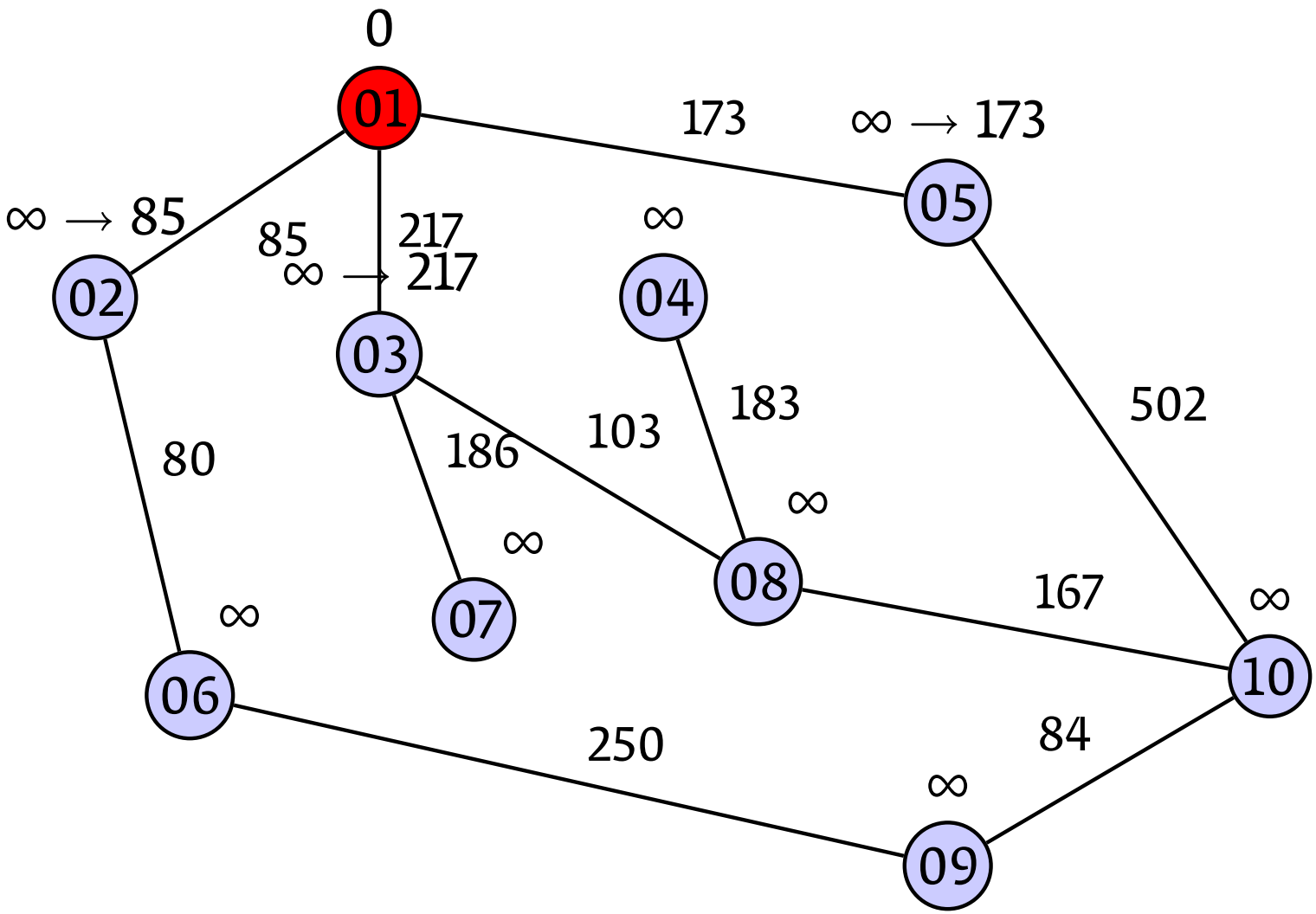
- La différence essentielle est celle entre RELAX et PSEUDORELAX :

RELAX$_d(u, v, w)$:
  if $d(v) > \underline{d(u)} + w(u, v)$:
    $d(v) \leftarrow \underline{d(u)} + w(u, v)$
    $\pi(v) \leftarrow u$

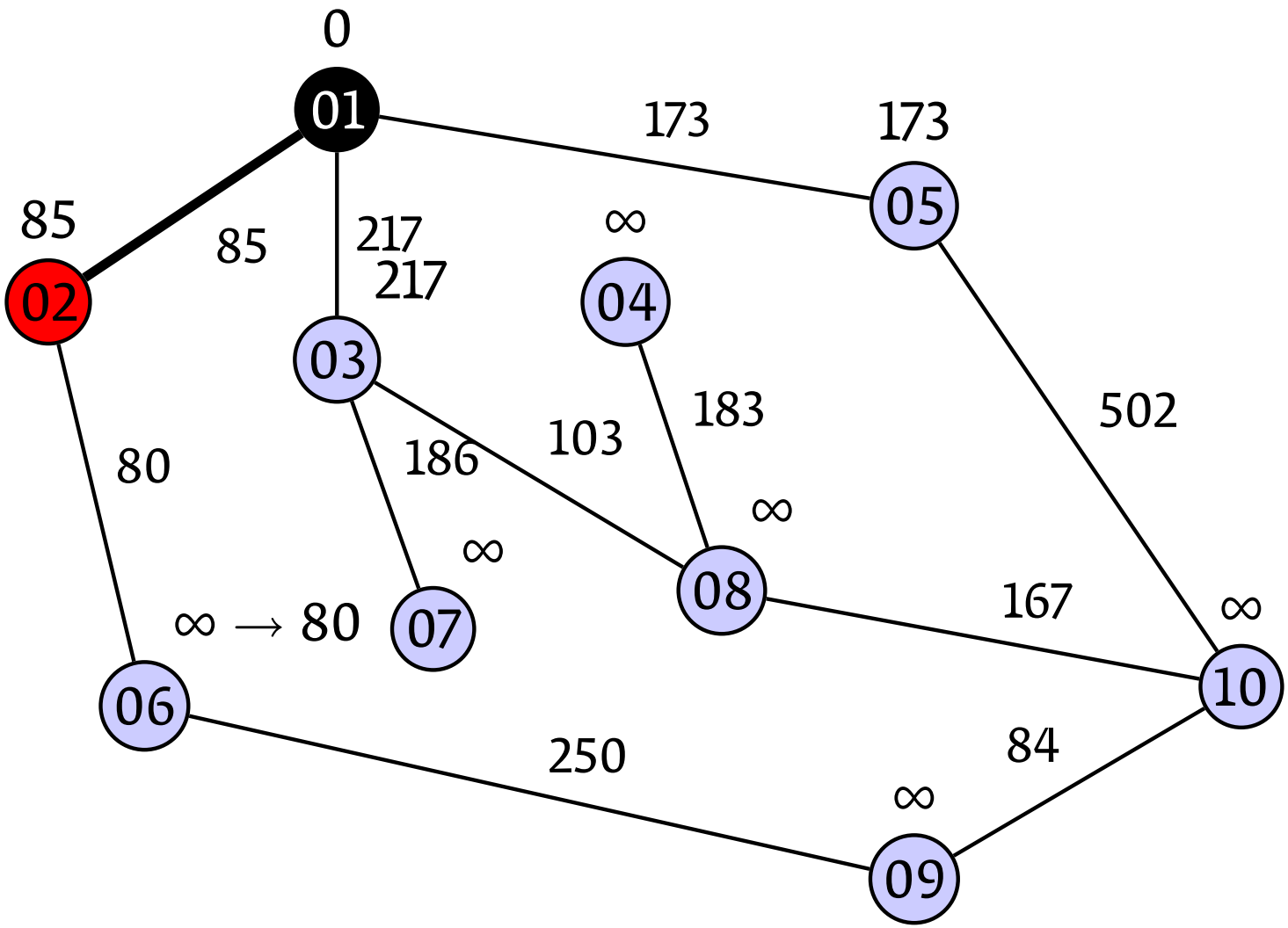
PSEUDORELAX$_\phi(u, v, w)$:
  if $\phi(v) > w(u, v)$:
    $\phi(v) \leftarrow w(u, v)$
    $\pi(v) \leftarrow u$

- Dans un cas, $d(v)$ récupère la valeur de $d(u)$ (plus celle de $w(u, v)$), ainsi la distance à partir du sommet d'origine est *répercutée à travers le graphe*.
- Dans l'autre cas, la valeur de $\phi(v)$ ne dépend que des arêtes adjacentes au sommet v, le reste du graphe n'intervient pas.

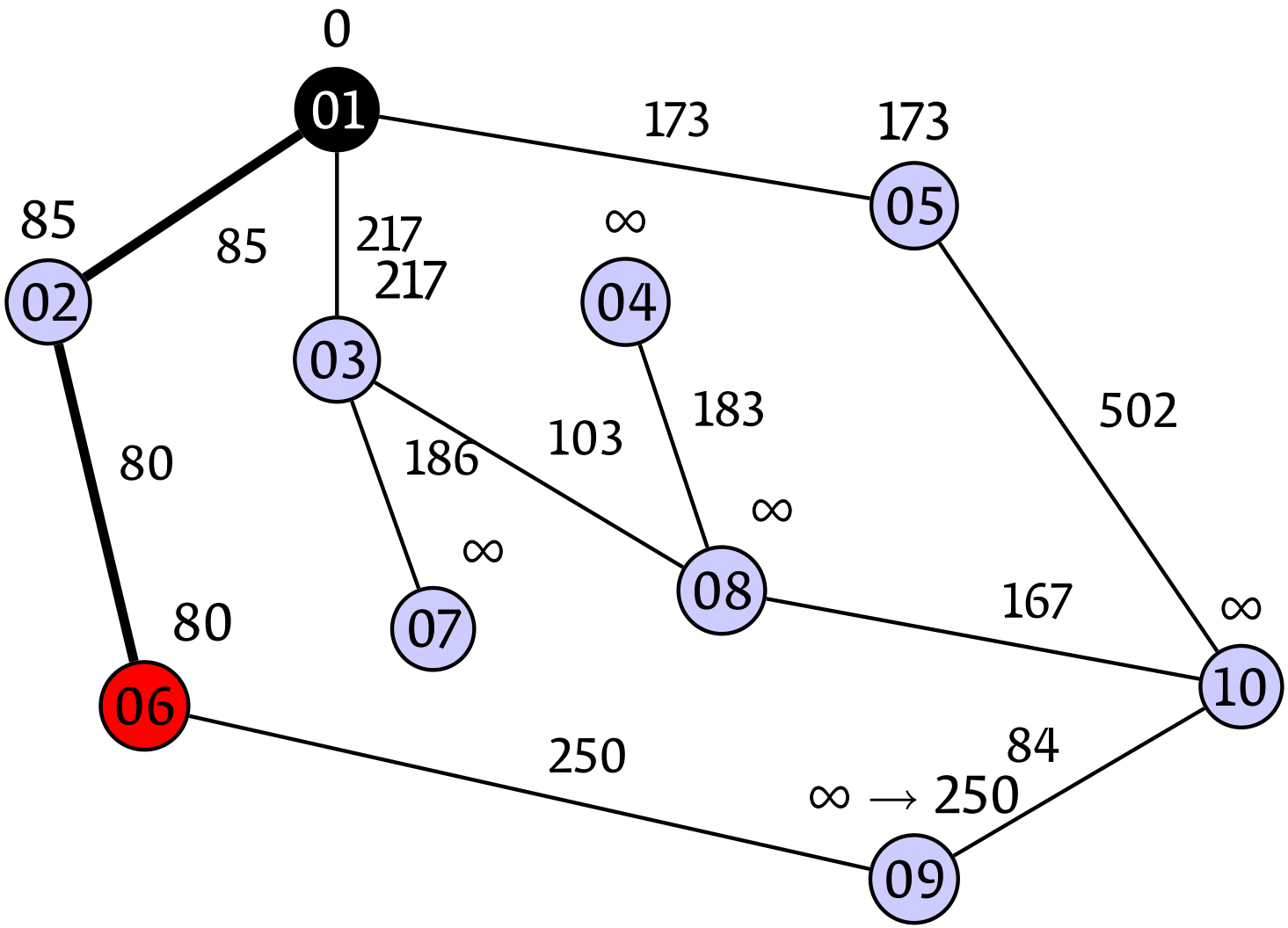
Exemple d'application de Prim 1/10



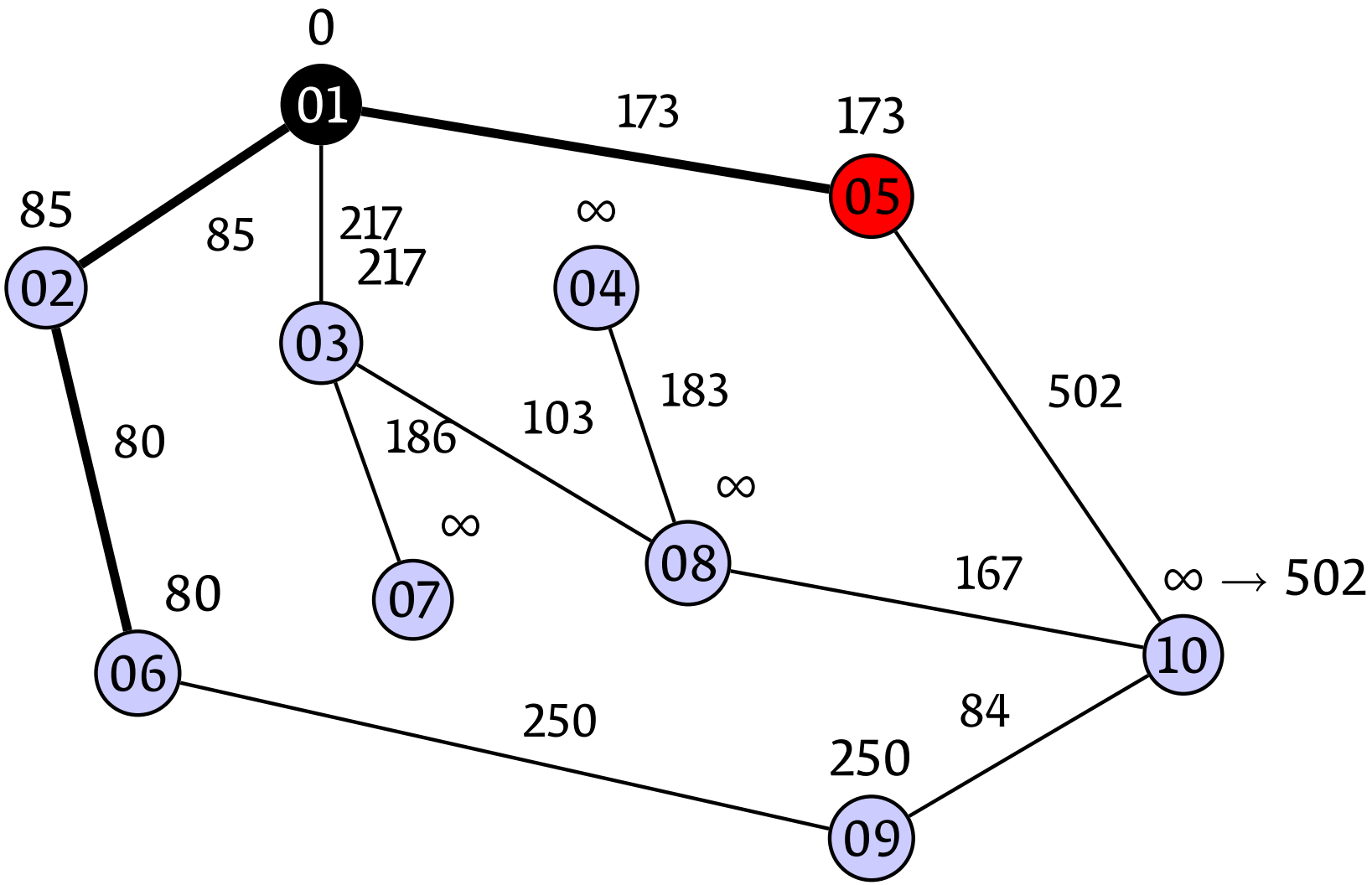
Exemple d'application de Prim 2/10



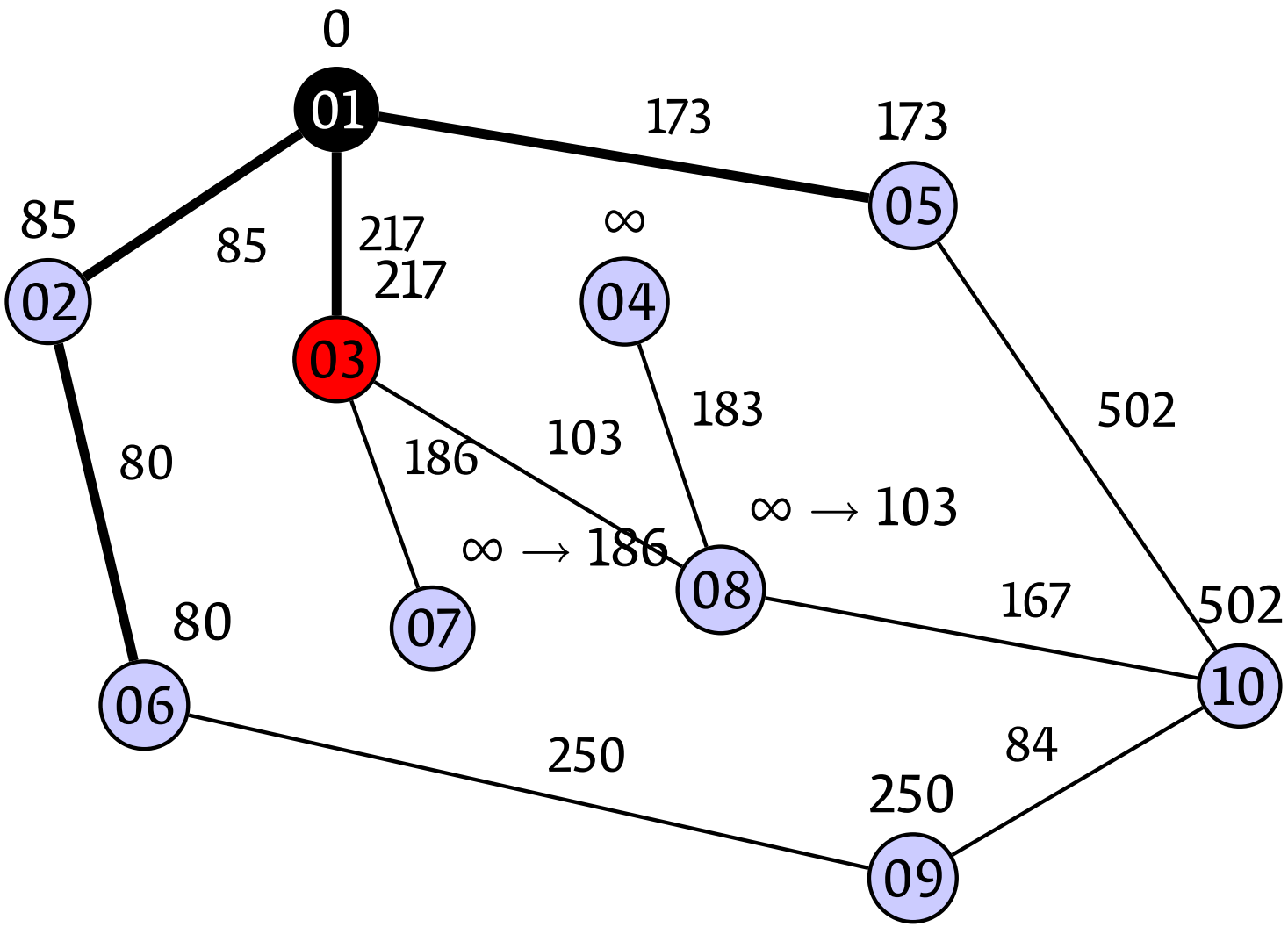
Exemple d'application de Prim 3/10



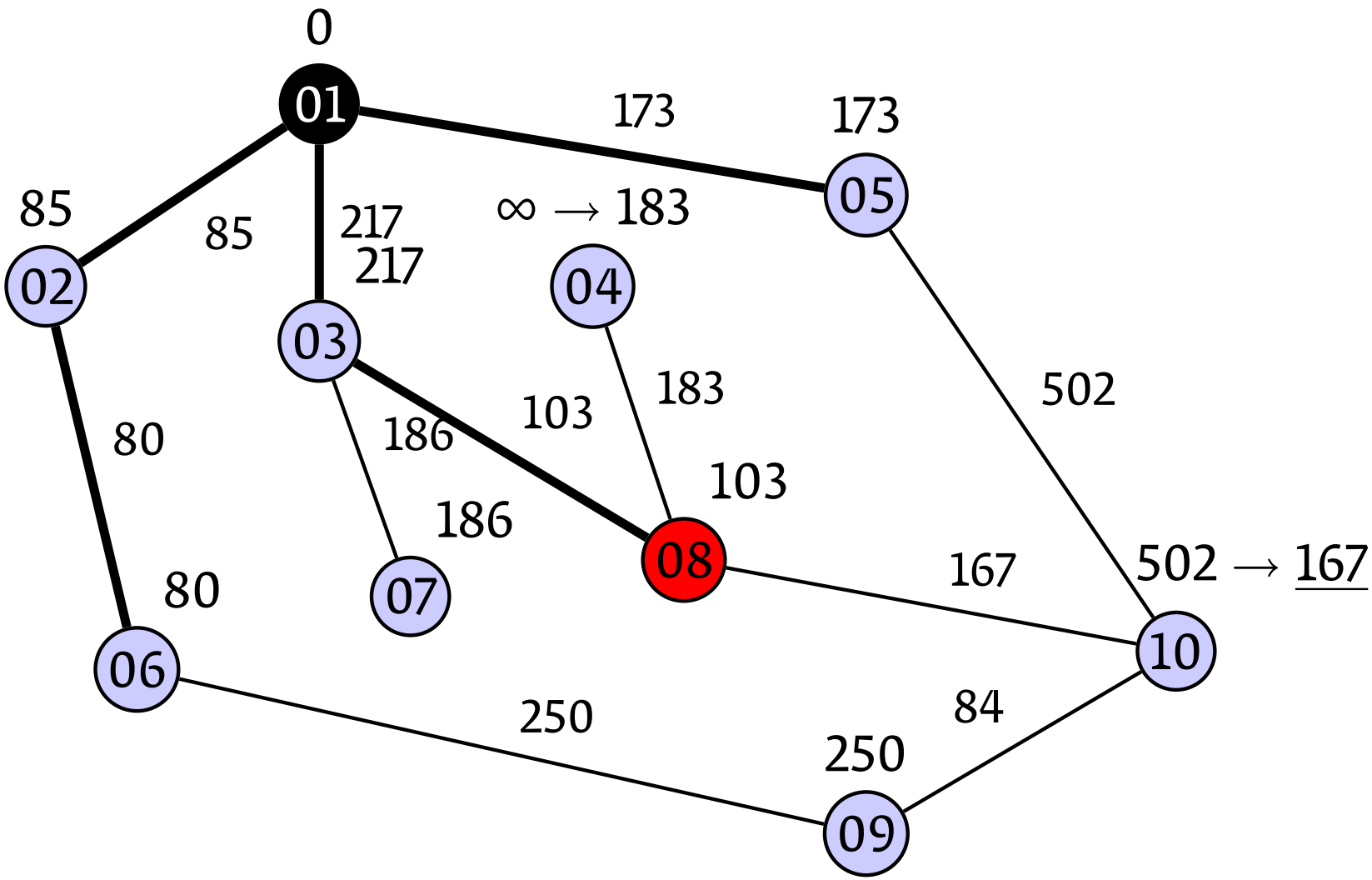
Exemple d'application de Prim 4/10



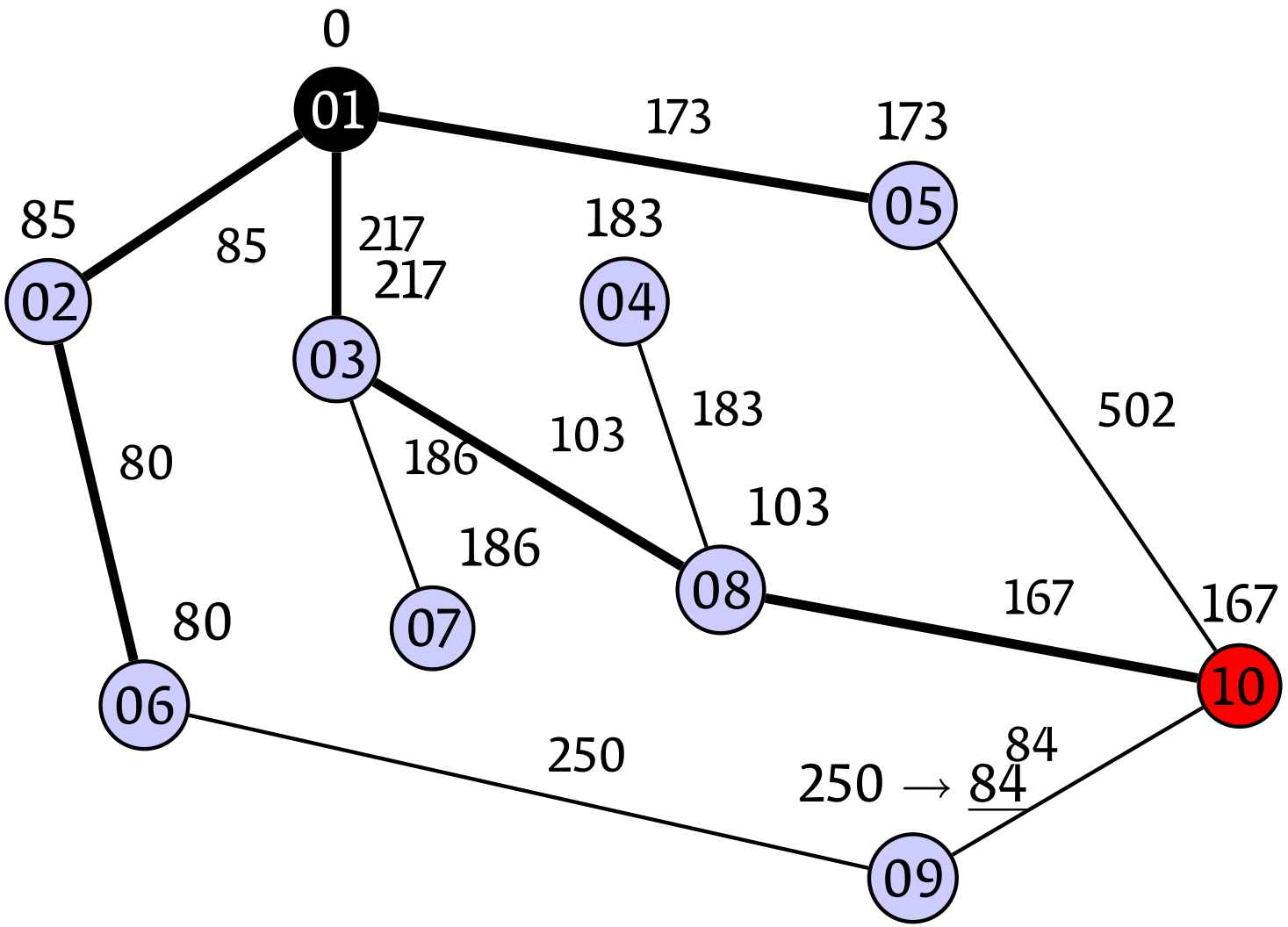
Exemple d'application de Prim 5/10



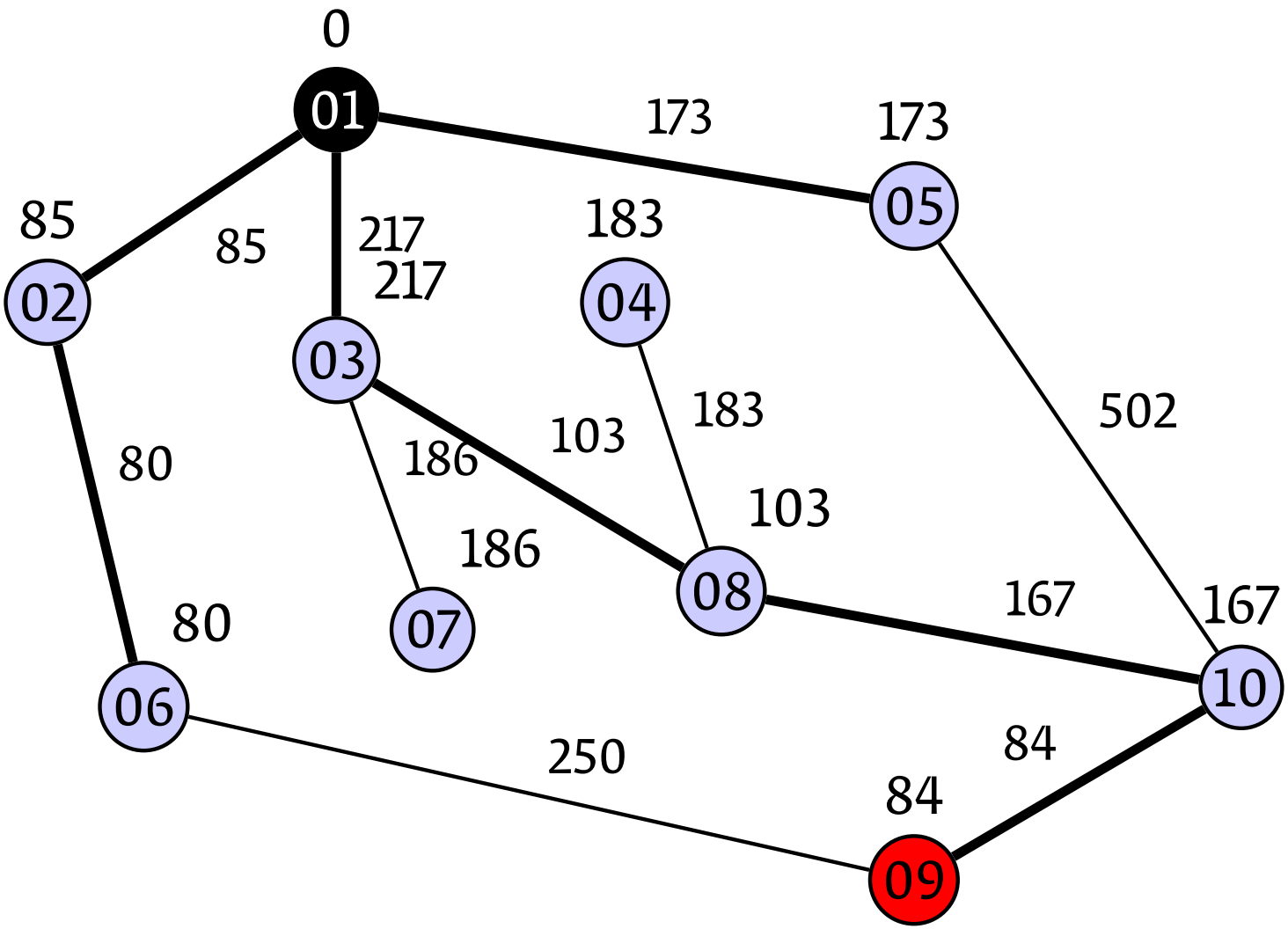
Exemple d'application de Prim 6/10



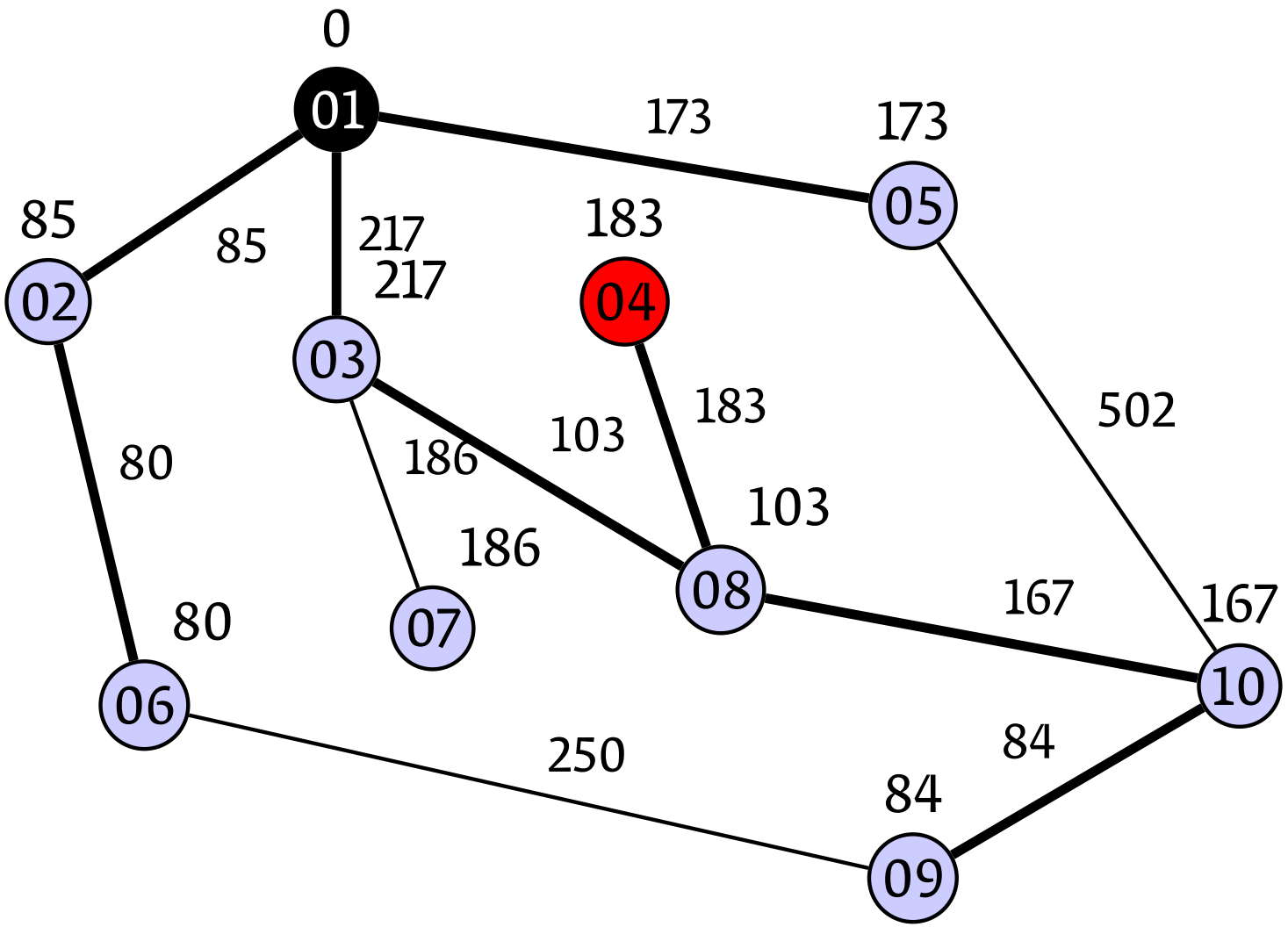
Exemple d'application de Prim 7/10



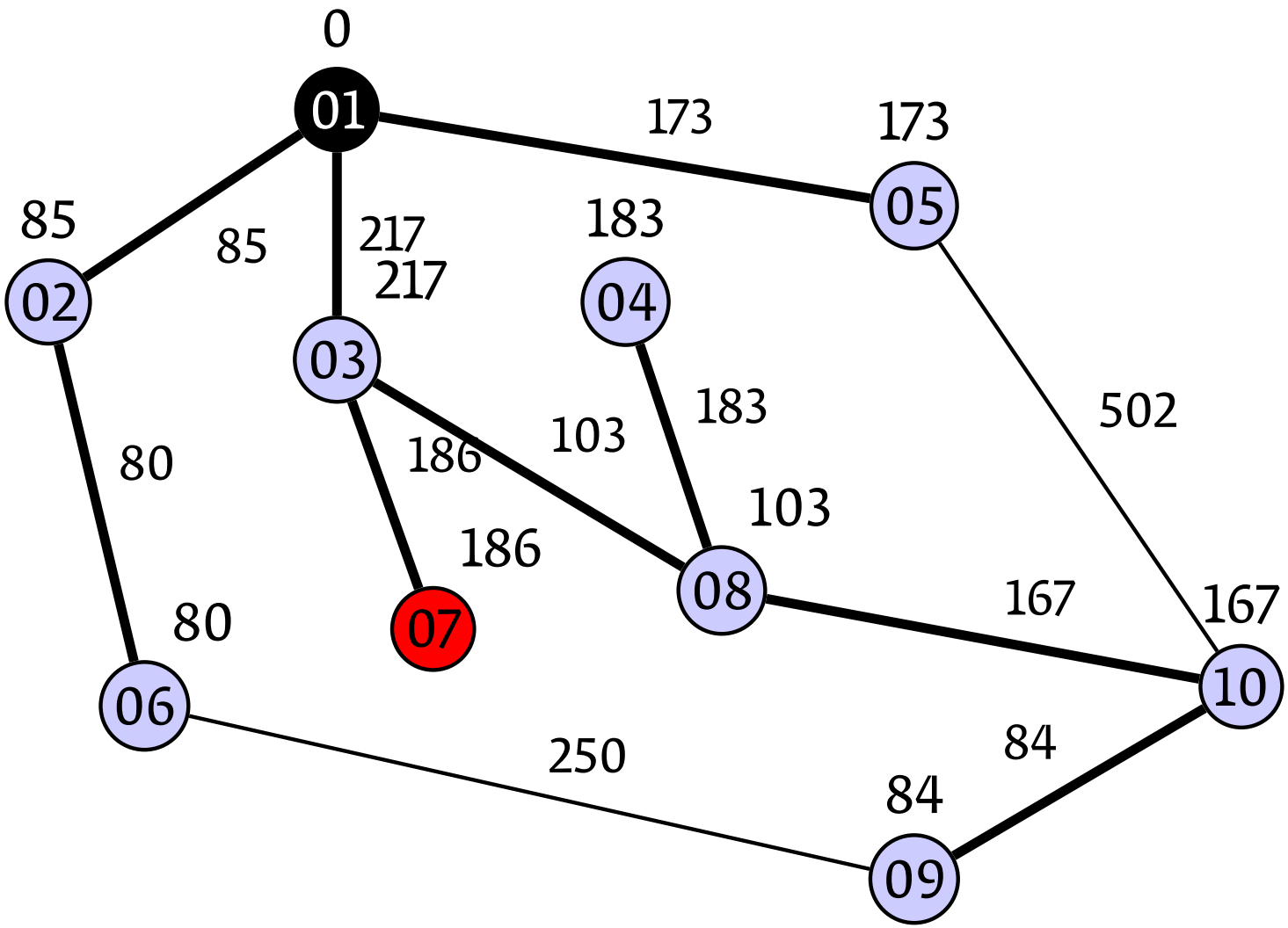
Exemple d'application de Prim 8/10



Exemple d'application de Prim 9/10



Exemple d'application de Prim 10/10



Observations sur l'exemple qui précède

- Chaque figure représente deux étapes : (a) choix du prochain sommet à annexer à l'arbre, et (b) calcul des clés des voisins du nouveau sommet, avec des éventuelles relaxations.
- D'ailleurs on constate deux relaxations non triviales, aux étapes 6 et 7 : $502 \to 167$ pour le sommet 10 et $250 \to 84$ pour le sommet 9.
- On remarque aussi que les clés des sommets de degré 1 ne peuvent qu'être égales au poids des arêtes les reliant au graphe.
- Cela illustre bien le fait que le calcul est essentiellement *local* et qu'il n'y a pas de *répercussion* de l'information à travers l'arbre.

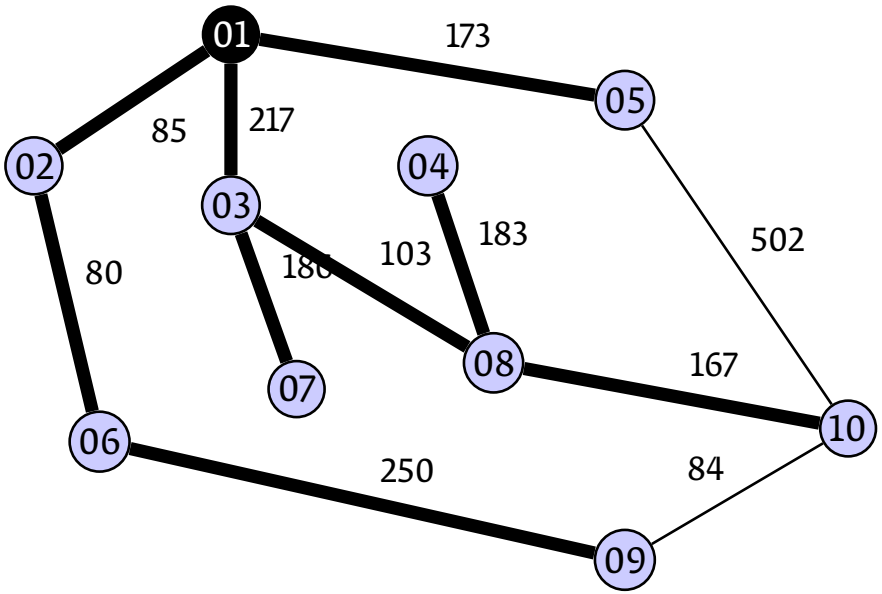
Prim sous graph-tool

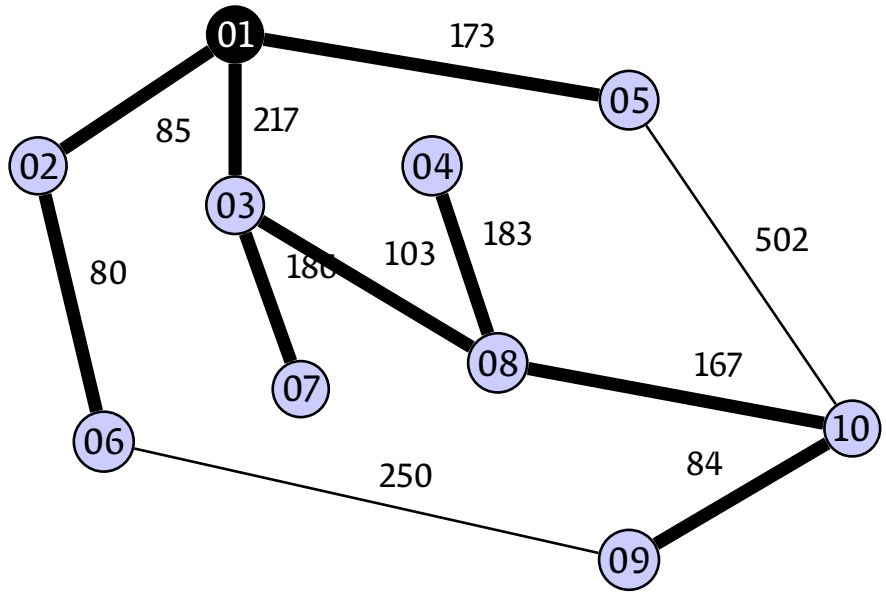
```
from graph_tool.all import *
g = Graph(directed=True)
weight = g.new_edge_property("int16_t")
g.add_vertex(10)
weights=[85,217,173,80,186,103,183,502,250,167,84]
my_edges=[(0,1),(0,2),(0,4),(1,5),(2,6),(2,7),(7,3),\
    (4,9),(5,8),(7,9),(8,9)]
for ((a,b),w) in zip(my_edges,weights):
    e = g.add_edge(a,b)
    weight[e]=w
pr=min_spanning_tree(g,weights=weight,source=g.vertex(0))
for e in g.edges():
    if (pr[e]==1):
        print(e)
```

(seule différence avec Kruskal : `source=g.vertex(0)`.)

Ne pas confondre les deux arbres

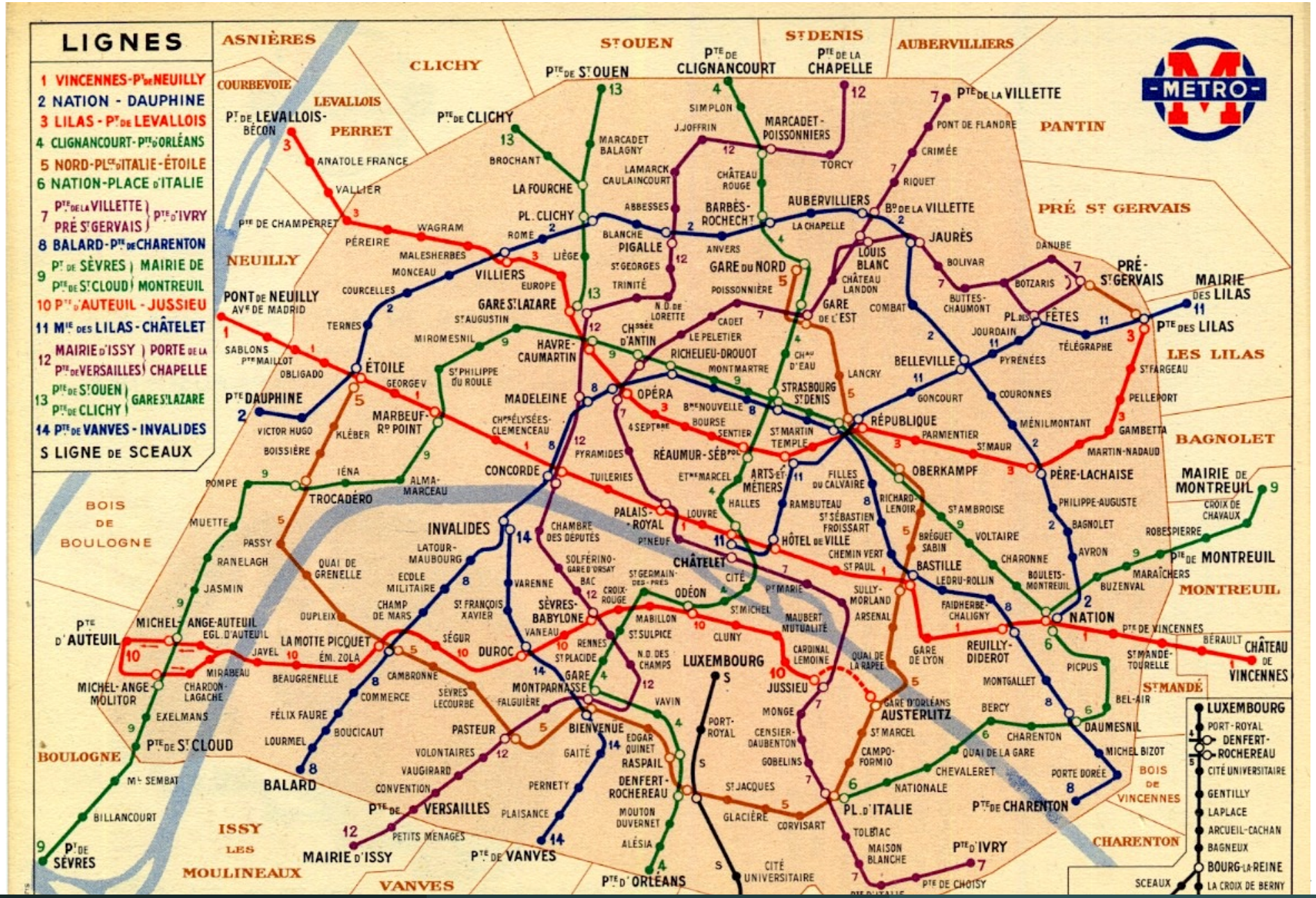
À noter que les arbres obtenus par Prim (ou Kruskal) et Dijkstra ne sont pas les mêmes :





À gauche l'*arbre des plus courts chemins*, obtenu par Dijkstra ou Bellman-Ford. À droite l'*arbre couvrant minimal*, obtenu par Prim ou Kruskal. Seules différences : les arêtes (6, 9) et (9, 10).

Réseaux de transport, flots, coupes

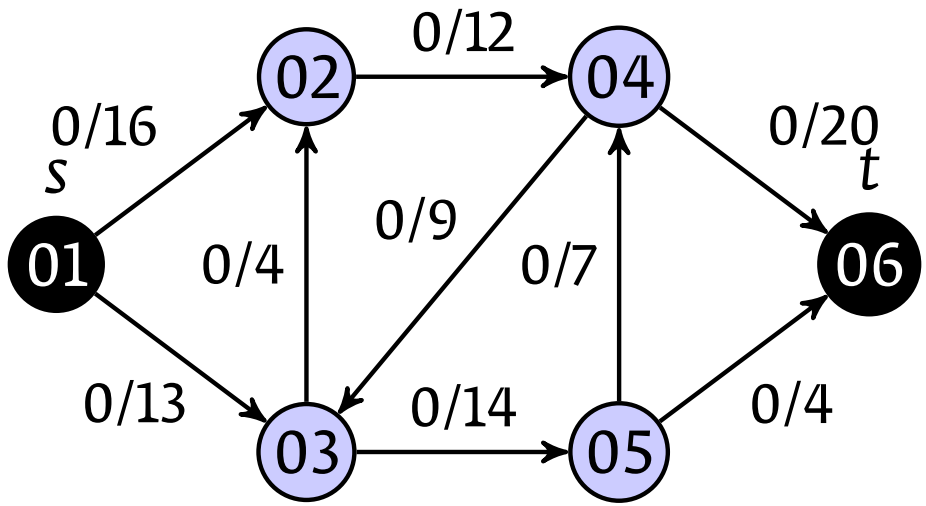


Réseaux de transport, flots, coupes

- Un *réseau de transport* est un graphe orienté $G = (V, E)$ dans lequel on a choisi deux sommets s et t, appelés resp. *source* (avec des arcs sortant uniquement) et *puits* (avec des arcs entrant uniquement), et une fonction positive sur les arêtes, appelée *capacité*.
- Une fonction positive f sur les arêtes est appelée un *flot* si :
  (1) pour un sommet v donné, autre que s et t, $\sum f(e^+) - \sum f(e^-) = 0$ pour tous les arcs entrant e^- et sortant e^+,
  (2) $f(e) \leq c(e)$ pour tout $e \in E$.
- La condition (1) correspond aux lois de Kirchhof en physique.
- On appelle $f(s)$ la *valeur du flot* et on l'écrit $|f|$.

Recherche de flot maximum, approche naïve

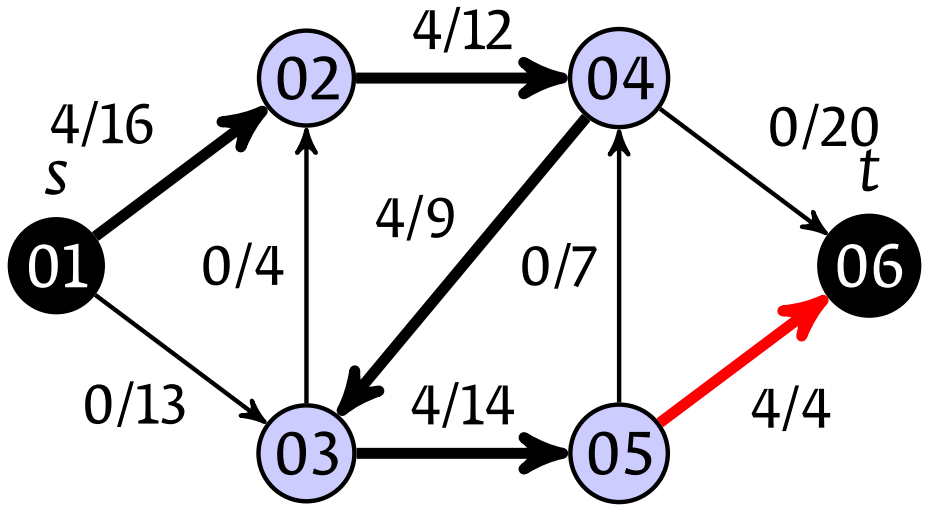
- On se propose de trouver le flot maximum qu'un réseau peut transporter.
- L'approche naïve consiste à ajouter du flot jusqu'à ce que le réseau soit saturé. Prenons le réseau suivant, doté du flot nul :



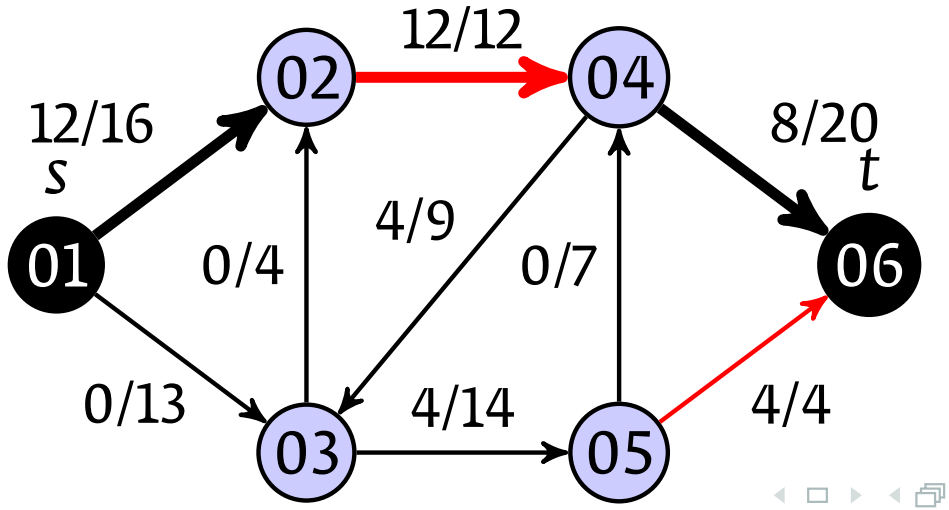
- Ici 0/16, par exemple, signifie que l'arête a une capacité de 16 et on lui a affecté un flot de 0.

Recherche de flot maximum, approche naïve

- Soyons déraisonnables : ajoutons naïvement un flot de 4 aux arêtes 1-2-4-3-5-6, de manière à saturer la 5-6 :

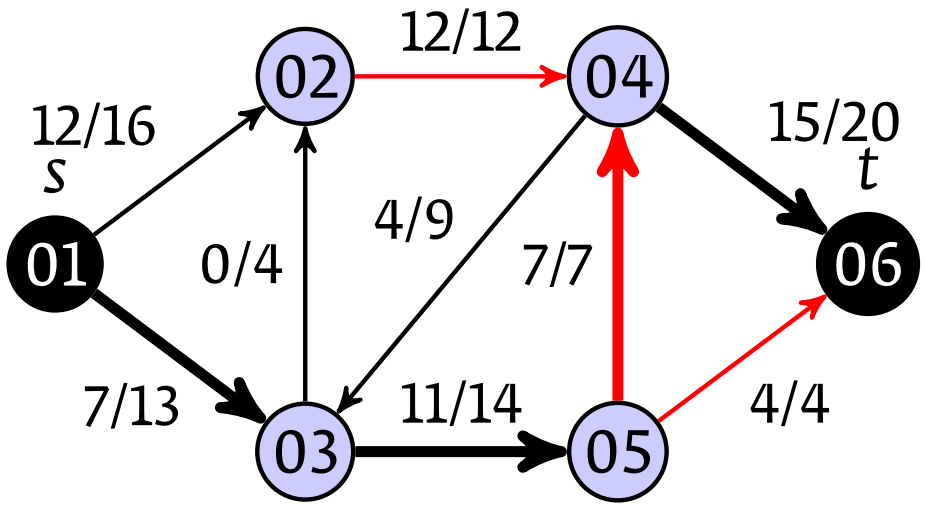


- Et ensuite 8 unités à 1-2-4-6, de manière à saturer la 2-4 :



Pourquoi la vie a du goût

- Il ne nous reste d'autre choix que d'ajouter un flot de 7 aux arêtes 1-3-5-4-6, de manière à saturer la 5-4 :



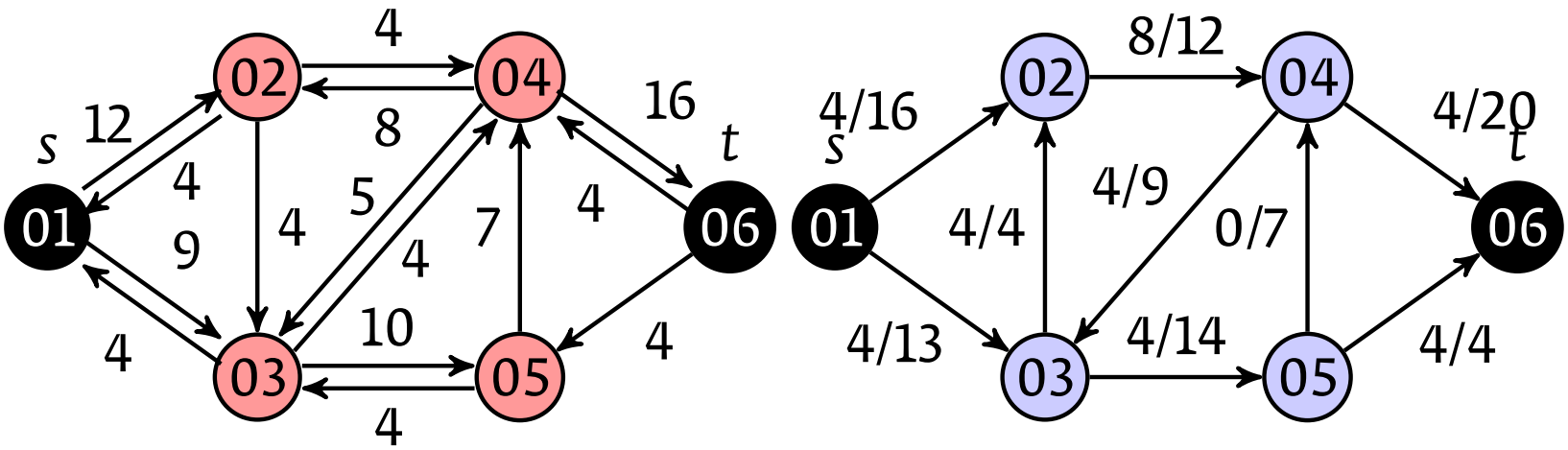
- La situation est bloquée puisque la saturation des arêtes 2-4, 5-4 et 5-6 nous empêche d'accéder à 6. Le flot total obtenu est de 19.
- Pourtant, en passant par 1-2-4-6 (12 unités), par 1-3-5-6 (4 unités) et par 1-3-5-4-6 (7 unités), on aurait eu un flot de 23.
- Peut-on éviter de faire des mauvais choix ? Réponse : NON !! (sinon la vie n'aurait pas de goût…)

Réseaux résiduels

- Comment faire alors ?
- On s'invente une infrastructure qui permet de changer d'avis et de redresser les torts causés. (Dans la vie cela n'existe pas toujours…)
- Cette infrastructure miraculeuse s'appelle un *réseau résiduel*.
- Soit G un graphe et f un flot, un *réseau résiduel* G_f est un graphe avec les mêmes sommets et :
  (1) pour chaque arête (u, v) de G, une arête dans le même sens et de capacité $c_f(u, v) = c(u, v) - f(u, v)$ dans G_f, ainsi qu'une arête (v, u) (dans l'autre sens) de capacité $c_f(v, u) = f(u, v)$,
  (2) on n'écrit pas les arêtes de capacité nulle dans G_f.
- Le réseau résiduel exprime *toutes* les possibilités dont on dispose pour modifier le flot de G.

Réseaux résiduels

- Attention : le réseau résiduel va changer à chaque fois que l'on va augmenter le flot, des arêtes vont apparaître ou disparaître.
- Voici un exemple de réseau résiduel :



- À droite, le graphe, avec un certain flot. À gauche, le réseau résiduel correspondant au graphe et au flot donnés.

Augmentation

- Soit f un flot de G et f' un flot de G_f, alors une *augmentation* $f \uparrow f'$ de f par f' est une fonction de $V \times V \to \mathbb{R}$ définie par

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{si } (u,v) \in E \\ 0 & \text{sinon.} \end{cases}$$

Lemma

$f \uparrow f'$ est un flot, de valeur $|f \uparrow f'| = |f| + |f'|$.

- Une *chaîne augmentante* p est une chaîne de G_f, allant de s à t. Sa *capacité résiduelle* $c_f(p)$ est le min $c_f(x,y)$ pour les $(x,y) \in p$.

Lemma

Soit $f_p(x,y) = c_f(p)$ si $(x,y) \in p$ et 0 sinon. Alors f_p est un flot de G_f et $|f_p| = c_f(p) > 0$. Alors $f \uparrow f_p$ est un flot de G, avec $|f \uparrow f_p| > |f|$.

Méthode de Ford-Fulkerson

- La *méthode de Ford-Fulkerson* s'écrit, en général :

  FORD-FULKERSON(G, s, t) :
    initialiser flot f
    while il existe une chaîne augmentante p dans G_f :
      augmenter f dans G, le long de p
    return f

- Soit G un graphe et f un flot, un *réseau résiduel* G_f est un graphe avec les mêmes sommets et :
  (1) pour chaque $(x, y) \in G$, une arête dans le même sens dans G_f, de capacité $c_f(x, y) = c(x, y) - f(x, y)$, ainsi qu'une arête (y, x) dans l'autre sens de capacité $c_f(y, x) = f(x, y)$,
  (2) on n'écrit pas les arêtes de capacité c_f nulle.
- Le réseau résiduel exprime les possibilités dont on dispose pour augmenter le flot de G.

Flot maximal ?

- La méthode de Ford-Fulkerson consiste à augmenter le flot en trouvant des chaînes augmentantes. Est-ce que cet algorithme s'arrête ? Comment savoir qu'on a alors un flot maximal ?

Théorème (Ford-Fulkerson)

On a équivalence entre :
1. f est un flot maximal de G,
2. le réseau résiduel G_f n'admet pas de chaîne augmentante.

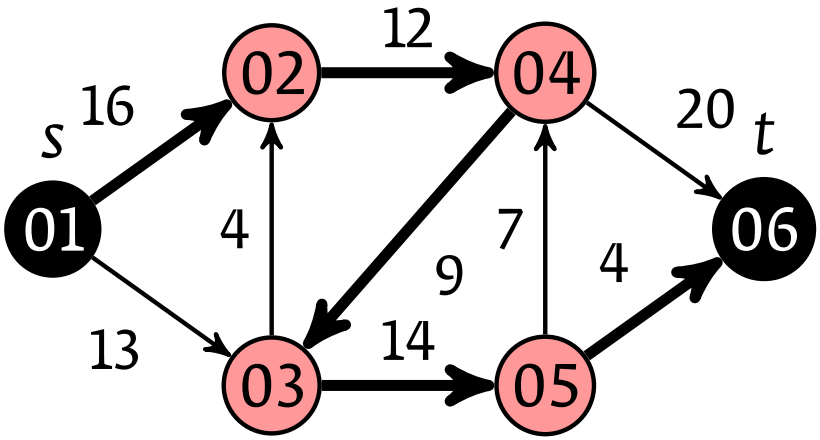
Algorithme de Ford-Fulkerson

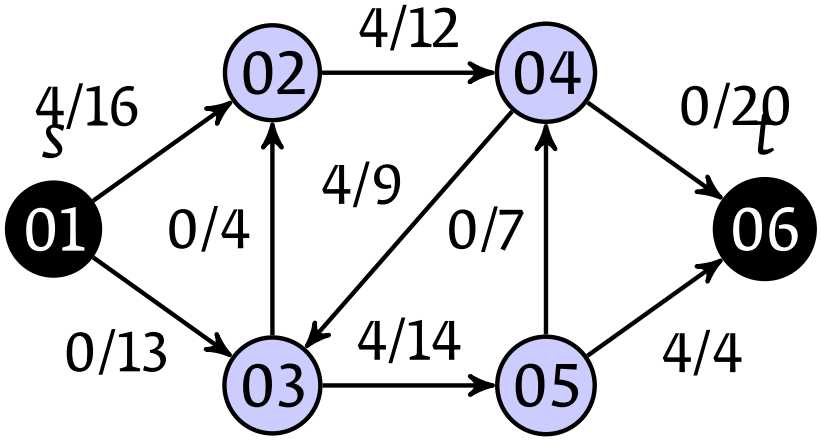
- L'*algorithme de Ford-Fulkerson* peut maintenant s'écrire :

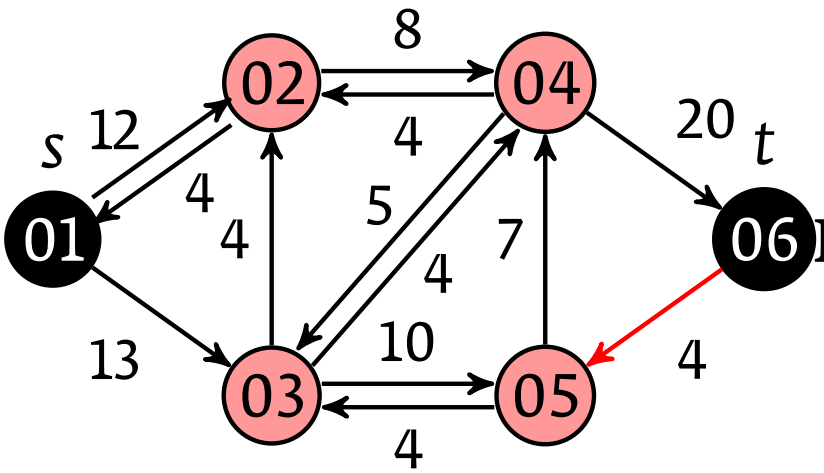
FORD-FULKERSON(G, s, t) :
  for toute arête (u, v) :
    $f(u, v) \leftarrow 0$
  while il existe une chaîne p de s à t dans G_f :
    $c_f(p) \leftarrow \min_{(u,v)\in p} c_f(u, v)$
    for toute arête (u, v) de p :
      if $(u, v) \in E$:
        $f(u, v) \leftarrow f(u, v) + c_f(p)$
      else :
        $f(v, u) \leftarrow f(v, u) - c_f(p)$
  return f

- Dans les illustrations qui suivent, on a mis en haut à gauche le graphe résiduel avec la chaîne augmentante trouvée, à droite le flot résultant, et en bas à gauche le nouveau graphe résiduel.

Exemple d'application de Ford-Fulkerson 1/5

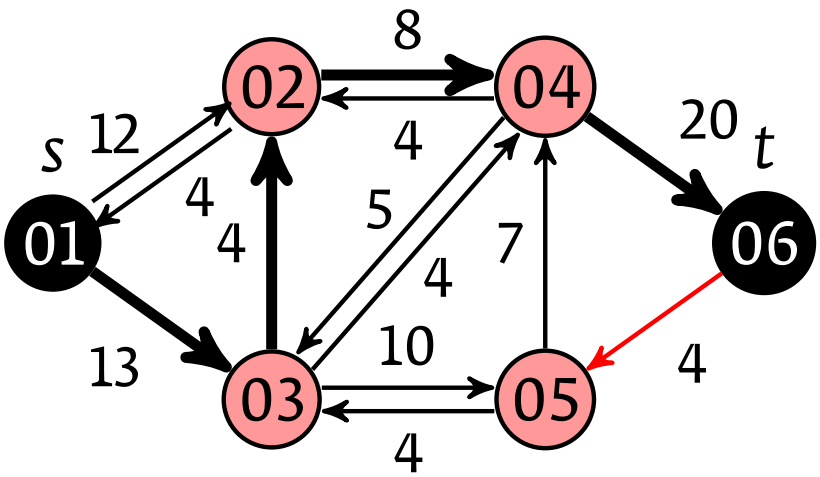


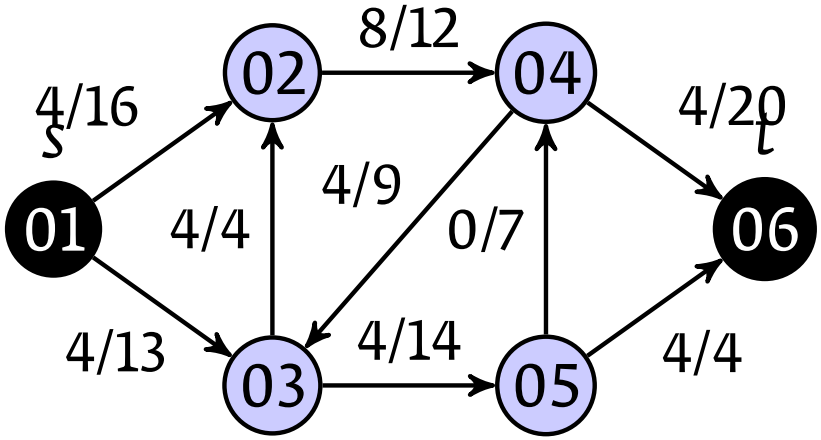


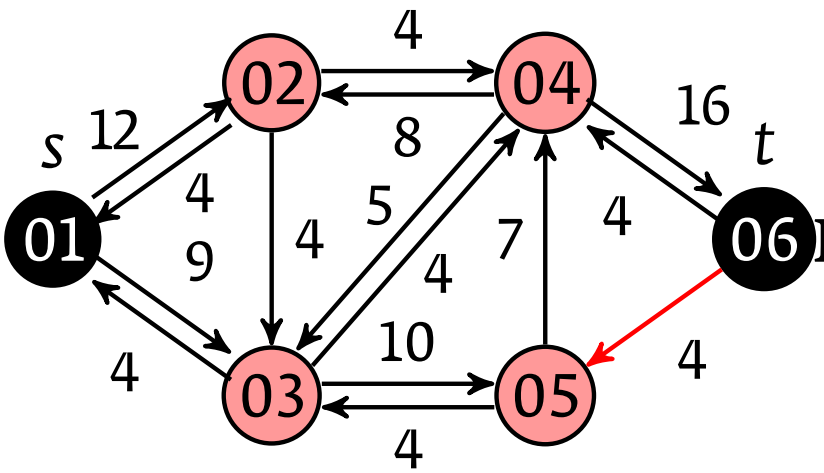


Flot total=4

Exemple d'application de Ford-Fulkerson 2/5

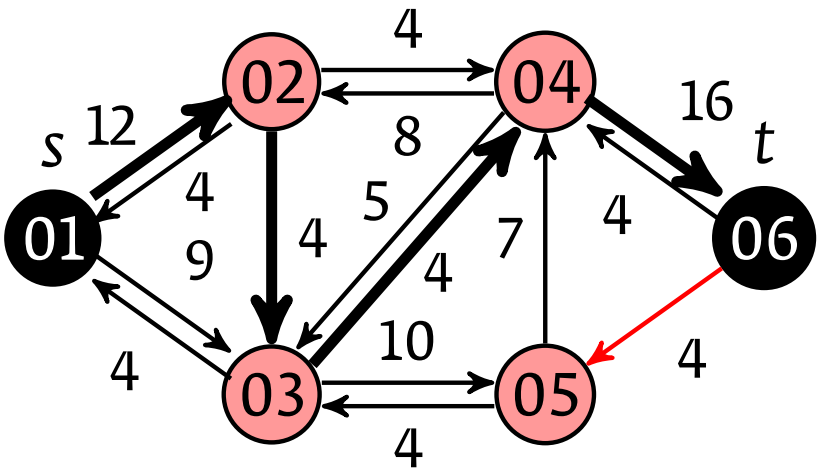


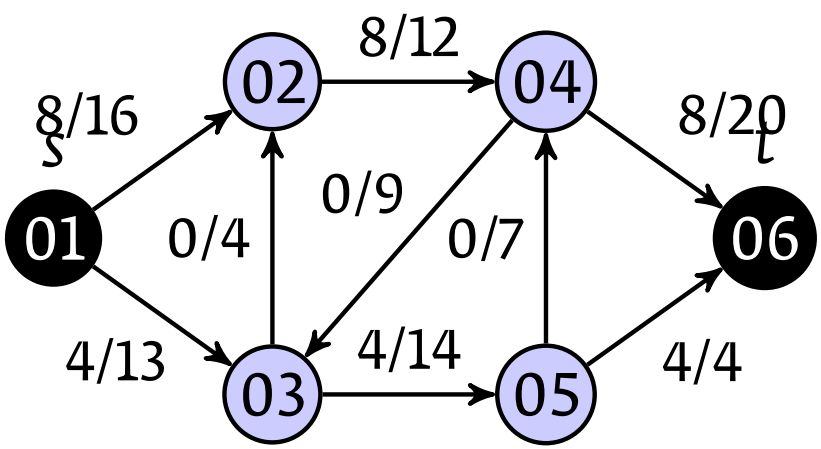


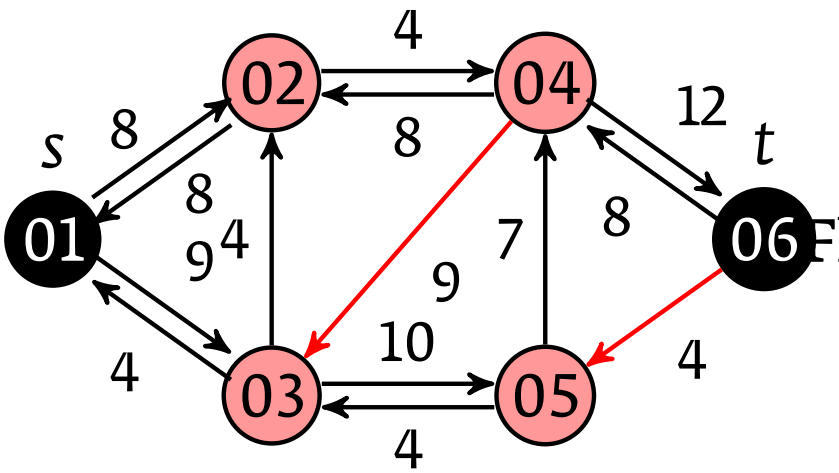


Flot total=8

Exemple d'application de Ford-Fulkerson 3/5

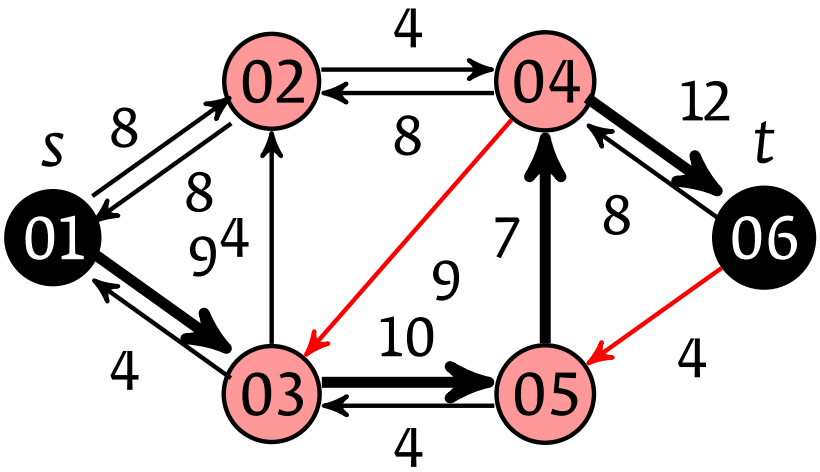


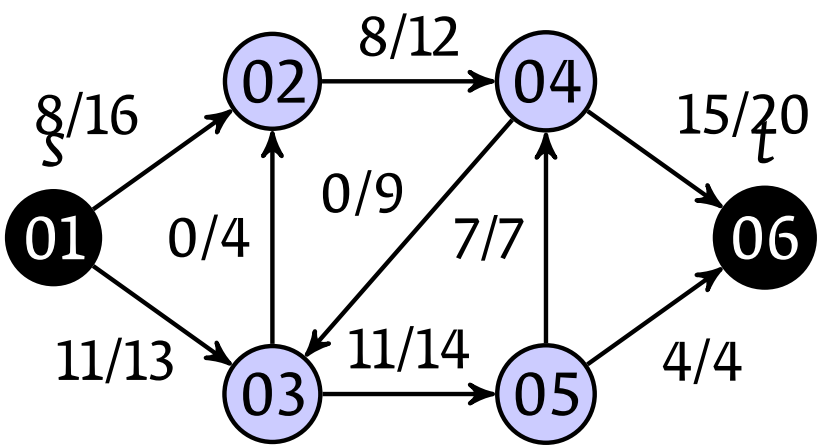


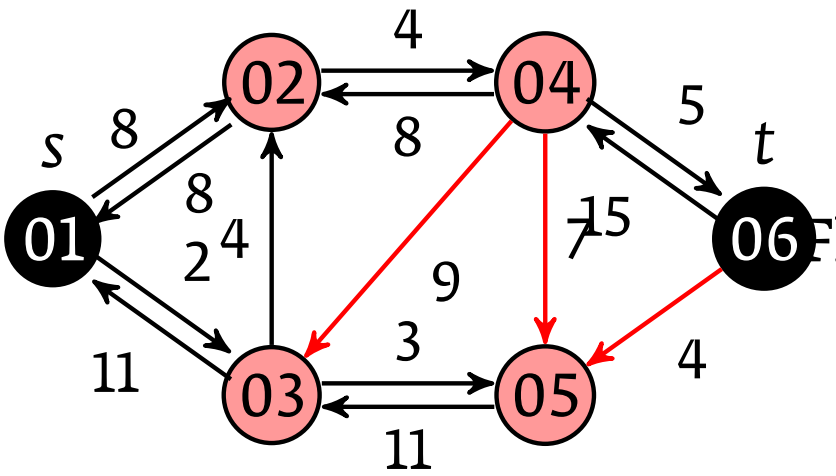


Flot total=12

Exemple d'application de Ford-Fulkerson 4/5

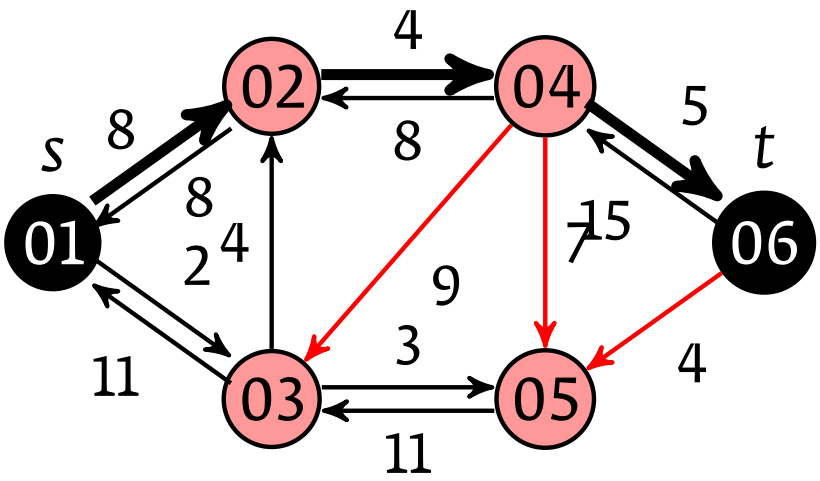


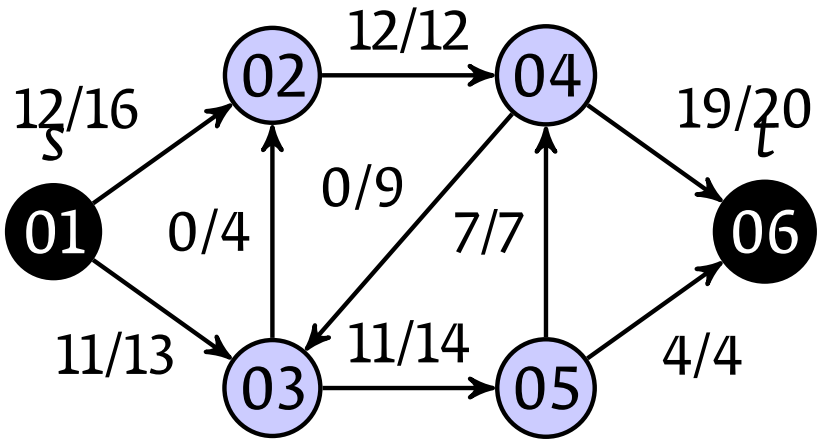


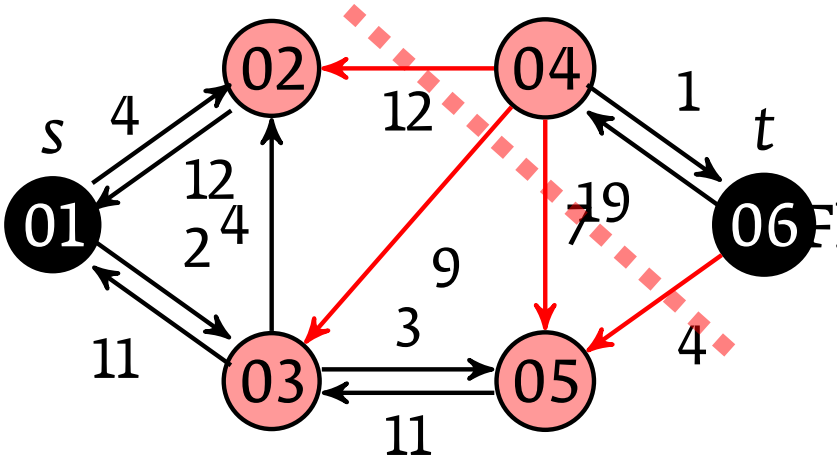


Flot total=19

Exemple d'application de Ford-Fulkerson 5/5







Flot total=23

Calcul de flot maximum sous graph-tool

```
from graph_tool.all import *
g = Graph(directed=True)
capa = g.new_edge_property("int16_t")
g.add_vertex(6)
weights=[16,13,4,12,9,14,7,20,4]
my_edges=[(0,1),(0,2),(2,1),(1,3),(3,2),(2,4),(4,3),(3,5),(4,5)]
for ((a,b),w) in zip(my_edges,weights):
    e = g.add_edge(a,b)
    capa[e]=w
resi=edmonds_karp_max_flow(g,g.vertex(0),g.vertex(5),capa)
resi.a = capa.a - resi.a
for e in g.edges():
    print(resi[e])
max_flow = sum(resi[e] for e in g.vertex(5).in_edges())
print(max_flow)
```

qui donne 12, 11, 12, 0, 11, 0, 19, 7, 4 et le flot maximal 23. (Edmonds-Karp est une amélioration de Ford-Fulkerson où l'on utilise BFS pour les choix d'arêtes.)

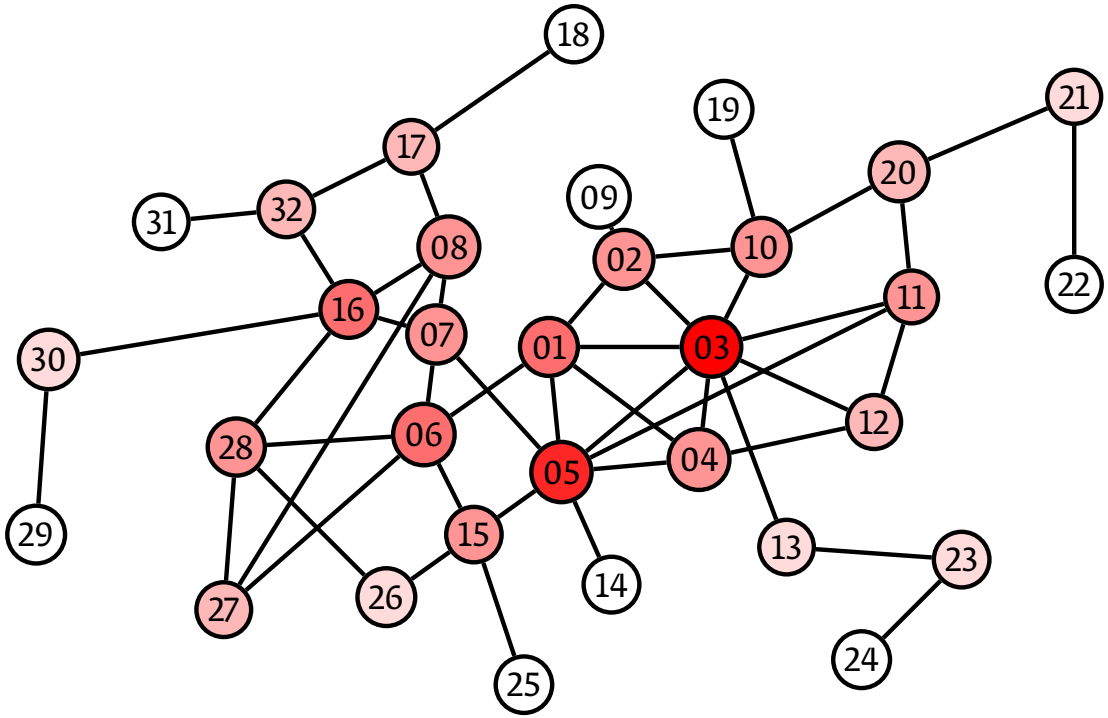
partie III

Mesures

Centralité

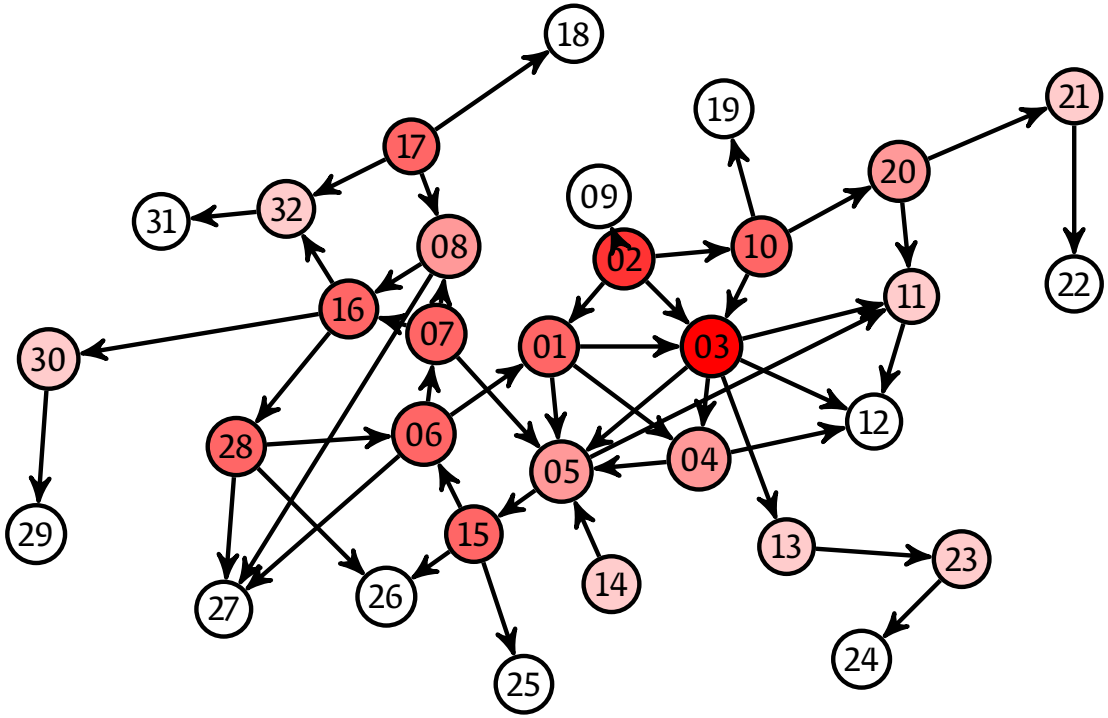
- Dans la suite nous allons étudier des graphes par leurs propriétés métriques.
- La première question que l'on peut se poser est : quels sont les sommets les plus importants / les plus centraux ?

Centralité par le degré



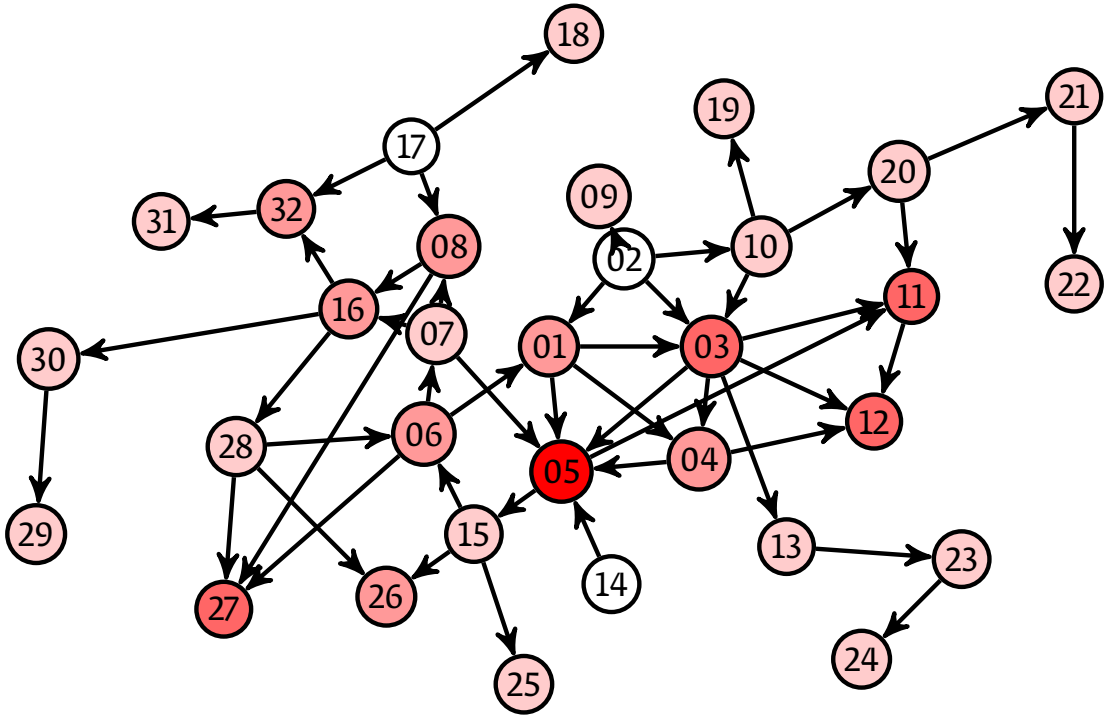
- Approche simpl(ist)e : *est central celui qui a le plus de voisins* : on appelle cette pondération, la *centralité par le degré*.
- On calcule pour chaque v_i la quantité $\frac{d(v_i)-d_{\min}}{d_{\max}-d_{\min}}$.
- À noter que v_{14} a une centralité très faible malgré le fait qu'il est très proche de v_5.

Centralité par le degré sortant



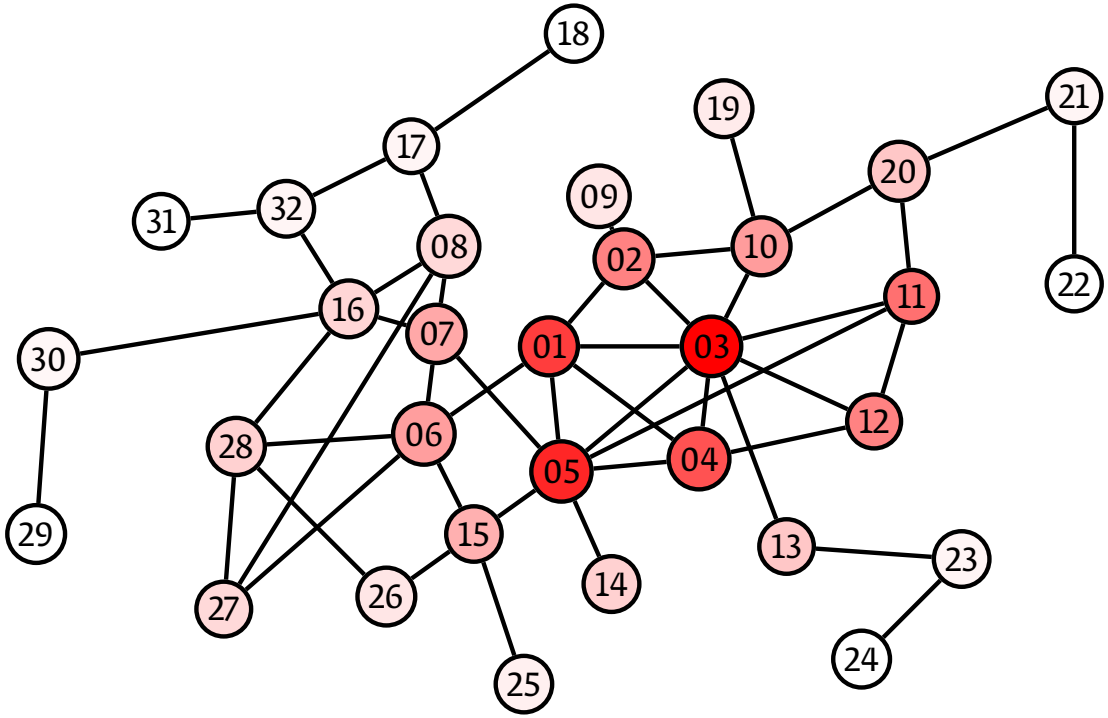
- Dans le cas orienté : *est central celui qui transmet à un maximum de voisins* : on appelle cette pondération, la *centralité par le degré sortant*.
- On calcule pour chaque v_i la quantité $\frac{d^+(v_i)-d^+_{\min}}{d^+_{\max}-d^+_{\min}}$.
- À noter que v_{12} a une centralité très faible malgré le fait qu'il est très proche de v_3.

Centralité par le degré entrant



- Dans le cas orienté : *est central celui qui reçoit à partir d'un maximum de voisins* : on appelle cette pondération, la *centralité par le degré entrant*.
- On calcule pour chaque v_i la quantité $\frac{d^-(v_i)-d^-_{\min}}{d^-_{\max}-d^-_{\min}}$.
- À noter que v_{14} a une centralité très faible malgré le fait qu'il est très proche de v_5.

Centralité non orientée par vecteur propre



- Plus sophistiqué : *est central celui qui a le plus de voisins, eux-même centraux* : on appelle cela, la *centralité par vecteur propre*.
- À noter que v_{14} est un peu plus central que v_{25}, lui-même plus central que v_{22}.

Centralité par vecteur propre

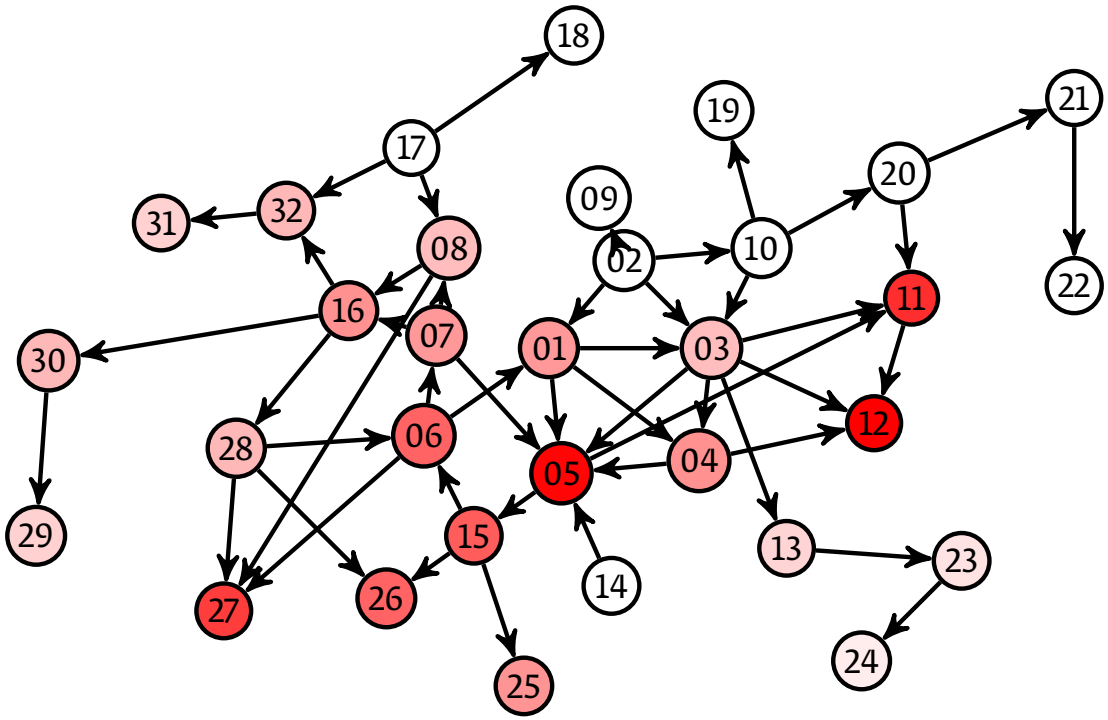
- Voici comment fonctionne la centralité par vecteur propre :
  1. on attache la valeur 1 à chaque sommet : $x_i = 1$,
  2. on fait un premier calcul : $x'_i = \sum_j a_{i,j} x_j$, autrement dit $X' = A \cdot X$
  3. après t étapes on aura $X(t) = A^t \cdot X(0)$,
  4. comme tout vecteur, $X(0)$ s'écrit en fonction des vecteurs propres de A : $X(0) = \sum_i c_i \vec{v_i}$,
  5. alors, si κ_i est la i-ième valeur propre (par ordre décroissant),

$$X(t) = A^t \cdot \sum_i c_i \vec{v_i} = \sum_i c_i \kappa_i^t \vec{v_i} = \kappa_1^t \sum_i c_i (\frac{\kappa_i}{\kappa_1})^t \vec{v_i},$$

     (La deuxième égalité vient du fait que si κ_i est valeur propre et v_i le vecteur propre correspondant, alors $(A - \kappa_i \mathrm{Id}) \cdot v_i = 0$ et donc $Av_i = \kappa_i v_i$ et $A^t v_i = \kappa_i^t v_i$.)

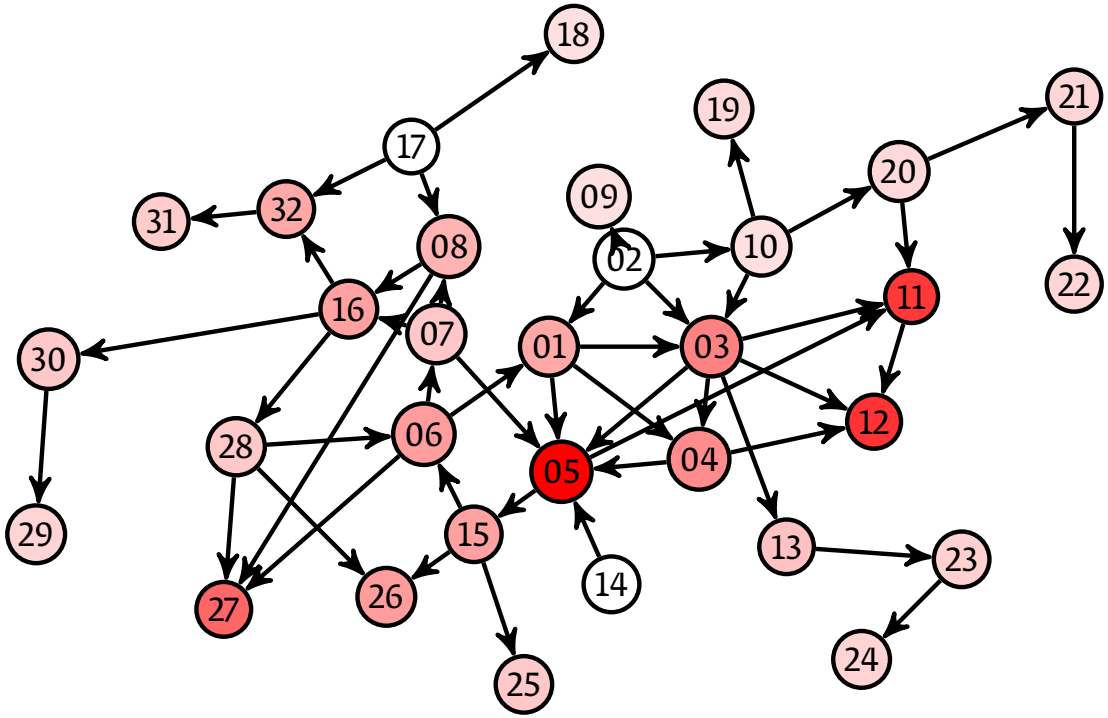
  6. quand $t \gg 0$, $X(t) \sim c_1 \kappa_1^t \vec{v_1}$ et donc la centralité est proportionnelle au premier vecteur propre.
- On définit donc la centralité X par vecteur propre comme la solution du système linéaire $A \cdot X = \kappa_1 X$.

Centralité entrante par vecteur propre



- La généralisation de la centralité par vecteur propre aux graphes orientés pose quelques problèmes.
- Comme on peut le voir : v_0 a centralité entrante nulle, il en est de même de v_{10} et de v_{20}, alors que ceux-ci ont des flèches entrantes…

Centralité de Katz

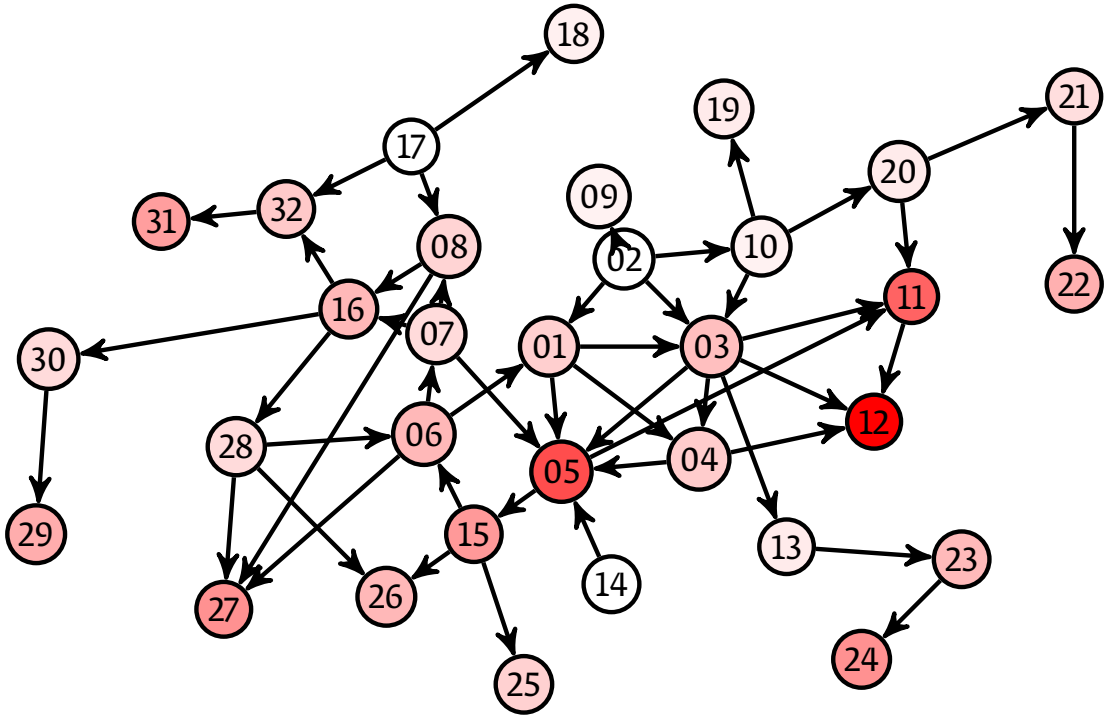


- Katz (1953) propose une amélioration de la centralité entrante par vecteur propre. On remarque que seuls les nœuds sans aucune arête entrante ont une centralité nulle.

Centralité de Katz

- L'idée de Katz : donner un petit bout de centralité à tout le monde et voir comment cela évolue : $x_i' = \alpha \sum_j a_{i,j} x_j + \beta$.
- En posant $\beta = 1$ on trouve $X = (\mathrm{Id} - \alpha A)^{-1} \cdot 1$, la *centralité de Katz*.
- Le choix de α est arbitraire, mais on ne veut pas que cela diverge.
- Or, cela diverge quand $\det(\mathrm{Id} - \alpha A) = 0$, c'est-à-dire pour $\alpha^{-1} = \kappa_*$, donc la première fois pour $\alpha = 1/\kappa_1$. On choisit donc un α légèrement inférieur à $1/\kappa_1$.
- Dans l'exemple on a pris $\alpha = \frac{1}{4}$.
- Pour faire les calculs on évite d'inverser la matrice A (complexité $O(n^3)$), on se sert plutôt de $X' \leftarrow \alpha A \cdot X + \beta \cdot 1$ que l'on réitère plusieurs fois.
- Ceci étant, il y a un problème avec Katz : si un sommet pointe vers un million d'autres, ils auront tous le même apport, alors que ceci devrait se diluer dans la masse des nœuds sortants.

Le PageRank



- Une solution à ce problème est donnée par l'algorithme *PageRank* qui a fait de Google une des plus grandes réussites commerciales de notre temps.
- On remarquera qu'à la différence des autres approches, les chaînes isolées (comme $v_{13}v_{23}v_{24}$ ou $v_{30}v_{29}$) font augmenter la centralité. On remarquera aussi que v_3 qui a 3 flèches entrantes et 3 sortantes est nettement plus faible que v_5 qui en a 5 et 2.

PageRank

- Le PageRank définit $x'_i = \alpha \sum_j a_{i,j} \frac{x_j}{d^+(v_j)} + \beta$, où $d^+(v_j)$ est le degré sortant de v_j.
- Problème : que faire quand $d^+(v_j) = 0$? On le remplace par $\max(d^+(v_j), 1)$.
- Soit D la matrice diagonale des $\max(d^+(v_j), 1)$. Alors on a $X = \alpha A \cdot D^{-1} \cdot X + \beta 1$ et donc
$$X = D \cdot (D - \alpha A)^{-1} \cdot 1.$$
- Il est communément admis que le succès de Google provient non de la pertinence de la totalité des résultats fournis, mais de la pertinence du tri selon l'importance perçue de ceux-ci.
- Et donc le succès de Google est bien dû à PageRank.
- Les mêmes calculs que tout à l'heure montrent que α doit être inférieur à 1. Il s'avère que Google utilise $\alpha = 0,85$.

PageRank et marches aléatoires

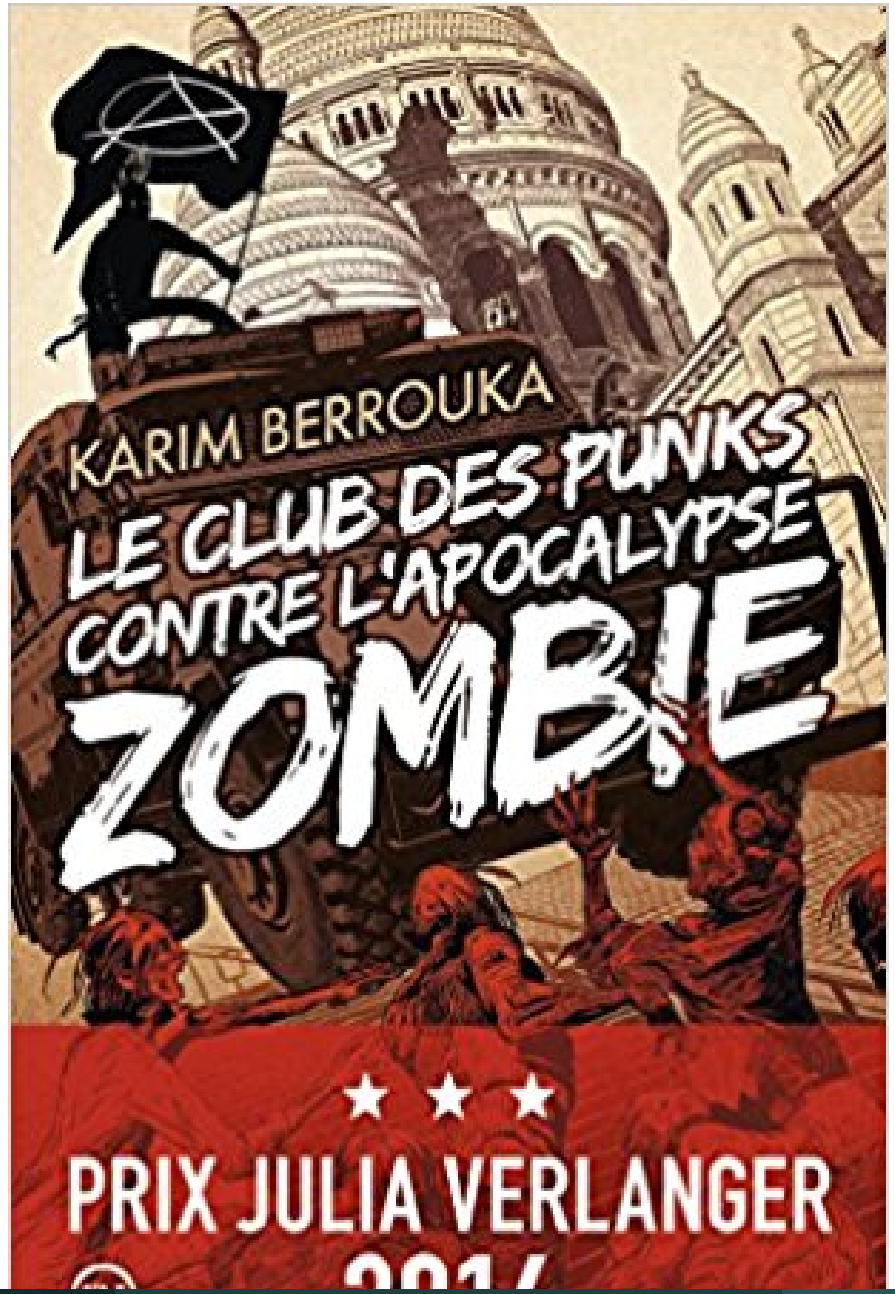
- Un fait notable concernant PageRank est qu'il peut être calculé à partir de *marches aléatoires* :

Définition

Une *marche aléatoire* est un chemin du graphe obtenu en choisissant à tout moment t le sommet suivant avec une probabilité $1/d$ où d est le degré du sommet courant.

- À la marche aléatoire *stricto sensu* nous ajoutons une autre opération : la *téléportation* (*Énergie Monsieur Sulu !*) : en chaque sommet, on a
  1. une probabilité α que le prochain sommet atteint soit non pas un voisin mais un sommet quelconque du graphe (à probabilité égale) ;
  2. une probabilité $(1-\alpha)$ que le pas suivant suive les règles d'une marche aléatoire classique.
- À comparer avec des hordes de touristes (ou de zombies) qui sillonnent Paris.

PageRank et marches aléatoires

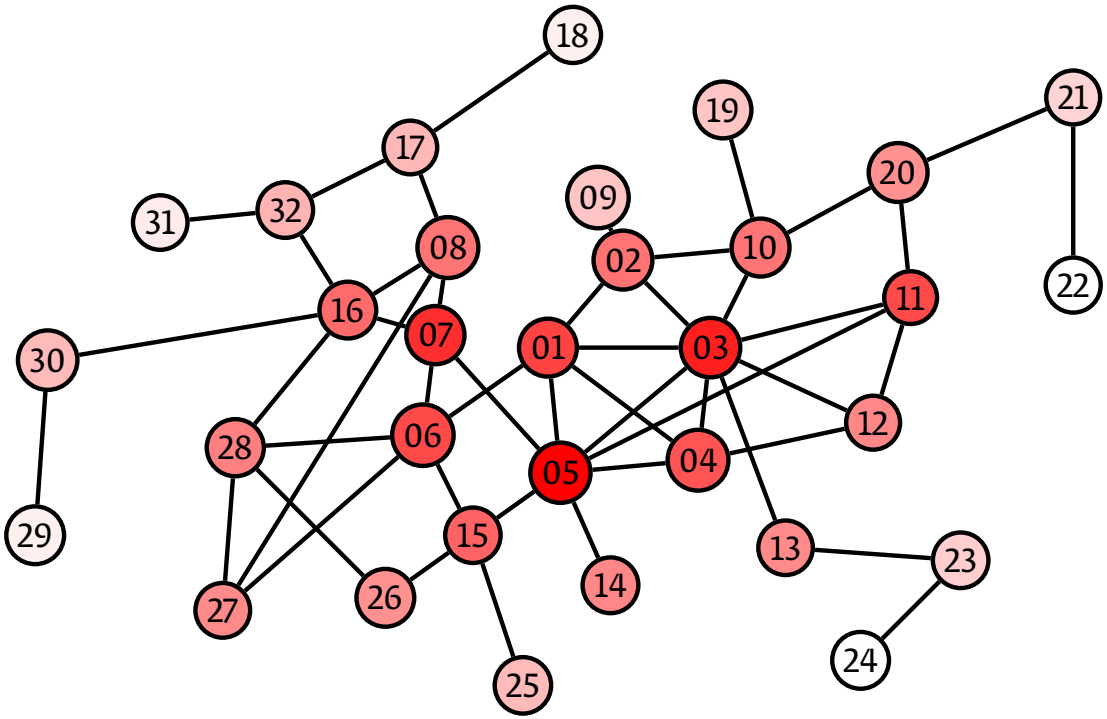


Centralité de proximité

- Il est temps de revenir à des considérations géométriques.
- Calculons la *distance géodésique moyenne* d'un sommet v_i à partir de tous les autres sommets : $\ell_i = \frac{1}{n} \sum_j d(v_i, v_j)$.
- Plus on est central, plus cette quantité diminue.
- On définit la *centralité de proximité* (*closeness*)

$$C_i := \frac{1}{\ell_i} = \frac{n}{\sum_j d(v_i, v_j)}.$$

Centralité de proximité



Centralité de proximité

- On a calculé la centralité de proximité pour les acteurs du site `imdb.com`, l'acteur le plus central est Christopher Lee avec un $C = 0,4143$ et l'actrice la moins centrale Leia Zanganeh avec $C = 0,1154$.
- Or il y a presque un demi-million d'acteurs entre ces deux valeurs. Les différentes valeurs sont très proches.
- Autre problème : dans un graphe non connexe, tous les C sont nuls...
- Si on se restreint aux composantes, on obtient des valeurs plus grandes pour les petites composantes.

Centralité de proximité

- Une solution à ce problème est d'utiliser la moyenne harmonique de distance géodésique :

$$C'_i = \frac{1}{n-1} \sum_{j \neq i} \frac{1}{d(v_i, v_j)}.$$

- Bonnes propriétés : si v_i et v_j appartiennent à des composantes différentes, la fraction est nulle.
- Les sommets éloignés comptent moins que les sommets proches.
- On définit aussi : la *distance géodésique moyenne*

$$\ell = \frac{1}{n} \sum_i \ell_i = \frac{1}{n^2} \sum_{i,j} d(v_i, v_j)$$

- et la *distance géodésique harmonique moyenne*

$$\ell' = \frac{n}{\sum_i C'_i} = n(n-1) \frac{1}{\sum_{i \neq j} \frac{1}{d(v_i, v_j)}}.$$

Centralité de synexité

- Dans un réseau où circulent des idées, être près de tous ne signifie pas toujours qu'on est important.
- Idée : peut aussi être important celui par qui *transite* un maximum de… (idées, opinions, paquets IP, marchandises,…)
- On veut quantifier le nombre de chemins géodésiques qui passent par un sommet donné.
- Définissons la *centralité de synexité* (du grec συνέχω < συνοχή = cohérence), en anglais *betweenness*, comme

$$x_i := \sum_{j,k} \frac{n_{i,j,k}}{g_{j,k}},$$

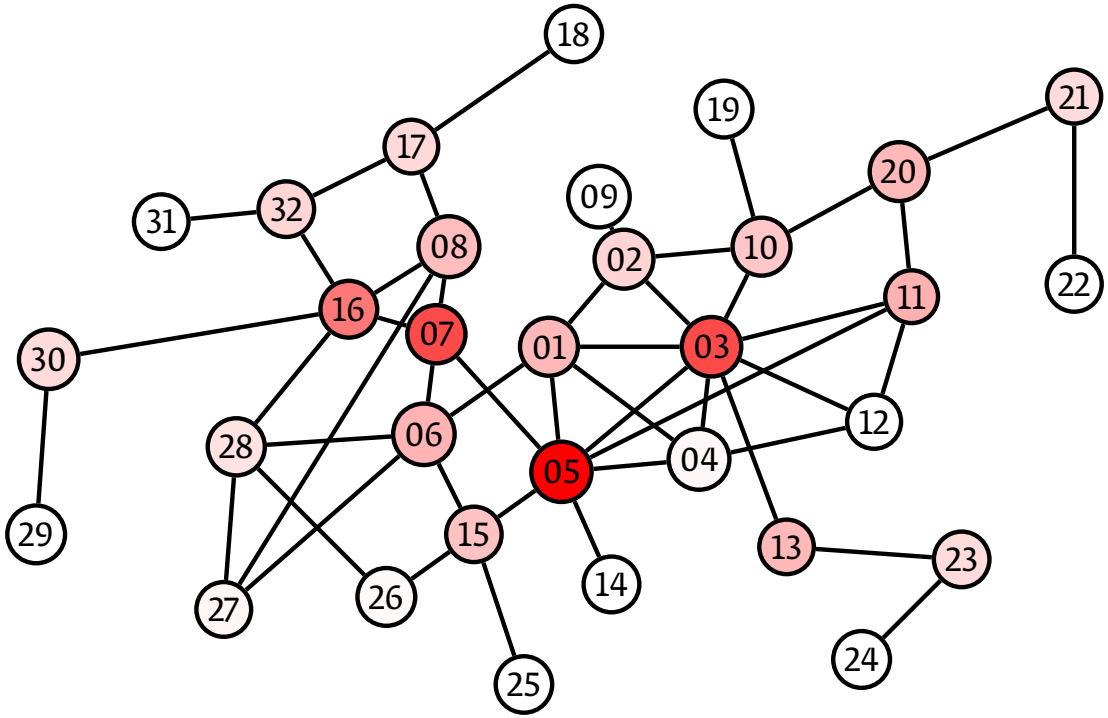
où :

- $n_{i,j,k}$ est le nombre total de chemins géodésiques entre v_j et v_k qui passent par v_i,
- $g_{j,k}$ le nombre de chemins géodésiques entre v_j et v_k.

Centralité de synexité

- La centralité de synexité est très différente des autres puisqu'elle est totalement indépendante du degré.
- Exemple : si $G = G_1 - v_i - G_2$ où G_1 et G_2 sont des composantes très peuplées, alors v_i aura une très forte centralité de synéxité, alors que son degré est simplement 2.
- Un tel sommet est appelé un *courtier* (*broker*).

Centralité de synexité



Ici, v_7 est un cas typique de courtier : son degré est inférieur à celui de v_6, et pourtant il a une centralité de synexité plus élevée.

Centralité de synexité

- La valeur maximale de la centralité de synexité est n^2 (penser au centre d'un graphé étoilé).
- La valeur minimale est $2n - 1$ (penser à une feuille de graphe : $n - 1$ chemins de la feuille vers le reste, $n - 1$ dans l'autre sens, 1 chemin de longueur 0).
- Dans IMDB, l'acteur avec la plus grande valeur est Fernando Rey, pour avoir joué dans des films US et européens, au cinéma et à la télé, avec $x = 7,47 \cdot 10^8$. Le deuxième gagnant est (encore) Christopher Lee avec $x = 6,46 \cdot 10^8$, une différence de 14%.
- La centralité de synexité est donc beaucoup plus stable que la centralité de proximité.
- On définit la *centralité de synexité normalisée*, dont les valeurs se situent entre 0 et 1 :

$$x_i = \frac{1}{n^2} \sum_{j,k} \frac{n_{i,j,k}}{g_{j,k}}.$$

Centralités sous graph-tool

- La centralité par vecteur propre :

```
poids=eigenvector(g)
```

- La centralité de Katz :

```
poids=katz(g,alpha=0.01)
```

- La centralité par PageRank :

```
poids=pagerank(g,damping=0.85)
```

- La centralité de proximité :

```
poids=closeness(g,norm=True,harmonic=False)
```

- La centralité de synexité :

```
poids=betweenness(g,norm=True)
```

Groupes de sommets

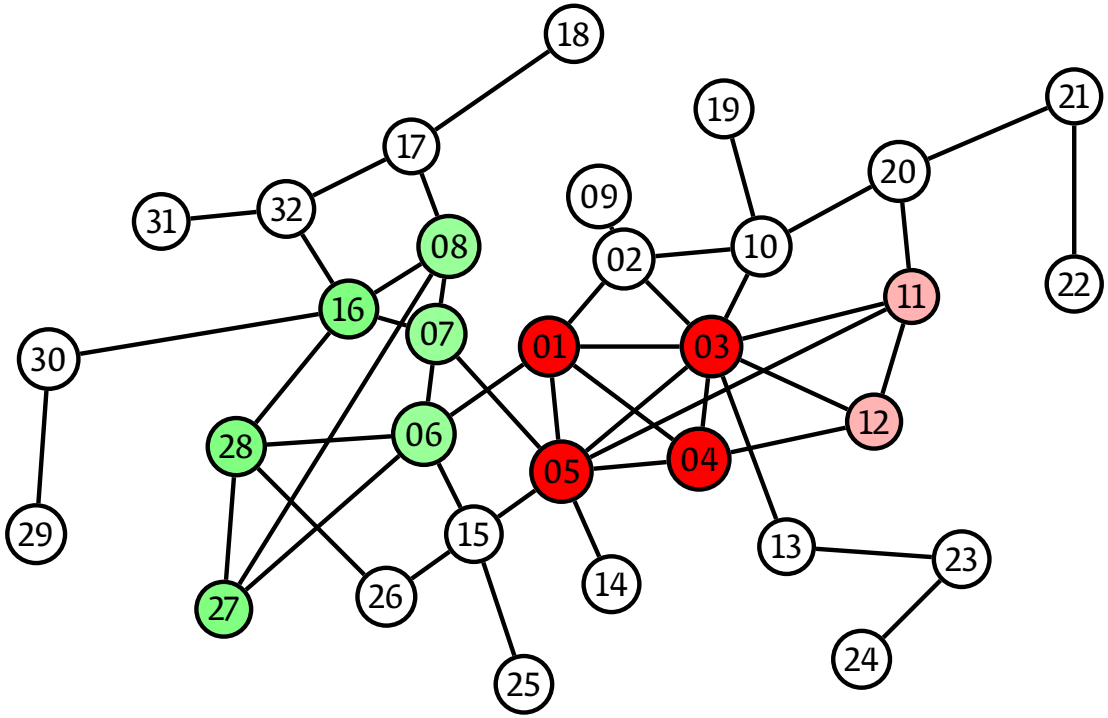
- Jusqu'à maintenant nous avons traité la centralité des nœuds individuels.
- Il est également intéressant d'étudier la présence et l'importance de groupes de sommets dans un graphes.
- L'exemple type : les communautés de personnes dans les groupes sociaux.

Cliques, ou presque : k-plexes

- On a déjà défini les cliques. Et si on relâchait un peu les contraintes ?
- Un *k-plexe* de taille n est un groupe de n sommets formant une composante connexe et dont chacun est lié à au moins $n - k$ autres.
- En prenant $k = 1$ on retombe sur la notion de clique. Pour $k \geq n$ on trouve tous les groupes connexes de n sommets.
- Dans l'exemple :

	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n \geq 8$
$k = 1$	11	1				
$k = 2$	125	20	11			
$k = 3$	125	363	32	**2**		
$k = 4$	125	363	1032	55		
$k = 5$	125	363	1032	2840	111	

Exemples de k-plexes

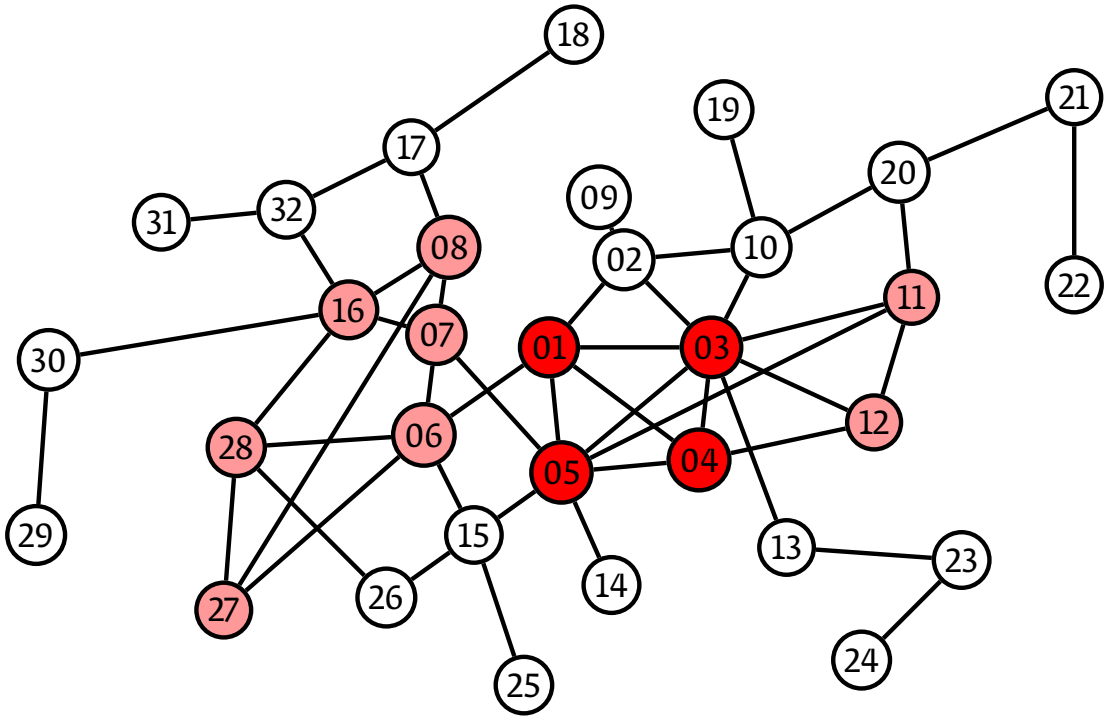


En rouge, la clique 1-3-4-5, qui est sous-graphe du 3-plexe (de taille 6) 1-3-4-5-11-12 (en rose). En vert un autre 3-plexe de taille 6 : 6-7-8-16-27-28.

Autre variante : les k-noyaux

- Si les k-plexes sont maximalistes, les k-noyaux sont minimalistes.
- Un *k-noyau* est un groupe maximal connexe de sommets dont chacun est lié à au moins k autres.
- Pour n sommets, un $(n-k)$-noyau est un k-plexe, et en particulier, un $(n-1)$-noyau est une clique.
- Dans l'exemple on a 2705 2-noyaux (pour $n \leq 12$) mais seulement 7 3-noyaux, dont un seul pour $n = 12$.

Exemple de k-noyau



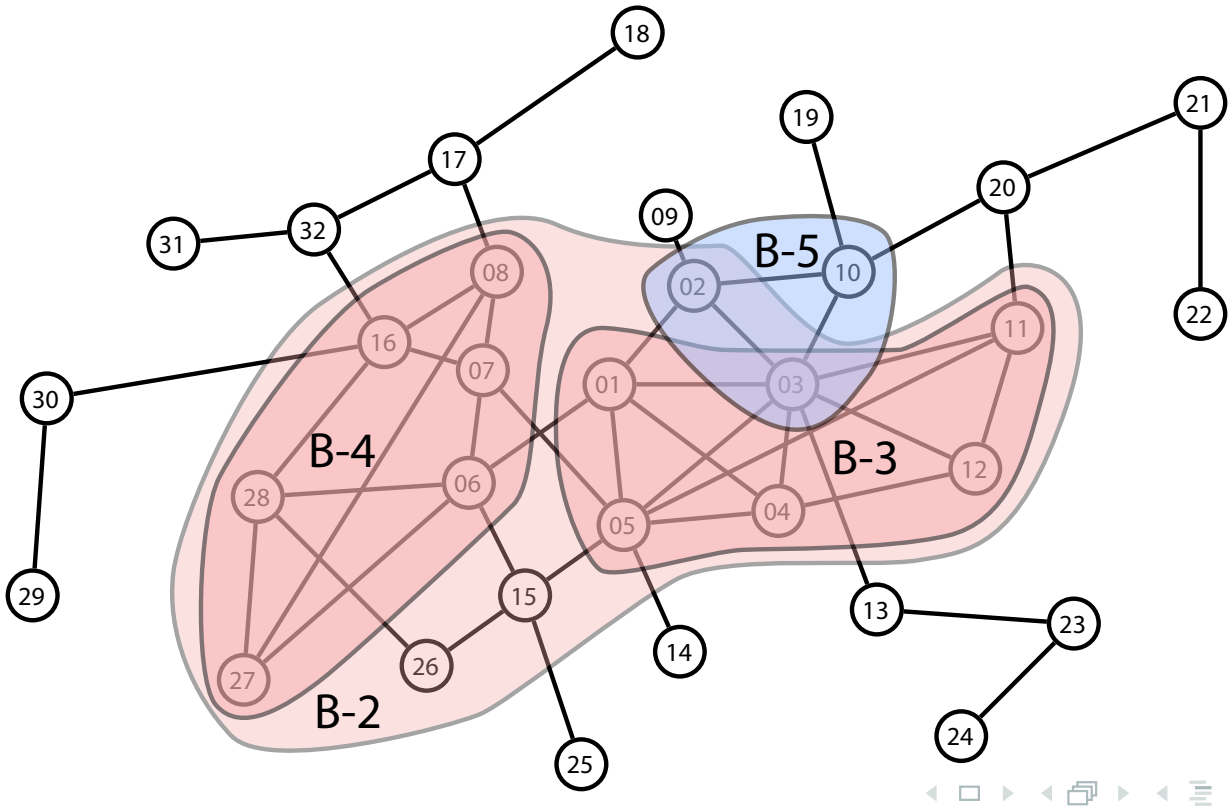
En rose, l'unique 3-noyau de taille 12, qui est en fait induit par les deux 3-plexes de taille 6.

Cohésion, k-composantes

- Le protocole TCP/IP a été défini en vue de rendre un réseau indestructible : si un serveur cesse d'exister, le paquet trouvera un autre chemin pour aller à bon port.
- Cette propriété peut être traduite par non seulement il existe un chemin entre deux sommets, mais il en existe même plusieurs.
- On peut être un peu plus sévère et dire : il existe plusieurs chemins indépendants (vis-à-vis des sommets).
- Une *k-composante* est un sous-graphe tel que toute paire de sommets v_i et v_j peut être reliée par au moins k chemins indépendants (vis-à-vis des sommets).
- Pour $k = 1$ on retrouve la notion de connexité habituelle.
- Inversement, une $n-2$-composante de taille n est un graphe complet.
- On appelle le k maximal des k-composantes d'un graphe, sa *cohésion*. Plus un graphe est cohésif, plus il est robuste.

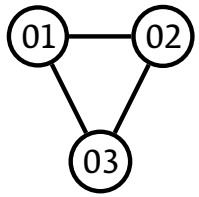
Calcul de k-composantes

```
> cohesive.blocks(G)
B-1          c 1, n 32
'- B-2       c 2, n 15   oooooooo.. oo..oo.... .....ooo.. ..
   '- B-3    c 3, n  6   o.ooo..... oo........ .......... ..
   '- B-4    c 3, n  6   .....ooo.. .....o.... ......oo.. ..
'- B-5       c 2, n  3   .oo......o .......... .......... ..
```



Transitivité

- En algèbre, une relation $*$ est transitive quand $a * b, b * c \Rightarrow a * c$.
- Dans les réseaux sociaux cela se traduit par l'ami de mon ami est mon ami.
- Un sous-graphe complet K_3 est appelé une *triade*.



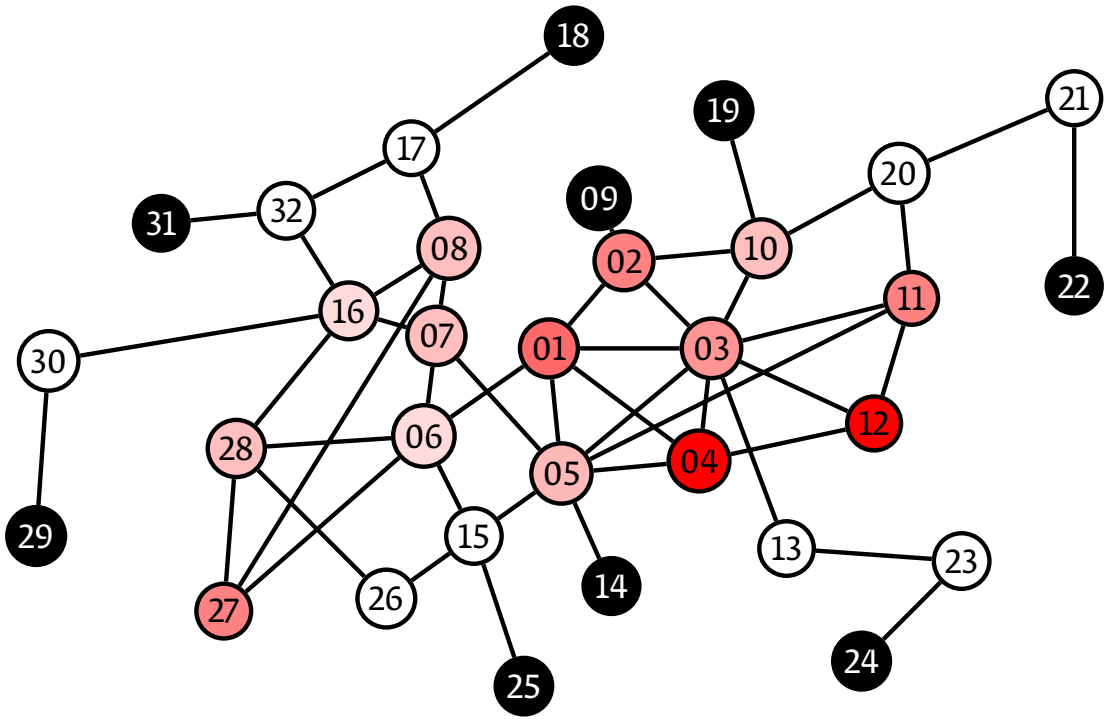
- On appelle *coefficient de clustering global* d'un graphe le ratio

$$C = \frac{\text{nombre de triades}}{\text{nombre de chemins de longueur 2}}.$$

- On peut aussi définir un *coefficient de clustering local* pour chaque sommet v_i de degré ≥ 2 :

$$C_i = \frac{\text{nombre de voisins de } v_i \text{ connectés}}{\text{nombre de paires de voisins de } v_i}.$$

Calcul de coefficient de clustering local



Les nœuds noirs sont de degré 1 et donc leur C_i n'est pas défini. On constate que v_4 et v_{12} sont les sommets les plus transitifs du graphe.

La transitivité sous graph-tool

- Le coefficient local de transitivité (pour chaque sommet qui ne soit pas une feuille) :

  ```
  poids=local_clustering(g,undirected=True)
  ```

- La coefficient global de transitivité :

  ```
  ((c,sigma),triangles,triples)=global_clustering(g,\
      ret_counts=True)
  ```

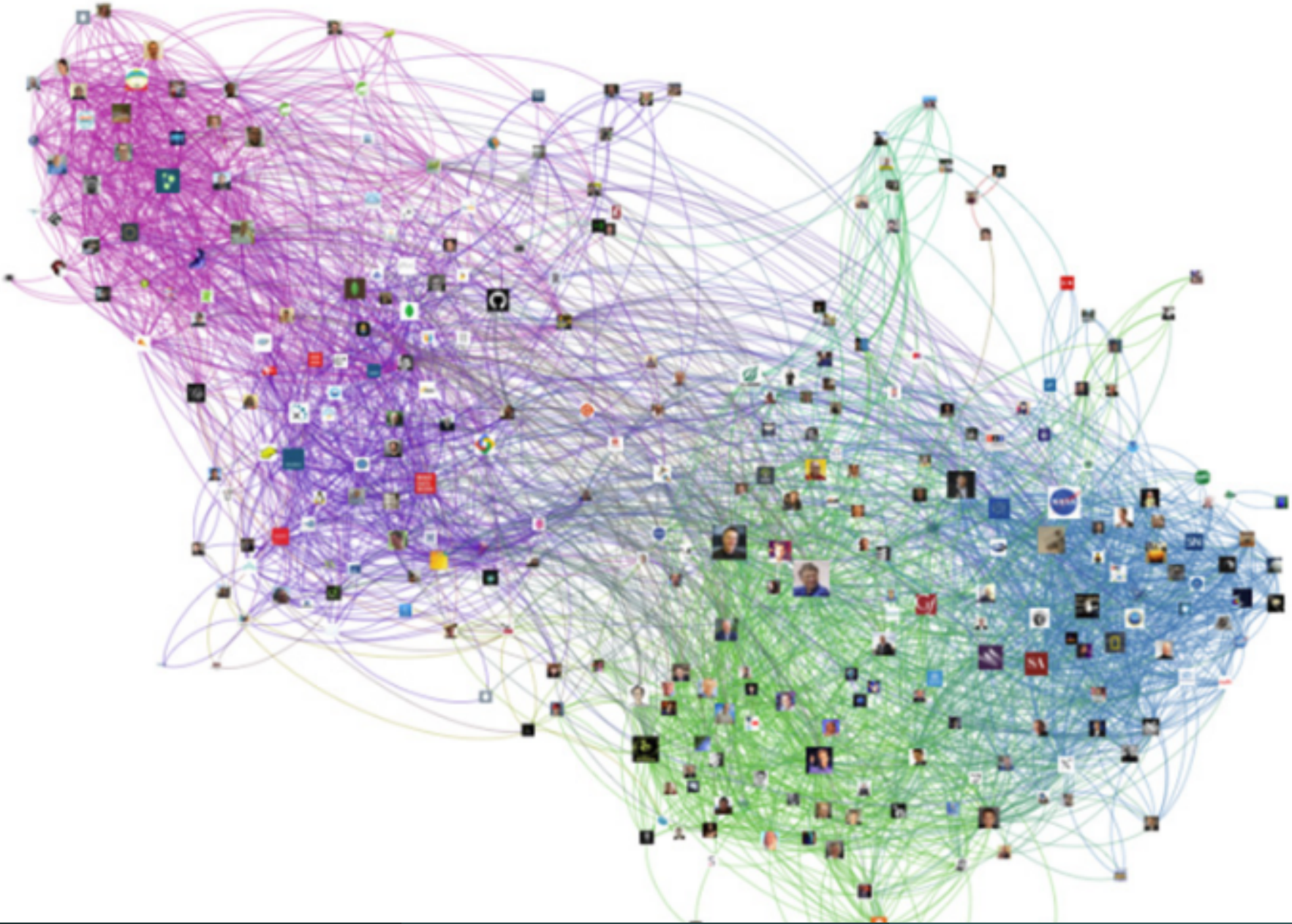
  où

  - `c` est le coefficient global de transitivé,
  - `sigma` est l'écart-type entre coefficient global et coefficients locaux,
  - `triangles` est le nombre de triades,
  - `triples` est le nombre de chemins de longueur 2.

partie IV

Détection de communautés

Détection de communautés dans les rés. sociaux



Assortativité/homophilie

- L'*assortativité* est la mesure du fait que les arêtes d'un graphe relient des sommets qui partagent une certaine propriété. Dans les réseaux sociaux on parle aussi d'*homophilie*.
- Considérons que chaque sommet v_i appartient à une classe c_i.
- Le nombre total d'arêtes intra-classe est

$$\sum_{\text{arêtes } v_i v_j} \delta_{c_i c_j} = \frac{1}{2} \sum_{i,j} A_{ij} \delta_{c_i c_j}$$

  où δ est le symbole de Kronecker.
- Faire le calcul ci-dessus n'a aucun intérêt en l'absence d'une valeur avec laquelle on pourra comparer.
- Nous allons comparer avec le nombre d'arêtes intra-classe (statistiquement) *attendues*.

Modularité

- Soit k_* le degré de v_* et $m := |E|$. Alors il y a $2m$ extrémités d'arêtes et pour chaque arête parmi les k_i qui quittent v_i, la probabilité d'atteindre v_j est $k_j/2m$.

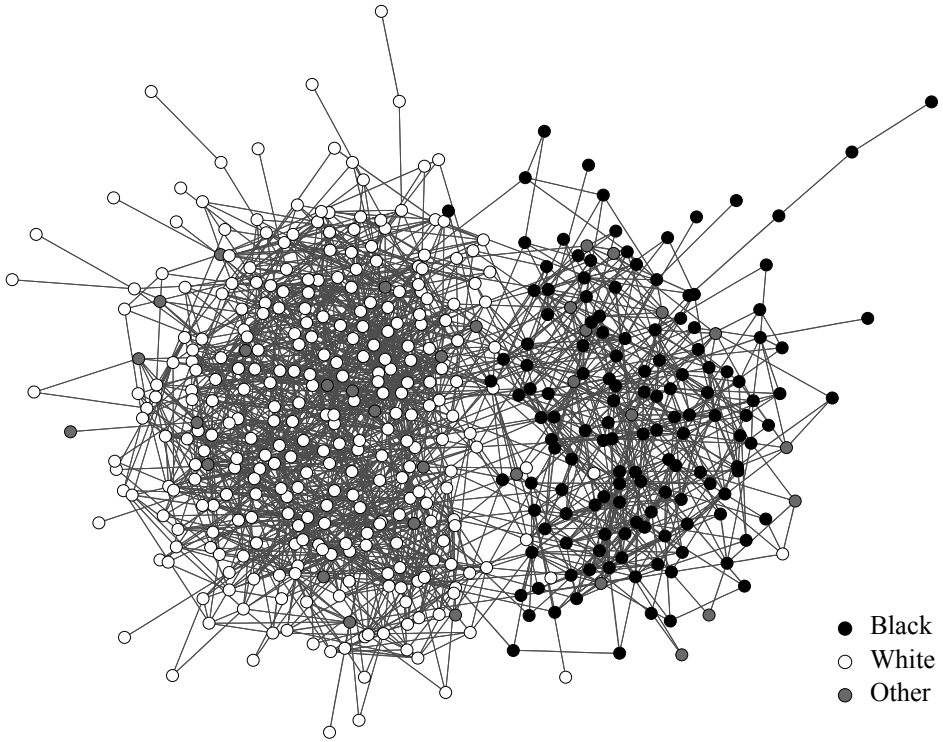
- Donc le nombre *attendu* d'arêtes assortatives est

$$\frac{1}{2}\sum_{i,j}\frac{k_i k_j}{2m}\delta_{c_i c_j}.$$

- Prenons la moyenne des différences entre nombres attendus et constatés d'arêtes assortatives, on l'appellera *modularité* du graphe :

$$Q = \frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})\delta_{c_i c_j}.$$

Étude statistique faite aux US



- Exemple tiré d'un ouvrage américain : le réseau des amitiés entre 470 élèves de lycée US (âges 14–18) en fonction de la couleur de peau (Newman, 2010). la valeur de Q pour cet exemple est de 0,305.
- (Q prend des valeurs dans [−1, +1].)

Assortativité vis-à-vis d'un scalaire

- Soit x_* des valeurs scalaires associées aux sommets.
- Soit $\mu = \frac{1}{2m} \sum_i k_i x_i$ la moyenne des x_* en tant qu'extrémités d'arêtes.
- Soit la généralisation de la formule de la modularité, aux valeurs scalaires :

$$\frac{\sum_{i,j} A_{ij}(x_i - \mu)(x_j - \mu)}{2m} = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) x_i x_j .$$

Cette quantité agit comme une covariance : quand x_i et x_j (v_i et v_j étant les extrémités d'une arête) tendent à prendre les mêmes valeurs, elle est de plus en plus positive, s'ils tendent à prendre des valeurs qui ne soient pas du même côté autour de la moyenne, elle est de plus en plus négative.

Assortativité vis-à-vis d'un scalaire

- Dans le cas d'un graphe *parfaitement assortatif* ($A_{ij} = 1 \Rightarrow x_i = x_j$), la formule atteint son maximum :

$$\frac{1}{2m}\sum_{ij}(A_{ij}x_i^2 - \frac{k_i k_j}{2m}x_i x_j) = \frac{1}{2m}\sum_{ij}(k_i\delta_{ij} - \frac{k_i k_j}{2m})x_i x_j .$$

[Explication : quand le graphe est parfaitement assortatif, $A_{ij}x_i x_j = A_{ij}x_i^2$, mais aussi $x_i^2 = x_i x_j \delta_{ij}$, so $A_{ij}x_i^2 = A_{ij}\delta_{ij}x_i x_j$. Et $\sum_{i,j} A_{ij}\delta_{ij}x_i x_j$ est $\sum_{i,j} k_i\delta_{ij}x_i x_j$ puisque sur la ligne j il y a k_i unités dans A.]

Assortativité vis-à-vis d'un scalaire

- Pour normaliser, divisons par le maximum.
- Nous définissons le *coefficient d'assortativité* comme le quotient de la valeur calculée par la valeur maximale possible :

$$r = \frac{\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})x_i x_j}{\sum_{i,j}(k_i \delta_{ij} - \frac{k_i k_j}{2m})x_i x_j}.$$

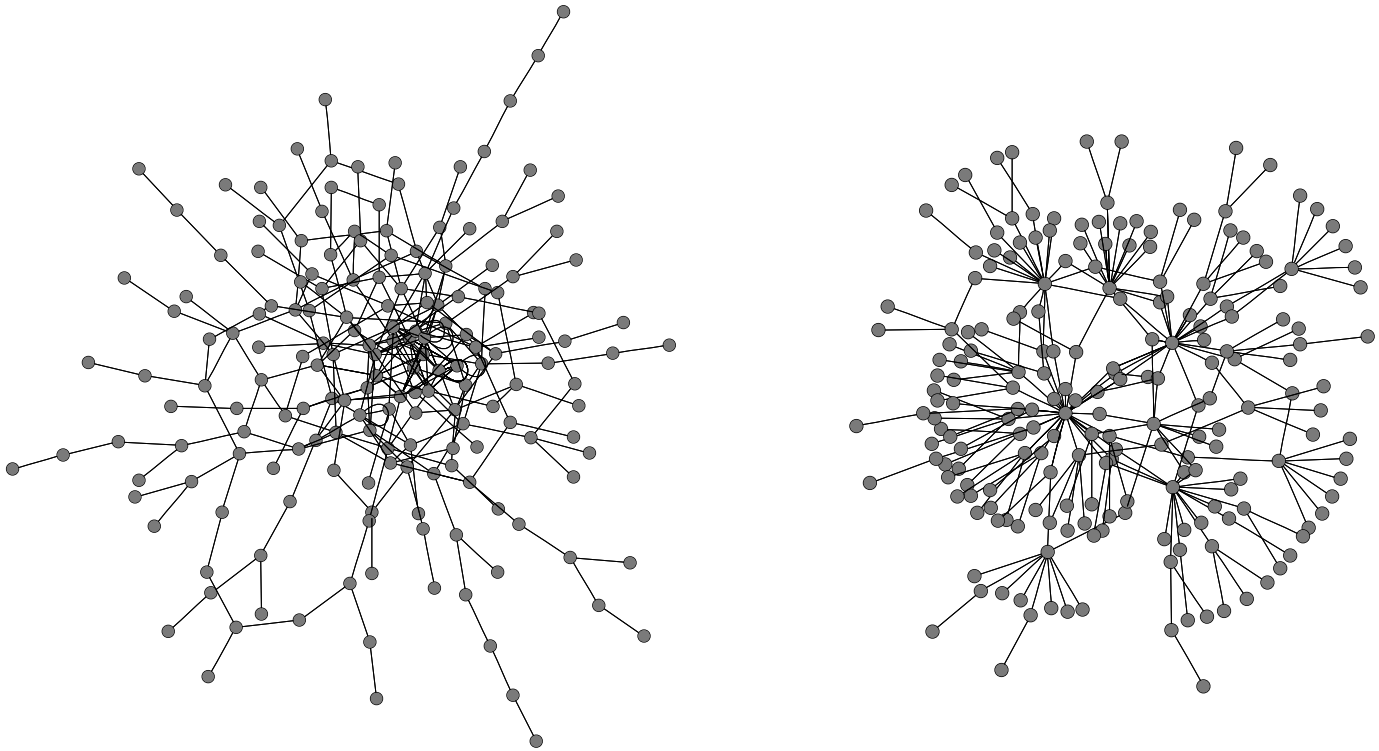
- C'est un coefficient de correlation et de ce fait il prend des valeurs entre −1 (*graphe parfaitement disassortatif*) and +1 (*graphe parfaitement assortatif*).

Assortativité vis-à-vis du degré

- Parmi les valeurs scalaires intrinsèques que l'on peut associer à un sommet, il y a le *degré*.
- Intuitivement, l'*assortativité vis-à-vis du degré* vérifie dans quelle mesure les « ermites fréquentent d'autres ermites et les gens sociables fréquentent d'autres gens sociables ».
- Dans les graphes d'assortativité importante visà-vis du dégre, nous observons une *structure de noyau/périphérie*.
- La formule de l'*indice d'assortativité vis-à-vis du degré* r devient :

$$r := \frac{\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})k_i k_j}{\sum_{i,j}(k_i \delta_{ij} - \frac{k_i k_j}{2m})k_i k_j}.$$

Assortativité vis-à-vis du degré



À gauche un graphe assortatif vis-à-vis du degré, à droite un graphe disassortatif (Newman, 2010).

Assortativité sous graph-tool

- Dans graph-tool il existe une méthode qui calcule aussi bien l'assortativité vis-à-vis d'un scalaire que celle vis-à-vis du degré :

  ```
  (r,var)=scalar_assortativity(g,deg)
  ```

  où
  - `r` est le coefficient obtenu,
  - `var` sa variance,
  - quand `deg` est une propriété de sommet, c'est une assortativité vis-à-vis d'un scalaire (donné par `deg`),
  - quand `deg` est `"in"` (resp. `"out"` ou `"total"`), il s'agit d'assortativité visà-vis du degré entrant (resp. sortant ou total).

Communautés et modularité

- Considérons les classes c_i attachées aux sommets v_i comme des « communautés auxquelles ils appartiennent ».
- Une définition intuitive de la notion de communauté est que :
  - les membres d'une communauté ont un maximum d'arêtes entre eux,
  - et un minimum d'arêtes avec les membres des autres communautés.
- On retrouve la notion de modularité : *plus un graphe est modulaire par rapport à des classes c_i, plus celles-ci induisent des communautés* au sens donné ci-dessus.
- La notion de modularité nous permettra de *détecter* des communautés dans un graphe.

La méthode de Louvain

- Parmi les différentes méthodes de détection de communautés nous allons en décrire une qui a fait ses preuves.
- L'article (Blondel *et al.*, 2008) décrit une méthode intéressante de détection (non-supervisée) de communautés, appelée *méthode de Louvain*[2].
- Le cadre est un peu plus général que celui envisagé dans les transparents précédents : elle s'applique aux graphes à arêtes pondérées. La définition de modularité est toujours

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) \delta_{c_i c_j}$$

sauf qu'ici, A est la matrice des poids des arêtes, et k_i la somme des poids des arêtes dont v_i est extrémité.

[2] À tort : elle aurait dû s'appeler « méthode de Louvain-la-Neuve »...

La méthode de Louvain

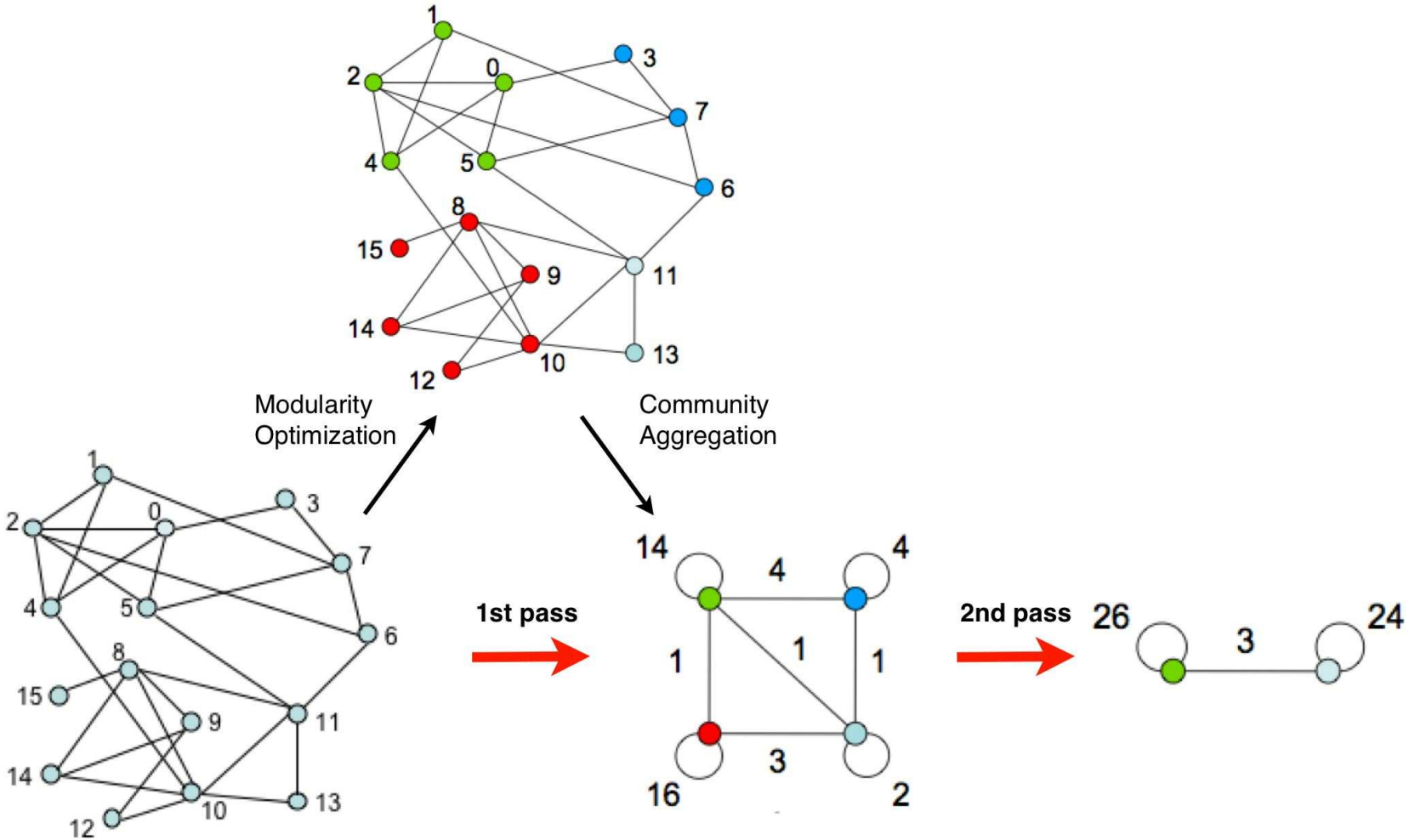
- L'algorithme procède en deux étapes.
- Étape 1.
  1. On commence on considérant chaque sommet comme une communauté distincte,
  2. pour chaque sommet v_i, prendre ses voisins v_j et calculer le gain de modularité lorsqu'on affecte v_j à la communauté de v_i – on effectue la ré-affectation à chaque fois que ce gain est strictement positif,
  3. on s'arrête lorsqu'un maximum de modularité local est atteint, c'est-à-dire quand on ne peut plus ré-affecter des sommets sans baisser la modularité.

La méthode de Louvain

- Étape 2.
  1. On contracte tous les sommets d'un groupe en un seul (méta-)sommet. Les poids des arêtes inter-groupes deviennent des poids d'arêtes entre les nouveaux sommets. Les arêtes intra-groupes deviennent des lacets pondérés.
  2. À chaque étape on mesure la modularité du graphe.
  3. On continue ainsi jusqu'à qu'il ne reste plus qu'un groupe.
  4. On revient ensuite pour choisir la combinaison de nombre de communautés et de valeur de modularité la plus adaptée.

- Cet algorithme est très rapide et, grâce à sa méthode de contraction, peut être appliqué à de très grands réseaux.
- Il a été implémenté pour le package networkx : `http://perso.crans.org/aynaud/communities/`

La méthode de Louvain



(Blondel *et al.*, 2008)

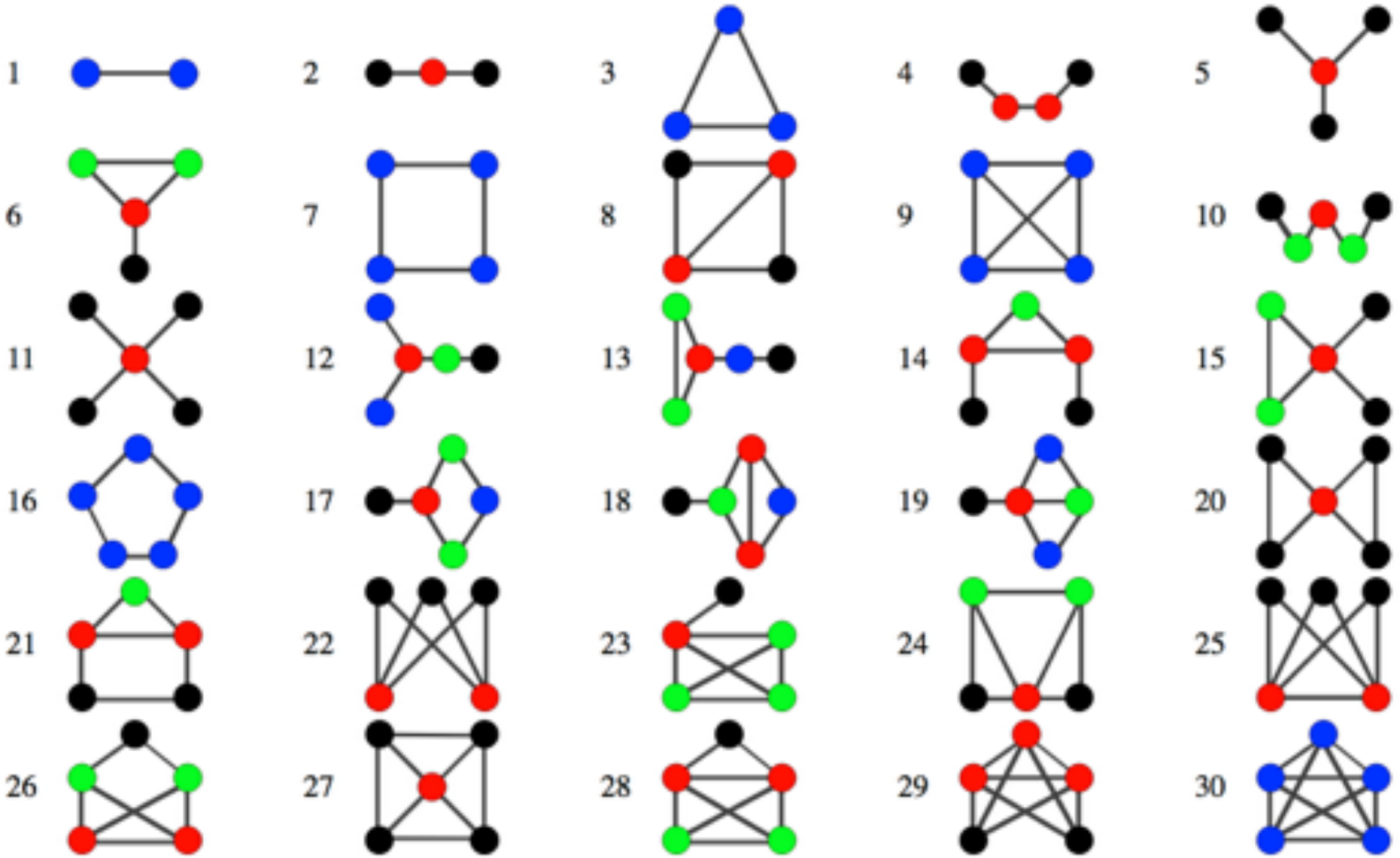
partie V

Parenthèse : Graphlets

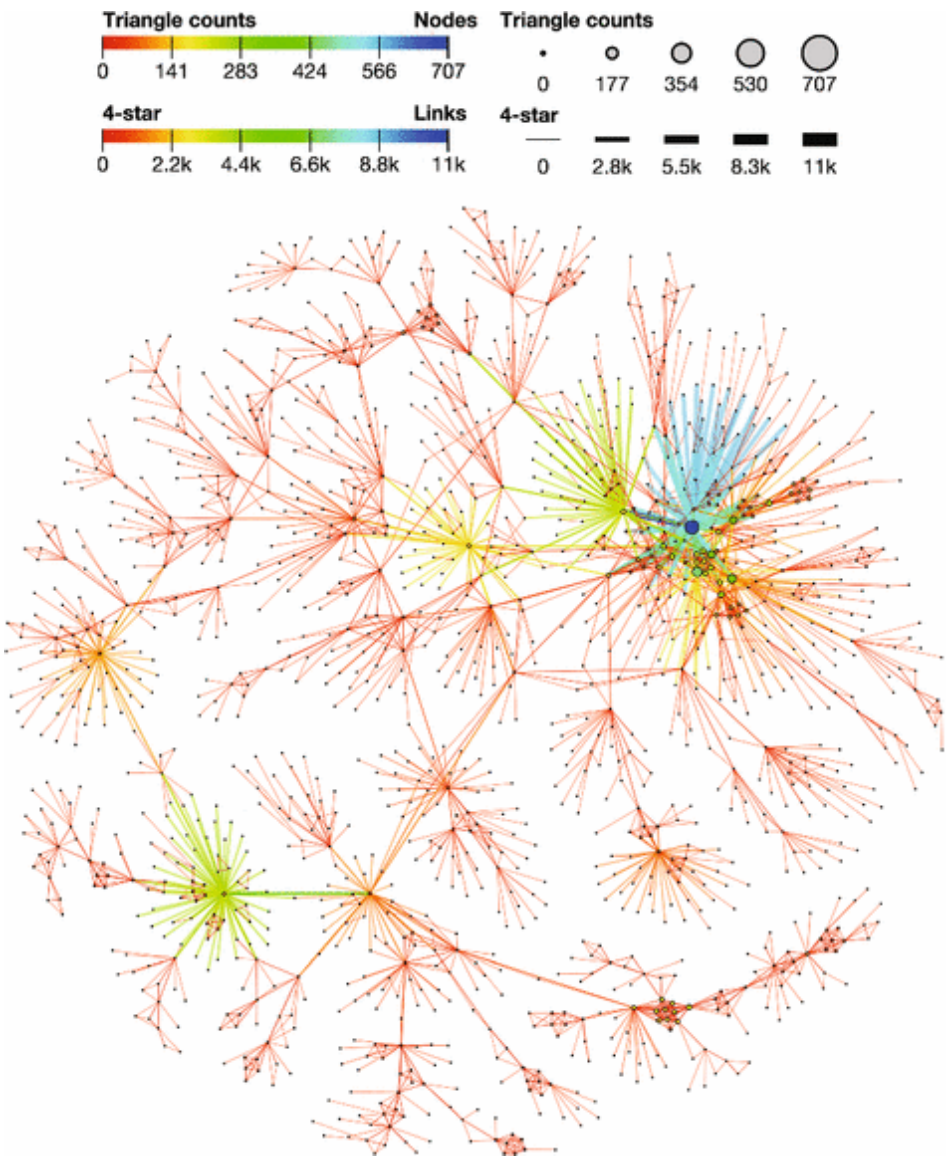
Graphlets

- Conformément à la méthode structuraliste, on peut se demander s'il y a des unités minimales distinctes qui permettent de décrire les graphes.
- Une réponse possible est celle des *graphlets* : introduits en bioinformatique (Pržulj, 2004) il s'agit de graphes d'ordres 2–5, non isomorphes entre eux.
- Il existe au total 30 graphlets d'ordres 2–5. Dans le transparent suivant des couleurs sont appliquées aux groupes de sommets interchangeables.
- Les graphlets sont implémentés dans le package igraph.

Graphlets


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

Graphlets



Graphe tiré de (Ahmed *et al.*, 2016) dont les sommets sont des gènes et des désordres génétiques. La coloration représente des communautés obtenues à partir des graphlets. En bleu clair : le cancer du colon (!), et en vert-jaune à gauche : la surdité.

partie VI

Coloration

Le nombre chromatique d'un graphe

- Une *k-coloration* G d'un graphe est une fonction $c : G \to \{1, \ldots, k\}$ telle que si xx' est une arête, $c(x) \neq c(x')$.
- Pour un graphe G le plus petit nombre k tel qu'il est existe une k-coloration est appelé *nombre chromatique* $\chi(G)$.

Application pratique

L'école de dresseurs de phoques de Plouzané propose 47 modules aux 349 élèves de première année. Chaque élève peut s'inscrire à 3 modules. Étant donné qu'un élève ne peut pas être physiquement présent à deux cours en même temps, comment mettre les cours en parallèle pour les faire en un minimum de temps ?

- G est discret ssi $\chi(G) = 1$. G est biparti (non discret) ssi $\chi(G) = 2$. $\chi(K_n) = n$.

Le nombre chromatique d'un graphe

Méthode de coloration : On énumère les sommets. On colorie chaque sommet v_j avec le i le plus petit tel qu'aucun voisin de v_j ne soit de cette couleur. Chaque v_j a $\|N(v_j)\| \leq \Delta(G)$ voisins (où $\Delta(G)$ est le plus grand degré de G), donc on peut toujours colorier avec $\Delta(G) + 1$ couleurs.

Théorème(s)

Proposition

On a pour tout graphe G : $\chi(G) \leq \frac{1}{2} + \sqrt{2\|G\| + \frac{1}{4}}$.

Théorème (Célèbre théorème des quatre couleurs)

On a $\chi(G) \leq 4$ pour tout graphe planaire.

- Problème posé par Guthrie en 1852.
- Solution donnée par Appel & Haken en 1976 (741 pages de listing...).
- On va démontrer une version plus faible :

Le nombre chromatique d'un graphe

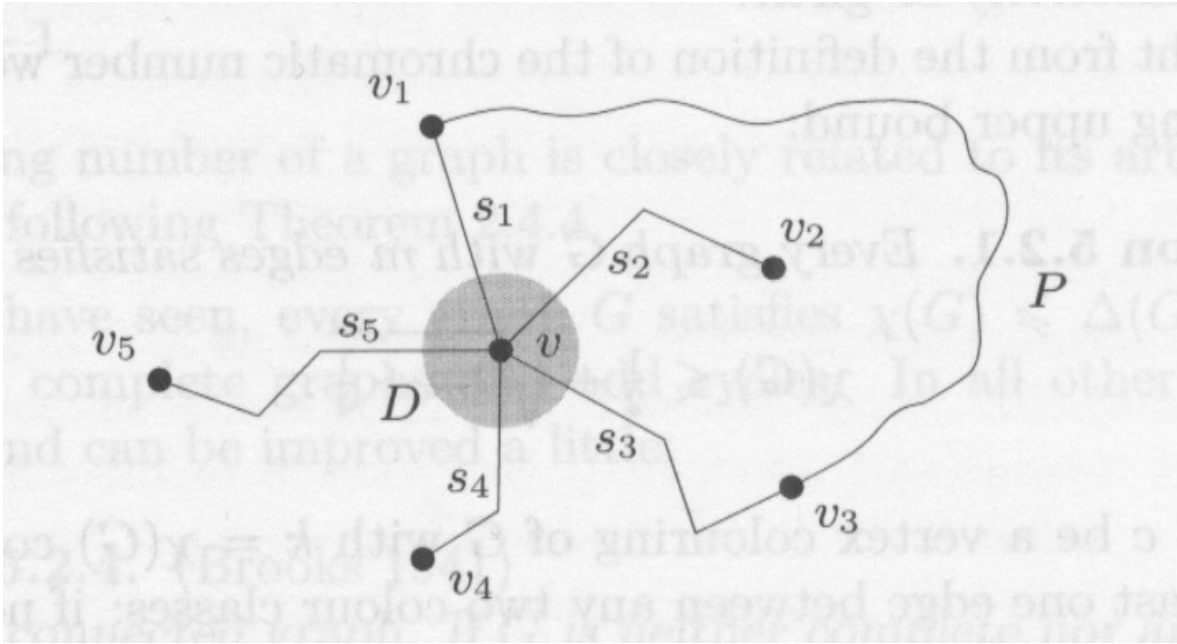
Théorème (Théorème des cinq couleurs)

On a $\chi(G) \leq 5$ pour tout graphe planaire.

Preuve : Par récurrence. Pour $n = 5$ cinq couleurs suffisent. Supposons l'assertion vraie pour les graphes d'ordre $|G| - 1$. Soit G planaire. Par le corollaire du théorème d'Euler, $\|G\| \leq 3|G| - 6$ et donc le degré moyen $d(G) = \frac{2\|G\|}{|G|} \leq \frac{2(3|G|-6)}{|G|} < 6$ ce qui signifie qu'on est sûr qu'il existe un sommet de degré ≤ 5. S'il existe un sommet de degré ≤ 4 on l'enlève, on applique l'hypothèse de récurrence, on le rajoute avec une couleur manquante et bingo.

Le nombre chromatique d'un graphe

Supposons que v est de degré 5 et que les 5 voisins v_i ont les 5 couleurs $\{1, 2, 3, 4, 5\}$:



Soit $H = G - v$. Montrons que tout chemin P de H (avec $v_2, v_4 \notin P$ qui relie v_1 à v_3 sépare v_2 de v_4. Dans G, le cycle vv_1Pv_3v sépare v_2 et v_4. Donc P les sépare dans H (montrer sur le dessin).
Soit maintenant $H_{i,j}$ le sous-graphe induit par H de couleurs i et j uniquement. De deux choses l'une : soit v_1 et v_3 sont dans la même composante connexe de $H_{1,3}$ soit ils ne le sont pas.

Le nombre chromatique d'un graphe

(a) Ils ne le sont pas. Alors j'échange les couleurs 1 et 3 dans la composante connexe de v_1, ils sont donc tous les deux de couleur 3, je mets v de couleur 1 et finito.
(b) Ils le sont. Alors $H_{1,3}$ contient un chemin qui relie v_1 et v_3 et qui ne passe pas par v_2 et v_4. Il sépare donc v_2 de v_4 dans H, ils se trouvent donc dans deux composantes connexes de $H_{2,4}$ différentes. Même truc.

Théorème (Théorème de Brooks (1941))

Soit G un graphe connexe qui n'est ni complet ni un cycle impair, alors on a $\chi(G) \leq \Delta(G)$.

Le nombre chromatique d'un graphe

Preuve : Par recurrence sur $\Delta(G)$. Pour $\Delta(G) \leq 2$ c'est ok (du moment que le cycle n'est pas impair !).
On considère que $\Delta(G) \geq 3$ et que ça marche pour des graphes plus petits.
Supposons que $\chi(G) > \Delta(G)$ (*). Soit $v \in G$ et $H = G - v$. H n'est peut-être pas connexe, je prends une comp.conn. H' de H quelconque, l'hypothèse s'applique : $\chi(H') \leq \Delta(H') \leq \Delta(G)$ sauf si H' est complet ou un cycle impair, dans lequel cas $\chi(H') = \Delta(H') + 1$. Mais ceci est toujours $\leq \Delta(G)$ puisque dans ce cas, chaque sommet de H' est à degré maximal et une parmi elles était reliée à v. Donc H peut être $\Delta(G)$-colorié mais pas G.
Donc : toute $\Delta(G)$-coloration de H utilise toutes les couleurs $\{1, \ldots, \Delta(G)\}$ au niveau des voisins de v et en particulier $d(v) = \Delta(G)$.

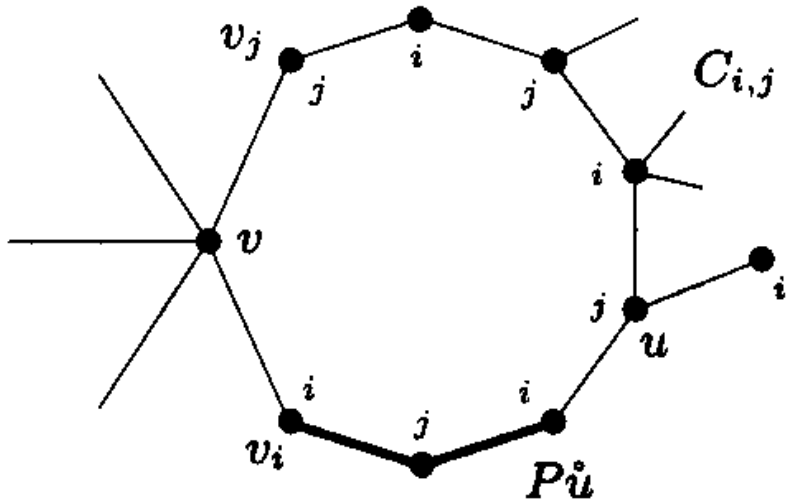
Le nombre chromatique d'un graphe

On appelle ces voisins $v_1, \ldots, v_{\Delta(G)}$ et $H_{i,j}$ comme pour le th. des 5 coul. On a forcément : *pour tout i, j, les sommets v_i et v_j se trouvent sur la même comp. conn. $C_{i,j}$ de $H_{i,j}$* (**). Si ce n'était pas le cas, on pourrait appliquer le truc du th.des5c. et avoir une couleur de moins.

Le nombre chromatique d'un graphe

On peut dire même plus : *$C_{i,j}$ est un chemin reliant v_i et v_j*. Pour le montrer par l'absurde, soit P un v_i-v_j-chemin ds $C_{i,j}$. Comme $d_H(v_i) \leq \Delta(G) - 1$, les voisins de v_i ont des couleurs différentes (sinon on aurait pu donner une autre couleur à v_i et la sienne à v, ce qui contredit l'hypothèse (*)). Donc il n'y a qu'un seul voisin de v_i de couleurs i ou j, idem pour v_j.

Si $C_{i,j} \neq P$ cela veut dire qu'il existe un sommet intérieur à P de couleur, par exemple, j qui a (au moins) trois voisins de couleur i. Soit u le premier tel sommet.



Le nombre chromatique d'un graphe

Autre constatation : *pour i, j, k distincts, $C_{i,j} \cap C_{j,k} = v_j$.* (***) Si c'était le cas contraire, avec $u \in \cap$, alors u aurait deux voisins i et deux voisins k, donc en appliquant le même truc, on pourrait recolorier u et contredire (**).
Finissons donc la preuve du théorème : si les voisins de v sont voisins entre eux, comme chacun a déjà $\Delta(G)$ voisins (avec tous les $C_{*,*}$ qui partent) dans le voisinage de v, il ne peut pas en avoir plus, donc G se résume à $N(v) \cup v$, et donc il est complet, ce qu'il ne devait pas être.
Soit $v_1v_2 \not\in G$. Donc après v_1 dans $G_{1,2}$ il y a un u de couleur 2 qui n'est pas v_2. Mais alors ce u appartient aussi à $C_{2,3}$ et donc $u \in C_{1,2} \cap C_{2,3}$, ce qui contredit (***).

partie VII

Hamilton et Euler

Graphes euleriens

- On appelle *parcours* P, un chemin $x_1 \dots x_n$ dont toutes les arêtes sont distinctes. Un parcours fermé est appelé un *tour*.
- Un tour est *eulérien* s'il passe par *toutes* les arêtes de G. On dit alors que le graphe est eulérien.

Théorème (Euler (1736))

Un graphe connexe est eulérien ssi tous ses sommets sont de degré pair.

Graphes hamiltoniens et semi-hamiltoniens

- On appelle *chemin hamiltonien* un chemin qui passe une fois et une seule par tous les sommets d'un graphe.
- Un graphe qui admet un chemin hamiltonien est appelé *graphe semi-hamiltonien*.
- $x_1x_2 \dots x_nx_1$ est un *cycle hamiltonien* si $x_1x_2 \dots x_n$ est un chemin ha
- Un graphe qui admet un cycle hamiltonien est appelé *hamiltonien*.
- Une *orientation* d'un graphe est un choix d'orientation pour chaque arête. Une orientation d'un graphe *complet* est appelée un *tournoi*.

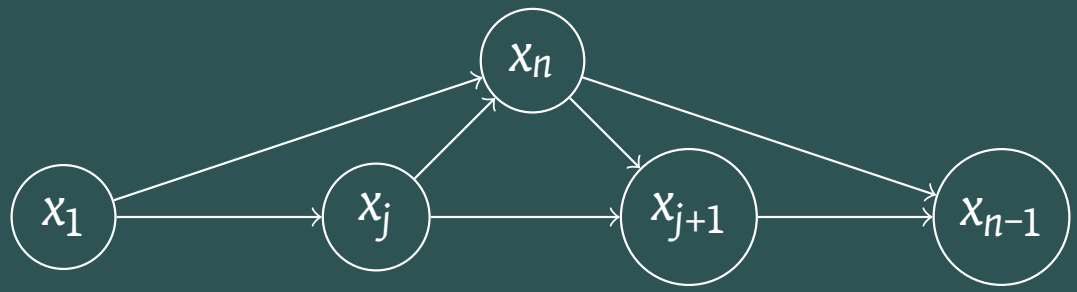
Graphes hamiltoniens et semi-hamiltoniens

Proposition

Tout tournoi est un graphe semi-hamiltonien.

Preuve : Par récurrence.
K_2 admet un chemin hamiltonien.
Supposons que K_{n-1} admet un ch. ham. Ajoutons un point x_n et toutes les arêtes pour en faire K_n. Supposons qu'il existe une orientation des nouvelles arêtes telle qu'il n'y a pas de chemin hamiltonien. Tracer les lignes :

Graphes hamiltoniens et semi-hamiltoniens



Si la première flèche va vers le bas, alors on a un chemin hamiltonien. Donc elle va vers le haut. Idem, la dernière va forcément vers le bas. Si celle de x_j va vers le haut et celle de x_{j+1} vers le bas on un chemin hamiltonien. Donc ce phénomène ne peut pas se produire. Soit k le plus grand nombre pour lequel la flèche va vers le haut (k est au moins 1). Si $k = n - 2$ on a un chemin hamiltonien. Sinon la flèche de $k + 1$ va forcément vers le bas, absurde. Donc cette orientation n'existe pas. Donc on a toujours un chemin hamiltonien.

Graphes hamiltoniens et semi-hamiltoniens

Théorème (Dirac (1952))

Tout graphe G avec $|G| \geq 3$ et degré minimum $\geq \frac{|G|}{2}$ est hamiltonien.

Application : On a un groupe de n personnes dont chacune connaît au moins $\frac{n}{2}$ autres dans le groupe. Alors on peut s'asseoir autour une table de manière à ce que chacun soit assis entre deux de ses connaissances.
Preuve : Soit G un tel graphe. G est forcément connexe, sinon : si on prend la plus petite des deux composantes, le degré de tout sommet serait $< \frac{n}{2}$ puisque cette composante a au plus $\frac{n}{2}$ sommets et le degré est toujours plus petit de 1 que le nombre de sommets.

Graphes hamiltoniens et semi-hamiltoniens

Soit $P = x_0 \dots x_k$ un chemin le plus long de G. Par maximalité tous les voisins de x_0 et de x_k sont sur P. Comme au moins $\frac{n}{2}$ des sommets $x_0, \dots, x_{k-1}$ sont adjacents à x_k, au moins $\frac{n}{2}$ parmi ces mêmes sommets ont la propriété qu'il existe une arête x_0x_{i+1} (ATTENTION, on a au moins $\frac{n}{2}$ sommets qui ont la propriété que leur voisin suivant est connecté à x_0, donc : *la propriété porte sur le voisin* !!), et n est plus grand que k, donc forcément quelque part au milieu il existe un i qui a les deux propriétés, c'est-à-dire tel qu'on ait aussi bien des arêtes x_0x_{i+1} que x_ix_k.
Alors le cycle $C = x_0x_{i+1}Px_kx_iPx_0$ est hamiltonien : si ce n'était pas le cas, on aurait un sommet $y \notin C$. G est connexe donc il existe des chemins à partir de tous les points de C vers y, prenons le plus court $y \dots x_j$. Il suffirait alors de prendre $y \dots x_jCx_{j-1}$ pour avoir un chemin plus long que P : absurde.

Références recommandées 1/2

- Cormen, Stein, Rivest & Leiserson, *Introduction to Algorithms*, 3e édition, MIT Press, 2009. Connu comme « CSRL », c'est la Bible des algorithmes. Les solutions des exercices sont fournis aux enseignants après vérification de leur statut. Le même, traduit en français : *Algorithmique*, 3e édition, Dunod, 2010.
- Newman, *Networks*, 2e édition, Oxford University Press, 2018. Les solutions des exercices sont fournis aux enseignants après vérification de leur statut.
- Berge, *Graphes et hypergraphes*, Dunod, 1970. Un chef d'œuvre de clarté et d'élégance. Pour amateurs de bonnes mathématiques à la française.
- Berge, « Qui a tué le duc de Densmore ? », *La Bibliothèque oulipienne*, n° 67, Paris, 1994, pages 75–90. Un roman à l'Agatha Christie, dont la solution est donnée par la théorie des graphes.

Références recommandées 2/2

- Blondel, Guillaume, Lambiotte & Lefebvre, « Fast unfolding of communities in large networks », *Journal of Statistical Mechanics : Theory and Experiment*, 2008 (10) P10008.
- Pržulj, Corneil & Jurisica, « Modeling Interactome, Scale-Free or Geometric ? », *Bioinformatics* 2004, 20 (18), 3508–3515.
- Ahmed *et al.*, « Graphlet Decomposition : Framework, Algorithms, and Applications », *Knowledge and Information Systems* 2016, 50 (3) : 689–722.