Graph Theory in Operational Research
(a brief overview)

Damien Leprovost

Laboratoire LIMICS
Inserm – UPMC – Paris 13
http://www.damien-leprovost.fr

Lecture objectives

- **Understanding the benefits of operational research**
- **Master the useful basics of graph theory**
- **Be able to model a concrete algorithm, from an abstract problem**
- **. . . And know how to solve it!**

Plan du cours

Outline

1. Introduction: some definitions
   - Operational research
   - Graphs

Definition of Operational research

Definition

A set of rational methods and techniques for analysis and synthesis of organizational processes, used to develop better decisions.

Operational research defines neither the criteria, nor objectives, nor the decisions!

Origines

- World War II (Staff of the British Navy)
  - Patrick Blackett, physicist (1940)
  - Application to military *operations*: supply paths, location of radars, ...

- In fact much older
  - *Expected value*, of Blaise Pascal and Pierre de Fermat (1654)
  - *Décision dans l'incertain* (Decision under uncertainty) of Jacques Bernoulli (1713)

Application domain

- Domains where common sense is deficient.
- Especially:
  - combinatorial problems;
  - random;
  - competitive situations.

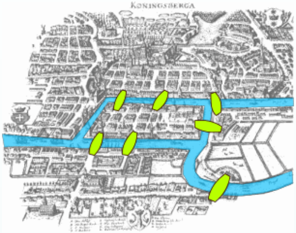
Outline

1. Introduction: some definitions
   - Operational research
   - Graphs

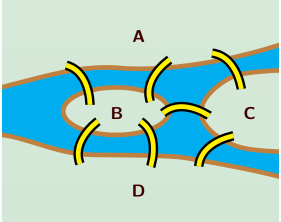
Definition of graphs

- A graph is first of all:
  - a set of elements;
  - a set of relations between these elements.
- Two families:
  - undirected graphs;
  - directed graphs.
- Many possible conceptualizations and models

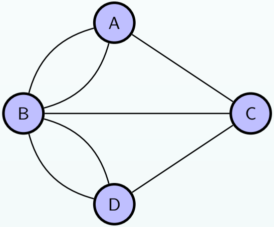
Seven Bridges of Königsberg (Euler, 1735)



$\Leftrightarrow$



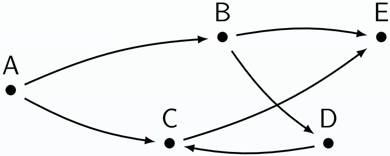
Representation as a graph:



$\Leftrightarrow$

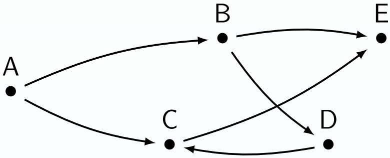
$$G = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \left| \begin{array}{cccc} - & 2 & 1 & - \\ 2 & - & 1 & 2 \\ 1 & 1 & - & 1 \\ - & 2 & 1 & - \end{array} \right| \end{array}$$

Vocabulary of directed graphs



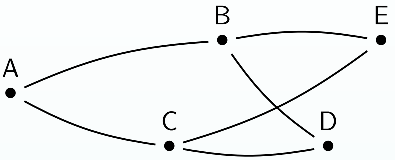
- Set of vertices $X = \{A, B, C, D, E\}$
- Arcs : ordered pairs of vertices, subset of X
  - $U \subset X \times X$, binary relation of X
  - $U = \{(A, B), (A, C), (B, D), (B, E), (C, E), (DC)\}$
  - $G = (X, U)$ is a possible notations of G

Vocabulary of directed graphs



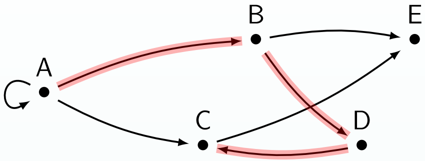
- Map Γ^+, as successor map, defined on X.
  - $\Gamma^+(A) = \{B, C\}$; A is a *graph input*
  - $\Gamma^+(E) = \emptyset$; E is a *graph output*
  - $G = (X, \Gamma^+)$ et $G = (X, \Gamma^-)$ are two possible notations of G

Vocabulary of undirected graphs



- Unordered pairs of vertices, called egdes
  - $[A, B] \Leftrightarrow [B, A]$
- At any directed graph corresponds a single undirected graph
  - "Disorientation" :
    $U = \{(A, C), (B, D), (D, B)\} \Rightarrow U = \{[A, C], [B, D]\}$

Vocabulary of graphs

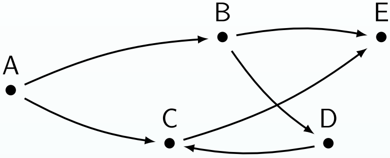


- Walk: sequence of vertices and edges
  - Called closed walk if its first and last vertices are the same
  - Traditionnally, open walks are refered as paths
- Trail: a walk in which all the edges are distinct
- Chain: a walk in which all the vertices are distinct
  - A closed chain is a cycle
- A 1-length path is a loop
- A graphe is **connected** when there is a path between every pair of vertices

Vocabulary of graphs

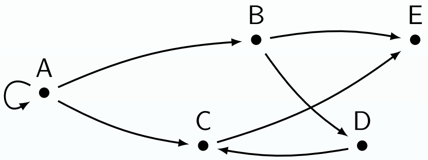
- Special trails:
  - A trail is called Eulerian if it uses all edges precisely once
    - A graph with one Eulerian trail is Eulerian, and *traversable*
  - It is Hamiltonian if it uses all vertices precisely once
    - A graph with one Hamiltonian trail is Hamiltonian

Vocabulary of graphs



- Outdegree of x : $d_x^+ = \text{card } \Gamma^+(x)$
- Indegree of x : $d_x^- = \text{card } \Gamma^-(x)$
- Degree de x : number of edges having a extremity on x

Matrix representation

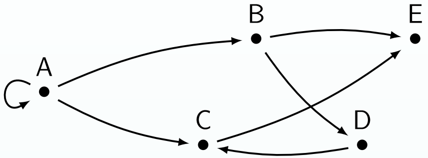


Definition of an adjancency matrix

$$M = \begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 1 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \Rightarrow \begin{array}{l} \Gamma^+(A) = \{A, B, C\} \\ \Gamma^+(B) = \{D, E\} \\ \Gamma^+(C) = \{E\} \\ \Gamma^+(D) = \{C\} \\ \Gamma^+(E) = \{\emptyset\} \end{array}$$

- Example of use: determining degrees

Matrix representation



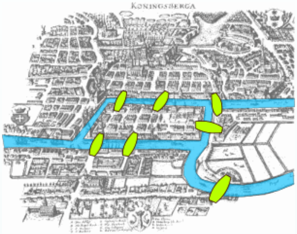
- Other example: M^k = cardinality of unique paths with a length of k

$$M = \begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 1 & 1 & 1 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \qquad M^3 = \begin{array}{c|ccccc|} & A & B & C & D & E \\ \hline A & 1 & 1 & 2 & 1 & 2 \\ B & 0 & 0 & 0 & 0 & 1 \\ C & 0 & 0 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

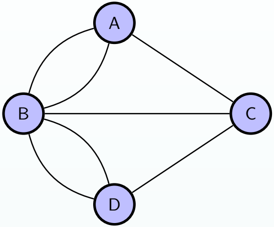
Utility in Operational Research

- Represent all kind of situation in organizational phenomena
- Modelize, for example:
  - Transportation network
    - Using the Kirchhoff's circuit laws
  - Relations systems
    - Using the transitive law
    - Scheduling problems
  - Systems of states and transitions
    - Markov chains and processes
    - Petri Nets

Test: Are you Euler?

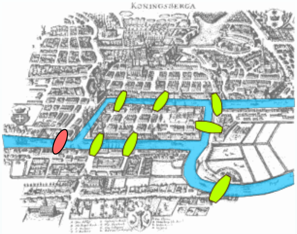


$\Rightarrow$

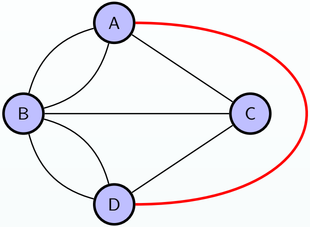


$$
G = \begin{array}{c|cccc|}
 & A & B & C & D \\
\hline
A & - & 2 & 1 & - \\
B & 2 & - & 1 & 2 \\
C & 1 & 1 & - & 1 \\
D & - & 2 & 1 & - \\
\hline
\end{array}
\Rightarrow
\begin{array}{l}
\Gamma(A) = 3 \\
\Gamma(B) = 5 \\
\Gamma(C) = 3 \\
\Gamma(D) = 3
\end{array}
$$

Test: Are you Euler?

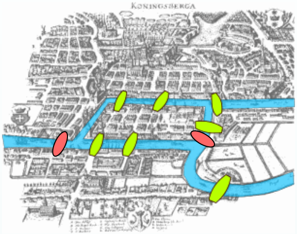


$\Rightarrow$

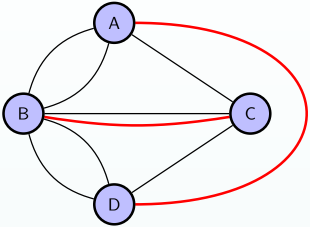


$$G = \begin{array}{c|cccc|} & A & B & C & D \\ \hline A & - & 2 & 1 & - \\ B & 2 & - & 1 & 2 \\ C & 1 & 1 & - & 1 \\ D & - & 2 & 1 & - \\ \hline \end{array} \Rightarrow \begin{array}{l} \Gamma(A) = 4 \\ \Gamma(B) = 5 \\ \Gamma(C) = 3 \\ \Gamma(D) = 4 \end{array}$$

Test: Are you Euler?



$\Rightarrow$



$$G = \begin{array}{c|cccc|} & A & B & C & D \\ \hline A & - & 2 & 1 & - \\ B & 2 & - & 1 & 2 \\ C & 1 & 1 & - & 1 \\ D & - & 2 & 1 & - \\ \hline \end{array} \Rightarrow \begin{array}{l} \Gamma(A) = 4 \\ \Gamma(B) = 6 \\ \Gamma(C) = 4 \\ \Gamma(D) = 4 \end{array}$$

Outline

2 Graph algorithms

- Definition of an algorithm
- Computational Complexity
- First algorithms: graph traversal
- Dynamic programming

Definition of an algorithm[1]

Definition

An algorithm is a finite and unambiguous serie of transactions or instructions for solving a problem.

- Property of Knuth:

  - Finiteness: *"An algorithm must always terminate after a finite number of steps"*
  - Definiteness: *"Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"*
  - Input: *"... quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects"*
  - Output: *"... quantities which have a specified relation to the inputs"*
  - Effectiveness: *"... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil"*

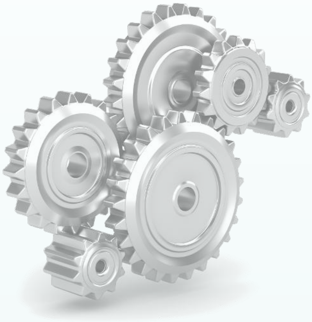
[1] After the name of the Persian mathematician *Al-Khwârizmî* ($\sim$ 780 – 850)

Outline

2. Graph algorithms
   - Definition of an algorithm
   - Computational Complexity
   - First algorithms: graph traversal
   - Dynamic programming

Computational Complexity

- Evaluate and compare the effectiveness of algorithms
- Two main criteria:
  - Calculation time
  - Memory size
- Calculation time is always limited!



The notation O

- O class of function “top-confining”
- $\exists c > 0, \exists n_0$ such that $\forall n \geq n_0, g(n) \leq c \cdot f(n) \Leftrightarrow g \in O(f)$
- $O(1)$ all functions bounded above by a constant from a certain rank
- $O(c \cdot f) = O(f)$: The complexity is used modulo a multiplicative constant (asymptotic behavior)
  - $O(100000n^2) = O(0,01n^2) = 0(n^2)$
  - $\forall P(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_p n^p, P \in O(n^p)$

Execution time of the usual functions[2]

$f(n) =$	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\log n$	10^{-9} s	$2 \cdot 10^{-9}$ s	$3 \cdot 10^{-9}$ s	$6 \cdot 10^{-9}$ s	$9 \cdot 10^{-9}$ s
n	10^{-8} s	10^{-7} s	10^{-6} s	10^{-3} s	1 s
$n \log n$	10^{-8} s	$2 \cdot 10^{-7}$ s	$3 \cdot 10^{-6}$ s	$6 \cdot 10^{-3}$ s	9 s
n^2	10^{-7} s	10^{-5} s	10^{-3} s	1000 s	32 years
n^3	10^{-6} s	10^{-3} s	1 s	32 years	$3 \cdot 10^4$ My
2^n	10^{-6} s	$3 \cdot 10^8$ My	10^{273} My	–	–

- $O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(2^n) \subset O(e^n) \subset O(n^n)$

[2] On the basis of one billion operations per second (1Ghz)

Outline

2. Graph algorithms
   - Definition of an algorithm
   - Computational Complexity
   - First algorithms: graph traversal
   - Dynamic programming

Depth-first search: computing connectivity

- Goal: Determining connectivity of a graph, the number of connected components
- Method: Depth-first search (DFS)

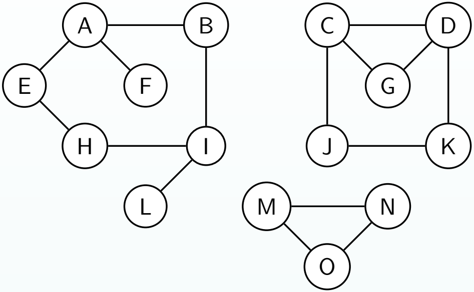
Depth-first search

An unmarked vertex is open if and only if it is adjacent to the last vertex previously open[3]. If such a vertex does not exist, the last opened vertex is closed.

- Using a stack (abstract data type)
  - Last In, First Out (LIFO)

[3] *successor of the last vertex* in an oriented graph

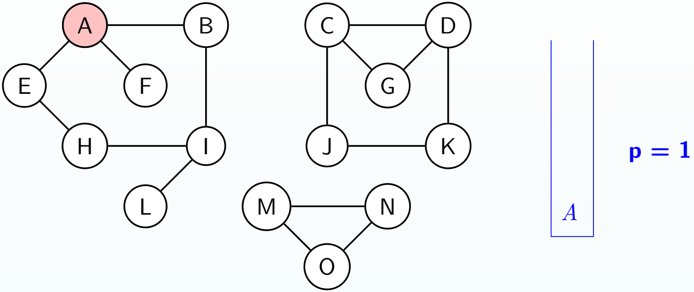
Depth-first search: computing connectivity



p = 0

```
all vertices are unmarked; p → 0; Stack → ∅
while it exists an unmarked vertex s, open and push s; p → p + 1
  while Stack is not empty, do
    if it exists an unmarked vertex y adjacent to the top x of Stack, do
       open and push y;
    else close and pop x; c(x) = p
```

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p + 1$

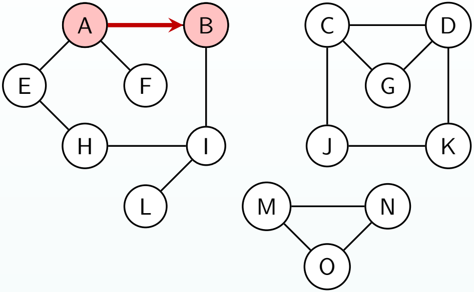
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

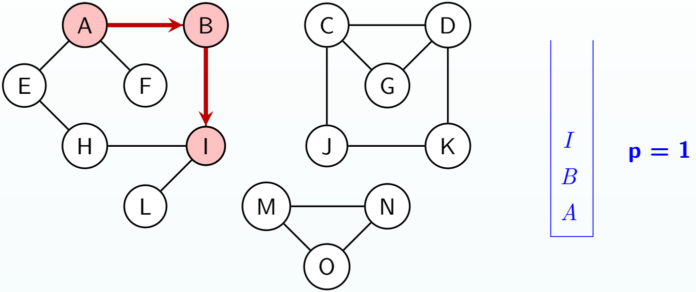
Depth-first search: computing connectivity



$\mathbf{p = 1}$

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$
while it exists an unmarked vertex s, open and push s; $p \to p+1$
  while $Stack$ is not empty, do
    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do
      open and push y;
    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

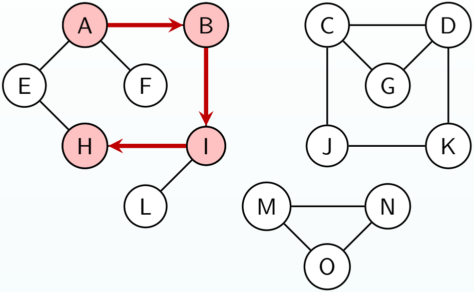
3: while $Stack$ is not empty, do

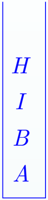
4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity





$\mathbf{p = 1}$

1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

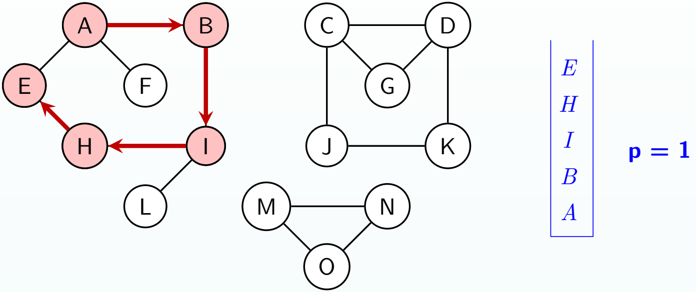
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

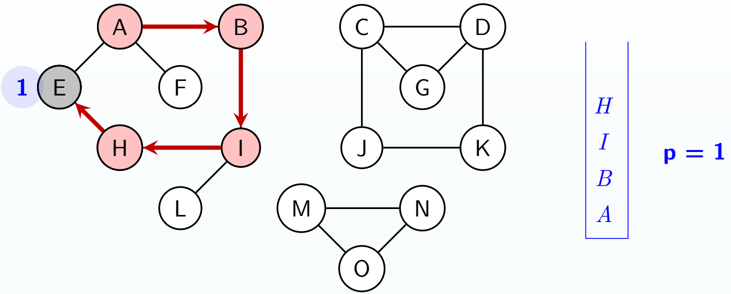
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

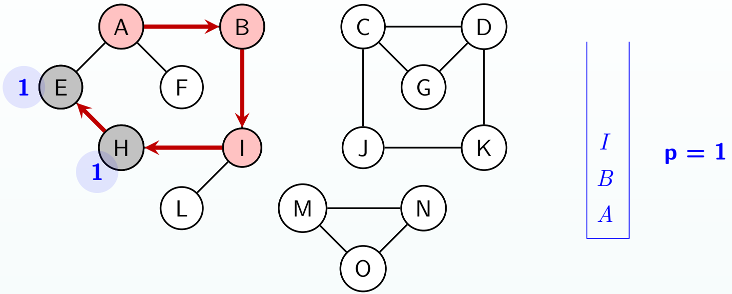
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

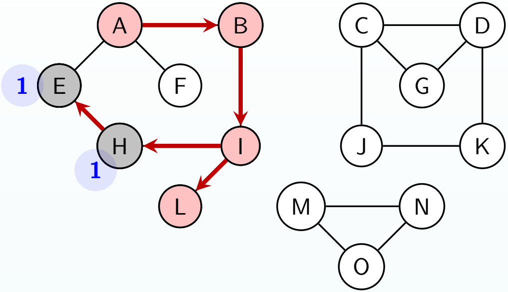
3: while $Stack$ is not empty, do

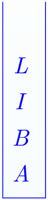
4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity





p = 1

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p + 1$

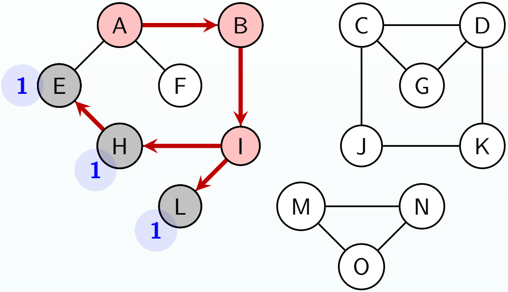
  while $Stack$ is not empty, do

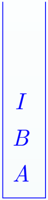
    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity





p = 1

1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p + 1$

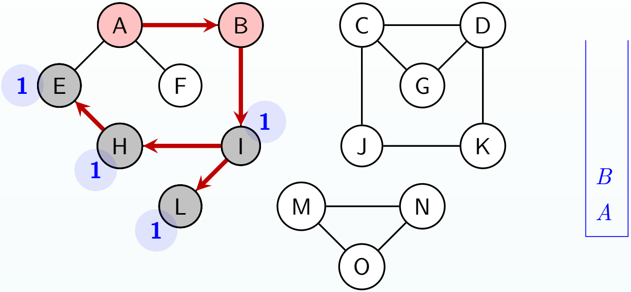
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



p = 1

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

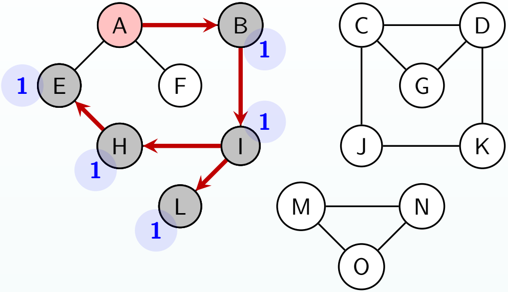
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



p = 1

1: all vertices are unmarked; $p \rightarrow 0$; $Stack \rightarrow \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \rightarrow p + 1$

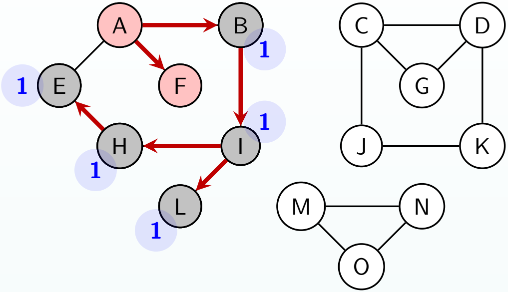
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

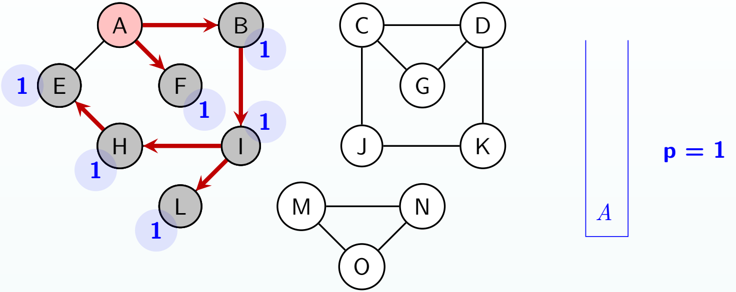
Depth-first search: computing connectivity



p = 1

```
all vertices are unmarked; p → 0; Stack → ∅
while it exists an unmarked vertex s, open and push s; p → p + 1
  while Stack is not empty, do
    if it exists an unmarked vertex y adjacent to the top x of Stack, do
       open and push y;
    else close and pop x; c(x) = p
```

Depth-first search: computing connectivity



all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

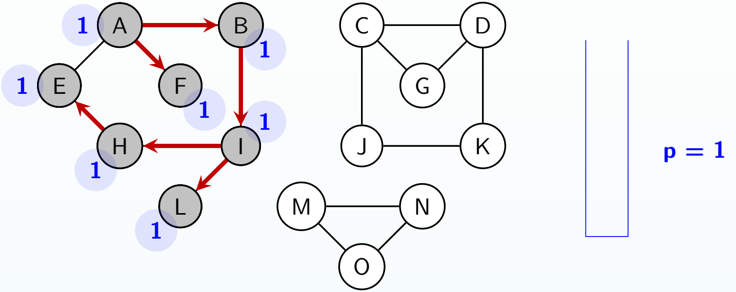
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

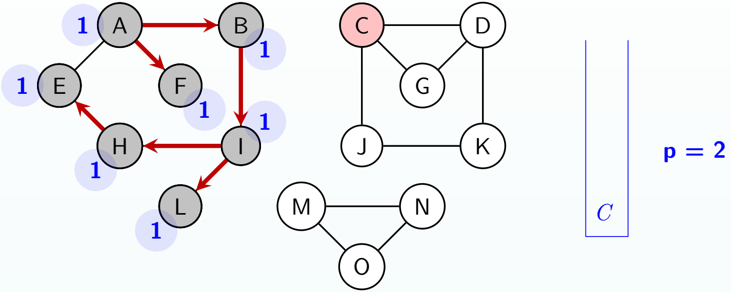
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

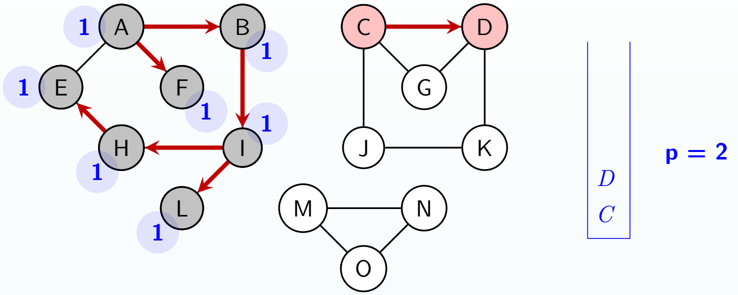
6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity




```
all vertices are unmarked; p → 0; Stack → ∅
while it exists an unmarked vertex s, open and push s; p → p + 1
  while Stack is not empty, do
    if it exists an unmarked vertex y adjacent to the top x of Stack, do
      open and push y;
    else close and pop x; c(x) = p
```

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

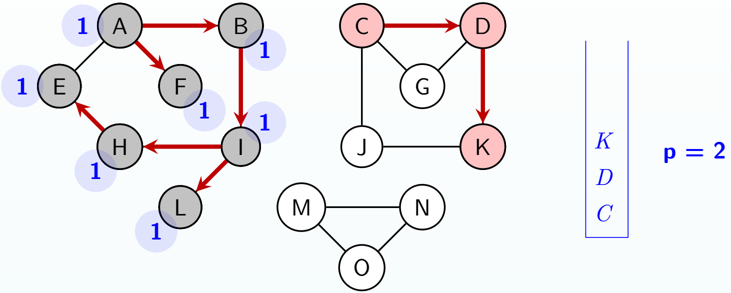
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

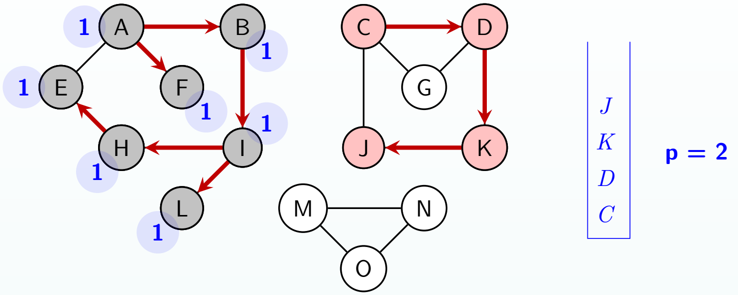
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \rightarrow 0$; $Stack \rightarrow \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \rightarrow p+1$

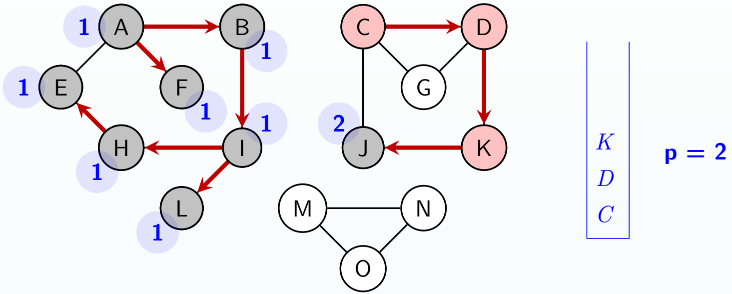
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

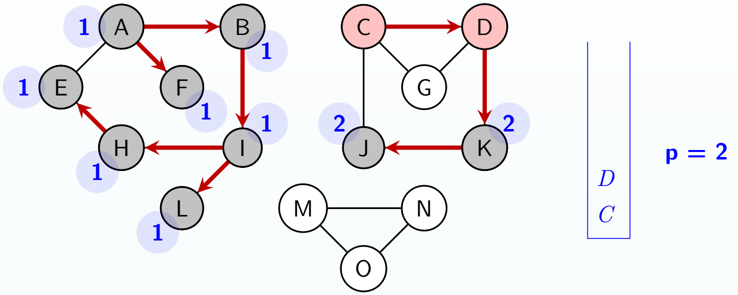
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

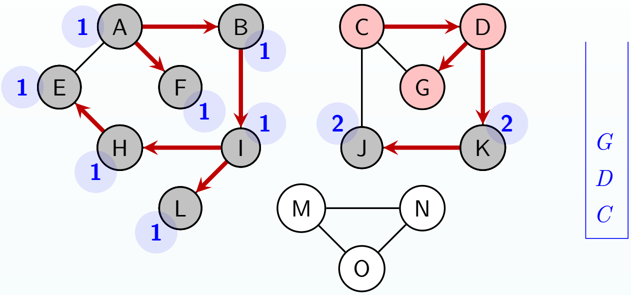
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



p = 2

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

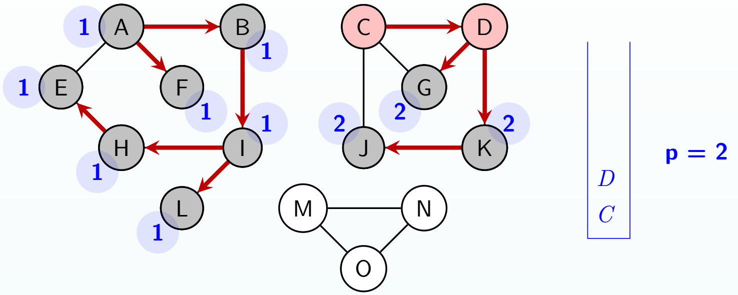
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p + 1$

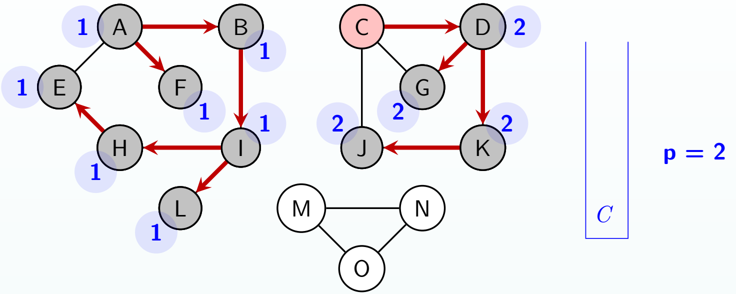
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p + 1$

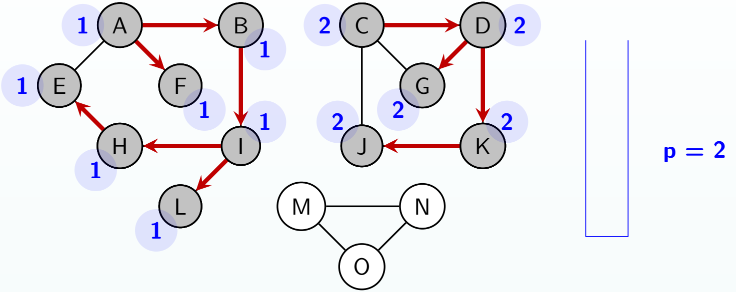
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

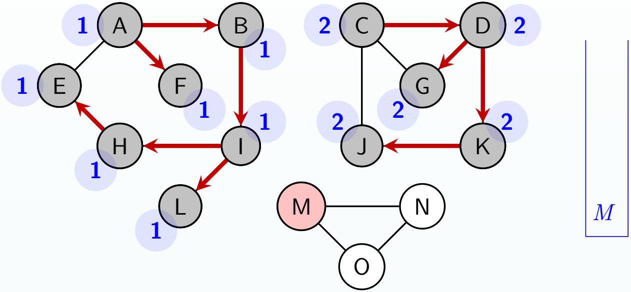
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



p = 3

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

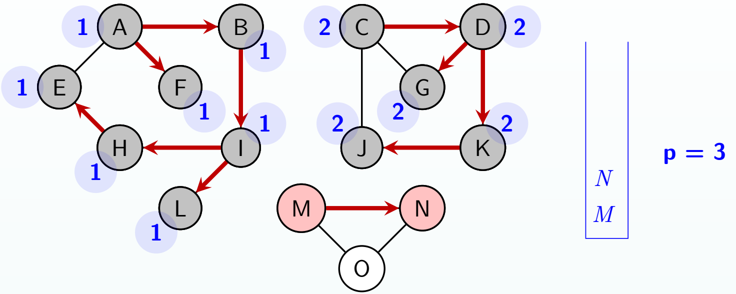
  while $Stack$ is not empty, do

    if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

      open and push y;

    else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

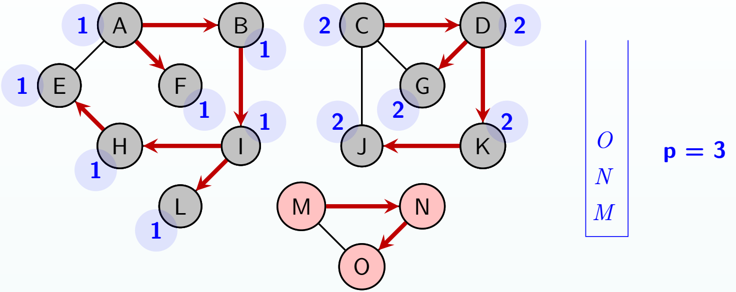
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

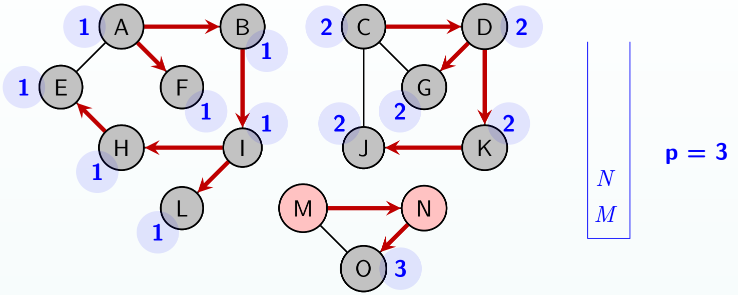
6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity




```
all vertices are unmarked; p → 0; Stack → ∅
while it exists an unmarked vertex s, open and push s; p → p + 1
  while Stack is not empty, do
    if it exists an unmarked vertex y adjacent to the top x of Stack, do
       open and push y;
    else close and pop x; c(x) = p
```

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p + 1$

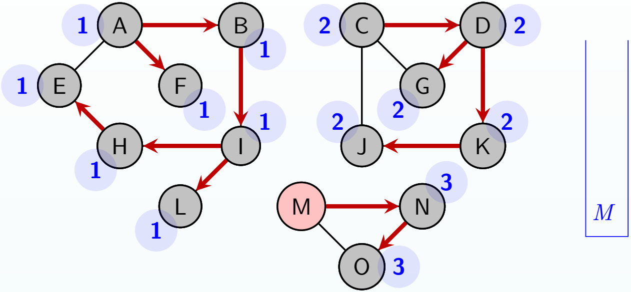
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



p = 3

all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

while it exists an unmarked vertex s, open and push s; $p \to p+1$

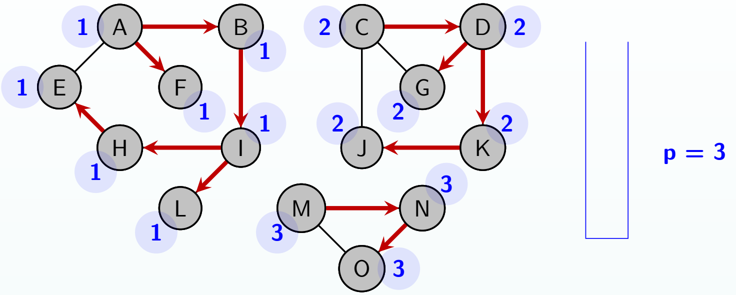
while $Stack$ is not empty, do

if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

open and push y;

else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



1: all vertices are unmarked; $p \to 0$; $Stack \to \emptyset$

2: while it exists an unmarked vertex s, open and push s; $p \to p+1$

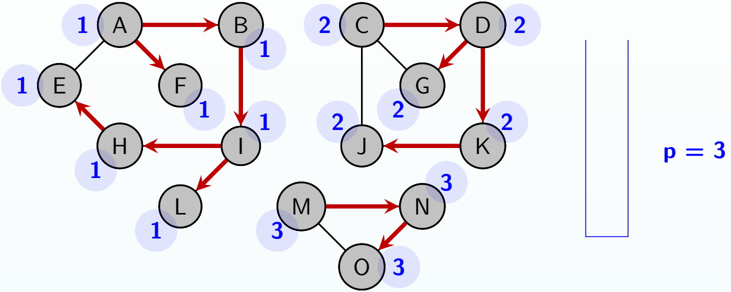
3: while $Stack$ is not empty, do

4: if it exists an unmarked vertex y adjacent to the top x of $Stack$, do

5: open and push y;

6: else close and pop x; $c(x) = p$

Depth-first search: computing connectivity



- The connectivity is determined, each vertex is labeled to a connected component
- Notable properties of Depth-first search :
  - Stack empty at each new connected component
  - Any non traveled egde indicates a chain

Breadth-first search: shortest path

- Goal: Determining the shortest path length[4] from a vertex s to others vertices of the graph
- Method: Breadth-first search

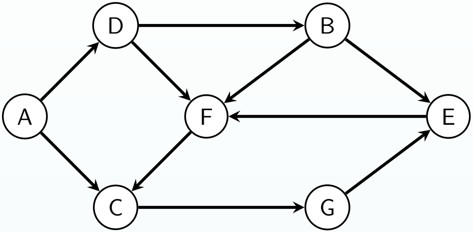
Breadth-first search

All unmarked successors of curent vertex are open successively. The next visited vertex at every step, among open vertices, is the one that was first opened.

- Using a queue (abstract data type)
  - First In, First Out (FIFO)

[4] Not to be confused with the value of a path of valued arcs

Breadth-first search: shortest path



all vertices are unmarked; $Queue \to \emptyset$

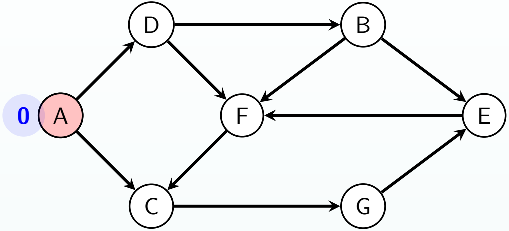
open and enqueue s; $d(s) \to 0$

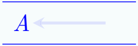
  while $Queue$ is no empty, do

    open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

    close and dequeue x;

Breadth-first search: shortest path





all vertices are unmarked; $Queue \to \emptyset$

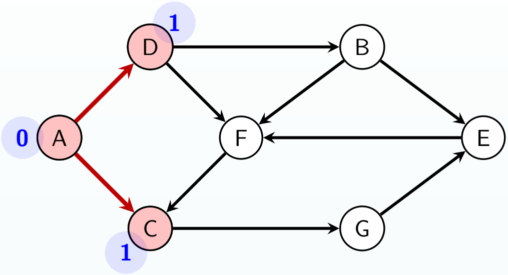
open and enqueue s; $d(s) \to 0$

  while $Queue$ is no empty, do

    open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

    close and dequeue x;

Breadth-first search: shortest path



$A\ D\ C$

all vertices are unmarked; $Queue \to \emptyset$

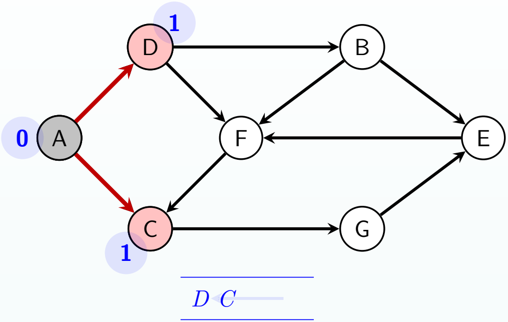
open and enqueue s; $d(s) \to 0$

while $Queue$ is no empty, do

open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

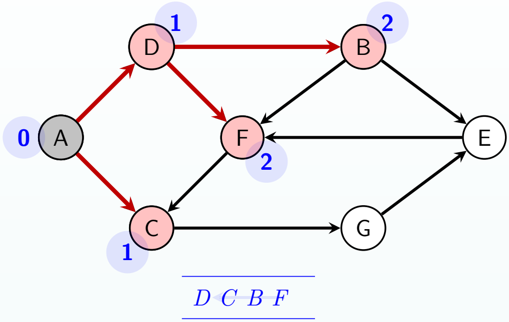
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

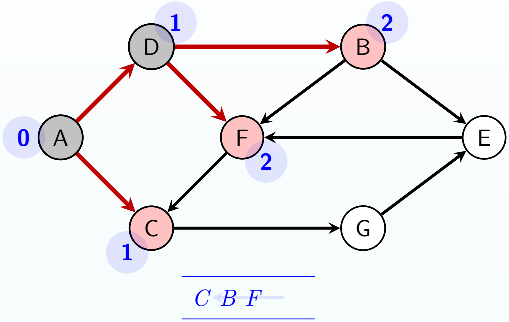
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

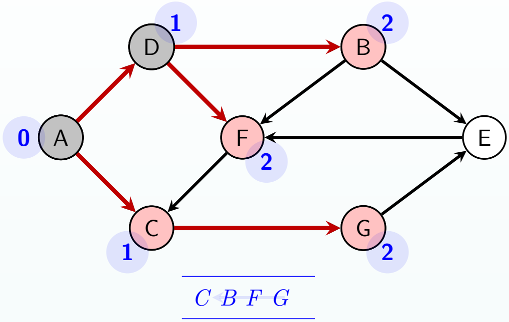
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

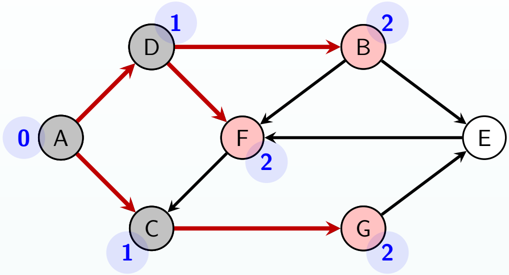
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



$B \; F \; G$

1: all vertices are unmarked; $Queue \rightarrow \emptyset$

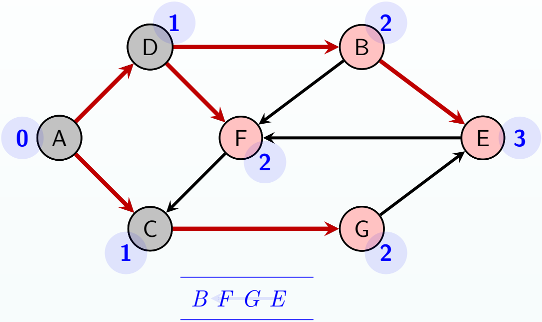
2: open and enqueue s; $d(s) \rightarrow 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

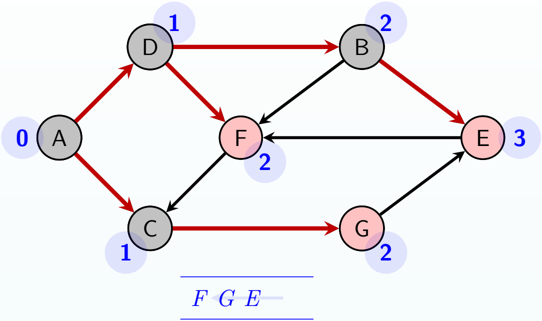
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \rightarrow \emptyset$

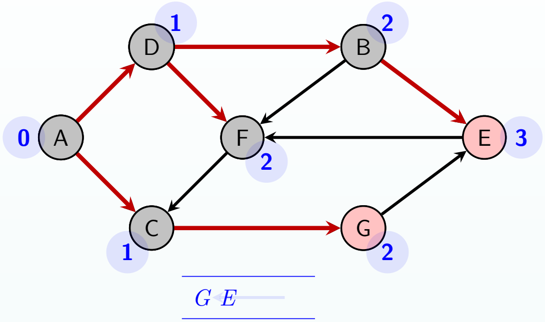
2: open and enqueue s; $d(s) \rightarrow 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

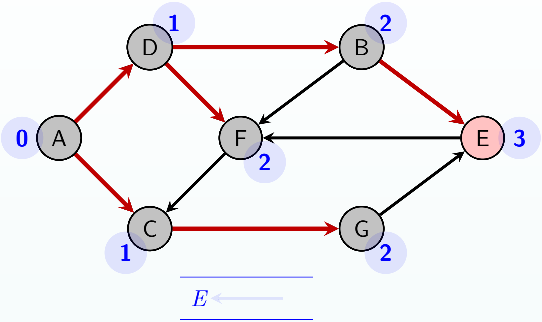
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \rightarrow \emptyset$

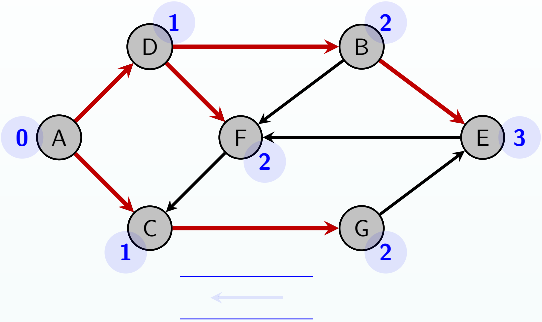
2: open and enqueue s; $d(s) \rightarrow 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



1: all vertices are unmarked; $Queue \to \emptyset$

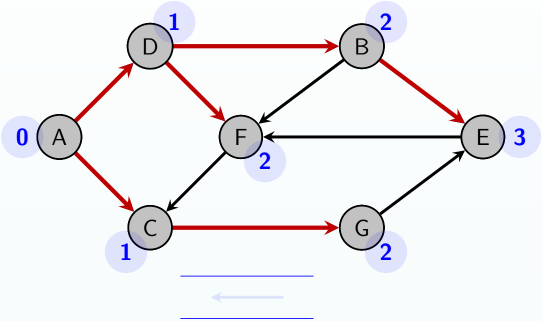
2: open and enqueue s; $d(s) \to 0$

3: while $Queue$ is no empty, do

4: open and enqueue all unmarked vertices y successors of the queue head x; $d(y) = d(x) + 1$;

5: close and dequeue x;

Breadth-first search: shortest path



- Each vertex is associated with its distance from the starting point
- The starting point determining
  - One connected component covered
  - Unmarked vertex at the end of the algorithm : inaccessible vertex from s

Outline

2 Graph algorithms

- Definition of an algorithm
- Computational Complexity
- First algorithms: graph traversal
- Dynamic programming

Origines

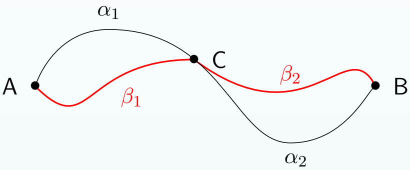
- Attributed to Richard Bellman ($\sim$ 1950)
  - Based on work of Pierre de Fermat in optics
- Implicit enumeration: avoiding some calculations by *lowering the complexity* of the problem
- Is based on the principle of optimality
- Solving problems of optimal paths (min or max)

Principle of optimality

Definition

Any portion (sub-path) of an optimal path is, itself, optimal.

- Easy demonstration by *reductio ad impossibilem* (Proof by contradiction)



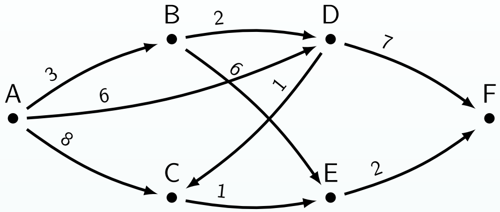
- Find a recursive formulation of the problem

Application areas

- Combinatorial distribution problems
  - Ski rental problem (limited elements)
  - Knapsack problem / Change-making problem (unlimited elements)
- Algorithmic of text
  - Calculation of the longest common subsequence
  - similarity computation (Levenshtein distance)
  - Local sequence alignment (Bioinformatics: Smith–Waterman algorithm)

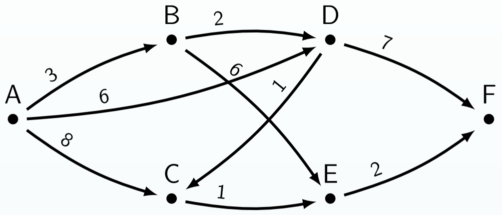
All dynamic programming algorithm can be reduced to the search for the shortest path in a graph (Martelli, 1976)

Raw example (Bellman algorithm)



- Once determined (A, B), shortest path between A and B, and (A, B, D, C), shortest path between A and C
- The shortest path between A and E is limited to the comparaison of (A, B, E) and (A, B, D, C, E)
  - (A, C, E) and (A, D, C, E) are excluded before calculation by the principle of optimality

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} \mathcal{D}_A(D) + v(D, F) \\ \mathcal{D}_A(E) + v(E, F) \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} \mathcal{D}_A(C) + v(C, E) \\ \mathcal{D}_A(B) + v(B, E) \end{pmatrix}$$

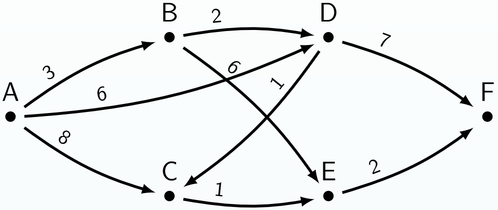
$$\mathcal{D}_A(D) = \min \begin{pmatrix} \mathcal{D}_A(B) + v(B, D) \\ \mathcal{D}_A(A) + v(A, D) \end{pmatrix}$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} \mathcal{D}_A(D) + v(D, C) \\ \mathcal{D}_A(A) + v(A, C) \end{pmatrix}$$

$$\mathcal{D}_A(B) = \mathcal{D}_A(A) + v(A, B)$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} \mathcal{D}_A(D) + 7 \\ \mathcal{D}_A(E) + 2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} \mathcal{D}_A(C) + 1 \\ \mathcal{D}_A(B) + 6 \end{pmatrix}$$

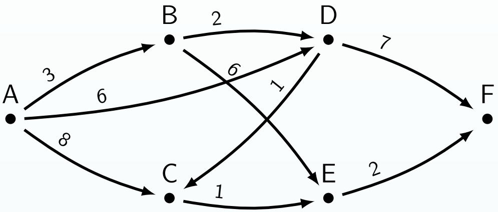
$$\mathcal{D}_A(D) = \min \begin{pmatrix} \mathcal{D}_A(B) + 2 \\ \mathcal{D}_A(A) + 6 \end{pmatrix}$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} \mathcal{D}_A(D) + 1 \\ \mathcal{D}_A(A) + 8 \end{pmatrix}$$

$$\mathcal{D}_A(B) = \mathcal{D}_A(A) + 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} \mathcal{D}_A(D) + 7 \\ \mathcal{D}_A(E) + 2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} \mathcal{D}_A(C) + 1 \\ \mathcal{D}_A(B) + 6 \end{pmatrix}$$

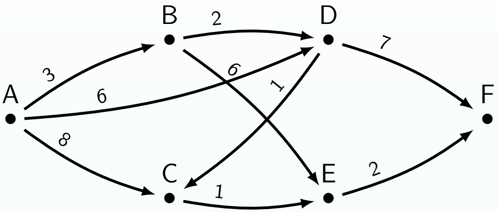
$$\mathcal{D}_A(D) = \min \begin{pmatrix} \mathcal{D}_A(B) + 2 \\ 0 + 6 \end{pmatrix}$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} \mathcal{D}_A(D) + 1 \\ 0 + 8 \end{pmatrix}$$

$$\mathcal{D}_A(B) = 0 + 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} \mathcal{D}_A(D) + 7 \\ \mathcal{D}_A(E) + 2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} \mathcal{D}_A(C) + 1 \\ 3 + 6 \end{pmatrix}$$

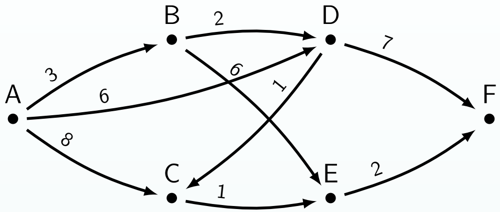
$$\mathcal{D}_A(D) = \min \begin{pmatrix} 3 + 2 \\ 0 + 6 \end{pmatrix}$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} \mathcal{D}_A(D) + 1 \\ 0 + 8 \end{pmatrix}$$

$$\mathcal{D}_A(B) = 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} 5 + 7 \\ \mathcal{D}_A(E) + 2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} \mathcal{D}_A(C) + 1 \\ 3 + 6 \end{pmatrix}$$

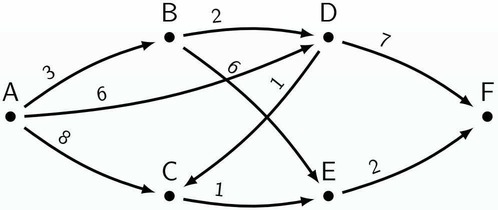
$$\mathcal{D}_A(D) = \min \begin{pmatrix} 3 + 2 \\ 0 + 6 \end{pmatrix} = 5$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} 5 + 1 \\ 0 + 8 \end{pmatrix}$$

$$\mathcal{D}_A(B) = 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} 5+7 \\ \mathcal{D}_A(E)+2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} 6+1 \\ 3+6 \end{pmatrix}$$

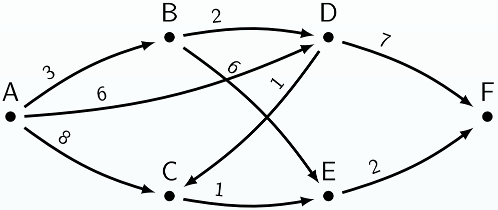
$$\mathcal{D}_A(D) = \min \begin{pmatrix} 3+2 \\ 0+6 \end{pmatrix} = 5$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} 5+1 \\ 0+8 \end{pmatrix} = 6$$

$$\mathcal{D}_A(B) = 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} 5+7 \\ 7+2 \end{pmatrix}$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} 6+1 \\ 3+6 \end{pmatrix} = 7$$

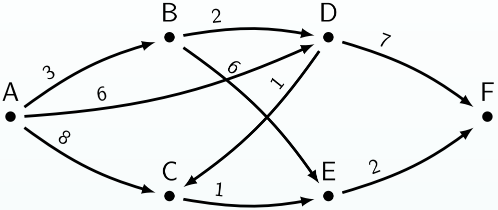
$$\mathcal{D}_A(D) = \min \begin{pmatrix} 3+2 \\ 0+6 \end{pmatrix} = 5$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} 5+1 \\ 0+8 \end{pmatrix} = 6$$

$$\mathcal{D}_A(B) = 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min \begin{pmatrix} 5+7 \\ 7+2 \end{pmatrix} = 9$$

$$\mathcal{D}_A(E) = \min \begin{pmatrix} 6+1 \\ 3+6 \end{pmatrix} = 7$$

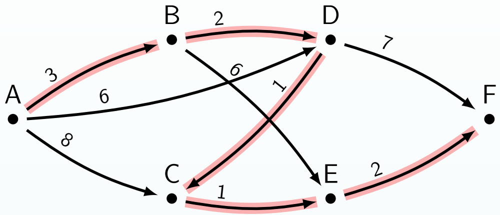
$$\mathcal{D}_A(D) = \min \begin{pmatrix} 3+2 \\ 0+6 \end{pmatrix} = 5$$

$$\mathcal{D}_A(C) = \min \begin{pmatrix} 5+1 \\ 0+8 \end{pmatrix} = 6$$

$$\mathcal{D}_A(B) = 3$$

$$\mathcal{D}_A(A) = 0$$

Raw example (Bellman algorithm)



$$\mathcal{D}_A(F) = \min\begin{pmatrix}\mathcal{D}_A(D) + v(D,F)\\ \mathcal{D}_A(E) + v(E,F)\end{pmatrix} = 9$$

$$\mathcal{D}_A(E) = \min\begin{pmatrix}\mathcal{D}_A(C) + v(C,E)\\ \mathcal{D}_A(B) + v(B,E)\end{pmatrix} = 7$$

$$\mathcal{D}_A(D) = \min\begin{pmatrix}\mathcal{D}_A(B) + v(B,D)\\ \mathcal{D}_A(A) + v(A,D)\end{pmatrix} = 5$$

$$\mathcal{D}_A(C) = \min\begin{pmatrix}\mathcal{D}_A(D) + v(D,C)\\ \mathcal{D}_A(A) + v(A,C)\end{pmatrix} = 6$$

$$\mathcal{D}_A(B) = v(A,B)$$

$$\mathcal{D}_A(A) = 0$$

Bellman algorithm

Initially $\mathcal{F}(x_0) = 0$; $\forall y \neq x_0, \mathcal{F}(y) = +\infty$
for k from 1 to $N-1$
  for all vertex y
    $\mathcal{F}'(y) = \min(\mathcal{F}(z) + l(z,y); z \in \mathcal{P}(y))$
  $\mathcal{F} = \mathcal{F}'$

- Shortest path to all vertices
- Non-absorbent cycles are possible
- Reversible
- Complexity $O(n^2)$

Outline

3. Flow problems
   - Definition
   - Ford-Fulkerson algorithm

Definition

- Modelize a transport network and the constraints associated with it
  - Road network
  - Power network
  - Water distribution network
  - ...
- Optimization of the flow
- Search key elements

Transport network

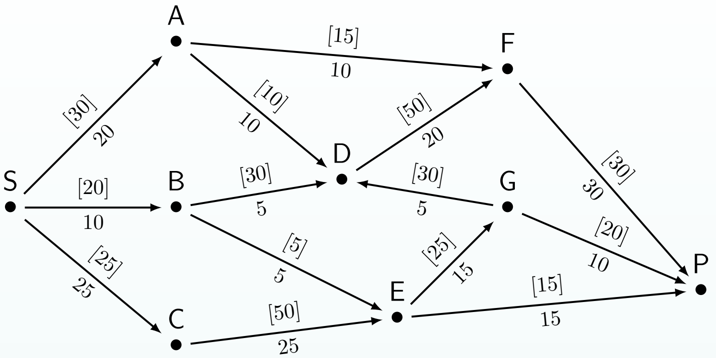
Definition

Is called a *transport network* an valued oriented graph G of n vertices without loop, with two vertices x_1 et x_n such that for all vertex x_k of G, there is at least one path from x_1 to x_n through x_k.

- x_1 is called *source* and x_n *sink* of the graph
- The value of the arc u, writed $c(u)$ is the capacity of the arc
- Is associated with each arc a flow φ, such that $0 \leq \varphi(u) \leq c(u)$
- The flows must respect the Kirchhoff's law
  - Accordingly:

$$\sum_{(x_1,x_i)\in U} \varphi(x_1, x_i) = \sum_{(x_j,x_n)\in U} \varphi(x_j, x_n)$$

Flow problems



$$\sum_{(x_1,x_i)\in U} c(x_1,x_i) = 75 \quad ; \quad \sum_{(x_j,x_n)\in U} c(x_j,x_n) = 65 \quad ; \quad \varphi = 55$$

- Is this flow optimal? Or can it be improved, and how?

Outline

3 Flow problems

- Definition
- Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

- Dectection of augmenting paths
- Flow optimization in the augmenting paths

Definition

An *augmenting path* is an elementary path from x_1 to x_n wherein no *direct arc* is saturated, and all the *indirect arcs* have a strictly positive flow.

- (x_a, x_b) is a direct arc if (x_a, x_b) is an arc of the graph
- (x_a, x_b) is a indirect arc if (x_b, x_a) is an arc of the graph
- An arc is saturated if $c(u) = \varphi(u)$

Ford-Fulkerson algorithm

- **Dectection of an augmenting path**

1: source labeled "$+$", others vertex unlabeled

2: while an arc (x, y) satisfies one of the two conditions

3: x is labeled, y is unlabeled and (x, y) is unsaturated: label « $+x$ » the vertex y

4: x is unlabeled, y is labeled and $\varphi(x, y)$ is not null: label « -y » the vertex x

5: if the sink is labeled, the current flow can be improved by the augmenting paths found by going back to the source with the labels

6: if the sink remains unlabeled, the current flow is optimal

Ford-Fulkerson algorithm

- **Increase the flow in the augmenting path**

1: $\delta^+ = \min\{c(u) - \varphi(u)\}$ where u is a direct arc of C

2: $\delta^- = \min\{\varphi(u)\}$ where u is an indirect arc of C

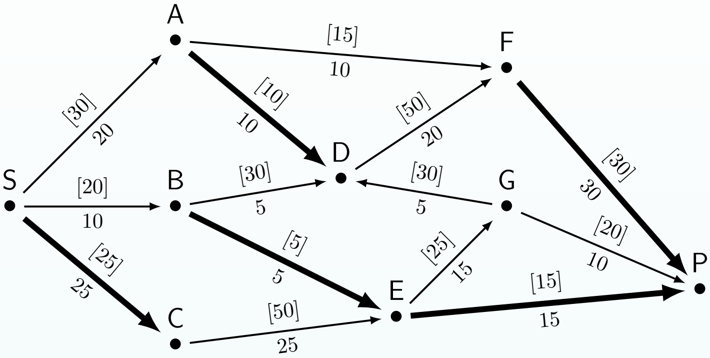
3: $\delta = \min\{\delta^+; \delta^-\}$

4: for all direct arc u of C: $\varphi_u \leftarrow \varphi_u + \delta$

5: for all indirect arc u of C: $\varphi_u \leftarrow \varphi_u - \delta$

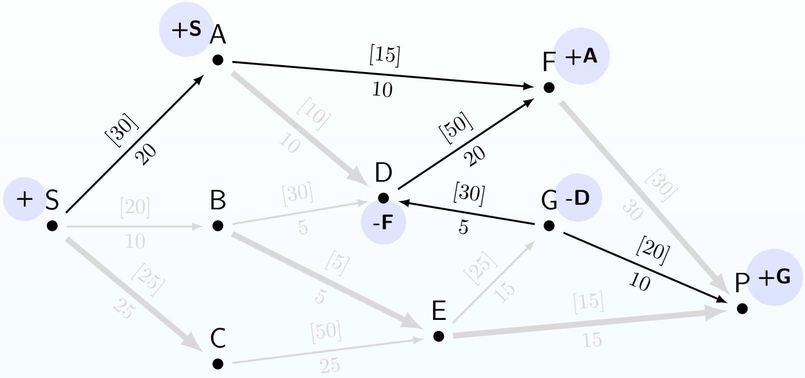
- $\delta > 0$, from the definition of the augmenting path

Ford-Fulkerson algorithm: Example



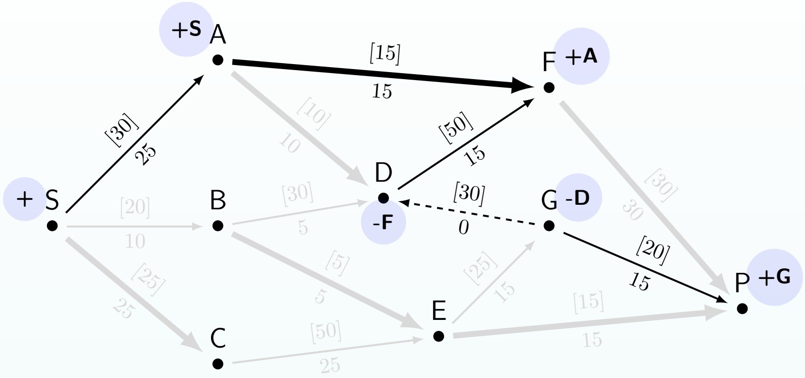
- Dectection of an augmenting path

Ford-Fulkerson algorithm: Example



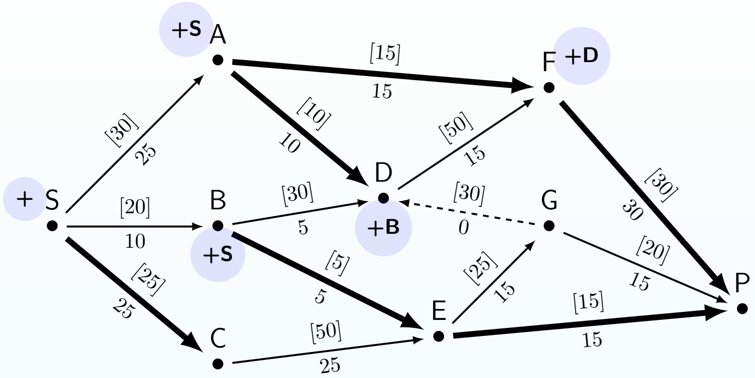
- Augmenting path (S, A, F, D, G, P)

Ford-Fulkerson algorithm: Example



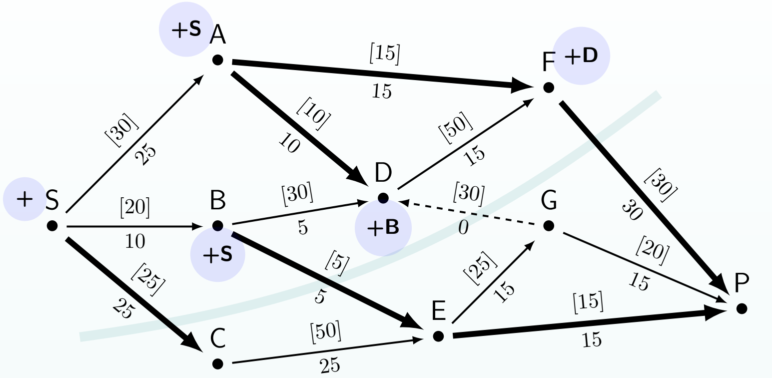
- Augmenting path (S, A, F, D, G, P): $\delta = 5$

Ford-Fulkerson algorithm: Example



- No augmenting path remaining, the flow is maximal: $\delta = 60$

Ford-Fulkerson algorithm: Example



- No augmenting path remaining, the flow is maximal: $\delta = 60$
- Detection of the minimal cut

Max-flot/min-cut theorem

Definition

The maximum flow from source to sink is equal to the *minimum capacity* that has to be removed from the graph to nullify the flow able to go from the source to the sink.

- The minimum cut separates the graph into tw sets of vertices, including:
  1. The removal of intermediate arcs nullifies the flow
  2. The sum of capacities of these arcs is minimal
- This sum of capacities is equal to the maximum flot of the graph

Flow problems

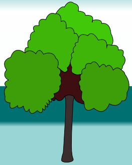
- Ford-Fulkerson algorithm
  - Getting a flow with a maximum value
  - Minimum cut
  - Proof of optimality
- Many other models
  - Maximum flow at minimal *cost*
  - Cuts
  - Assignment problems
  - ...

Outline

4 The trees

- Definitions
- Application examples

Definitions



Definition

A tree is a connected graph without cycle.

Definition

An arborescence or **rooted tree** is in an oriented graph, un tree with a vertice *root* for which there is a single path to any other vertex.

- Necessarily, the root can not admit predecessor and is unique (no cycle in the graph)

Attention !

In **Computer Science**, the term *tree* is commonly used to describe an *arborescence*!

Outline

4 The trees

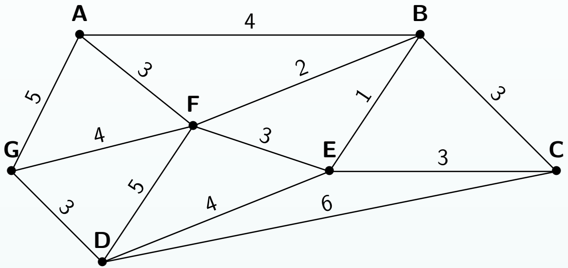
- Definitions
- Application examples

Application examples

- Optimal trees
  - Network routing optimization
- Transport programs
  - Organization between multiple sources and destinations
  - Unlimited flow but limited cost
- Tree search
  - "Branch and bound" algorithms
  - Optimal search of scheduling solution

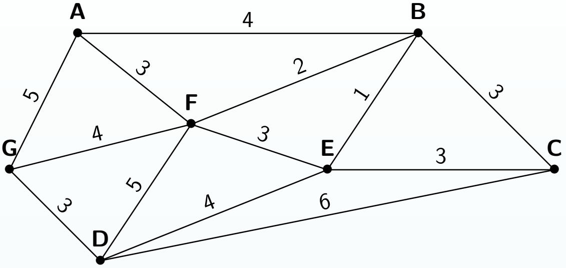
Optimal trees

- What tree from the graph minimizes the value of the edges?



- Using a greedy algorithm

Optimal trees



1: while possible, do

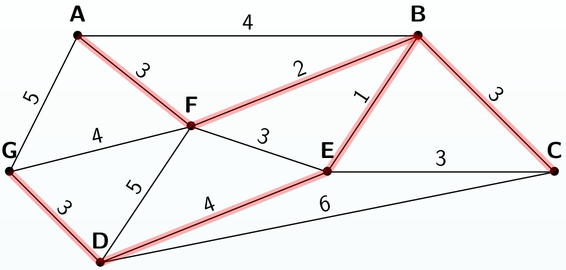
2: select the arc with the smallest possible valuation, whose the two ends are not all selected

3: select the ends of the arc

4: si multiple disconnected components

5: start again at the hypergraph

Optimal trees



1: while possible, do

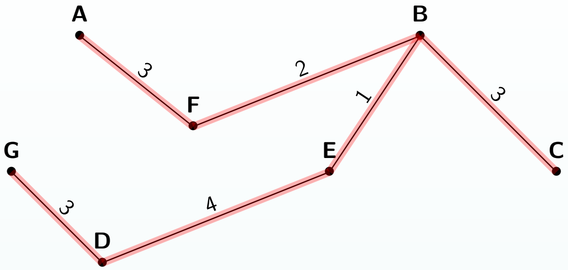
2: select the arc with the smallest possible valuation, whose the two ends are not all selected

3: select the ends of the arc

4: si multiple disconnected components

5: start again at the hypergraph

Optimal trees



1: while possible, do
2: select the arc with the smallest possible valuation, whose the two ends are not all selected
3: select the ends of the arc
4: si multiple disconnected components
5: start again at the hypergraph

Graph Theory in Operational Research (a brief overview)
Damien Leprovost
March 6, 2015

permalink: http://www.damien-leprovost.fr/enseignements/graphs.2015.pdf