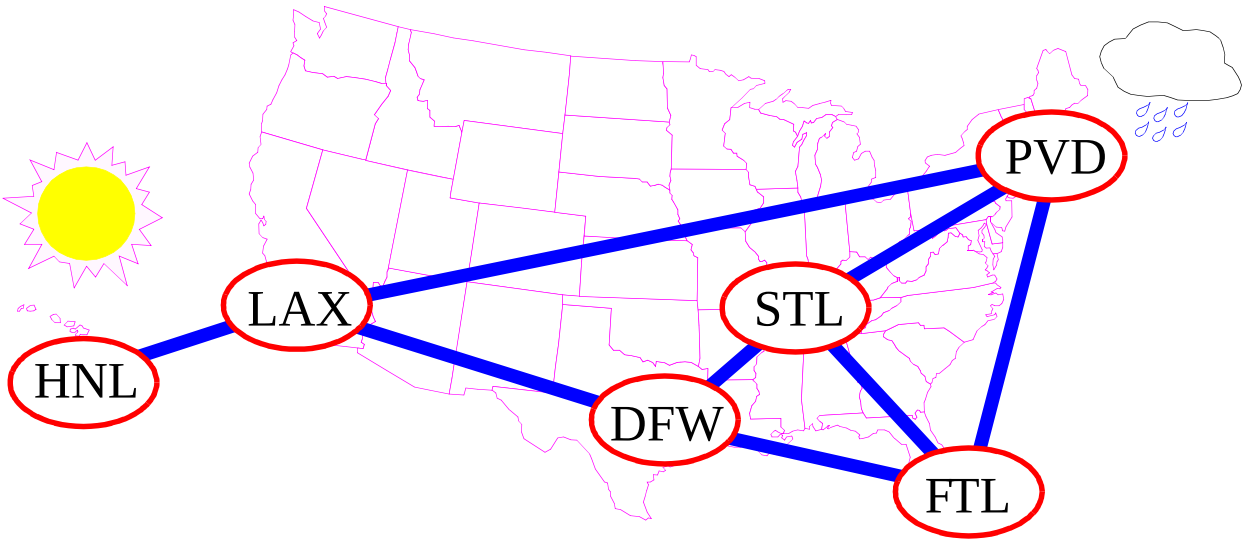
GRAPHES

- Définitions
- Le TAD Graphe
- Structures de données pour graphes



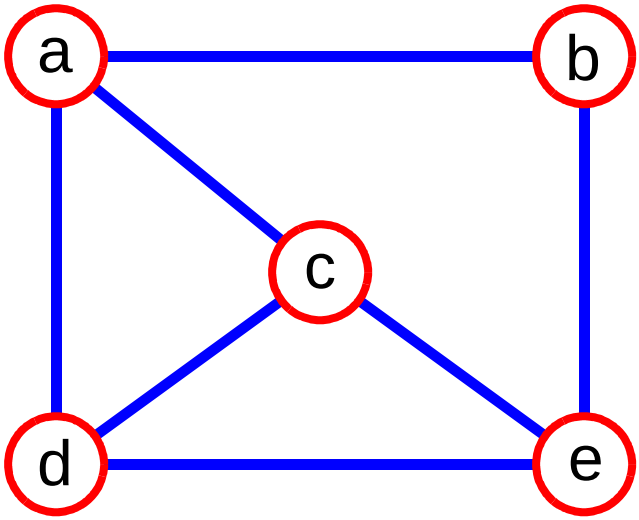
Qu'est-ce qu'un graphe?

- Un **graphe G** = (**V**,**E**) est composé de:

  V: ensemble de **sommets** (*vertices*)

  E: ensemble d'**arcs** (*edges*) reliant les sommets de **V**

- Un **arc e** = (u,v) est une paire de sommets
- Exemple:

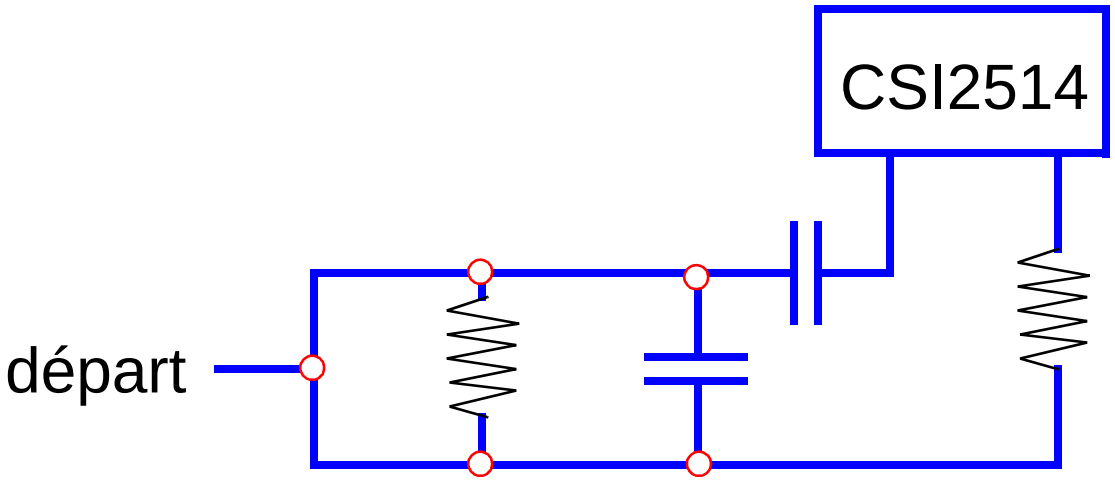


V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

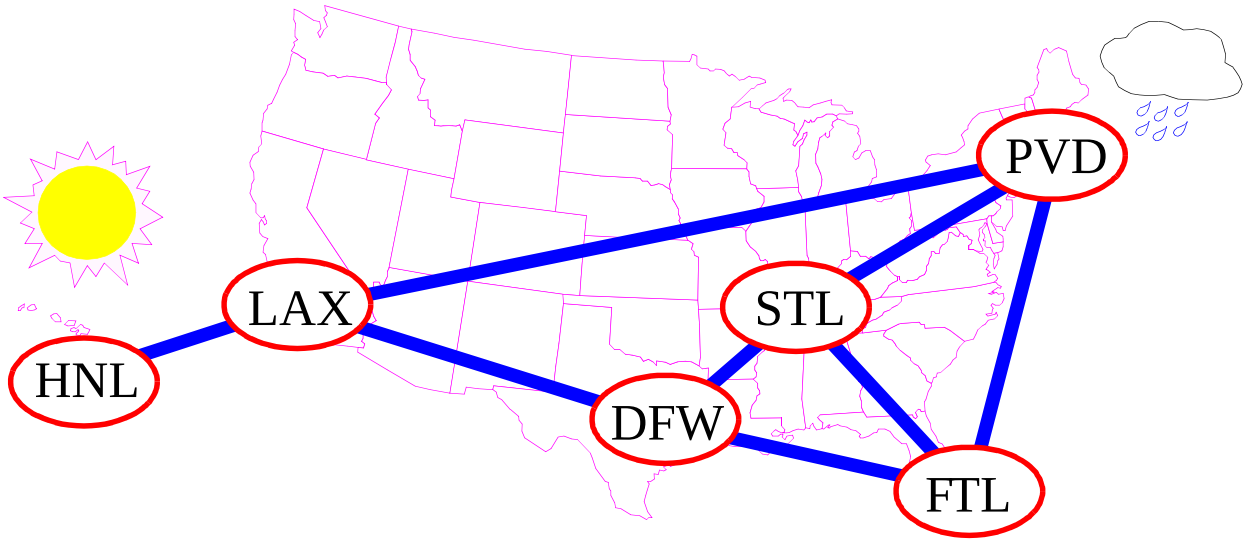
Applications

- circuits électroniques



trouvez le chemin à la moindre résistance menant à CSI2514

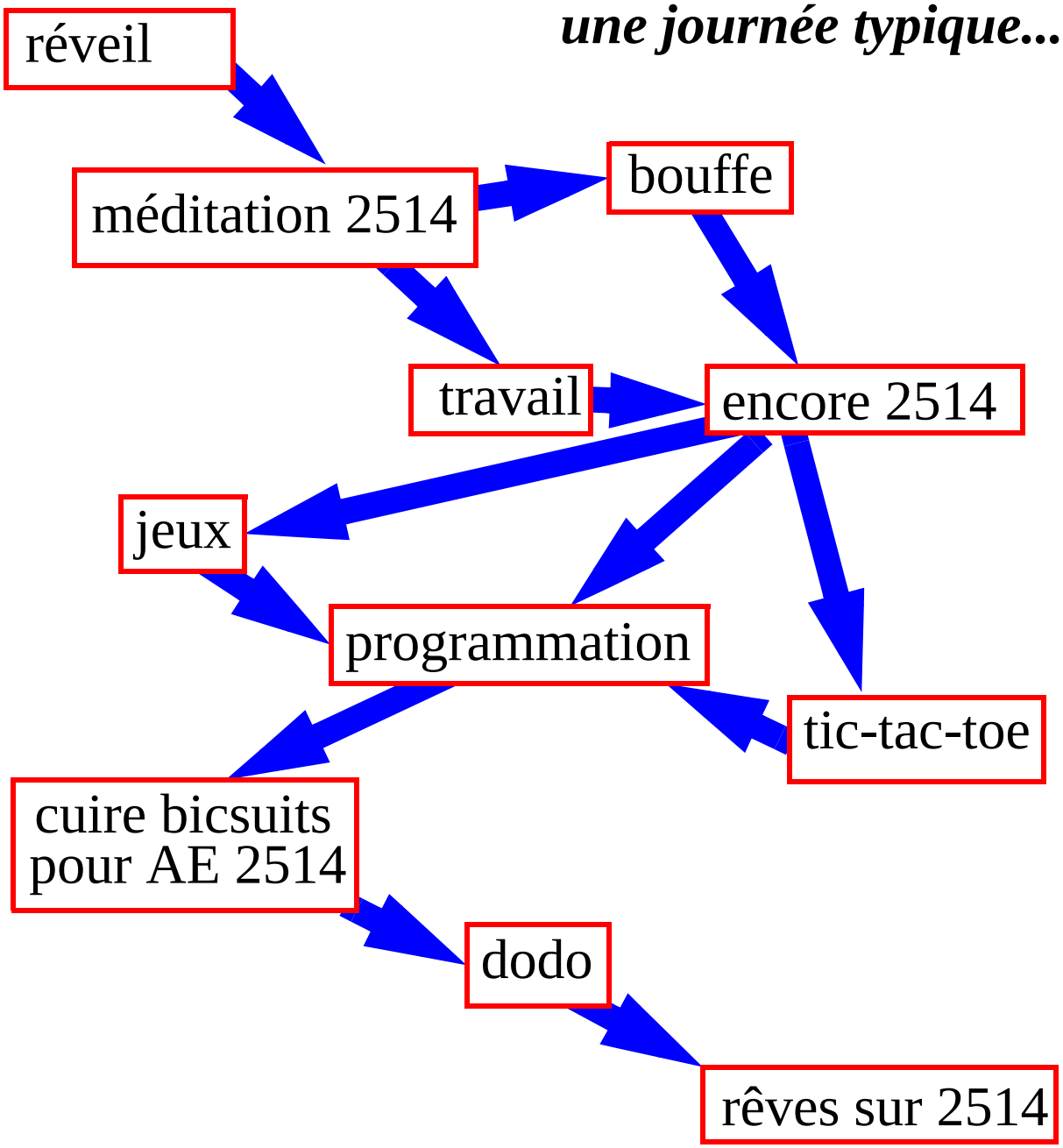
- réseaux (routiers, aériens, de communication)



De meilleurs exemples...

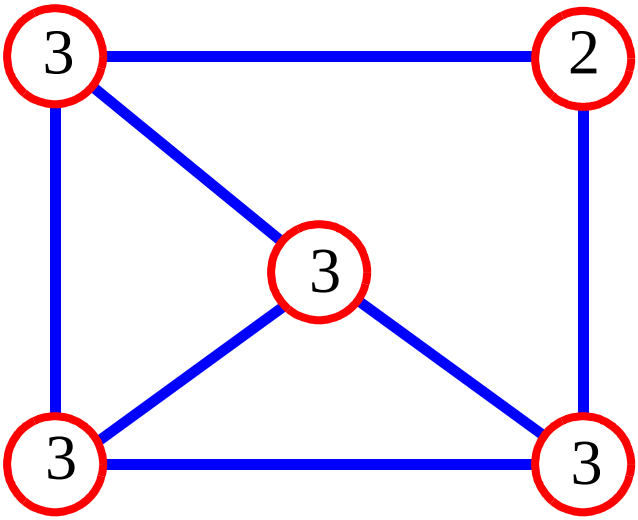
- conception d'horaires (planification de projet)

une journée typique...



Terminologie des graphes

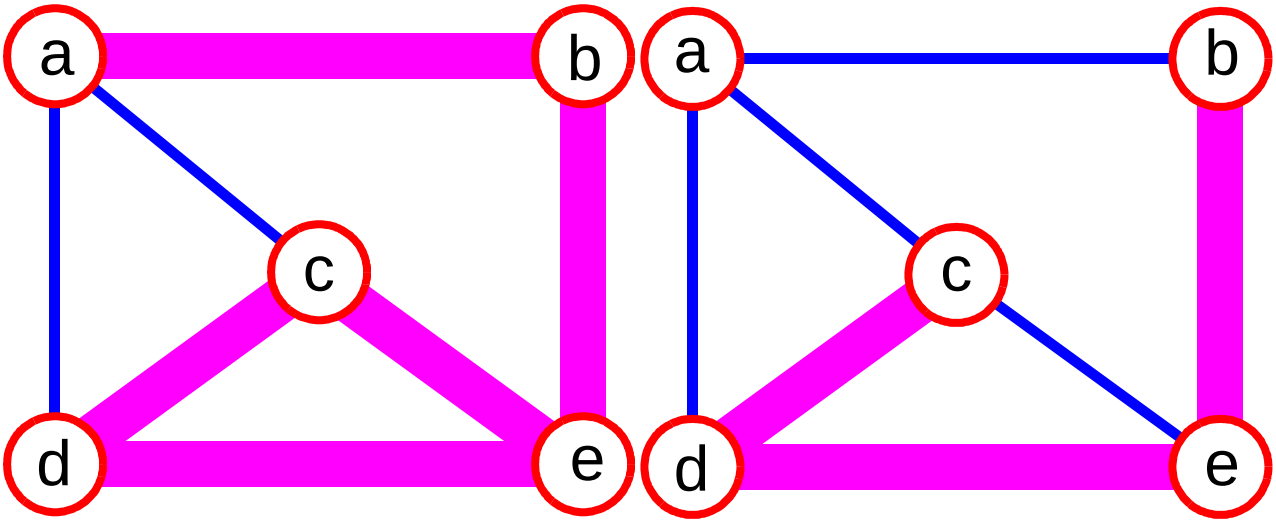
- sommets adjacents: reliés par un arc
- degré (d'un sommet): # de sommets adjacents



$$\sum_{v \in V} \deg(v) = 2(\#\text{ arcs})$$

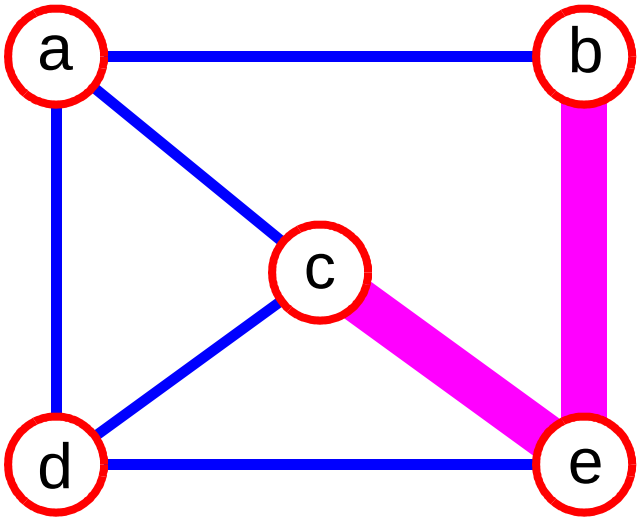
- Comme des sommets adjacents comptent tous deux l'arc les reliant, celui-ci sera compté deux fois.

chemin: séquence de sommets $v_1, v_2, \ldots v_k$ où les sommets consécutifs v_i et v_{i+1} sont adjacents.



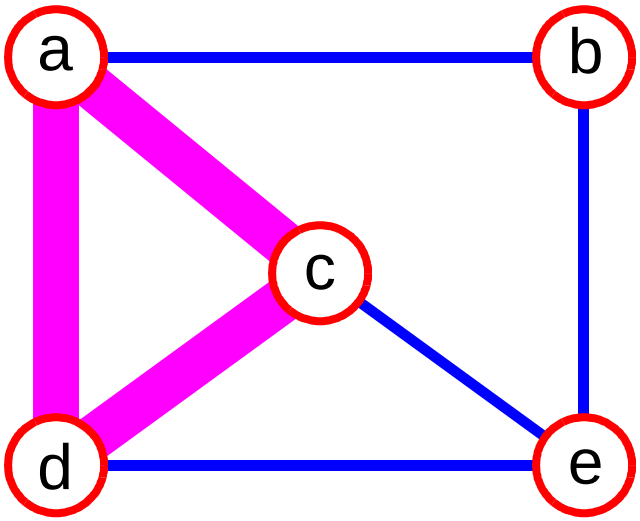
Encore de la terminologie...

- chemin simple: sans aucun sommet répété

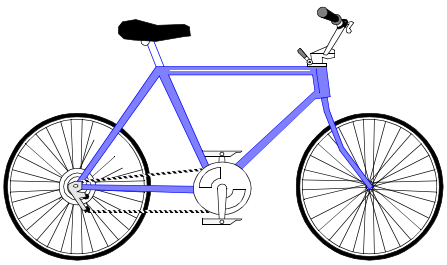


b e c

- cycle: chemin simple, sauf que le dernier sommet est le même que le tout premier

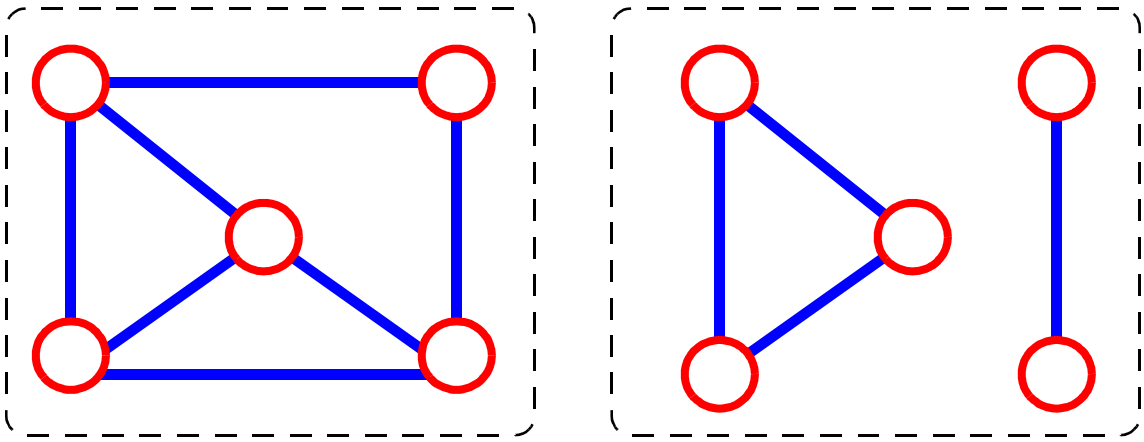


a c d a



Et encore de la terminologie...

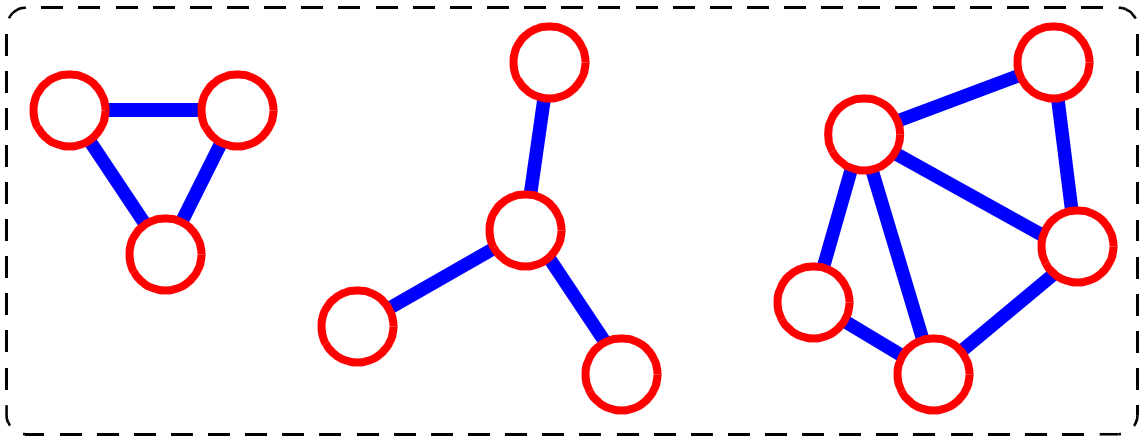
- graphe connexe: toutes les paires de sommets sont reliées par un chemin



connexe non-connexe

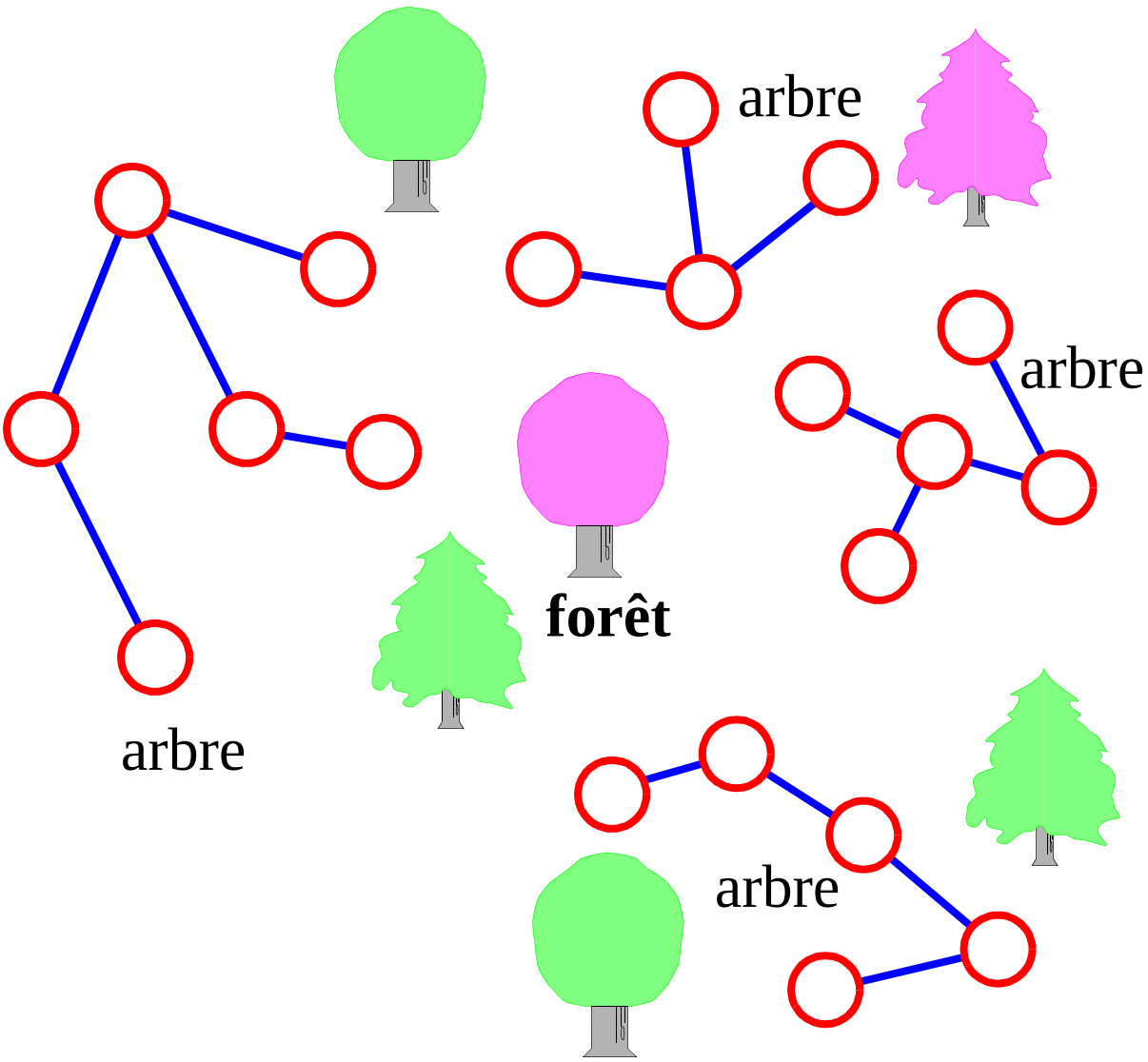
- sous-graphe: sous-ensemble de sommets et d'arcs formant un graphe

- composante connexe: sous-graphe connexe maximal. Par exemple, le graphe ci-dessous a 3 composantes connexes.



¡Caramba! Encore de la terminologie!

- arbre (libre) - graphe connexe sans cycle
- forêt - ensemble d'arbres



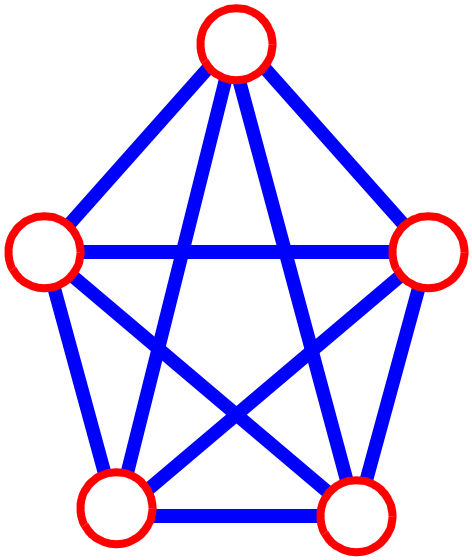
Connectivité

Soit **n** = #sommets
m = #arcs

- graphe complet - toutes les paires de sommets sont adjacentes

$$\mathbf{m} = (1/2)\sum_{v \in \mathbf{V}} \deg(\mathbf{v}) = (1/2)\sum_{v \in V} (\mathbf{n} - 1) = \mathbf{n}(\mathbf{n}-1)/2$$

- Chacun des **n** sommets est attaché à **n** - 1 arcs, cependant, nous aurons compté chaque arc deux fois!!! Ainsi, intuitivement, **m** = **n**(**n**-1)/2.



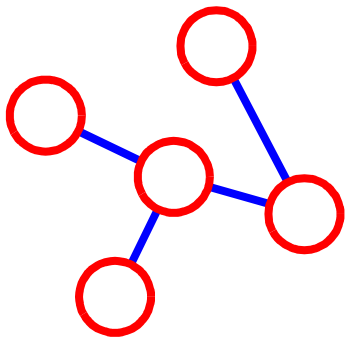
$\mathbf{n} = 5$
$\mathbf{m} = (5 * 4)/2 = 10$

- Donc, si un graphe n'est ***pas*** complet,
  m < **n**(**n**-1)/2

Plus de connectivité

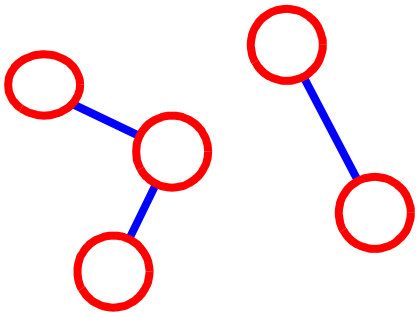
n = #sommets
m = #arcs

- Pour un arbre **m** = **n** - 1



n = 5
m = 4

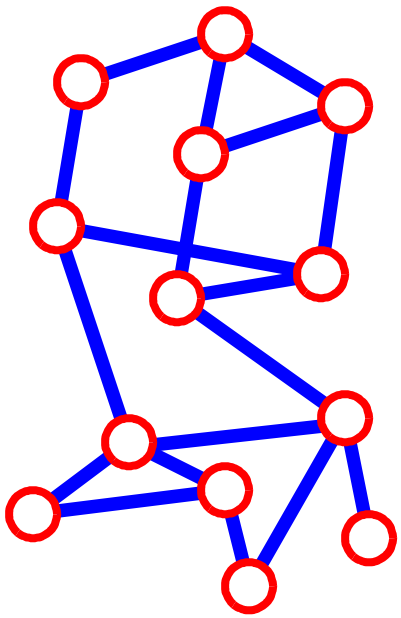
- Si **m** < **n** - 1, alors le graphe **G** n'est pas connexe



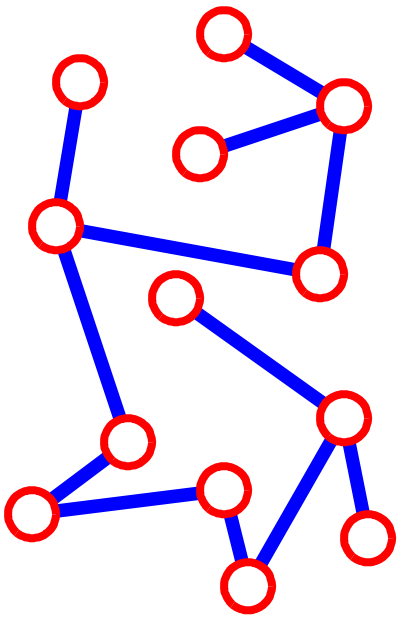
n = 5
m = 3

Arbre recouvrant (*Spanning Tree*)

- Un **arbre recouvrant** (*spanning tree*) de **G** est un sous-graphe qui:
  - est un arbre
  - contient tous les sommets de **G**



G



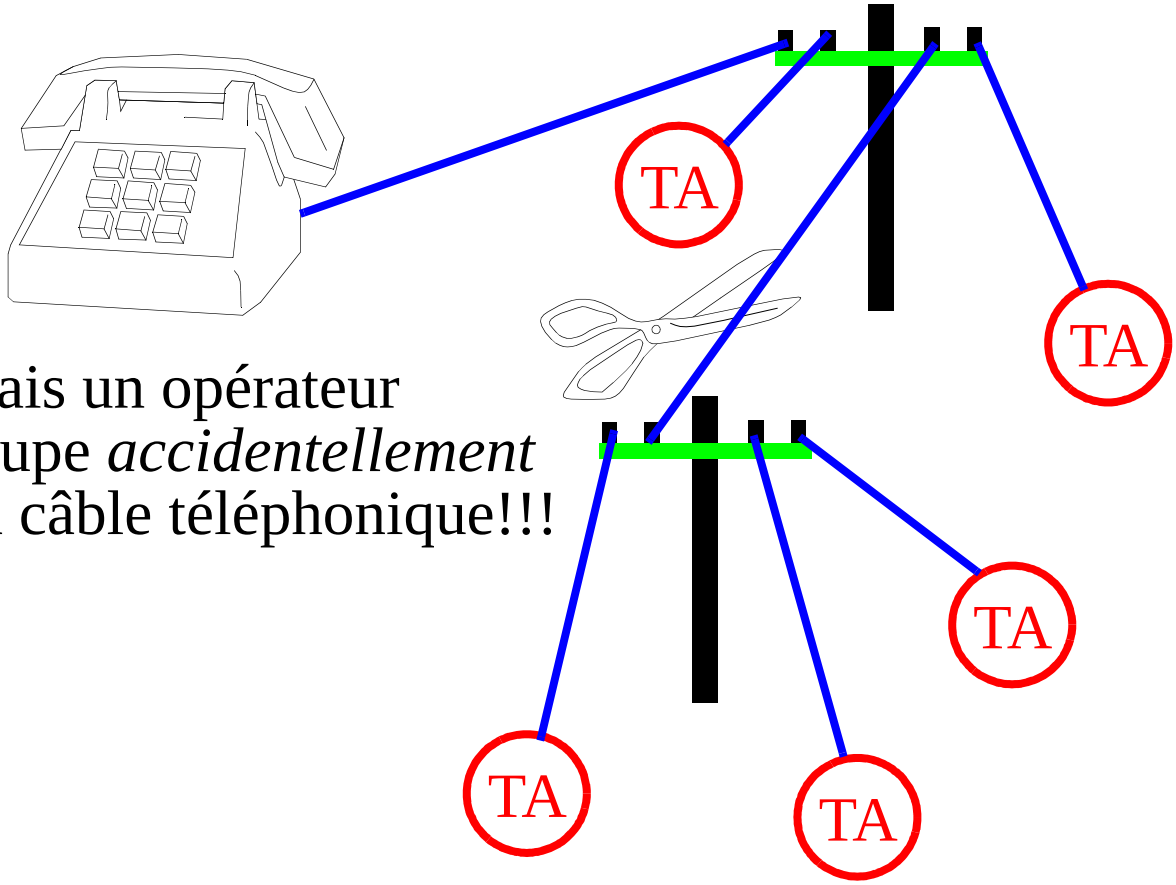
arbre recouvrant de **G**

- Une faute affectant n'importe quel arc rend le système non-connexe (l'arbre recouvrant est la configuration la moins tolérante aux fautes)

Bell Canada contre SM&T

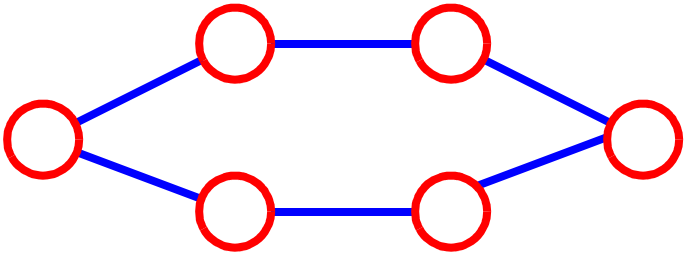
(Stan Matwin & Telephone)

- Stan désire appeler ses AE afin de suggérer une extension pour le prochain devoir...

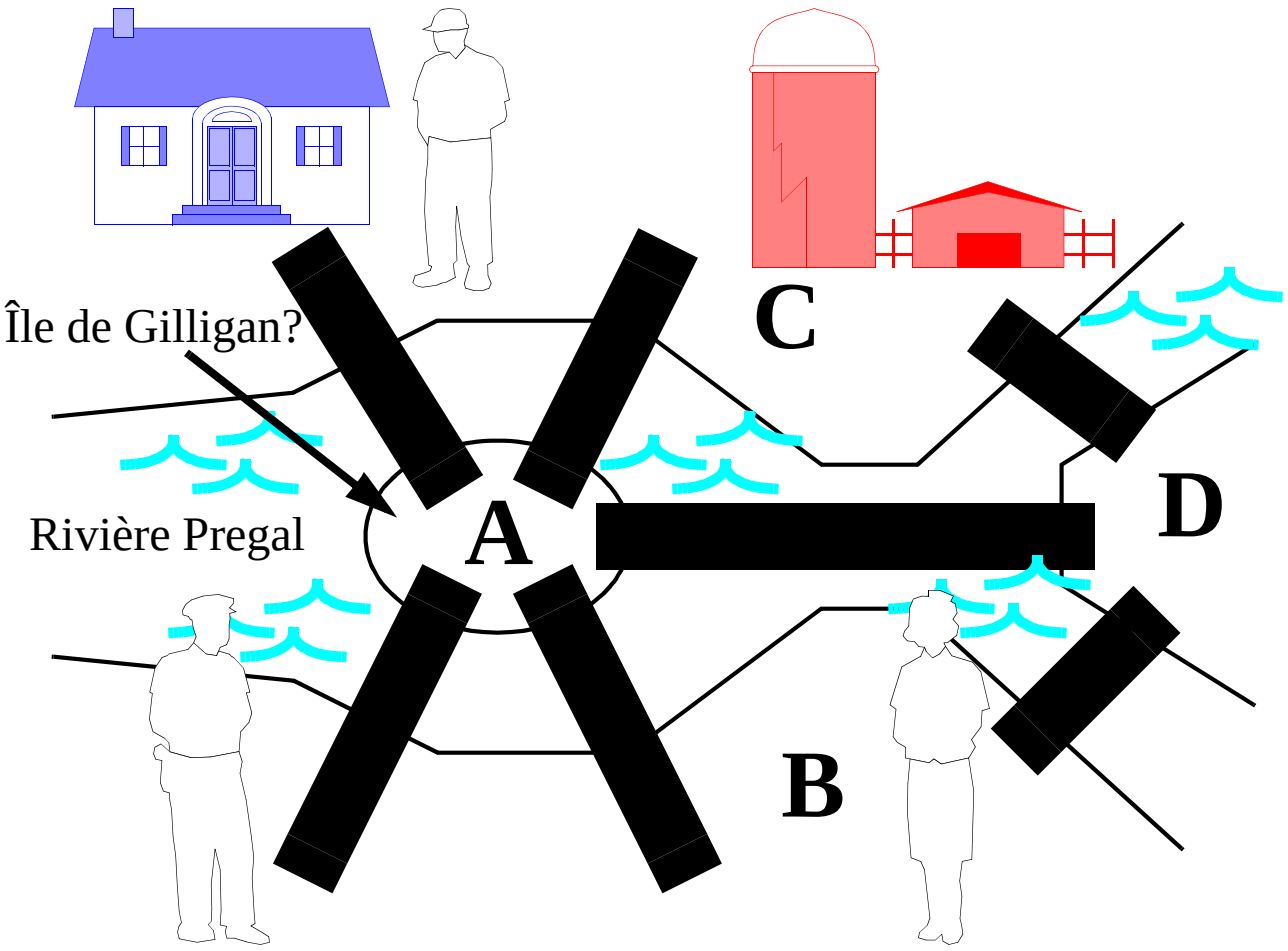


Mais un opérateur coupe *accidentellement* un câble téléphonique!!!

- Une faute va déconnecter une partie du graphe!
- Un cycle serait plus tolérant aux fautes et n'exige que **n** arcs.



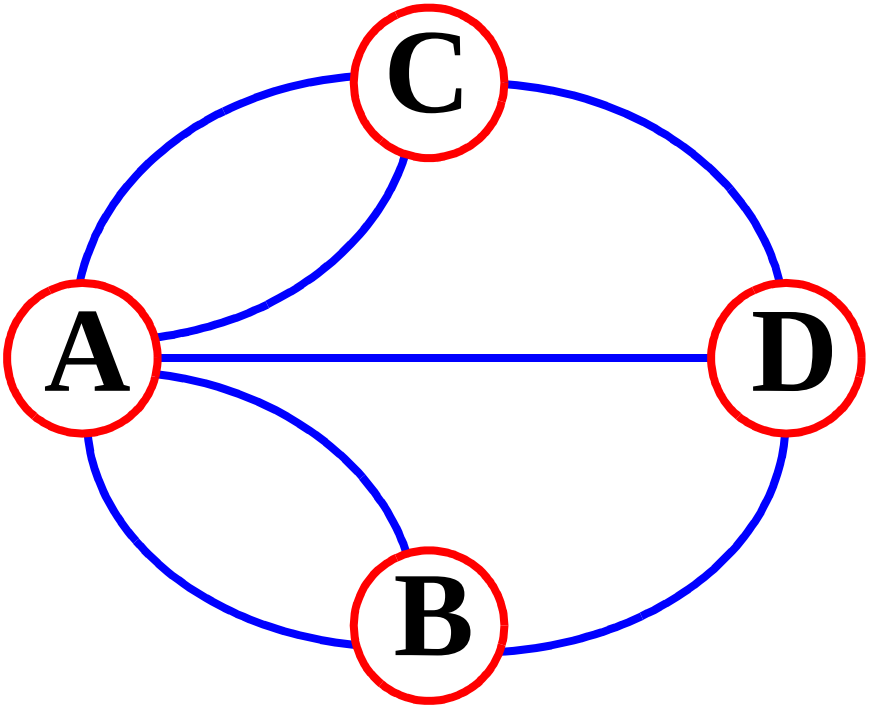
Euler et les ponts de Koenigsberg



Peut-on traverser chaque pont exactement une fois et retourner au point de départ?

- Mettez-vous à la place d'un conducteur de UPS ou de Fedex qui ne voudrait pas revenir sur son chemin.
- En 1736, Euler a prouvé que ce n'est pas possible.

Modèle de graphe (avec arcs parallèles)



- Tour d'Euler: chemin qui traverse chaque arc une fois exactement et qui retourne au premier sommet
- Théorème d'Euler: un graphe a un tour d'Euler si et seulement si tous les sommets ont un degré pair.
- Trouvez-vous intéressantes de telles idées?
- Aimeriez-vous passer une session entière à faire de telles preuves...? Il existe un tel cours!

Le TAD Graphe (*Graph*)

- Le **TAD Graphe** est un contenant positionnel dont les positions sont les sommets et les arcs du graphe.

  - size() Retourne le nombre de sommets plus le nombre d'arcs contenus dans *G*.
  - isEmpty()
  - elements()
  - positions()
  - swap()
  - replaceElement()

Notation: Graphe *G*; Sommets *v*, *w*; Arc *e*; Objet *o*

  - numVertices()
    Retourne le nombre de sommets de *G*.
  - numEdges()
    Retourne le nombre d'arcs de *G*.
  - vertices() Retourne une énumération des sommets de *G*.
  - edges() Retourne une énumération des arcs de *G*.

Le TAD Graphe (suite)

- directedEdges()
  Retourne une énumération de tous les arcs orientés de *G*.
- undirectedEdges()
  Retourne une énumération de tous les arcs non-orientés de *G*.
- incidentEdges(*v*)
  Retourne une énumération de tous les arcs attachés à *v*.
- inIncidentEdges(*v*)
  Retourne une énumération de tous les arcs entrant dans *v*.
- outIncidentEdges(*v*)
  Retourne une énumération de tous les arcs sortant de *v*.
- opposite(*v*, *e*)
  Retourne le sommet de l'arc *e* qui n'est pas *v*.
- degree(*v*)
  Retourne le degré de *v*.
- inDegree(*v*)
  Retourne le degré d'entrée de *v*.
- outDegree(*v*)
  Retourne le degré de sortie de *v*.

Encore des méthodes...

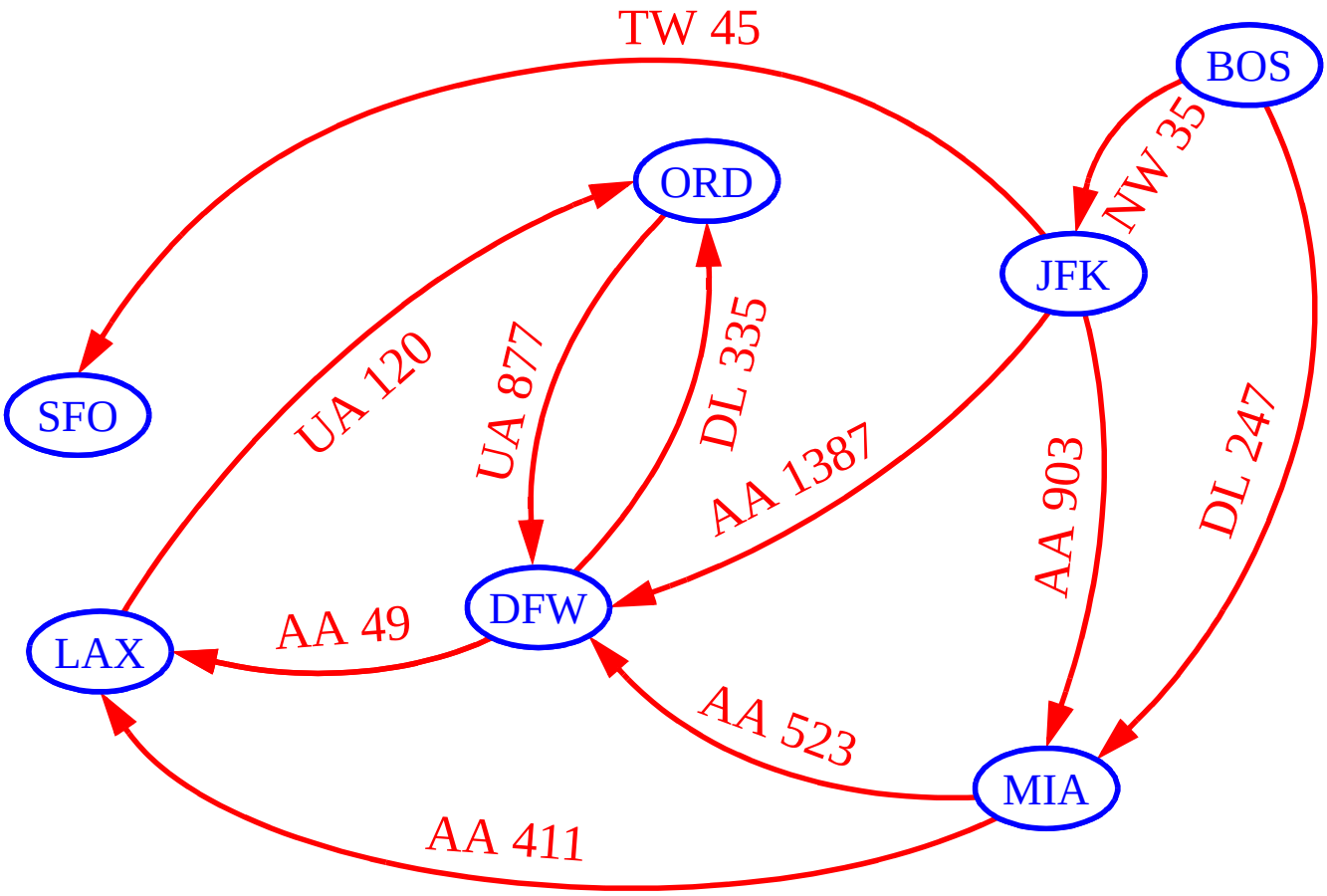
- adjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v*.
- inAdjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v* qui ont un arc entrant dans *v*.
- outAdjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v* qui ont un arc sortant de *v*.
- areAdjacent(*v*,*w*)
  Indique si les sommets *v* et *w* sont adjacents.
- endVertices(*e*)
  Retourne un vecteur de taille 2 emmagasinant les sommets aux bouts de *e*.
- origin(*e*)
  Retourne le sommet duquel *e* sort.
- destination(*e*)
  Retourne le sommet auquel *e* entre.
- isDirected(*e*)
  Retourne vrai ssi *e* est orienté.

Méthodes de mise à jour

- makeUndirected(*e*)
  Déclare *e* comme arc non-orienté.
- reverseDirection(*e*)
  Inverse les sommets d'origine et de destination de *e*.
- setDirectionFrom(*e*, *v*)
  Ajuste la direction de *e* de façon à sortir de *v*, l'un de ses sommets.
- setDirectionTo(*e*, *v*)
  Ajuste la direction de *e* de façon à entrer dans *v*, l'un de ses sommets.
- insertEdge(*v*, *w*, *o*)
  Insère et retourne un arc non-orienté entre *v* et *w*, tout en emmagasinant *o* à cette position.
- insertDirectedEdge(*v*, *w*, *o*)
  Insère et retourne un arc orienté entre *v* et *w*, tout en emmagasinant *o* à cette position.
- insertVertex(*o*)
  Insère et retourne un nouveau sommet (isolé) emmagasinant *o* à cette position
- removeEdge(*e*)
  Retire l'arc *e*.

Structures de données pour graphes

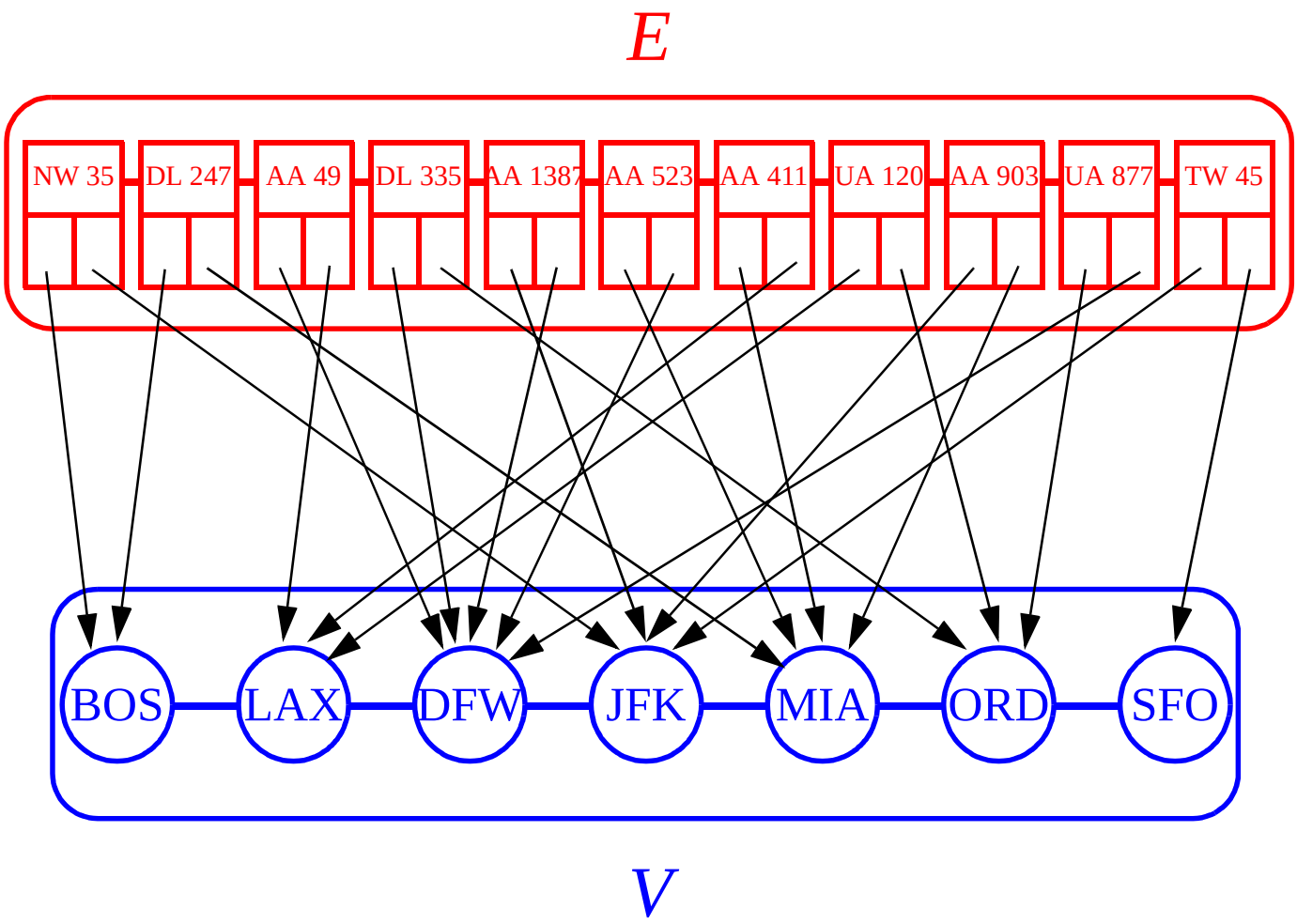
- Un graphe! Comment le représenter?
- Pour débuter, nous conservons les sommets et les arcs dans deux contenants, et chaque objet arc a des références vers les sommets qu'il relie.



- Des structures additionnelles peuvent être utilisées afin de mieux exécuter les méthodes de Graphe.

Liste d'arcs (*Edge List*)

- La structure **liste d'arcs** emmagasine tout simplement les sommets et les arcs dans des séquences non-triées.
- Facile à réaliser.
- Trouver l'arc attaché à un sommet donné n'est pas efficace parce que cela exige le parcours de la séquence d'arcs toute entière.

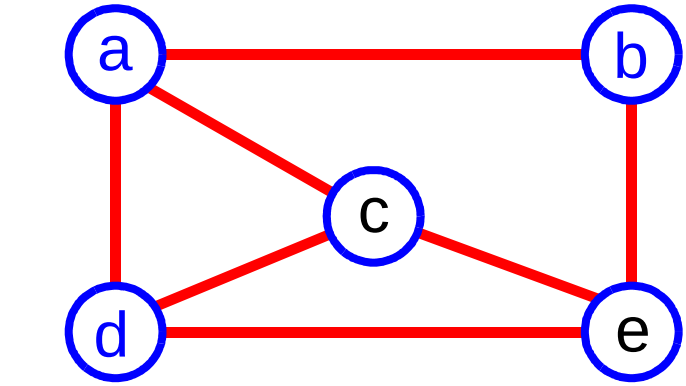


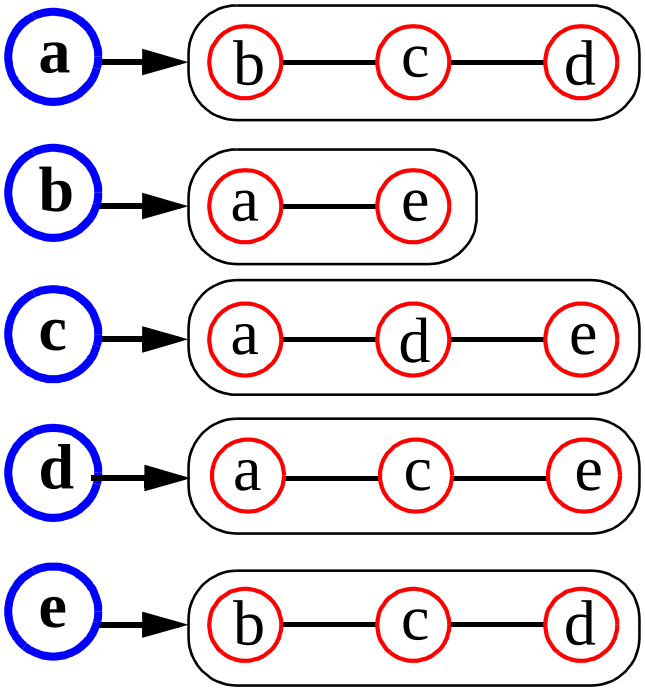
Performance de la structure Liste d'arcs

Opération	Temps
size, isEmpty, replaceElement, swap	*O(1)*
numVertices, numEdges	*O(1)*
vertices	*O(n)*
edges, directedEdges, undirectedEdges	*O(m)*
elements, positions	*O(n+m)*
endVertices, opposite, origin, destination, isDirected	*O(1)*
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices, areAdjacent, degree, inDegree, outDegree	*O(m)*
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	*O(1)*
removeVertex	*O(m)*

Liste d'adjacence (traditionnelle)

- Liste d'adjacence d'un sommet v:
  séquence de sommets adjacents à v
- Représentez le graphe par les listes d'adjacence de tous les sommets

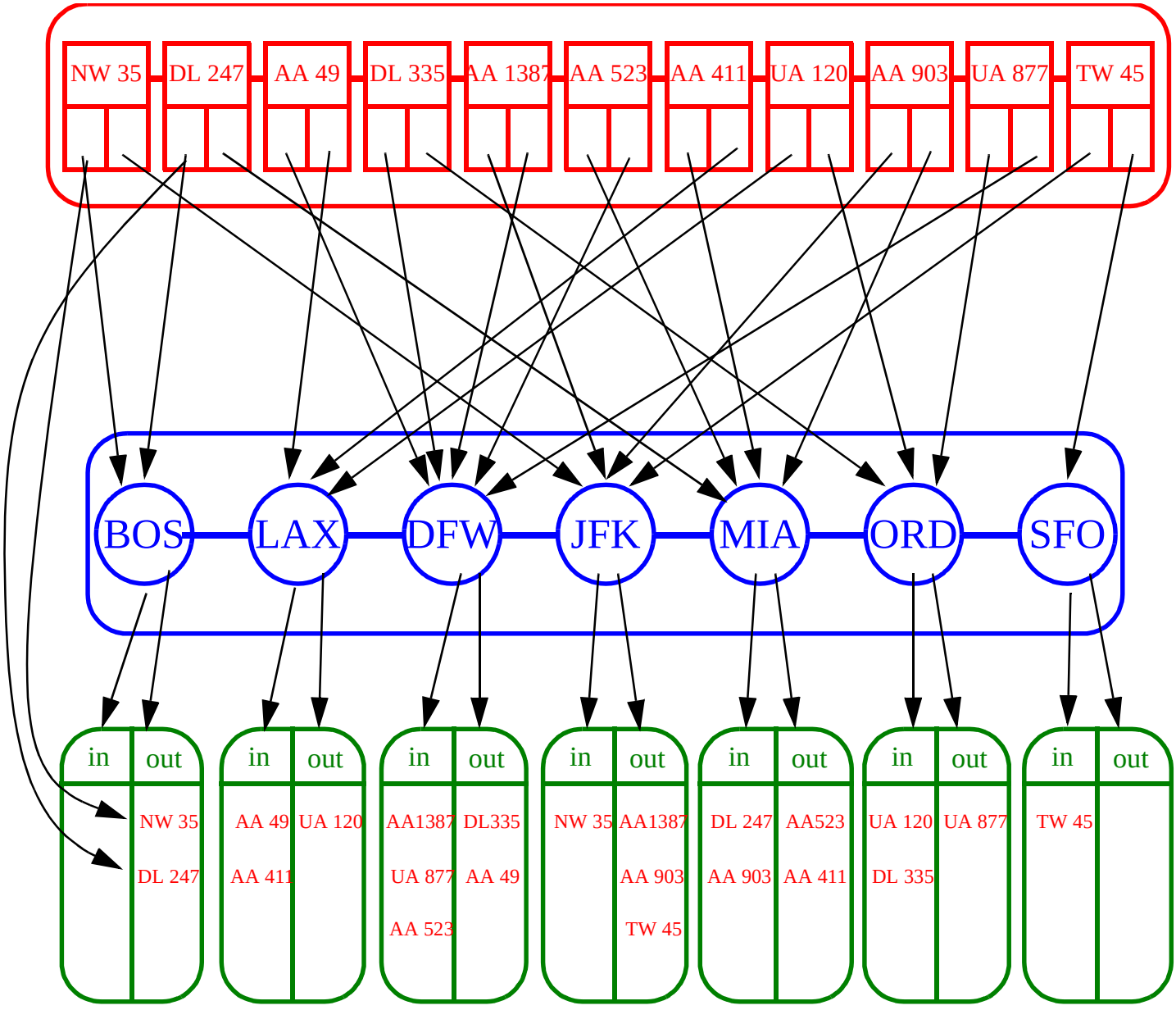




- Espace requis = $\Theta(\mathbf{N} + \sum \deg(\mathbf{v})) = \Theta(\mathbf{N} + \mathbf{M})$

Liste d'adjacence (moderne)

- La structure **liste d'adjacence** améliore la structure de liste d'arcs en ajoutant des contenants de liaison (*incidence containers*) à chaque sommet.

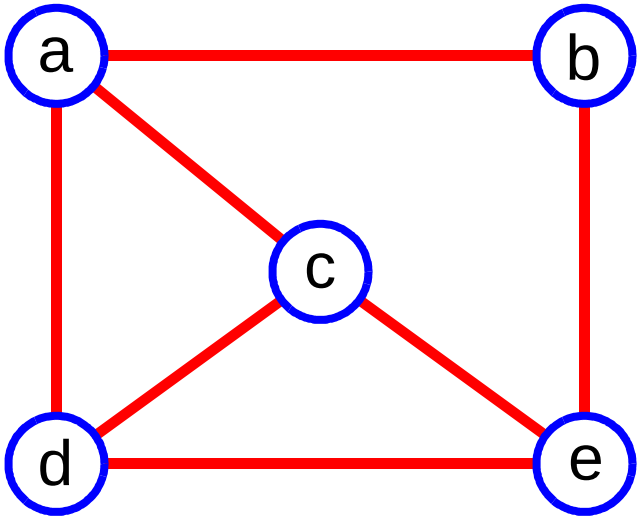


- L'espace requis est $O(n + m)$.

Performance de la structure Liste d'adjacence

Opération	Temps
size, isEmpty, replaceElement, swap	$O(1)$
numVertices, numEdges	$O(1)$
vertices	$O(n)$
edges, directedEdges, undirectedEdges	$O(m)$
elements, positions	$O(n+m)$
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	$O(1)$
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	$O(\deg(v))$
areAdjacent(u, v)	$O(\min(\deg(u), \deg(v)))$
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	$O(1)$
removeVertex(v)	$O(\deg(v))$

Matrice d'adjacence (traditionnelle)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- Matrice M avec entrées pour toutes les paires de sommets
- M[i,j] = vrai signifie qu'il y a un arc (i,j) dans le graphe.
- M[i,j] = faux signifie qu'il n'y a aucun arc (i,j) dans le graphe.
- Il y a une entrée pour chaque arc possible, donc:

  espace requis = $\Theta(\mathbf{N^2})$

Matrice d'adjacence (moderne)

- Les structures à matrice d'adjacence ajoutent à la structure liste d'arcs une matrice où chaque rangée et colonne correspond à un sommet.

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS 0 DFW 1 JFK 2 LAX 3 MIA 4 ORD 5 SFO 6

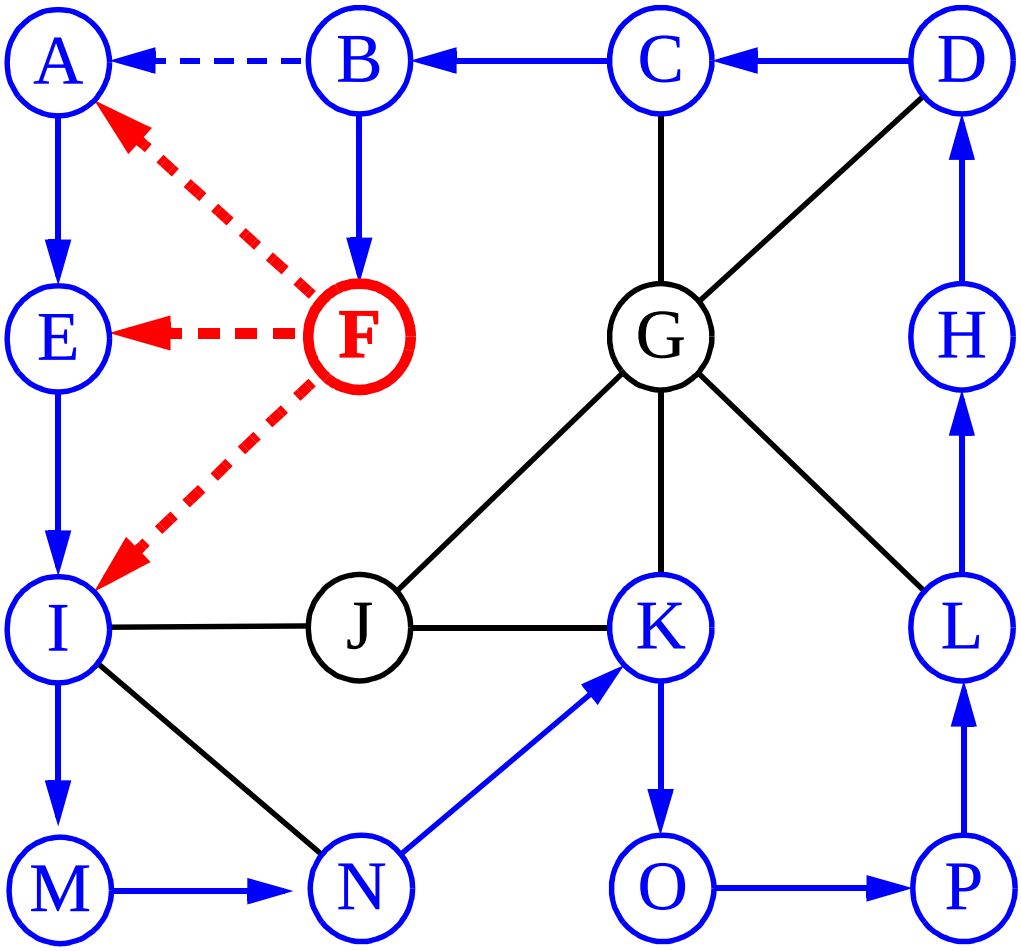
- L'espace requis est $O(n^2 + m)$

Performance de la structure Matrice d'adjacence

Opération	Temps
size, isEmpty, replaceElement, swap	$O(1)$
numVertices, numEdges	$O(1)$
vertices	$O(n)$
edges, directedEdges, undirectedEdges	$O(m)$
elements, positions	$O(n+m)$
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	$O(1)$
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	$O(n)$
areAdjacent	$O(1)$
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	$O(1)$
insertVertex, removeVertex	$O(n^2)$

TRAVERSÉES DE GRAPHES

- En profondeur (*Depth-First Search*)
- En largeur (*Breadth-First Search*)
- Patron de conception: méthode du gabarit (*Template Method Pattern*)



Explorer un labyrinthe sans se perdre

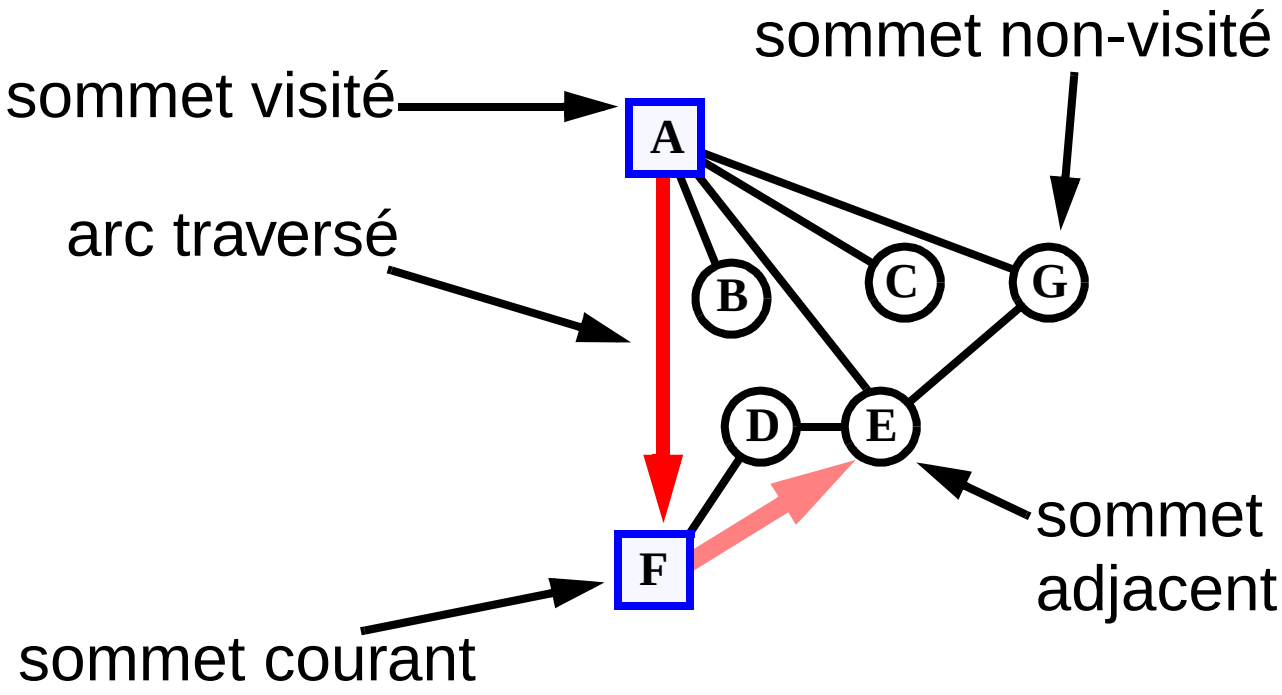
- Une **recherche en profondeur** **(***depth-first search* ou **DFS)** dans un graphe non-orienté G, c'est comme vagabonder dans un labyrinthe avec une corde et une cannette de peinture rouge, sans se perdre.
- Nous partons d'un sommet s, en attachant un bout de notre corde à ce point et en peinturant "visité" sur s. Ensuite, nous étiquetons s comme étant notre sommet courant appelé u.
- Maintenant, nous allons vers un arc arbitraire (u,v).
- Si l'arc (u,v) nous mène à un sommet v déjà visité, alors nous retournons à u.
- Si le sommet v n'a pas été visité, alors nous déroulons notre corde en allant à v, peinturons "visité" sur v, étiquetons v comme notre sommet courant, et répétons les étapes précédentes.
- Éventuellement, nous seront au point où tous les arcs attachés à u mènent à des sommets visités. Alors, nous revenons sur nos pas en déroulant la corde vers un sommet déjà visité v. Ainsi v devient notre sommet courant et nous répétons les étapes précédentes.

Explorer un labyrinthe sans se perdre (suite)

- Si tous les arcs attachés à v mènent à des sommets visités, alors nous revenons sur nos pas comme nous l'avons fait précédemment. Nous continuons à revenir sur nos pas en trouvant et en explorant les arcs inexplorés, et en répétant la procédure.
- Quand nous retournons au sommet *s* et qu'il n'y a plus d'arc inexploré attaché à ce point, alors nous avons terminé notre recherche DFS.

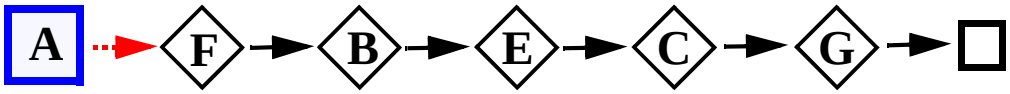
Recherche en profondeur DFS

Algorithme DFS(*v*);
Entrée: un sommet *v* dans un graphe
Sortie: un étiquetage des arcs comme étant découverts (*discovery edges*) ou arrières (*backedges*)
for chaque arc *e* attaché à *v* **do**
if l'arc *e* est inexploré **then**
soit *w* l'autre extrémité de *e*
if le sommet *w* est inexploré **then**
étiqueter *e* comme arc de découverte
appeler récursivement **DFS**(*w*)
else
étiqueter *e* comme arc arrière



Déterminer les arcs attachés

- DFS dépend de la façon dont ces arcs sont obtenus.
- Si nous commençons à A et examinons l'arc vers F, ensuite vers B, E, C, et enfin G:

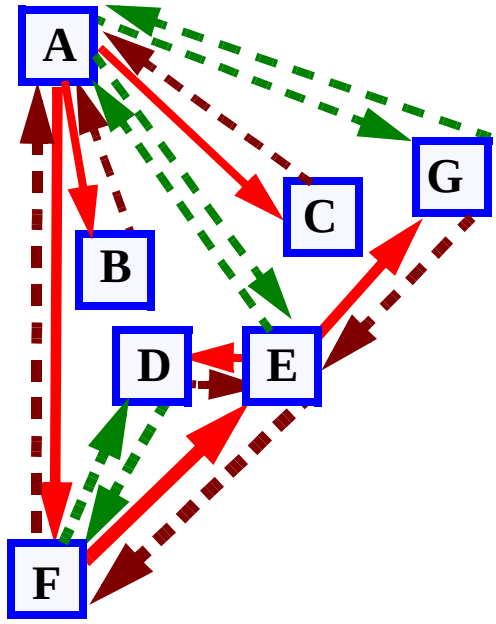


Le graphe résultant est:

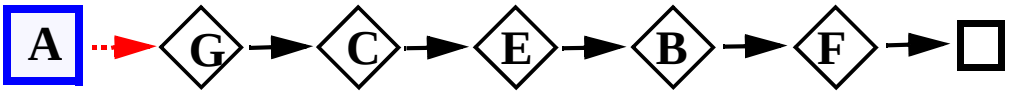
arc de découverte

arc arrière

retour d'une impasse

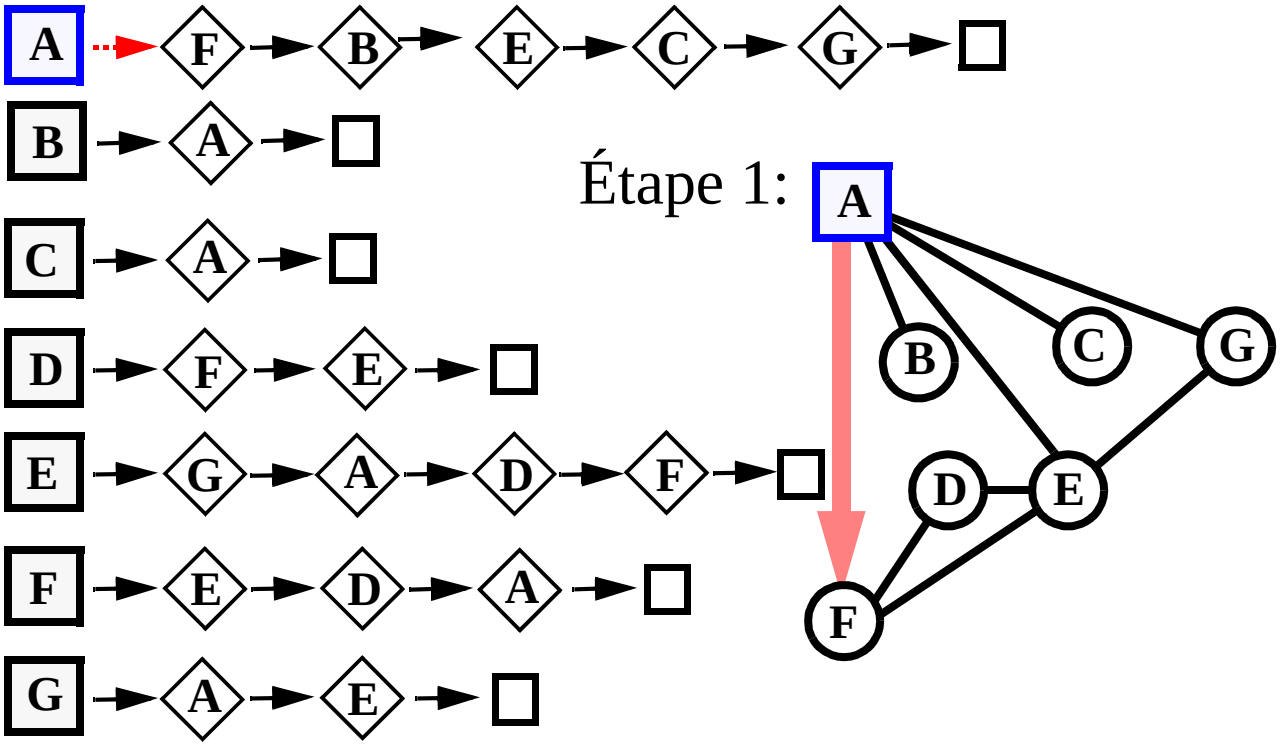


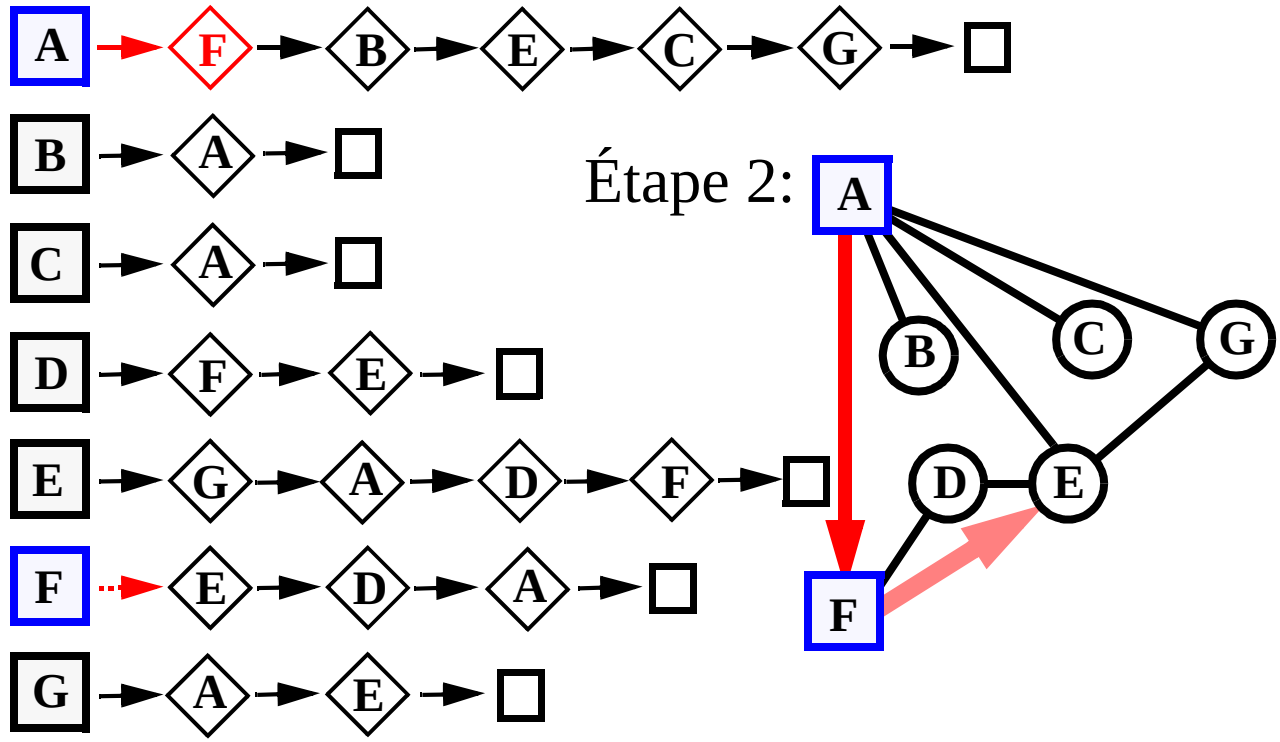
Si maintenant nous examinons l'arbre en commençant par A et ensuite G, C, E, B, et enfin F.

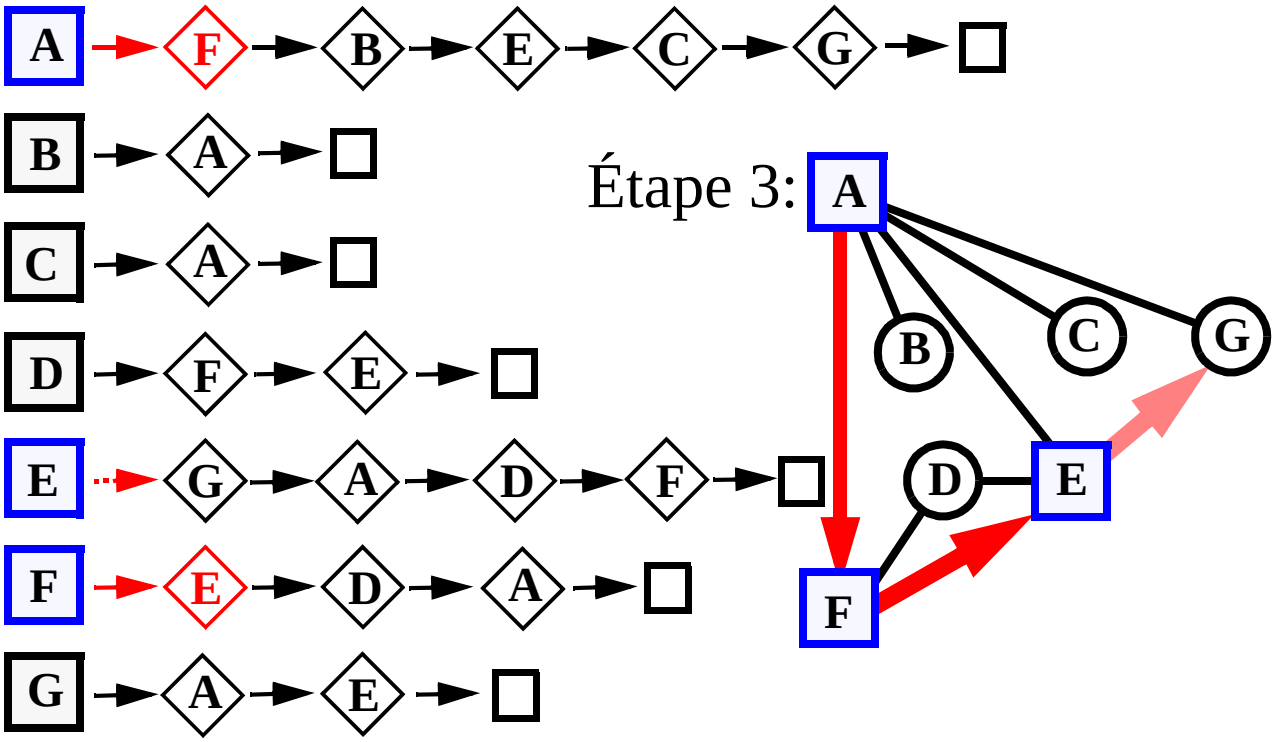


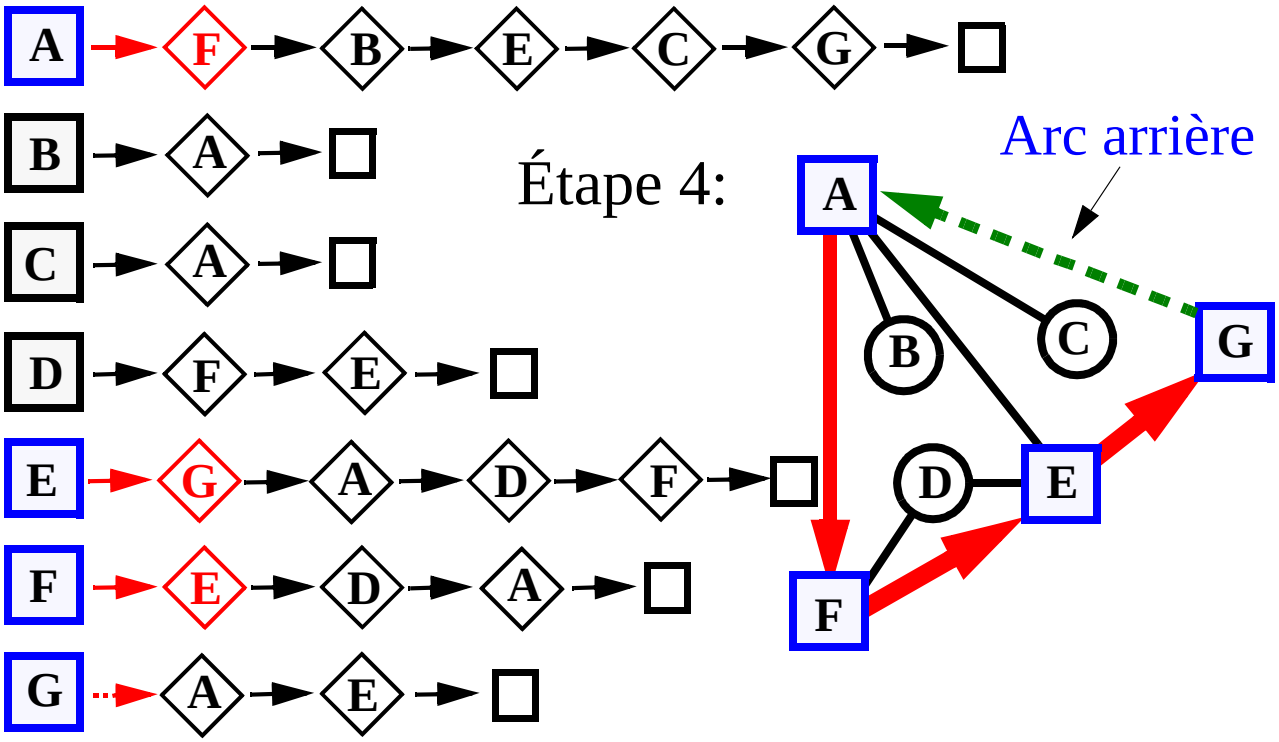
l'ensemble final d'arcs arrières et de découverte, de même que les points de retour, sont différents.

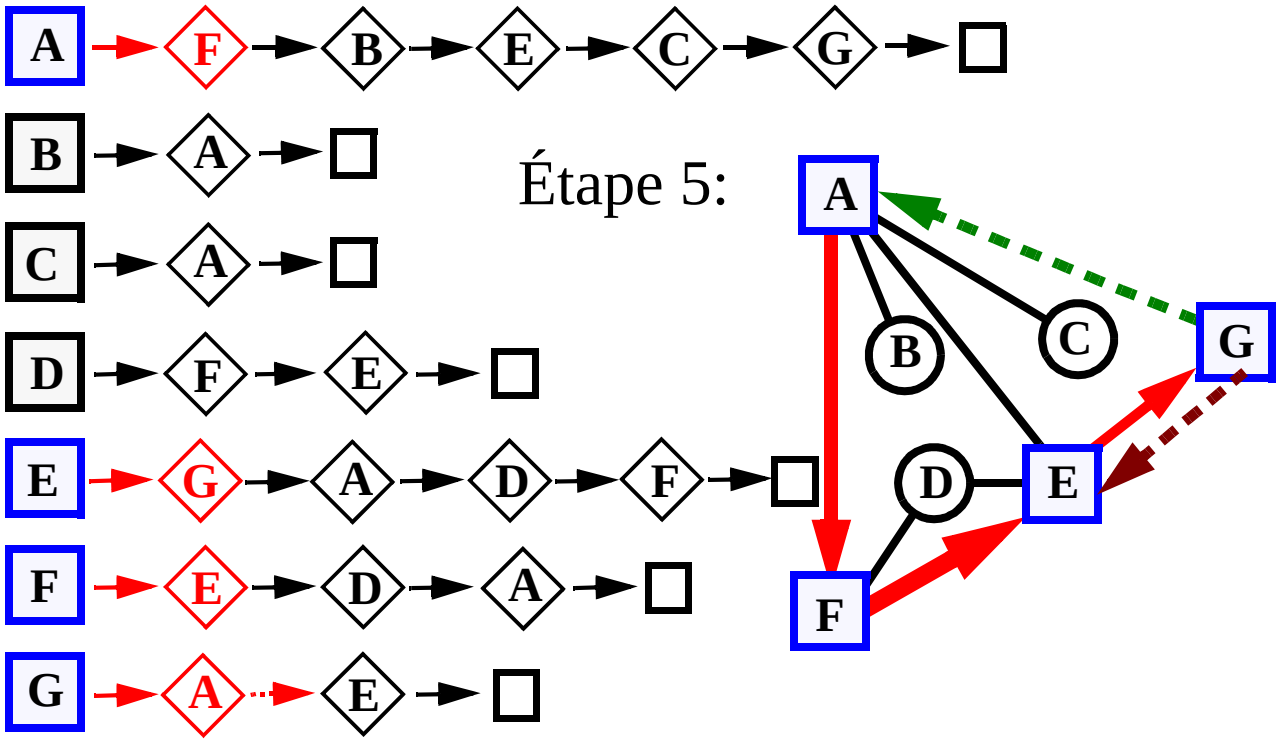
- Passons maintenant à un exemple de DFS.

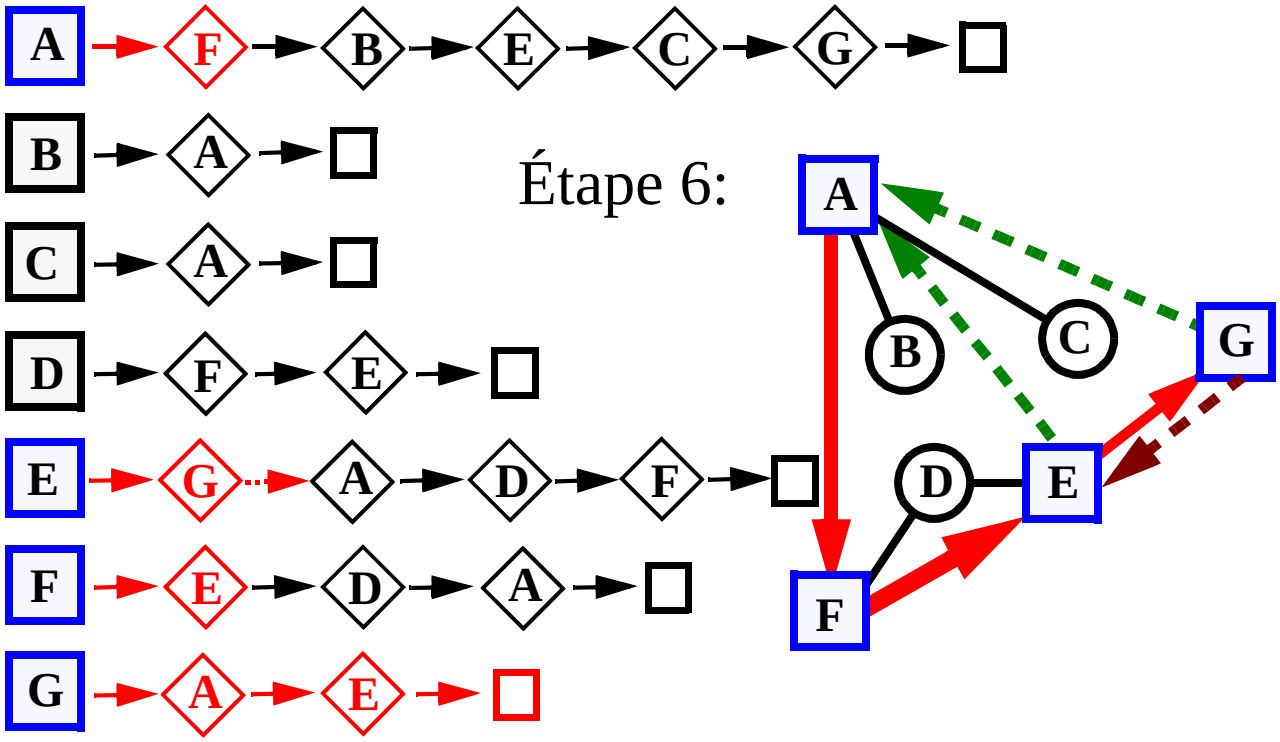

A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 1:
A
B
C
G
D
E
F

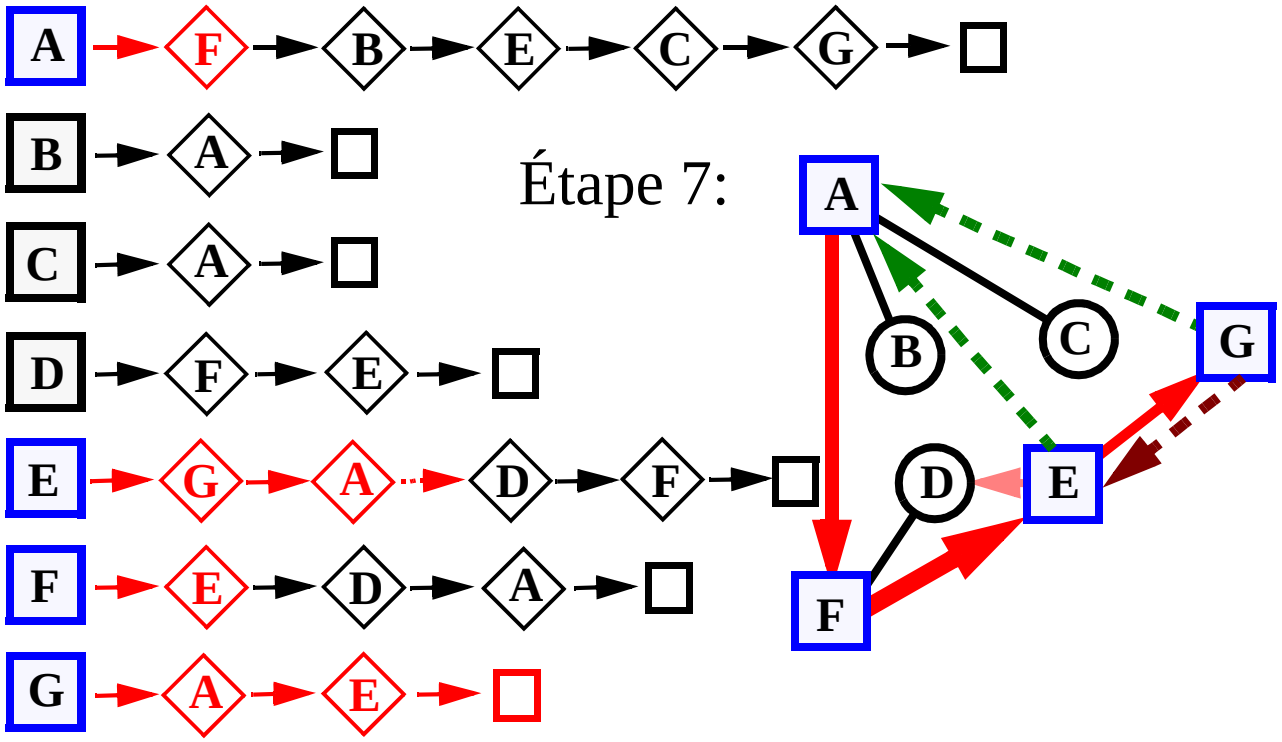

A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 2:
A
B
C
G
D
E
F

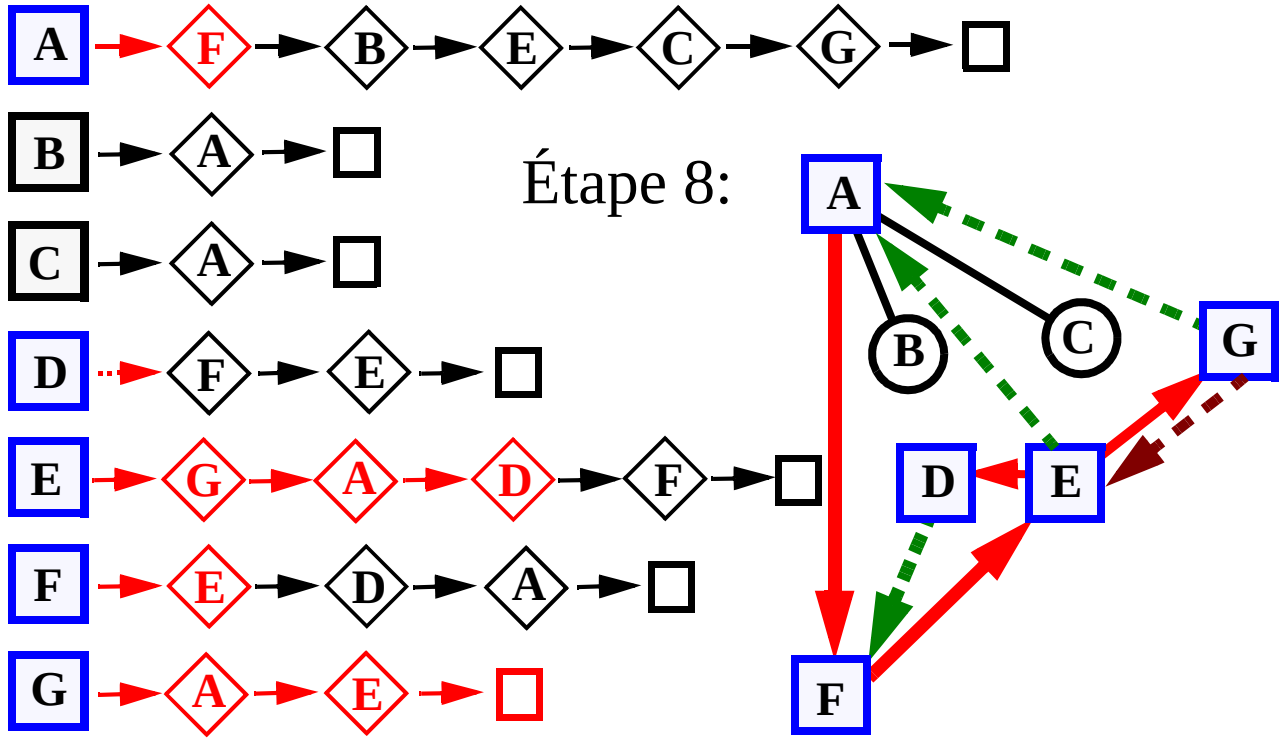

A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 3:
A
B
C
G
D
E
F

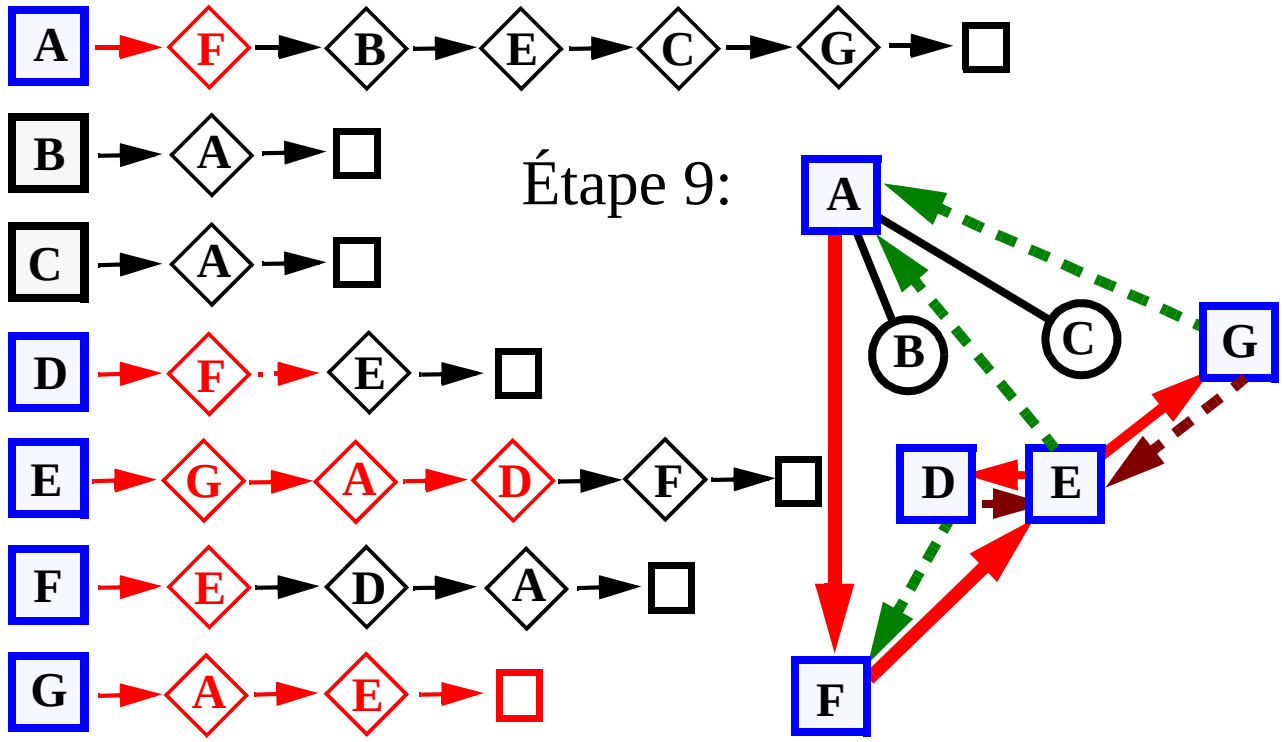

A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 4:
Arc arrière
A
B
C
G
D
E
F

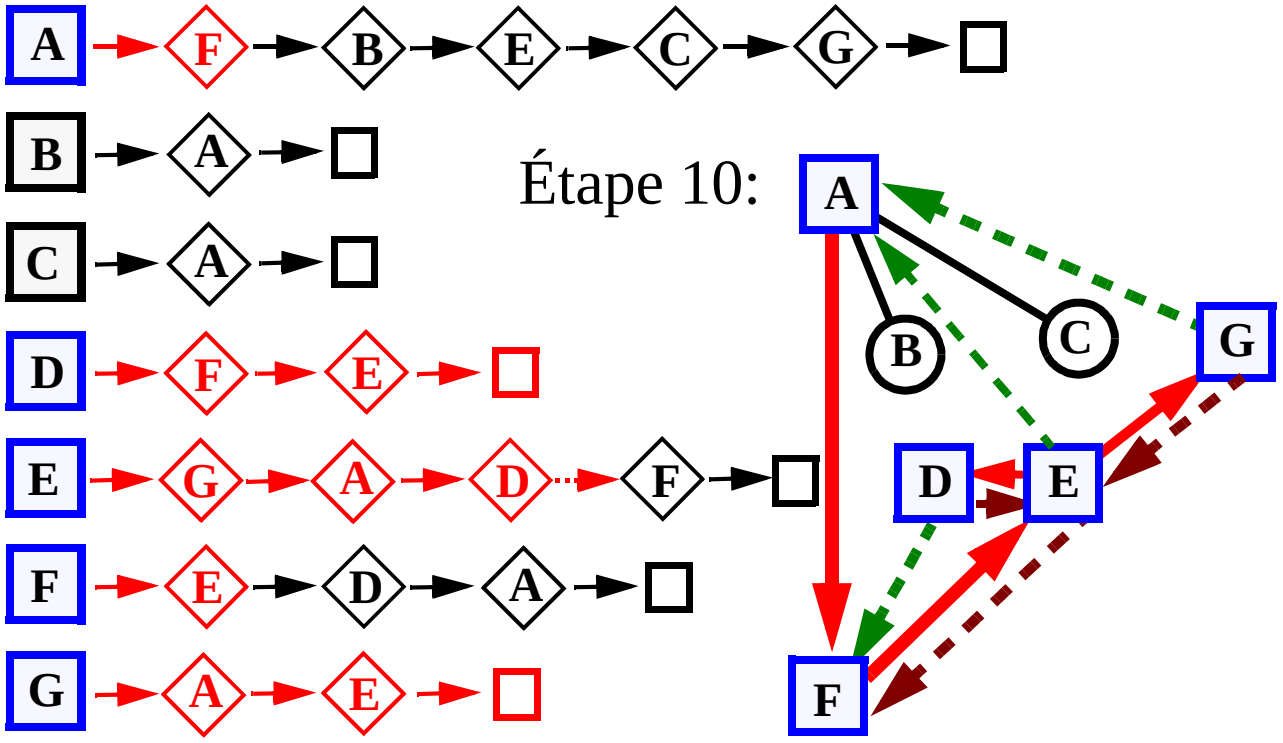

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 5:
A
B
C
G
D
E
F

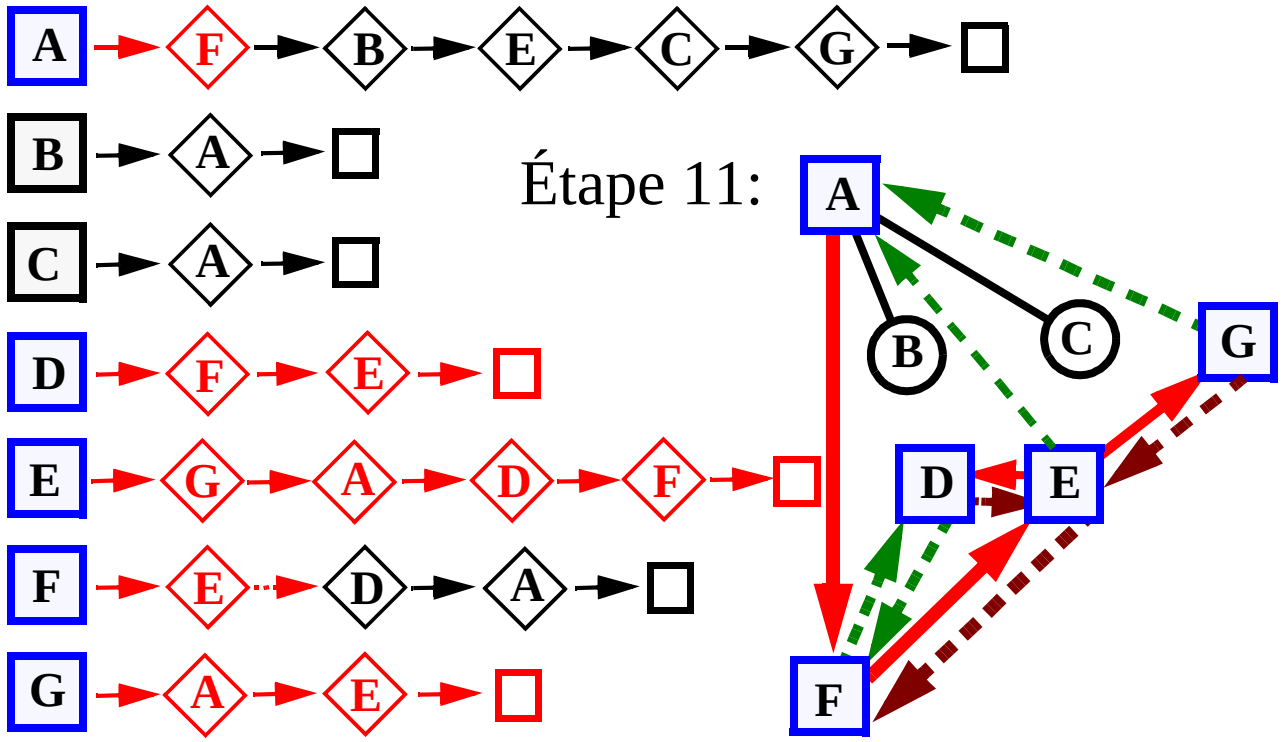

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 6:
A
B
C
G
D
E
F

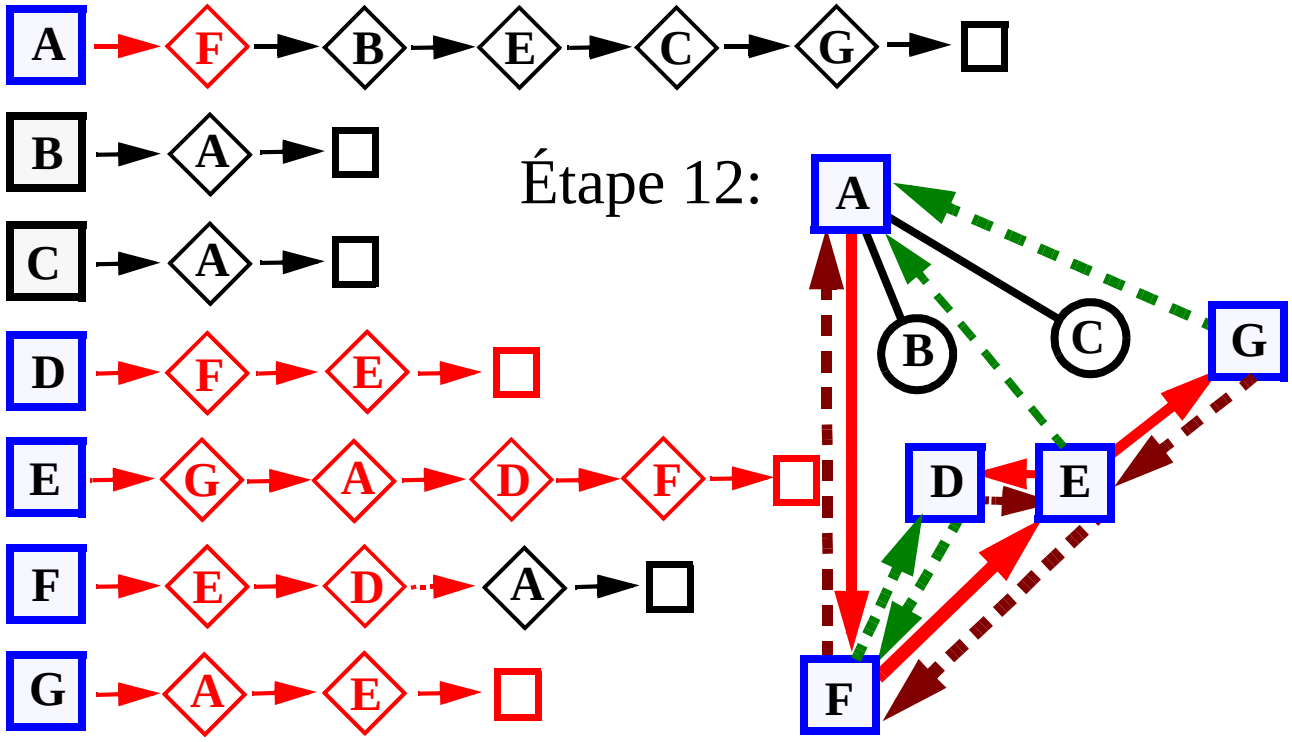

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 7:
A
B
C
G
D
E
F

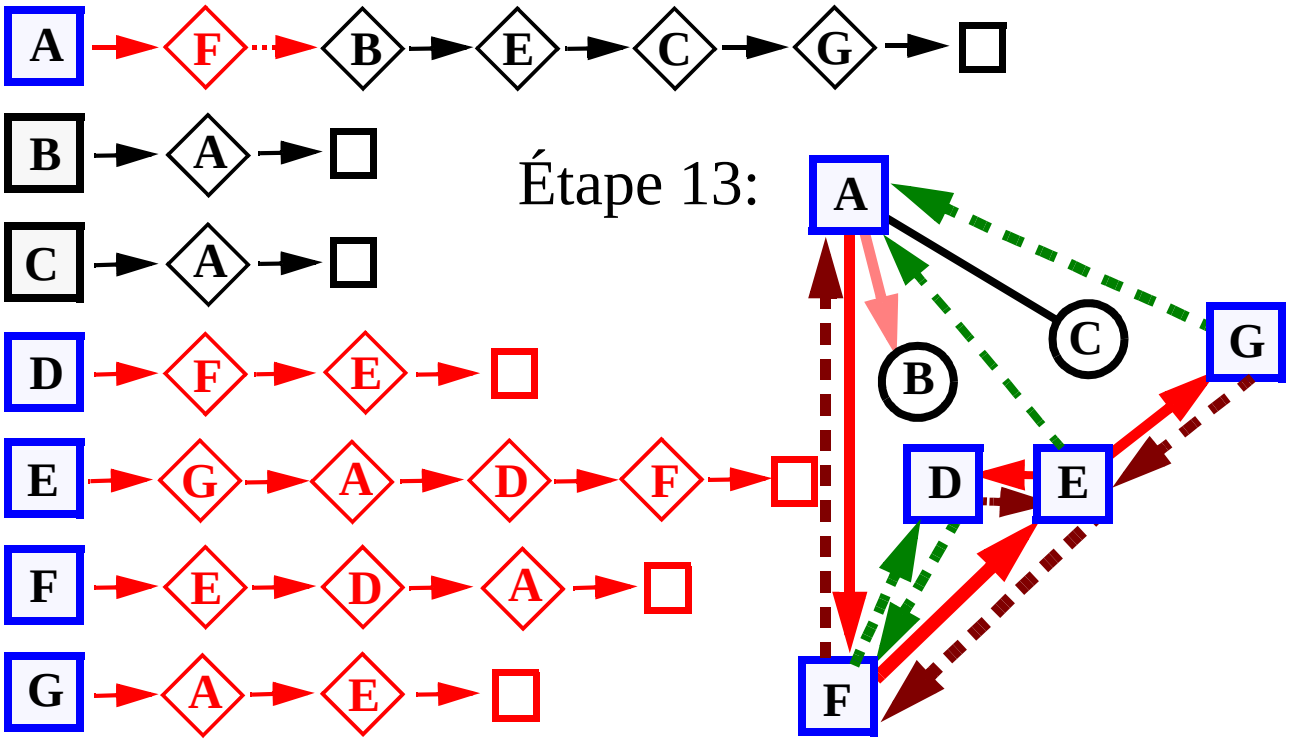

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 8:
A
B
C
G
D
E
F

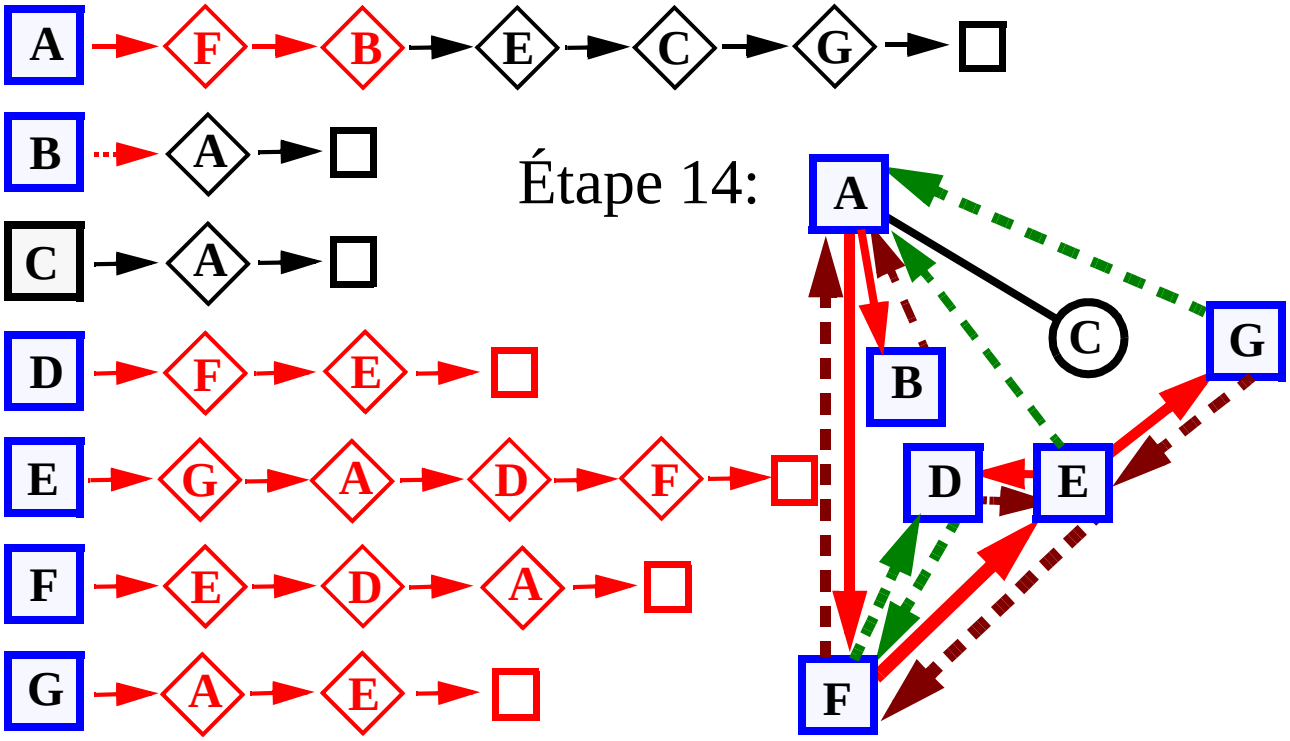

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 9:
A
B
C
G
D
E
F

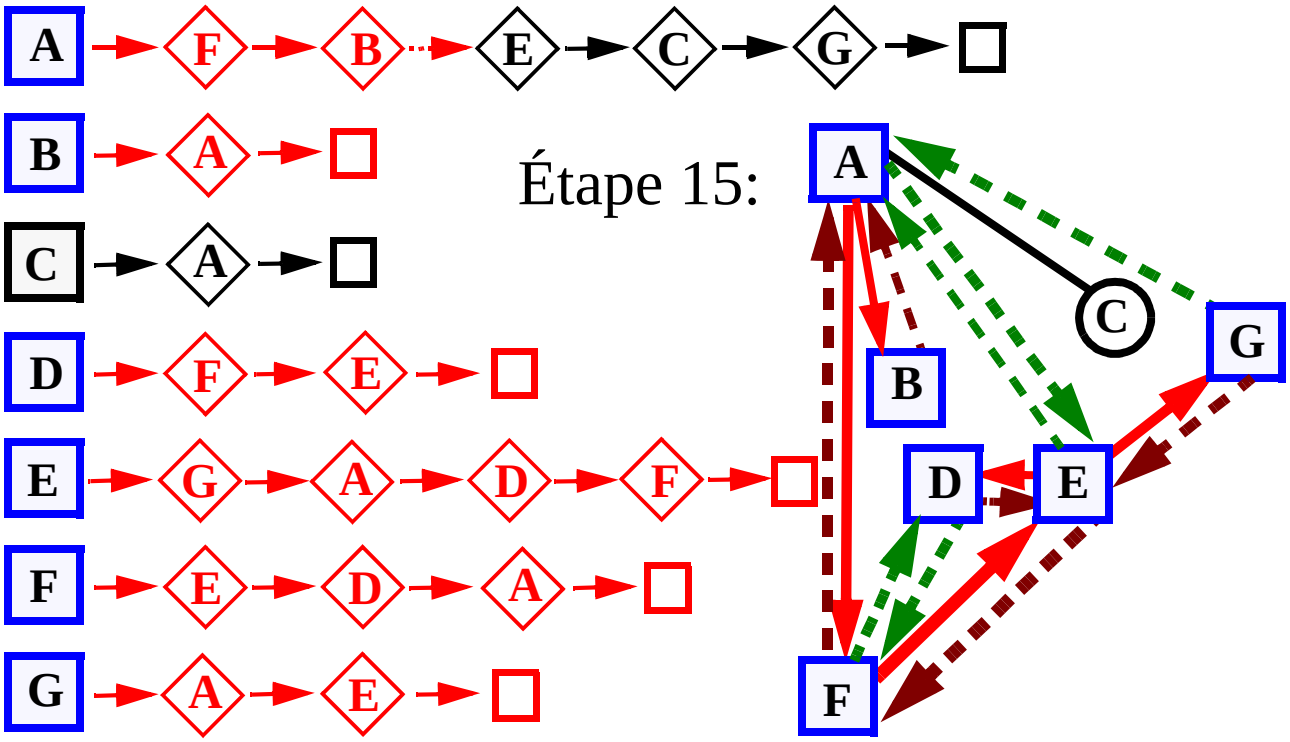

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 10:
A
B
C
G
D
E
F

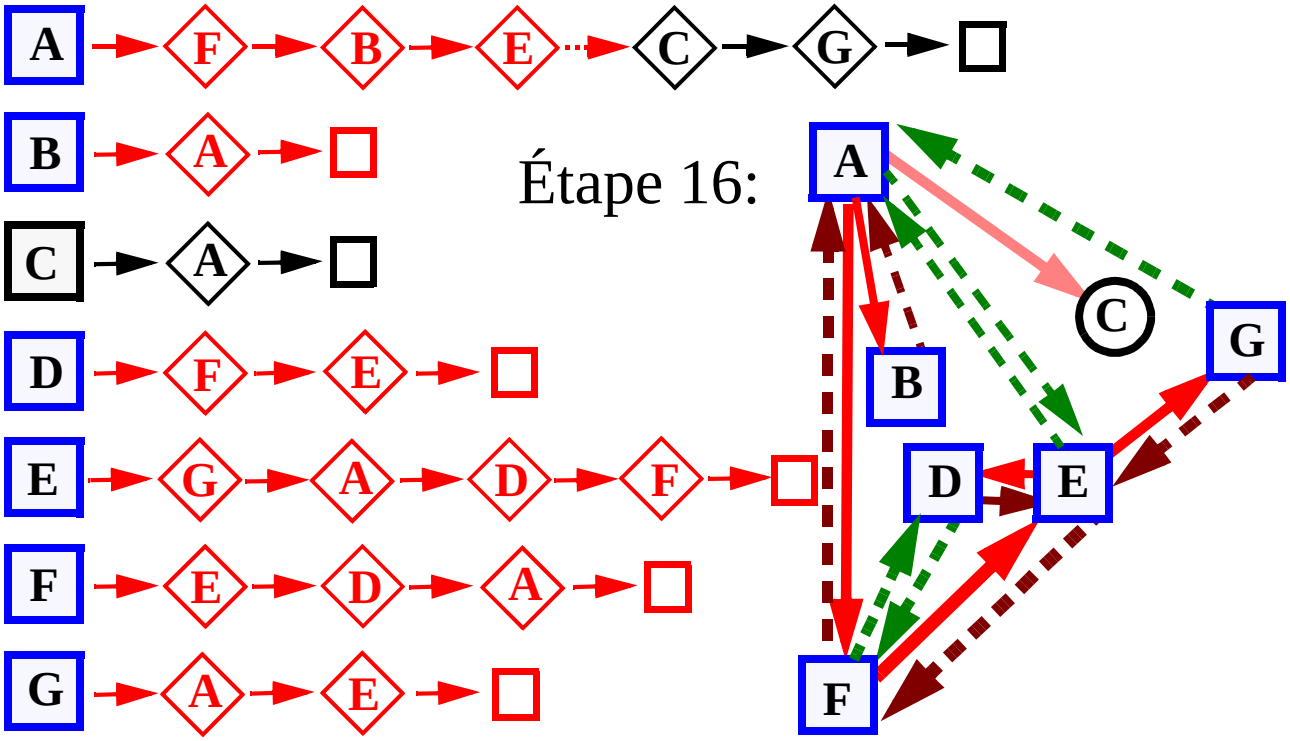

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 11:
A
B
C
G
D
E
F

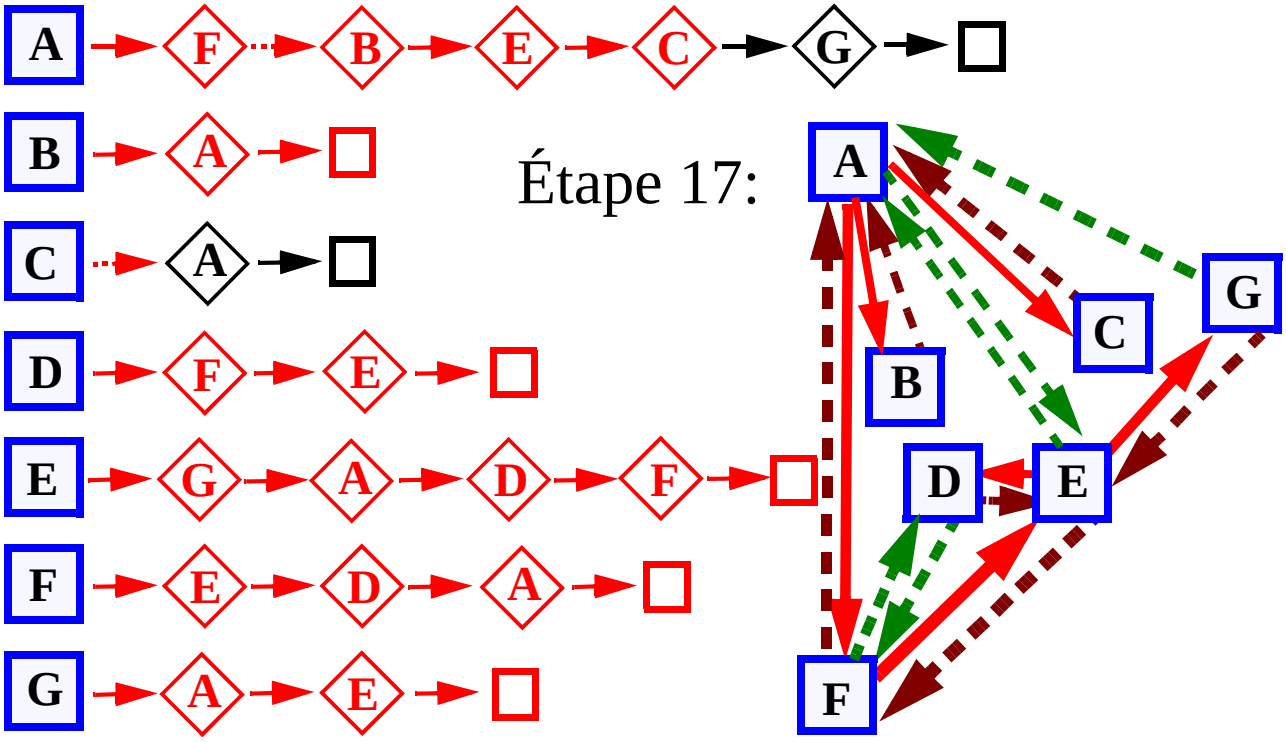

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 12:
A
B
C
G
D
E
F

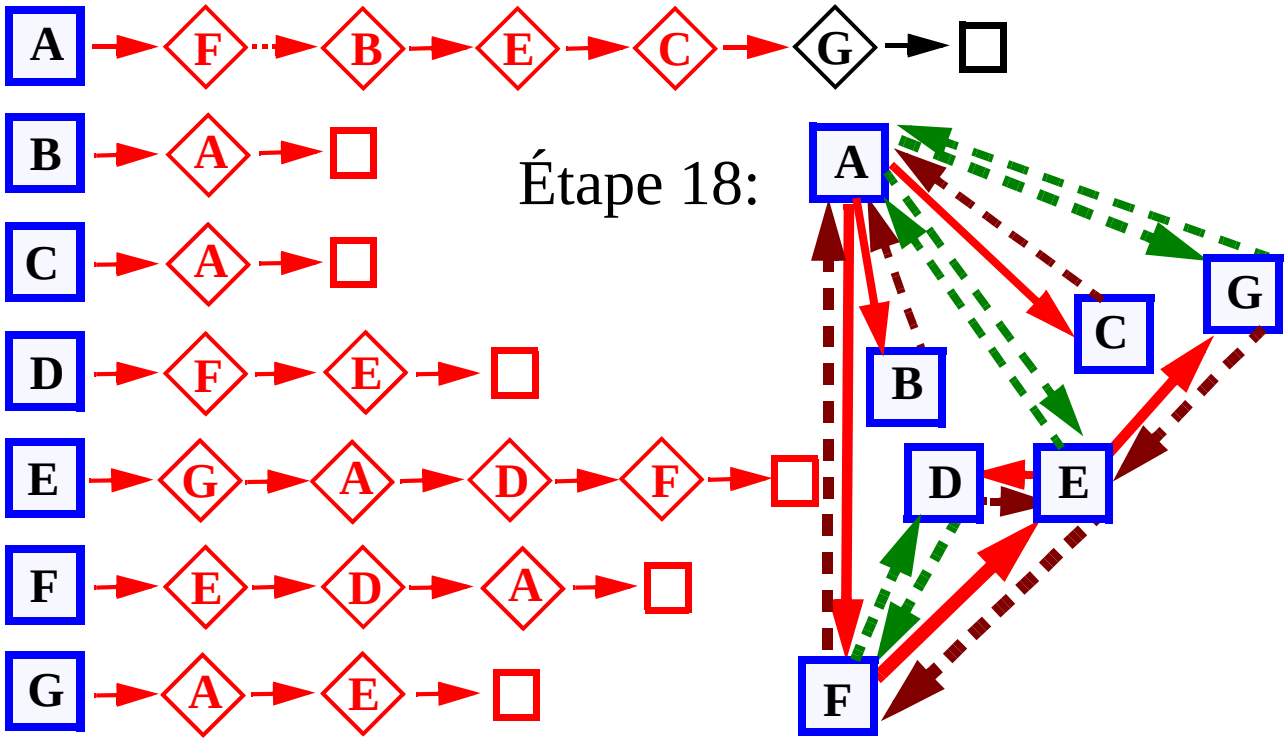

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 13:
A
B
C
G
D
E
F

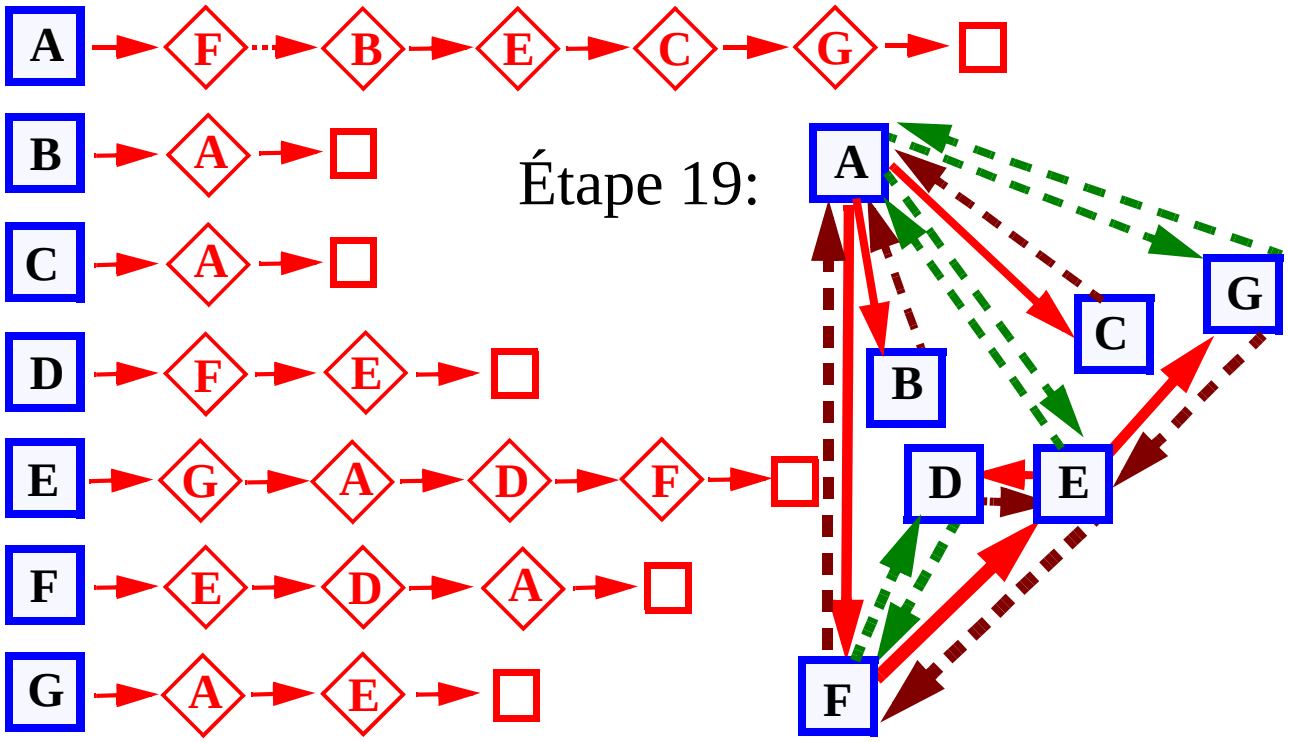

A F B E C G
B A
C A
D F E
E G A D F
F E D A
G A E
Étape 14:
A
B
C
G
D
E
F


A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 15:
A
C
G
B
D
E
F


A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 16:
A
C
G
B
D
E
F


A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 17:
A
C
G
B
D
E
F


A
F
B
E
C
G
B
A
C
A
D
F
E
E
G
A
D
F
F
E
D
A
G
A
E
Étape 18:
A
C
G
B
D
E
F



Et c’est tout!

Propriétés de DFS

- Proposition 9.12 : Soit G un graphe non-orienté sur lequel une traversée DFS commençant au sommet *s* a été faite. Alors:
  1) La traversée visite tous les sommets dans la composante connexe de *s*
  2) Les arcs de découverte forment un arbre recouvrant de la composante connexe de *s*

- Justification de 1):
  - Essayons une contradiction: supposons qu'il y ait au moins un sommet *v* non-visité et soit *w* le premier sommet non-visité sur un chemin de *s* à *v*.
  - Comme *w* est le premier sommet non-visité sur le chemin, il y a un voisin *u* qui a été visité.
  - Mais quand nous avons visité *u* nous devons avoir observé l'arc (*u*, *w*). Donc *w* doit avoir été visité.

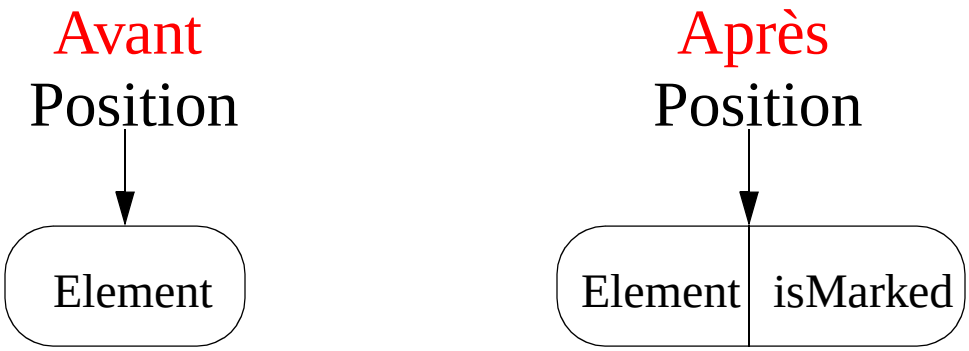
- Justification de 2):
  - Nous étiquetons seulement les arcs à partir du moment où nous allons vers des sommets non-visités. Ainsi, nous ne formons jamais de cycle d'arcs de découverte; ces arcs forment un arbre.
  - C'est un arbre recouvrant car DFS visite chaque sommet dans la composante connexe de *s*.

Analyse du temps d'exécution

- Souvenez-vous:
  - DFS sur chaque sommet une fois exactement.
  - Chaque arc est examiné exactement deux fois, une fois pour chacun de ses sommets.
- Pour n_s sommets et m_s arcs dans la composante connexe du sommet s, une DFS commençant à s s'exécute en un temps $O(n_s + m_s)$ si:
  - Le graphe est représenté dans une structure de données, comme une liste d'adjacence, où les méthodes pour les sommets et les arcs s'exécutent en un temps constant;
  - Étiqueter un sommet comme étant exploré et tester si un sommet a été exploré prend O(degré);
  - En étiquetant les nœuds visités, nous pouvons systématiquement considérer les arcs attachés au sommet courant, de façon à ne pas examiner le même arc plus d'une fois.

Étiquetage des sommets

- Étudions les façons d'étiqueter les sommets de façon à satisfaire les conditions mentionnées à la page précédente.
- Extension des positions de sommet pour inclure une variable servant à l'étiquetage.



- Utilisation d'un mécanisme de table de hachage qui satisfait ces conditions dans un sens probabiliste, parce qu'un tel mécanisme supporte les opérations d'étiquetage et de test en un temps attendu $O(1)$.

Patron de conception: méthode du gabarit (*Template Method Pattern*)

- le patron de conception **méthode du gabarit** offre un ***mécanisme de calcul générique*** qui peut être spécialisé en redéfinissant certaines étapes

- pour l'appliquer, nous concevons une classe qui:
  - réalise le ***squelette*** d'un algorithme
  - invoque des méthodes auxiliaires qui peuvent être redéfinies par ses sous-classes afin de faire des calculs utiles

- **Bénéfices**
  - fait que la rectitude des calculs spécialisés dépend de celle de l'algorithme squelette
  - démontre la puissance de l'héritage
  - promeut la réutilisation de code
  - encourage le développement de code générique

- **Exemples**
  - ***traversée générique d'un arbre binaire*** (qui inclut pré-ordre, in-ordre, et post-ordre) et ses applications
  - ***recherche en profondeur générique d'un graphe non-orienté*** et ses applications

Recherche en profondeur générique

```
public abstract class DFS {
 protected Object dfsVisit(Vertex v) {
    protected InspectableGraph graph;
    protected Object visitResult;
    initResult();
    startVisit(v);
    mark(v);
    for (Enumeration inEdges = graph.incidentEdges(v);
          inEdges.hasMoreElements();) {
      Edge nextEdge = (Edge) inEdges.nextElement();
      if (!isMarked(nextEdge)) { // found an unexplored edge
        mark(nextEdge);
        Vertex w = graph.opposite(v, nextEdge);
        if (!isMarked(w)) { // discovery edge
          mark(nextEdge);
          traverseDiscovery(nextEdge, v);
          if (!isDone())
            visitResult = dfsVisit(w); }
        else  // back edge
          traverseBack(nextEdge, v);
      }
    }
    finishVisit(v);
    return result();
  }
```

Méthodes auxiliaires de recherche DFS générique

```
public Object execute(InspectableGraph g, Vertex start,
                      Object info) {
  graph = g;
  return null;
}

protected void initResult() {}

protected void startVisit(Vertex v) {}

 protected void traverseDiscovery(Edge e, Vertex from) {}

protected void traverseBack(Edge e, Vertex from) {}

protected boolean isDone() { return false; }

protected void finishVisit(Vertex v) {}

protected Object result() { return new Object(); }
```

Observons maintenant 4 façons de spécialiser DFS générique!

- la classe FindPath spécialise DFS afin de retourner un chemin du sommet start vers le sommet target.

```
public class FindPathDFS extends DFS {
  protected Sequence path;
  protected boolean done;
  protected Vertex target;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    path = new NodeSequence();
    done = false;
    target = (Vertex) info;
    dfsVisit(start);
    return path.elements();
  }
  protected void startVisit(Vertex v) {
    path.insertFirst(v);
    if (v == target) { done = true; }
  }
  protected void finishVisit(Vertex v) {
    if (!done) path.remove(path.first());
  }
  protected boolean isDone() { return done; }
```

Autre spécialisation de DFS générique...

- FindAllVertices spécialise DFS afin de retourner une énumération des sommets dans la composante connexe contenant le sommet start.

```
public class FindAllVerticesDFS extends DFS {
   protected Sequence vertices;
   public Object execute(InspectableGraph g, Vertex start,
                         Object info) {
      super.execute(g, start, info);
      vertices = new NodeSequence();
      dfsVisit(start);
      return vertices.elements();
   }

   public void startVisit(Vertex v) {
      vertices.insertLast(v);
   }
}
```

Plus de spécialisations de DFS générique...

- ConnectivityTest utilise une spécialisation de DFS pour déterminer si un graphe est connecté.

```
public class ConnectivityTest {
   protected static DFS tester=new FindAllVerticesDFS();
   public static boolean isConnected(InspectableGraph g)
   {
      if (g.numVertices() == 0) return true; //empty is
                                             //connected
      Vertex start = (Vertex)g.vertices().nextElement();
      Enumeration compVerts =
                  (Enumeration)tester.execute(g, start, null);
      // count how many elements are in the enumeration
      int count = 0;
      while (compVerts.hasMoreElements()) {
         compVerts.nextElement();
         count++;
      }
      if (count == g.numVertices()) return true;
      return false;
   }
}
```

Et une autre spécialisation de DFS générique!

- FindCycle spécialise DFS afin de déterminer si la composante connexe du sommet start contient un cycle, et alors le retourne.

```
public class FindCycleDFS extends DFS {
   protected Sequence path;
   protected boolean done;
   protected Vertex cycleStart;
   public Object execute(InspectableGraph g, Vertex start,
                         Object info) {
      super.execute(g, start, info);
      path = new NodeSequence();
      done = false;
      dfsVisit(start);
      //copy the vertices up to cycleStart from the path to
      //the cycle sequence.
      Sequence theCycle = new NodeSequence();
      Enumeration pathVerts = path.elements();
```

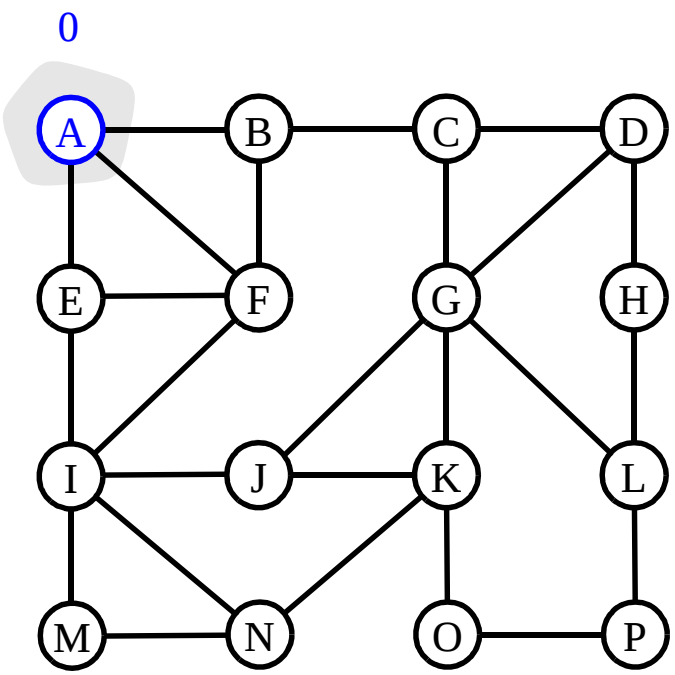
```
        while (pathVerts.hasMoreElements()) {
            Vertex v = (Vertex)pathVerts.nextElement();
            theCycle.insertFirst(v);
            if ( v == cycleStart) {
                break;
            }
        }
        return theCycle.elements();
    }
    protected void startVisit(Vertex v) {path.insertFirst(v);}
    protected void finishVisit(Vertex v) {
        if (done) {path.remove(path.first());}
    }
    //When a back edge is found, the graph has a cycle
    protected void traverseBack(Edge e, Vertex from) {
        Enumeration pathVerts = path.elements();
        cycleStart = graph.opposite(from, e);
        done = true;
    }
    protected boolean isDone() {return done;}
}
```

Recherche en largeur BFS (*Breadth-First Search*)

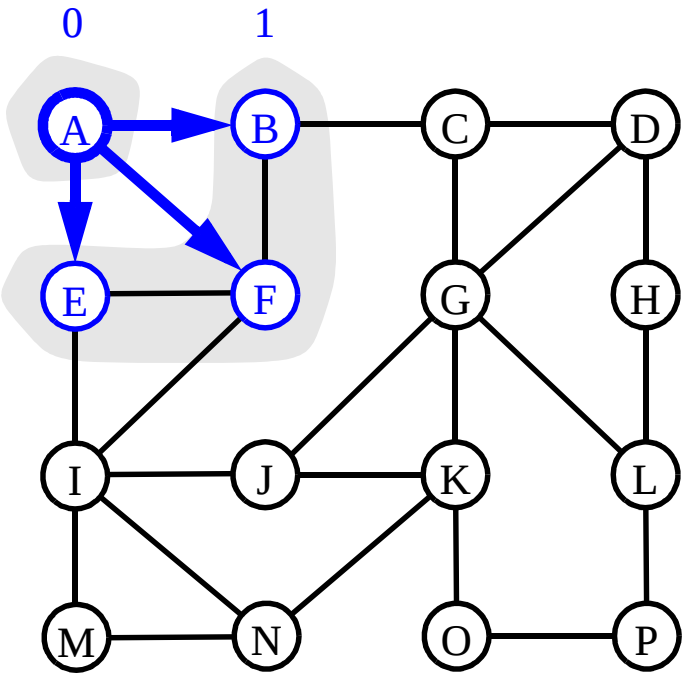
- Comme DFS, une recherche en largeur (BFS) traverse une composante connexe d'un graphe, et ce faisant définit un arbre recouvrant qui a quelques propriétés utiles
  - Le sommet de départ *s* a un niveau 0; comme dans DFS, définissons ce point comme point d'ancrage.
  - Au premier tour, la corde est déroulée de la longueur d'un arc, et tous les arcs à une distance d'un arc du point d'ancrage sont visités.
  - Ces arcs sont placés dans le niveau 1.
  - Au second tour, tous les nouveaux arcs qui peuvent être atteints en déroulant la corde d'une longueur de 2 arcs sont visités et placés dans le niveau 2.
  - Ceci se poursuit jusqu'à ce que tous les sommets aient été placés dans un niveau.
  - L'étiquette de tout sommet *v* correspond à la longueur du plus court chemin de *s* à *v*.

BFS - Une Représentation graphique

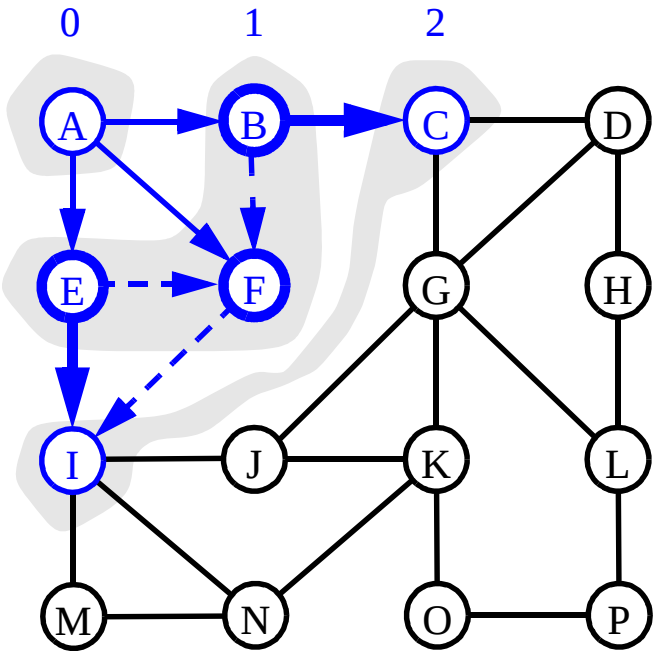
a)



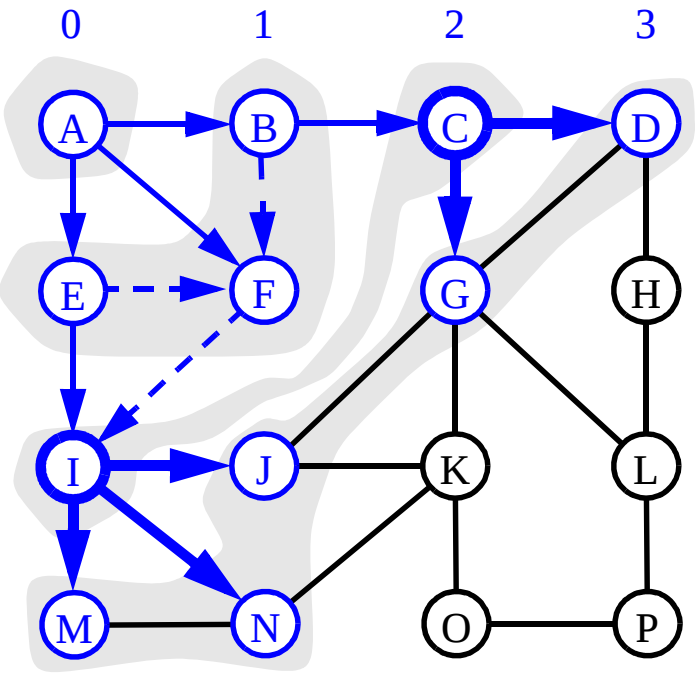
b)



c)

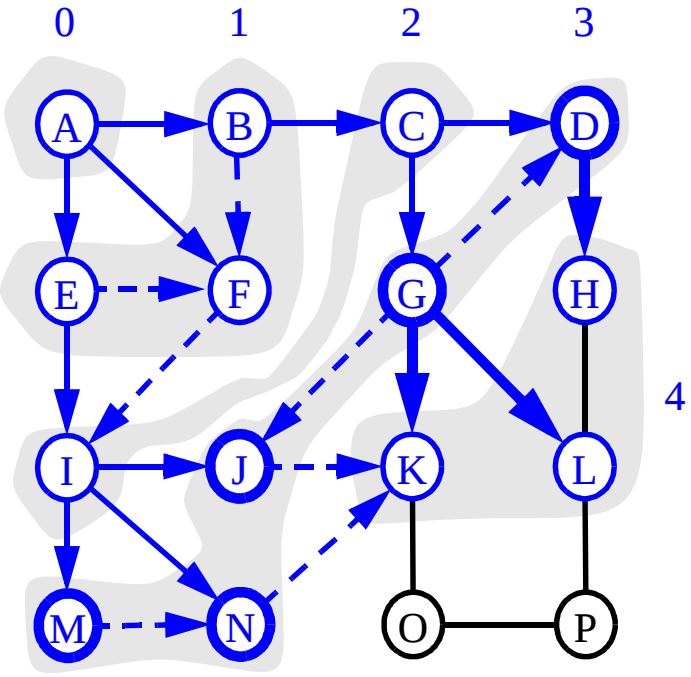


d)

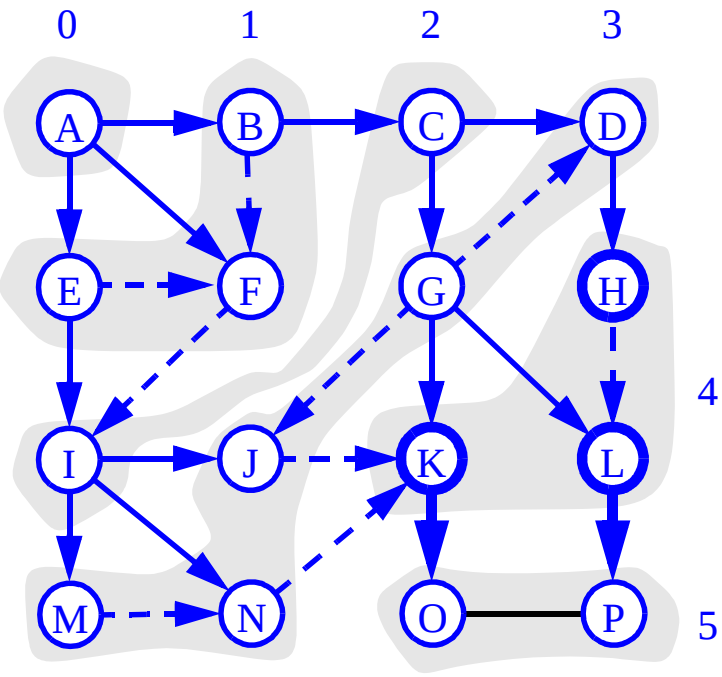


Encore BFS

e)



f)



Pseudo-code BFS

Algorithme BFS(s):
  Entrée: Un sommet s dans un graphe
  Sortie: Un étiquetage des arcs comme étant
    découverts (*discovery edges*) ou traversés
    (*cross edges*)
  initialiser le contenant L_0 avec le sommet s
  $i \leftarrow 0$
  while L_i n'est pas vide do
    créer le contenant L_{i+1} initialement vide
    for chaque sommet v dans L_i do
      if l'arc e est attaché à v do
        soit w l'autre extrémité de e
        if le sommet w est inexploré then
          étiqueter e comme arc de découverte
          insérer w dans L_{i+1}
        else
          étiqueter e comme arc traversé
  $i \leftarrow i + 1$

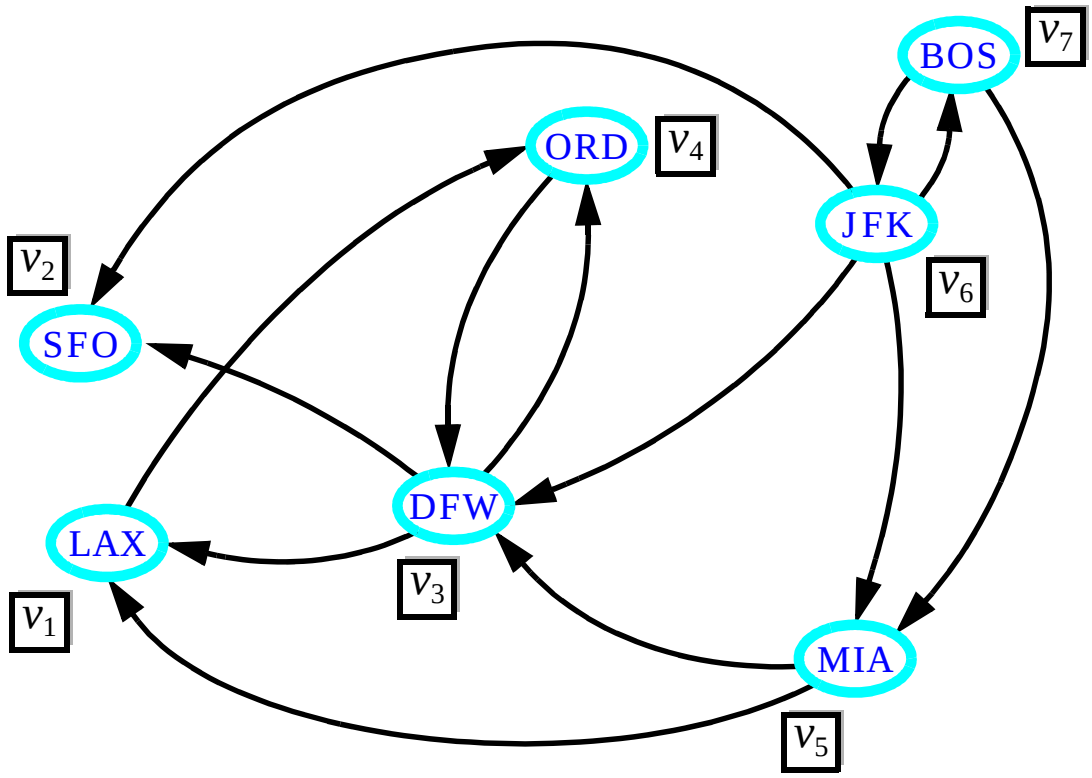
Propriétés de BFS

- Proposition: Soit G un graphe non-orienté sur lequel une traversée BFS débutant au sommet s a été faite. Alors:
  - La traversée visite tous les sommets dans la composante connexe de s.
  - Les arcs de découverte forment un arbre recouvrant T, que nous appelons arbre BFS, de composante connexe de s.
  - Pour chaque sommet v au niveau i, le chemin de l'arbre BFS T entre s et v a i arcs, et tout autre chemin de G entre s et v a au moins i arcs.
  - Si (u, v) est un arc qui n'est pas dans l'arbre BFS, alors les niveaux de u et v diffèrent de 1 au plus.

- Proposition: Soit G un graphe avec n sommets et m arcs. Une traversée BFS de G a un temps $O(n + m)$. Aussi, il existe des algorithmes au temps $O(n + m)$ basés sur BFS pour les problèmes suivants:
  - Tester si G est connexe
  - Calculer l'arbre recouvrant de G
  - Calculer les composantes connexes de G
  - Calculer, pour chaque sommet v de G, le nombre minimum d'arcs de tout chemin entre s et v.

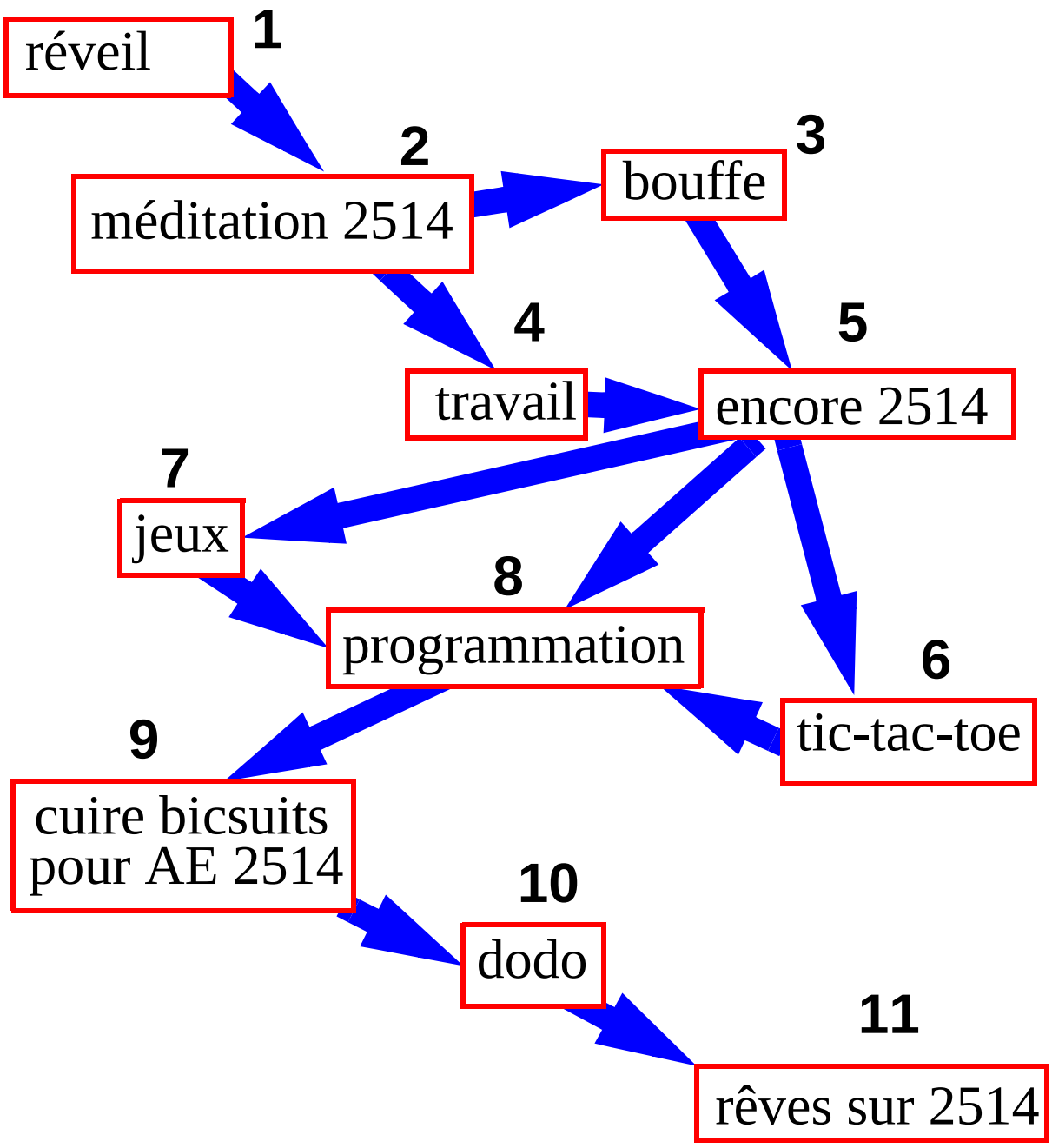
DIGRAPHES

- Accessibilité (*reachability*)
- Connectivité
- Fermeture transitive (*closure*)
- Algorithme de Floyd-Warshall



DIGRAPHES

une journée typique...

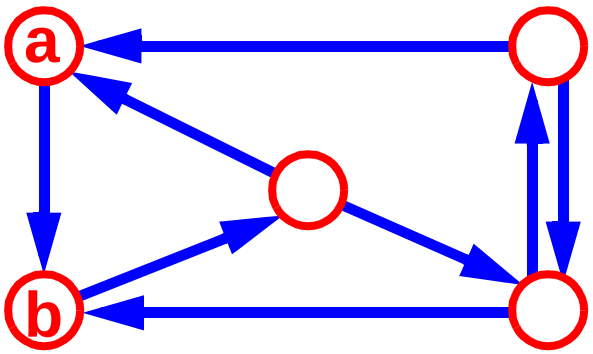


Qu'est-ce qu'un digraphe?

Un graphe orienté (de l'anglais *directed graph*)!

Chaque arc va dans une direction

L'arc (a,b) va de a à b, mais pas de b à a

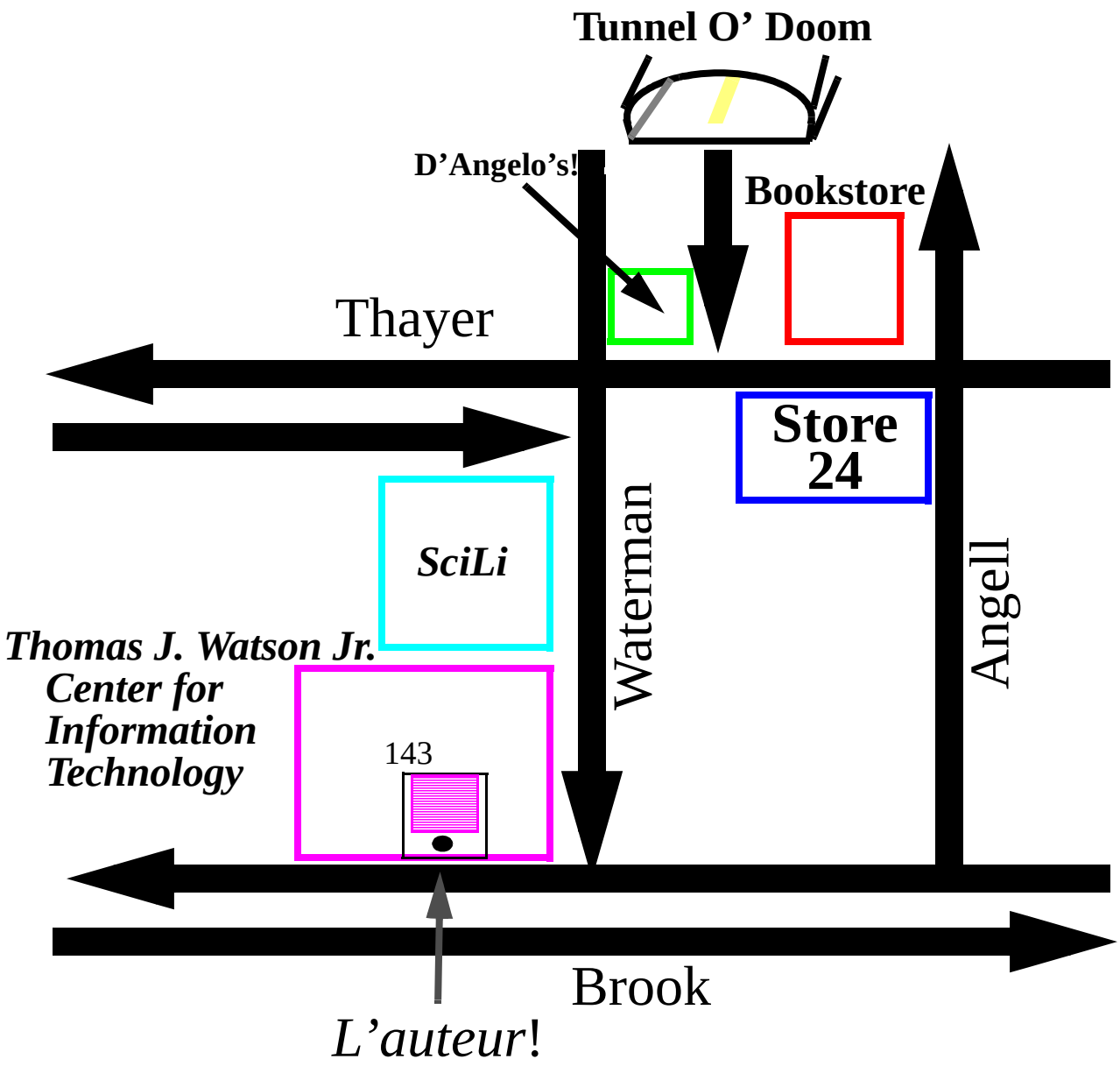


Vous dites sûrement: "Ouin, et si nous avions un exemple qui démontrerait combien nous pourrions être éclairés par l'utilisation de digraphes?!! – Et bien, si vous insistez. . .

Applications

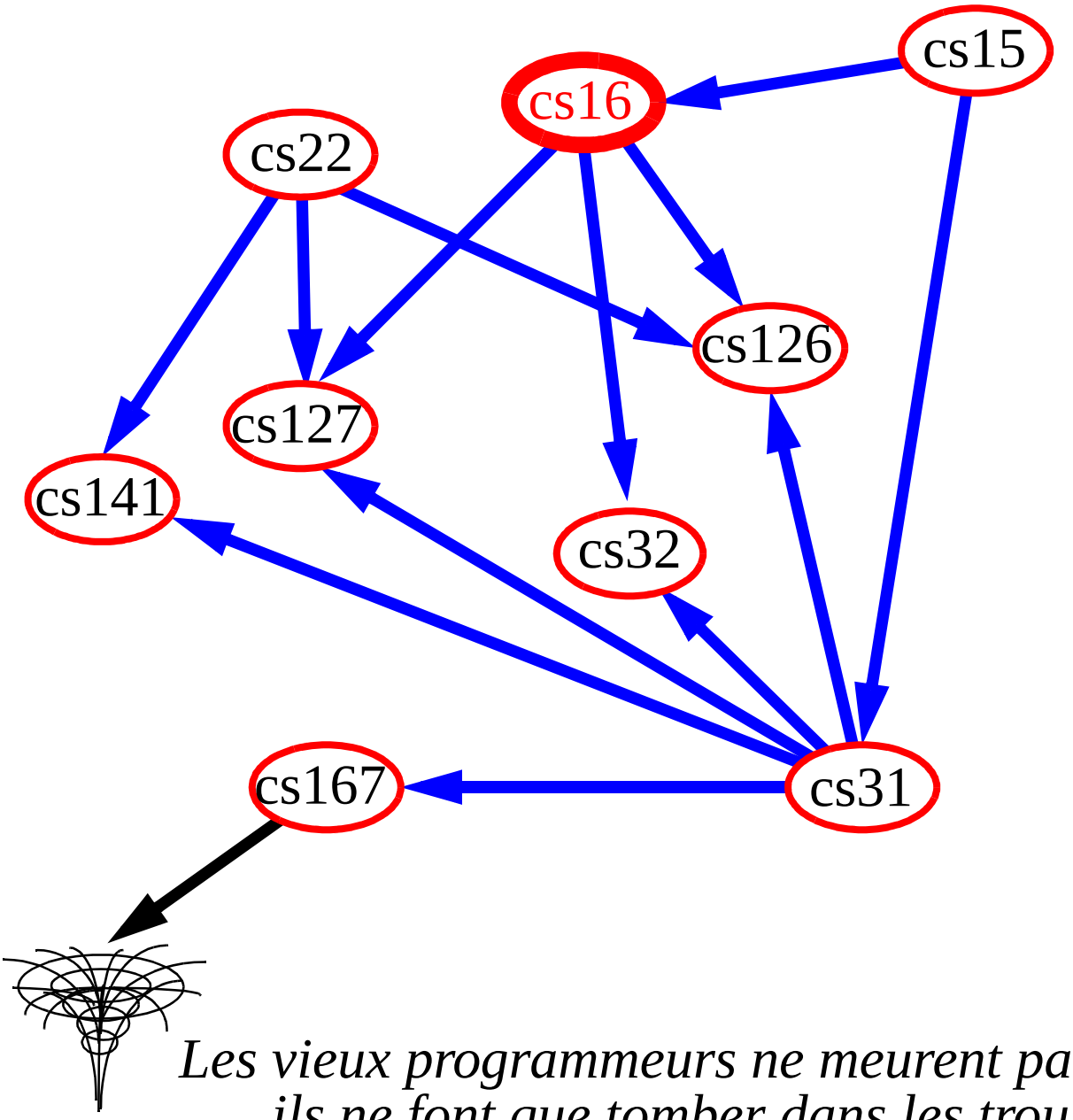
Cartes: les digraphes peuvent représenter les rues à sens unique

(utiles dans les grands centres-villes)



Une autre application

Planification d'horaires: l'arc (a,b) indique que la tâche a doit être complétée avant que b ne démarre.



Les vieux programmeurs ne meurent pas—
ils ne font que tomber dans les trous noirs

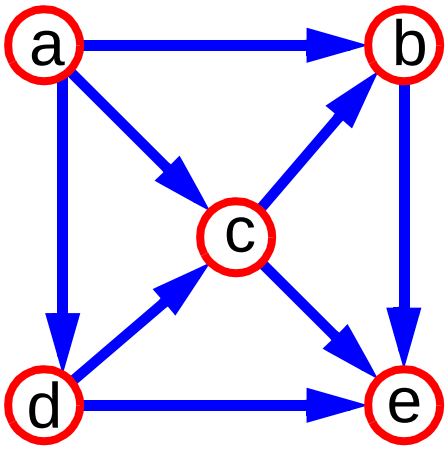
Les GOA!

GOA: Graphe Orienté Acyclique

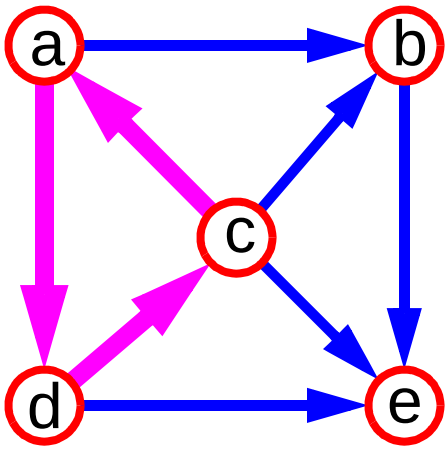
(de l'anglais *directed acyclic graph — DAG*)

Pardon?!!

C'est un graphe orienté sans cycles orientés



GOA

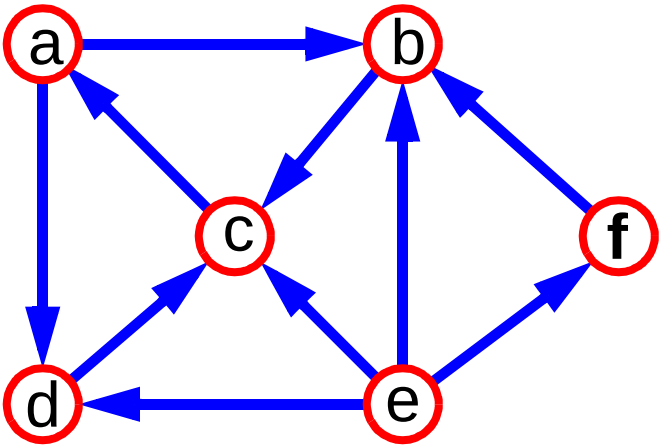


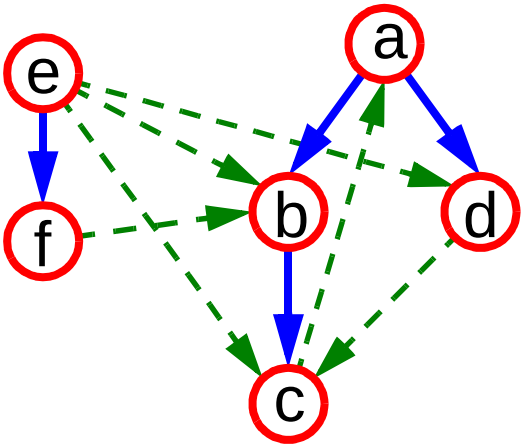
pas un GOA

Recherche en profondeur

Même algorithme que pour les graphes non-orientés

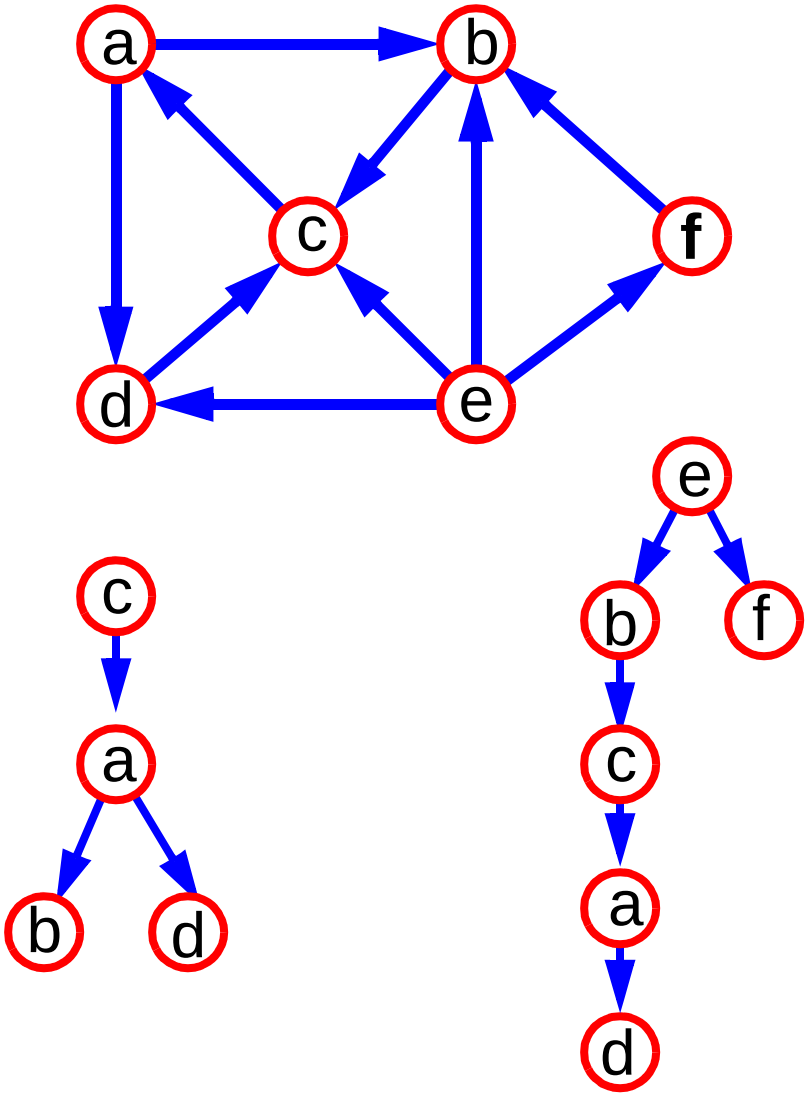
Sur un digraphe connexe, nous pouvons obtenir des arbres DFS non-connexes (c'est-à-dire, une forêt DFS)





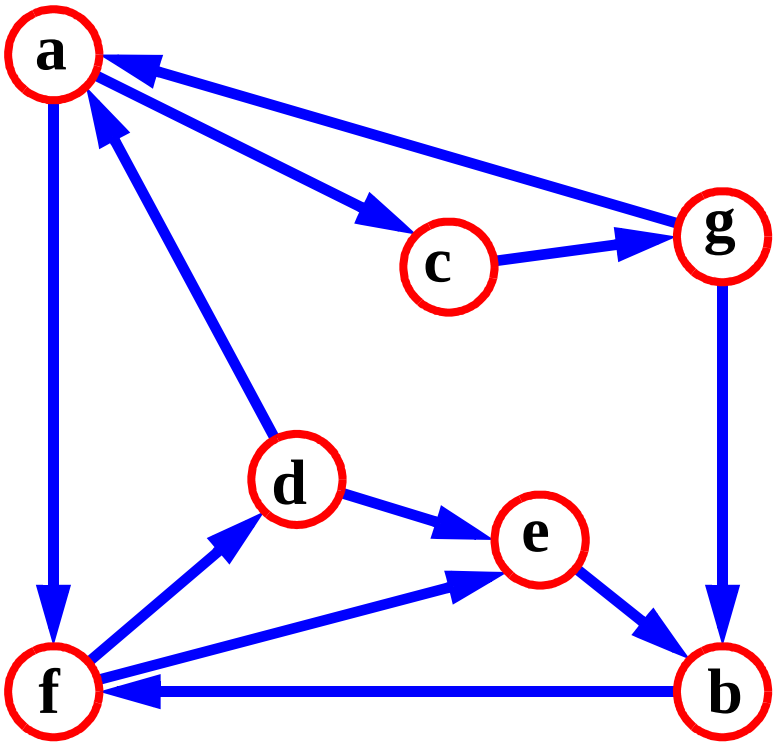
Accessibilité (*reachability*)

Arbre DFS avec racine v: sommets accessibles à partir de v via les chemins orientés

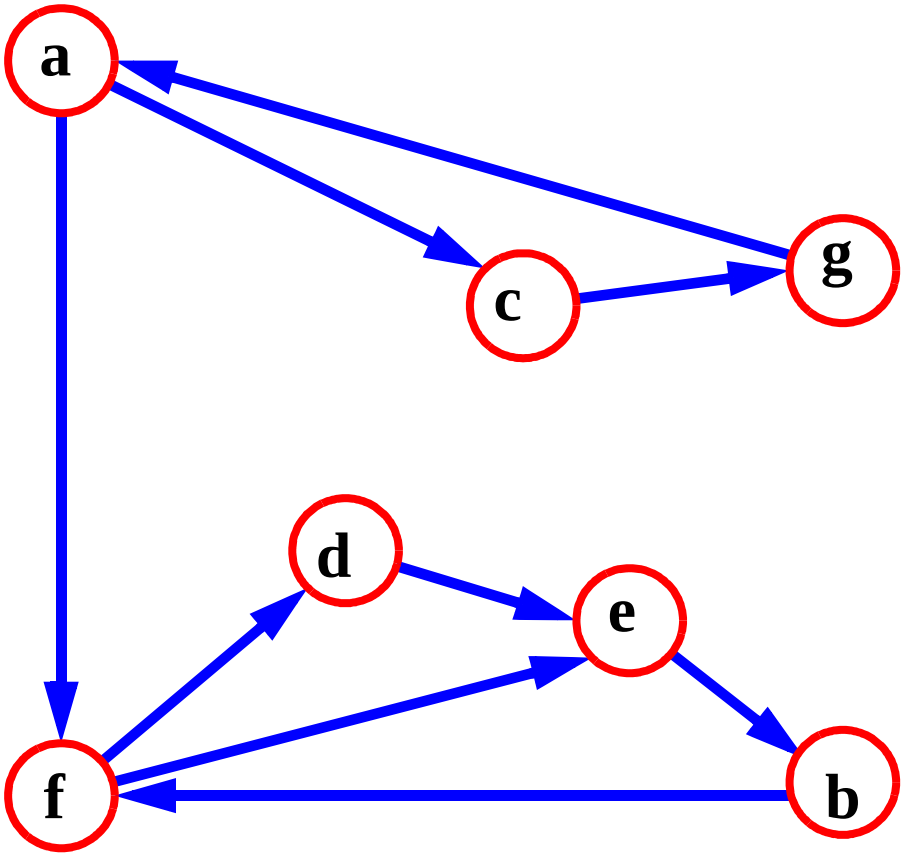


Digraphes fortement connexes

Chaque sommet peut atteindre tous les autres sommets



Composantes fortement connexes



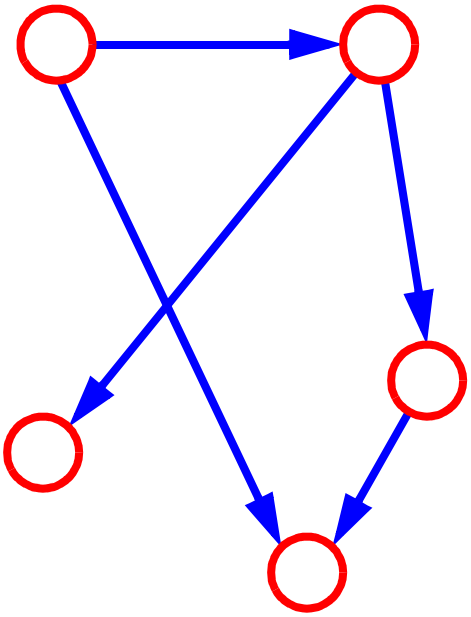
{ a , c , g }

{ f , d , e , b }

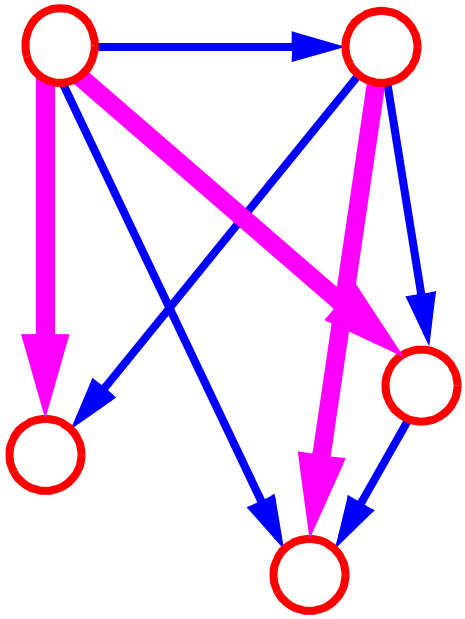
Fermeture transitive

Le digraphe G^* est obtenu de G en utilisant la règle:

Si il existe un chemin orienté dans **G** de a à b, alors ajouter l'arc (a,b) à $\mathbf{G}^*$



G



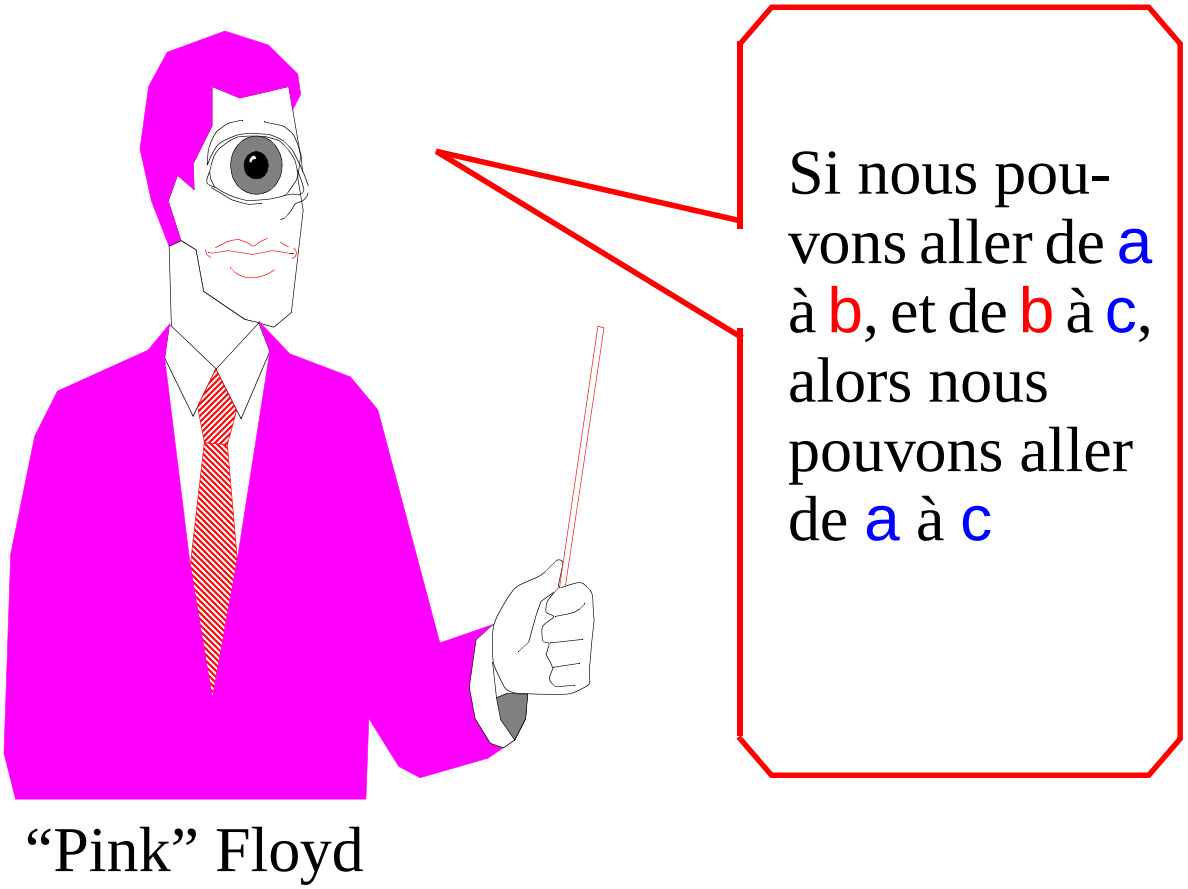
$\mathbf{G}^*$

Calculer la fermeture transitive

Nous pouvons utiliser DFS sur chaque sommet

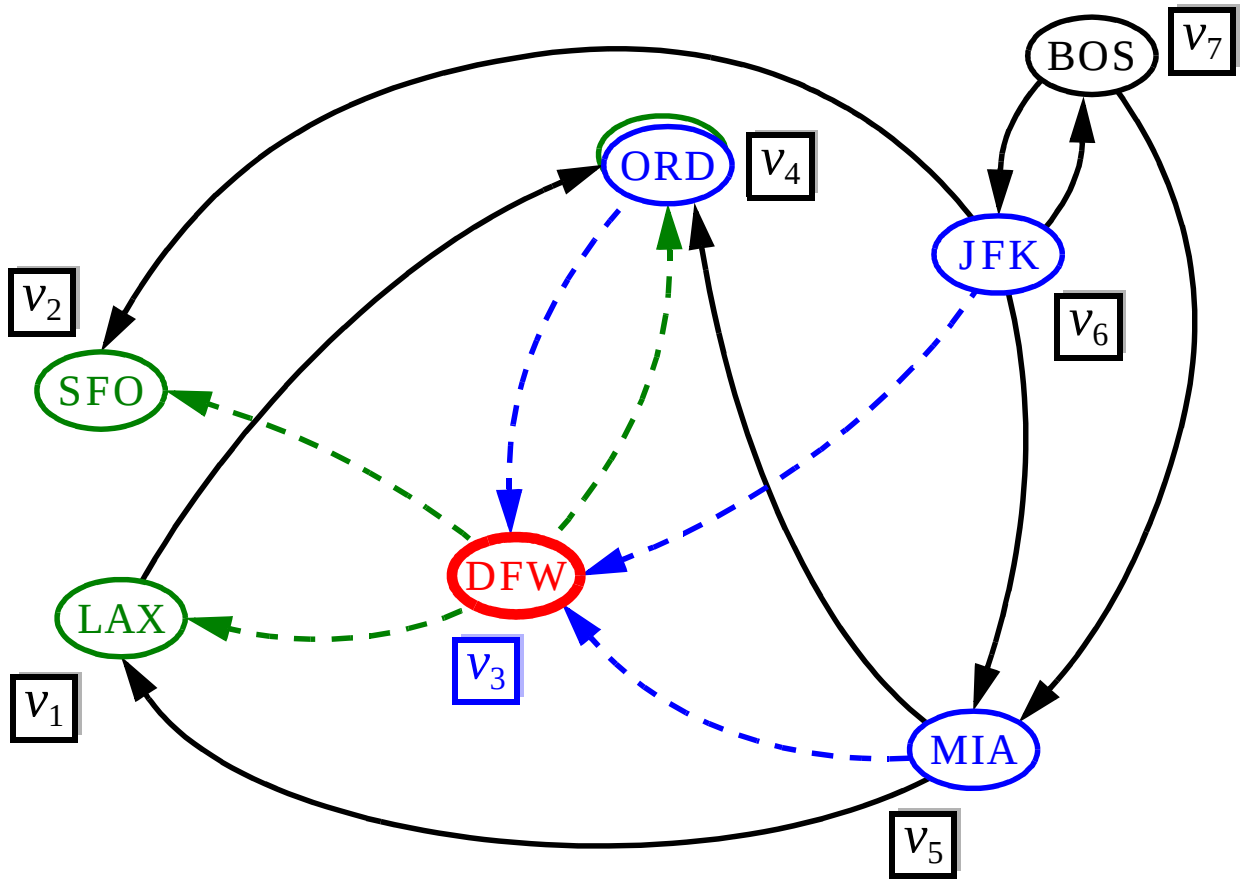
Temps: $O(n(n+m))$

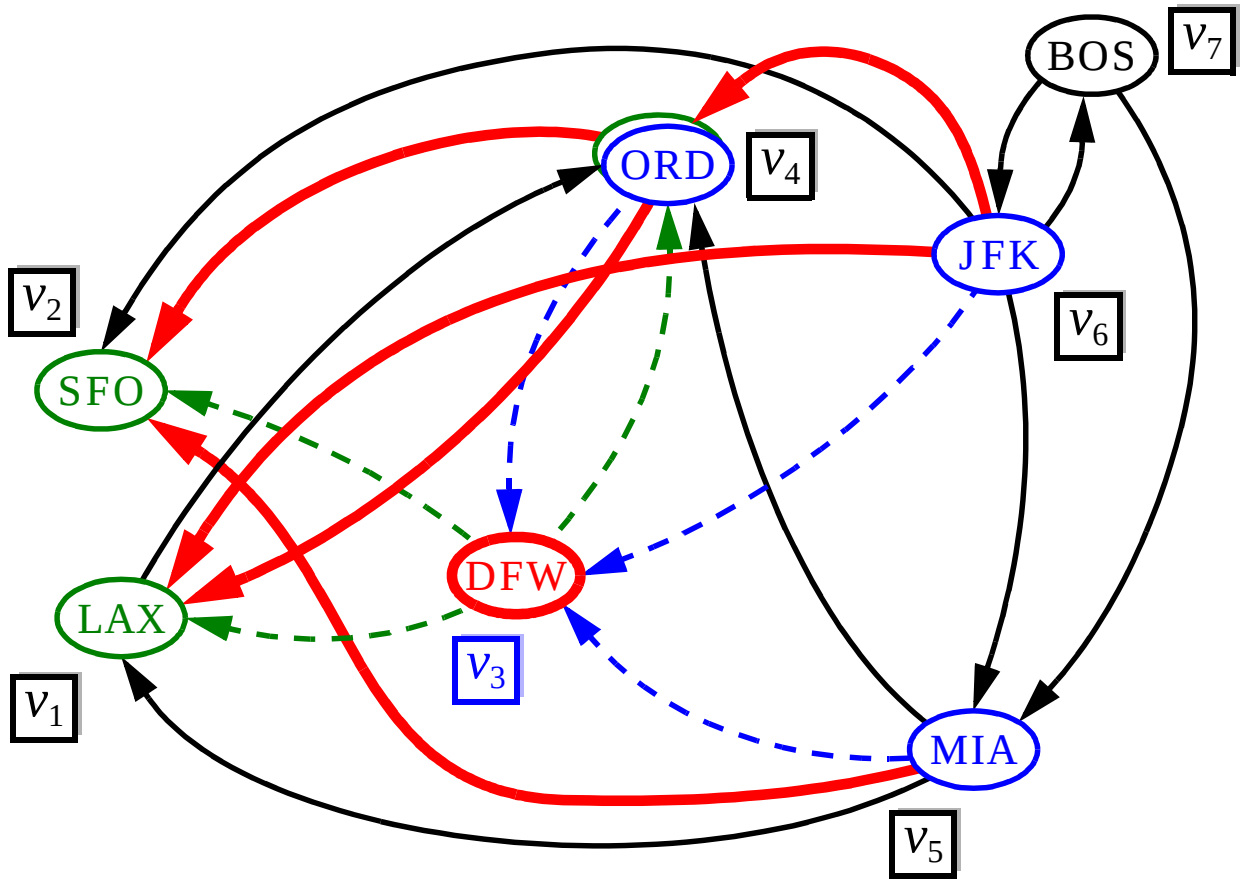
Ou encore... utiliser l'algorithme de Floyd-Warshall:



"Pink" Floyd

Exemple


BOS
v_7
ORD
v_4
JFK
v_2
v_6
SFO
DFW
LAX
v_3
v_1
MIA
v_5


BOS
v_7
ORD
v_4
JFK
v_2
v_6
SFO
DFW
LAX
v_3
v_1
MIA
v_5

Algorithme de Floyd-Warshall

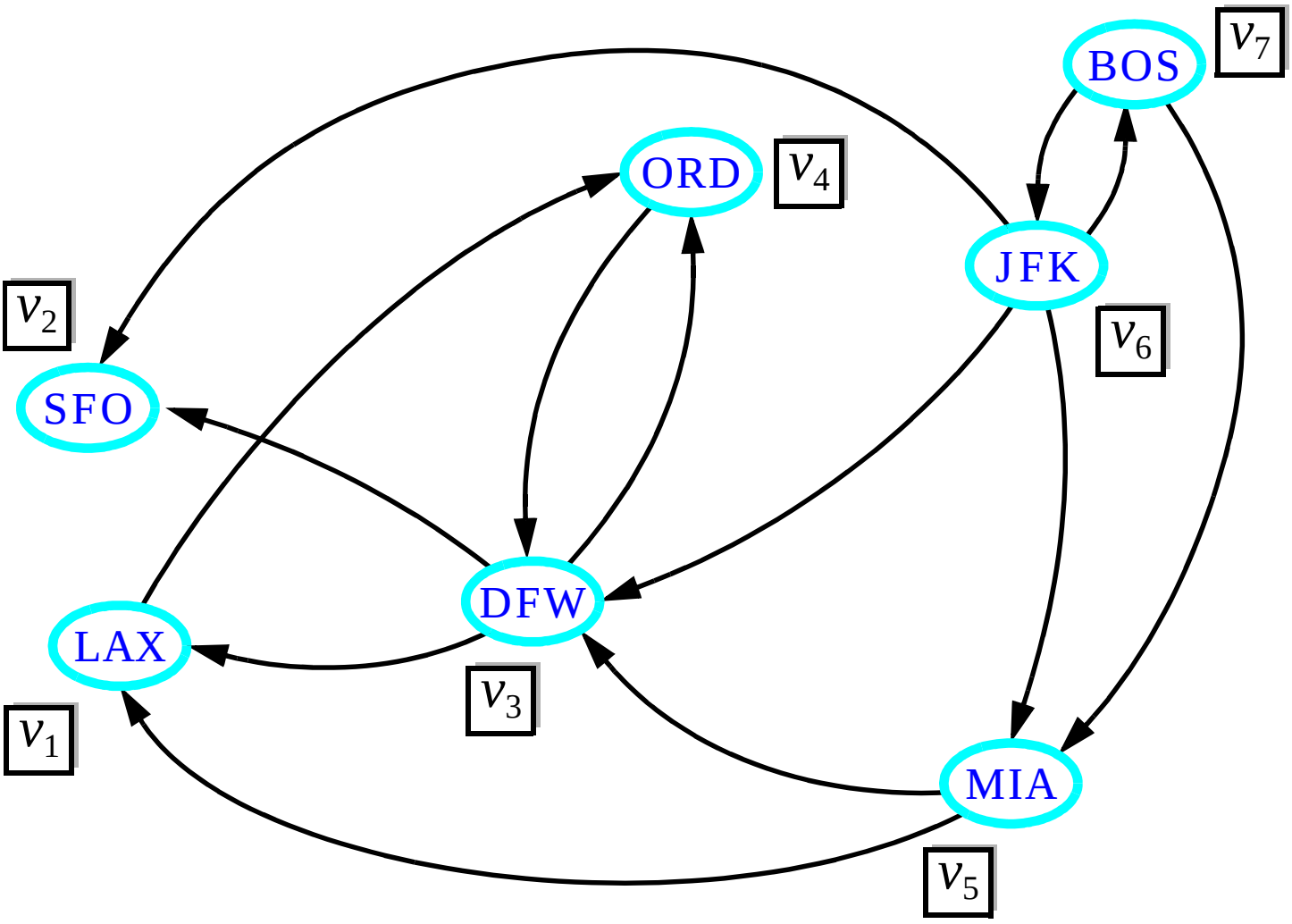
- Cet algorithme présuppose que les méthodes areAdjacent et insertDirectedEdge prennent un temps $O(1)$ (par exemple, structure en matrice d'adjacence)

Algorithme FloydWarshall(G)
  soit v_1 ... v_n un ordre arbitraire des sommets
  $G_0 = G$
  for k = 1 **to** n **do**
    // considérez tous les sommets de routage
    // possibles v_k
    $G_k = G_{k-1}$ // ce sont les seuls à conserver
    for each (i, j = 1, ..., n) (i != j) (i, j != k) **do**
      // pour chaque paire de sommets v_i et v_j
      if G_{k-1}.areAdjacent(v_i,v_k) **and**
        G_{k-1}.areAdjacent(v_k,v_j) **then**
          G_k.insertDirectedEdge(v_i,v_j,null)
    return G_n

- Le digraphe G_k est le sous-digraphe de la fermeture transitive de G induit par les chemins avec sommets intermédiaires dans l'ensemble { v_1, ..., v_k }
- Temps d'exécution: $O(n^3)$

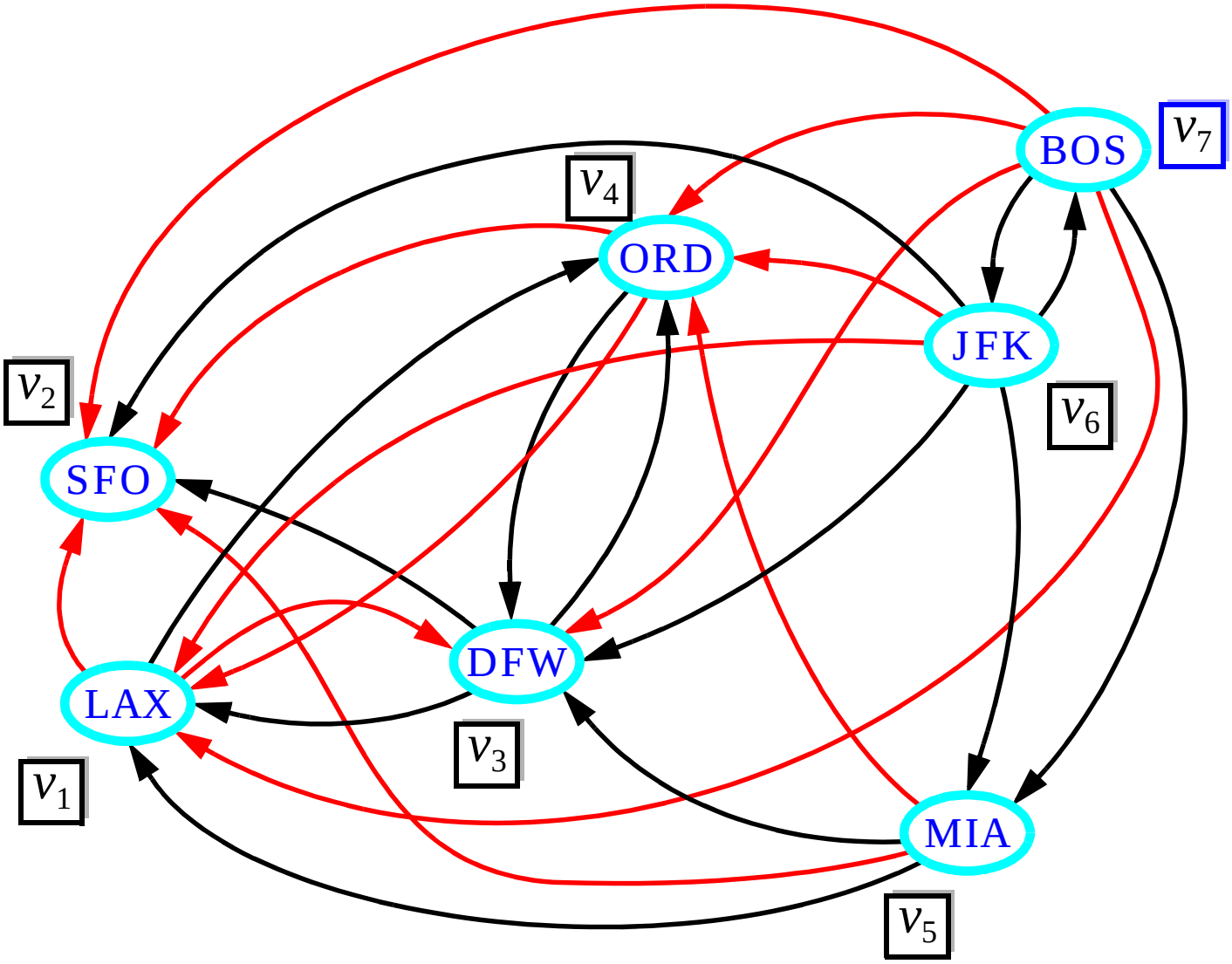
Exemple

- digraphe G



Exemple

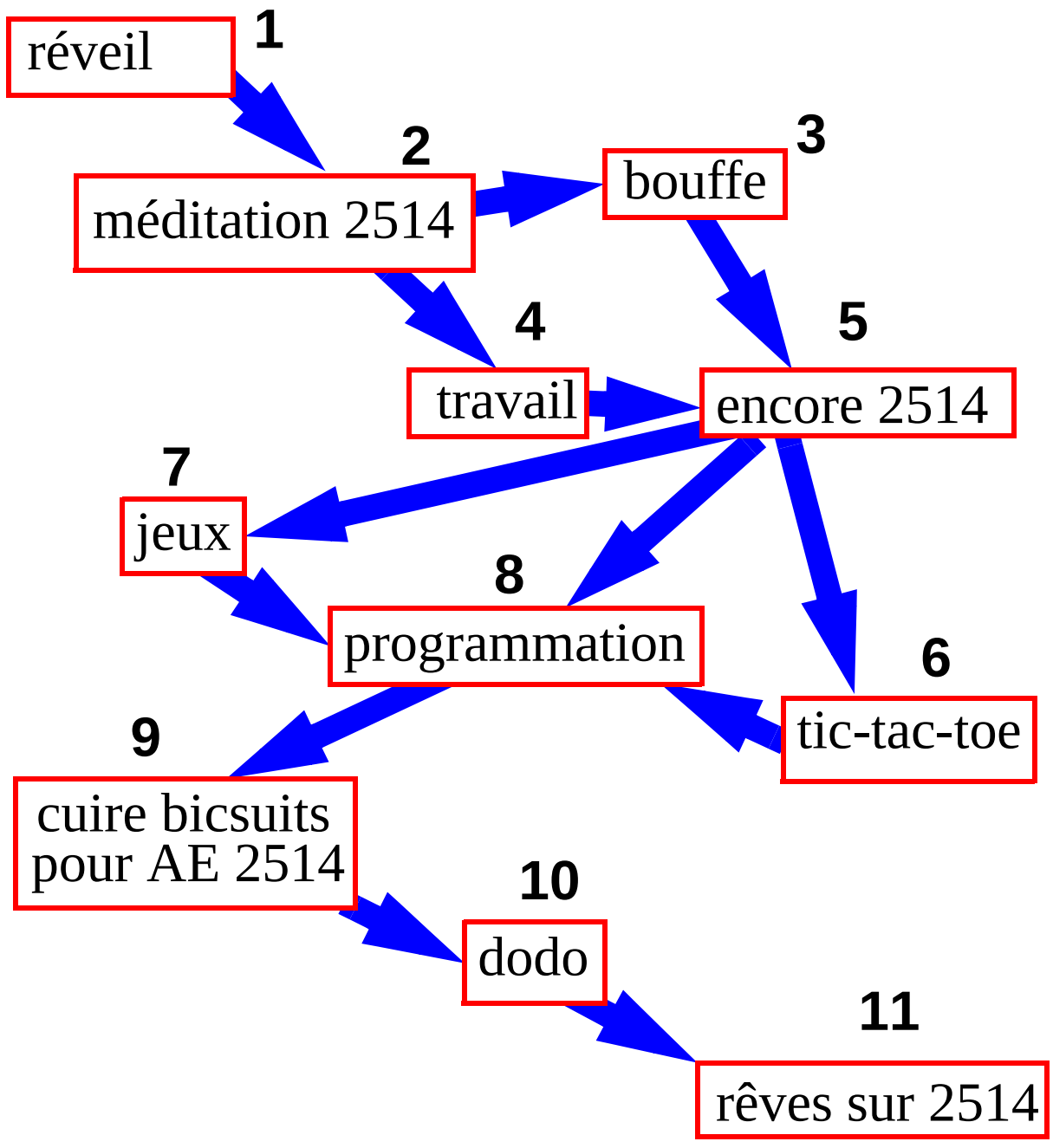
- digraphe G*



Tri topologique

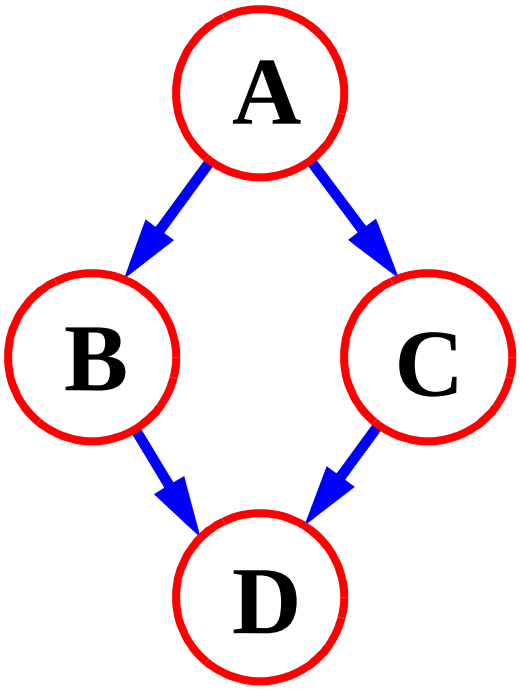
Pour chaque arc (u,v), le sommet u est visité avant le sommet v

une journée typique...



Tri topologique

Le résultat du tri topologique peut **ne pas** être unique



A B C D

ou

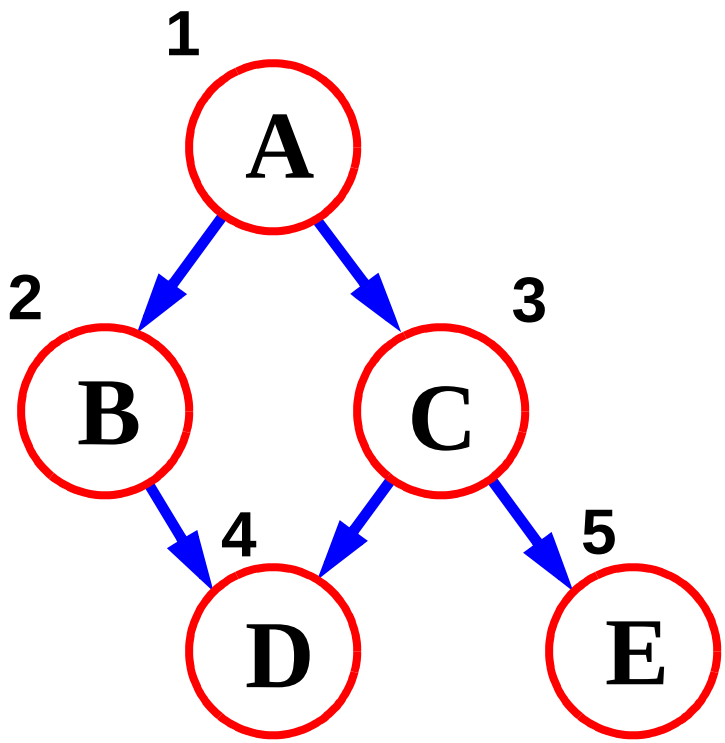
A C B D

– ***À vous*** *de décider*

Tri topologique

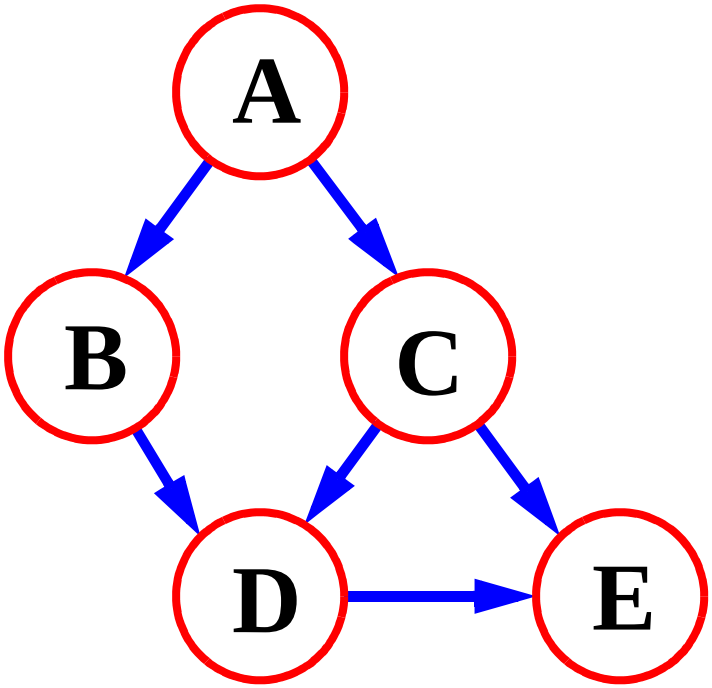
Les étiquettes augmentent le long d'un chemin orienté.

Un digraphe a un tri topologique *si et seulement si* il est acyclique (donc, un GOA)



Algorithme pour tri topologique

```
method TopologicalSort
   if il y a encore des sommets
         soit v une source;
               // un sommet sans arcs d'entrée
         étiqueter et supprimer v;
          TopologicalSort;
```



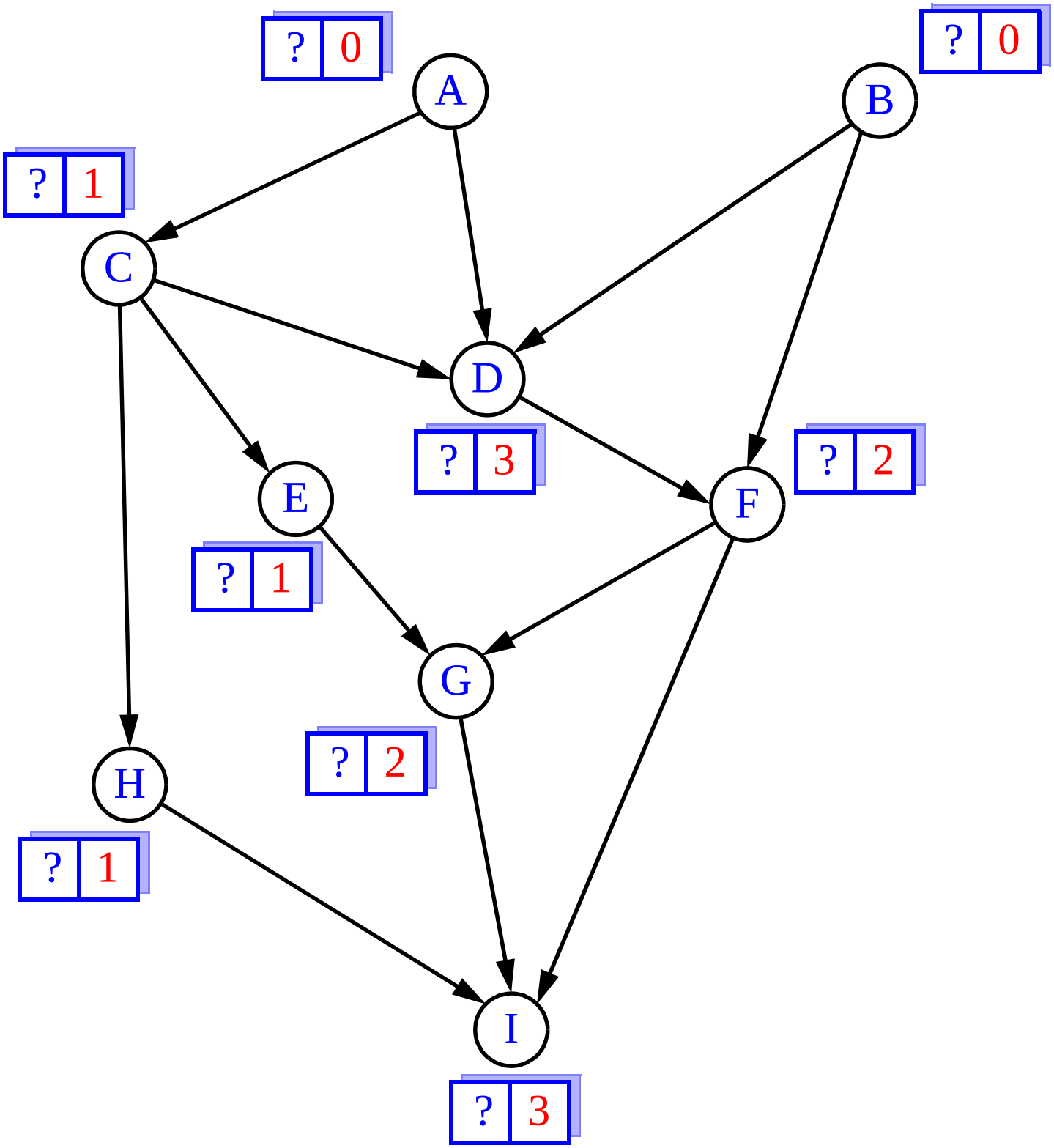
Algorithme (suite)

Simuler la suppression de sources en utilisant des compteurs de degré d'entrée

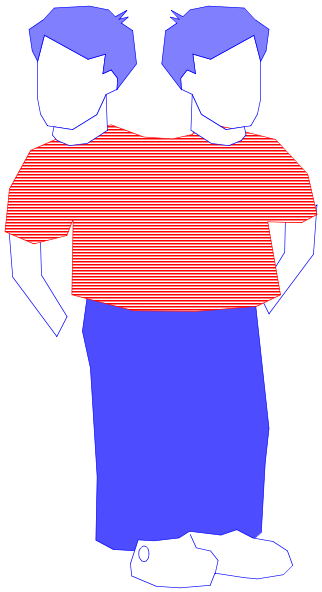
```
TopSort(Vertex v);
   étiqueter v;
   foreach arc(v,w)
            indeg(w) = indeg(w) – 1;
            if indeg(w) = 0
                 TopSort(w);
```

1. Calculer indeg(**v**) pour tous les sommets
2. Foreach sommet **v** do
   if **v** non étiqueté et indeg(**v**) = 0
   then TopSort(**v**)

Exemple



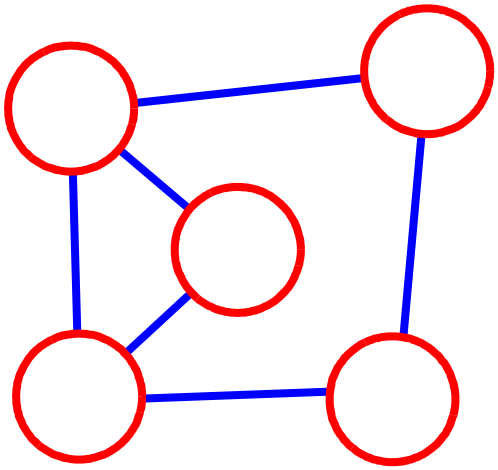
Connectivité et Biconnectivité



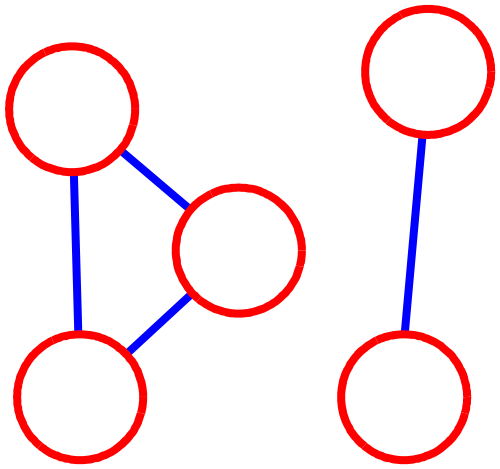
- composantes connexes
- sommets de séparation (*cutvertices*)
- composantes biconnexes

Composantes connexes

Graphe connexe: chaque paire de sommets reliée par un chemin.



connexe



non-connexe

Composante connexe:
sous-graphe connexe maximal d'un graphe

Relations d'équivalence

Une ***relation*** sur un ensemble S est un ensemble ordonné R composé de paires d'éléments de S et défini par une propriété quelconque.

Exemple:

- $S = \{1,2,3,4\}$
- $R = \{(i,j) \in S \times S \text{ tel que } i < j\}$
  $= \{(1,2),(1,3),(1,4),(2,3),(2,4),\{3,4)\}$

Une ***relation d'équivalence*** satisfait les propriétés suivantes:

- $(x,x) \in R, \forall\ x \in S$ (***réflexive***)
- $(x,y) \in R \Rightarrow (y,x) \in R$ (***symétrique***)
- $(x,y), (y,z) \in R \Rightarrow (x,z) \in R$ (***transitive***)

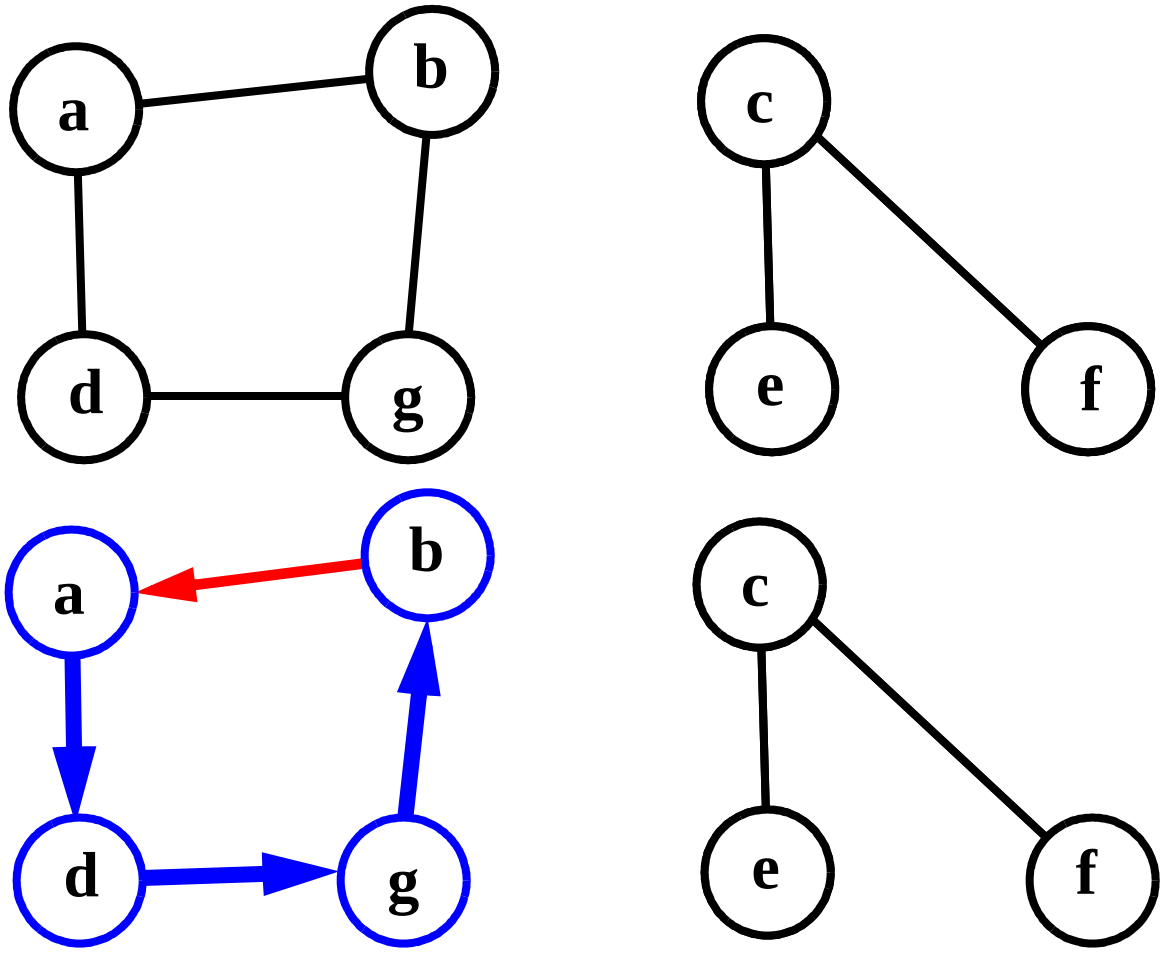
La relation C sur l'ensemble des sommets d'un graphe:

- $(\boldsymbol{u},\boldsymbol{v}) \in C \Leftrightarrow$ ***u*** et ***v*** sont dans la même composante connexe

est une relation d'équivalence.

DFS sur un graphe non-connexe

- DFS(*v*) visite tous les sommets et les arcs dans la composante connexe de *v*.



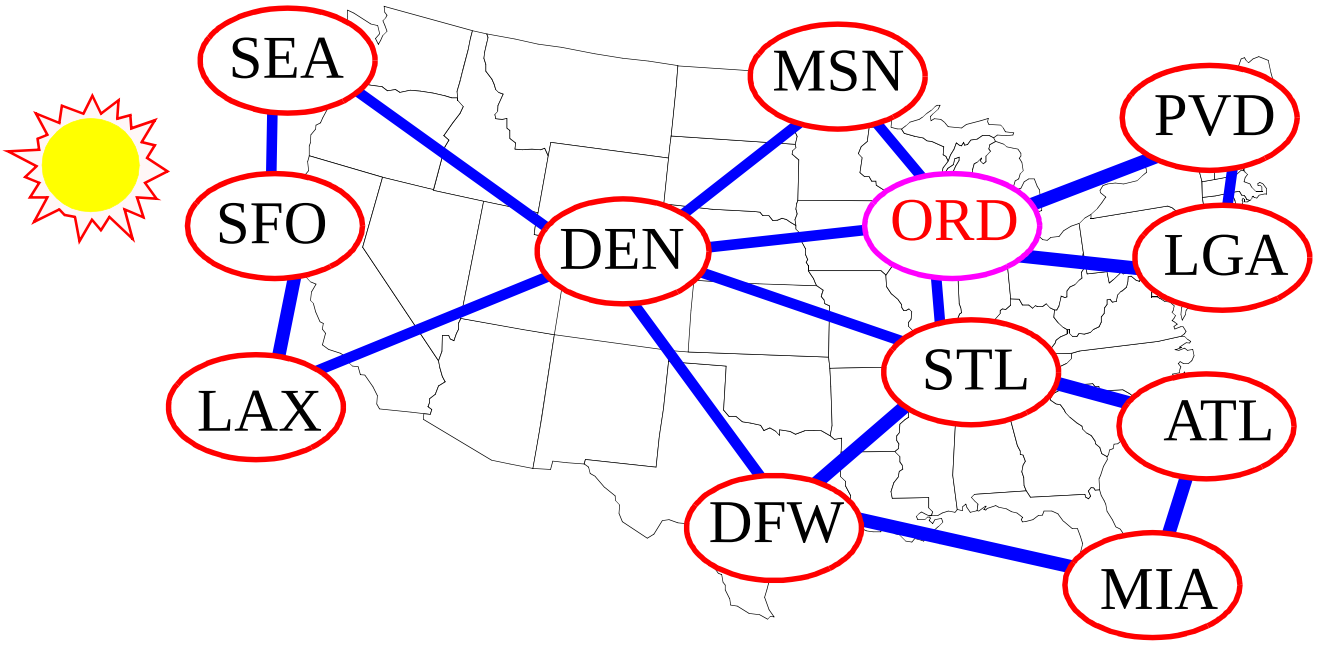
- Pour déterminer les composantes connexes:

```
k = 0 // compteur composante
foreach (vertex v)
    if unvisited(v)
    // ajouter à la composante k
    // les sommets atteints par v
    DFS(v, k++)
```

Sommets de séparation (*Cutvertices*)

Sommet de séparation (*cutvertex*):
son retrait rend le graphe non-connexe

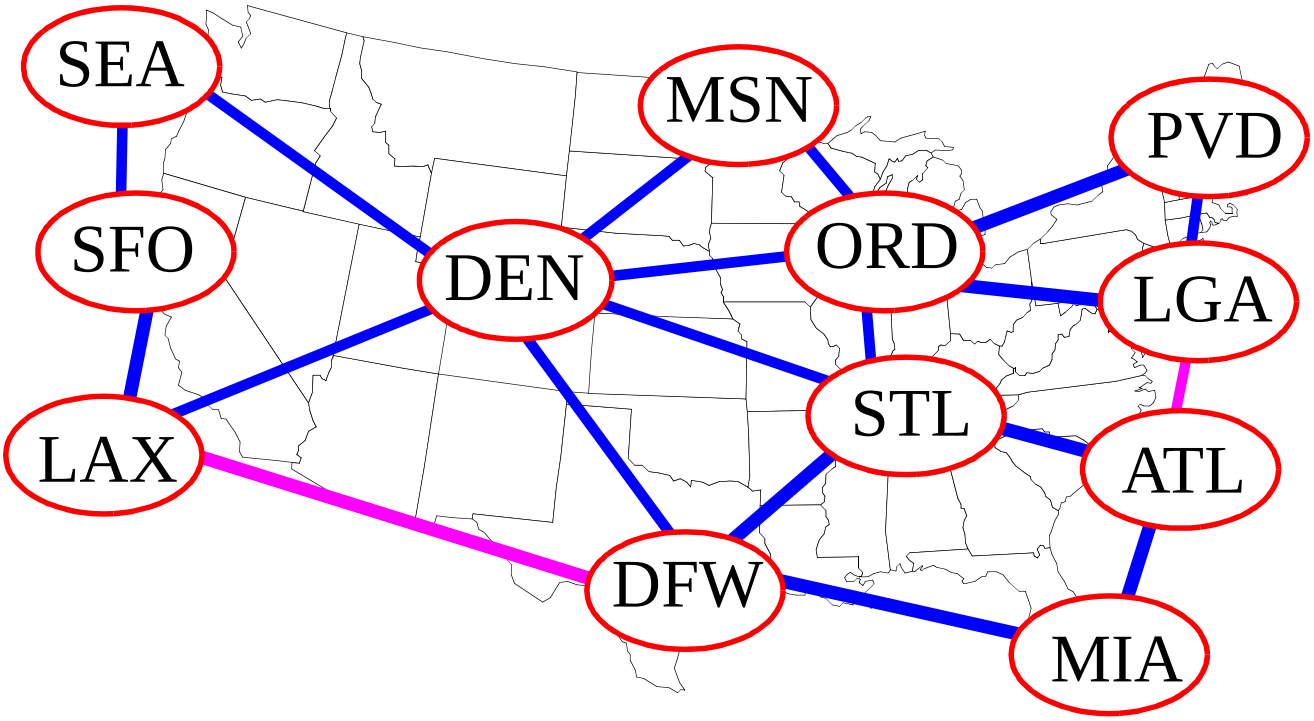
Si l'aéroport de Chicago est fermé, alors il n'y a aucun moyen d'aller dans les villes de la côte ouest à partir de Providence (PVD). Même chose pour l'aéroport de Denver.



- Sommets de séparation: ORD. DEN

Biconnectivité

Graphe biconnexe: n'a pas de sommet de séparation.

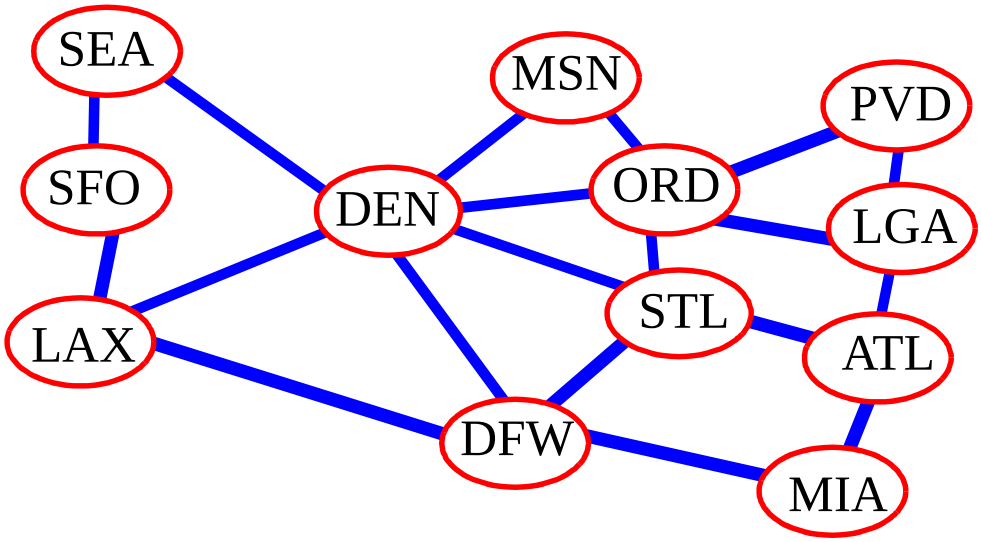


Nouveaux vols:

LGA-ATL et DFW-LAX

rendent le graphe biconnexe.

Propriétés des graphes biconnexes

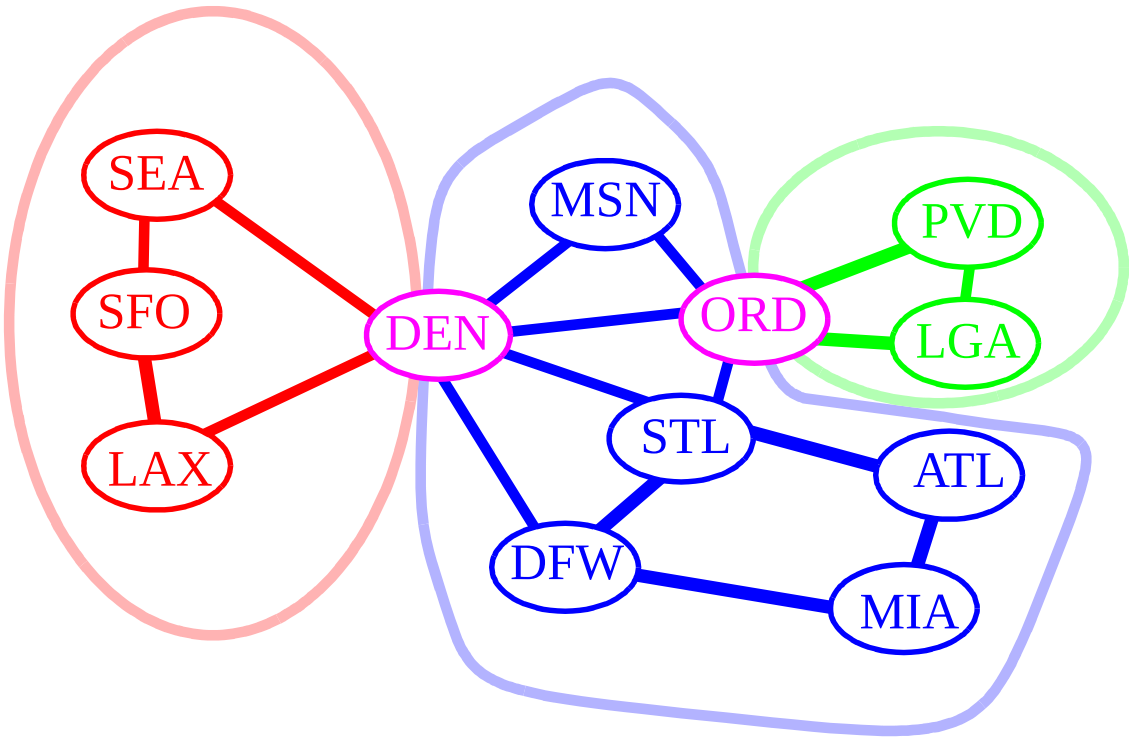


- Il y a ***deux chemins disjoint*** entre n'importe quelle paire de sommets.
- Il y a un ***cycle*** au travers de n'importe quelle paire de sommets.

Par convention, deux nœuds reliés par un arc forment un graphe biconnexe, mais ceci ne satisfait pas les propriétés mentionnées ci-haut.

Composantes biconnexes

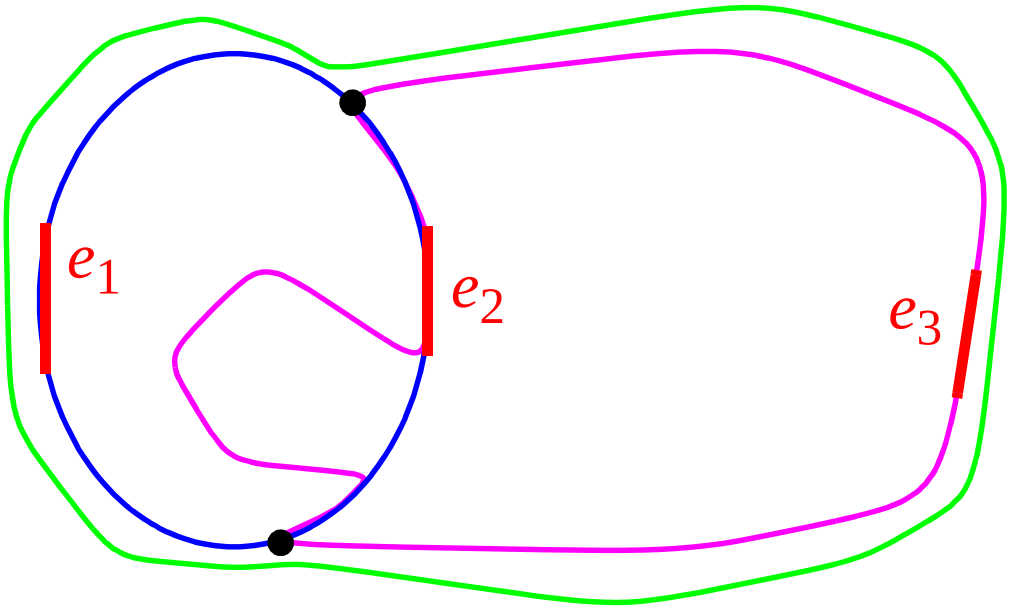
- Composante biconnexe (bloc): sous-graphe biconnexe maximal



- Les composantes biconnexes d'un graphe ne partagent pas d'arc, mais elles partagent des sommets de séparation.

Caractérisation des composantes biconnexes

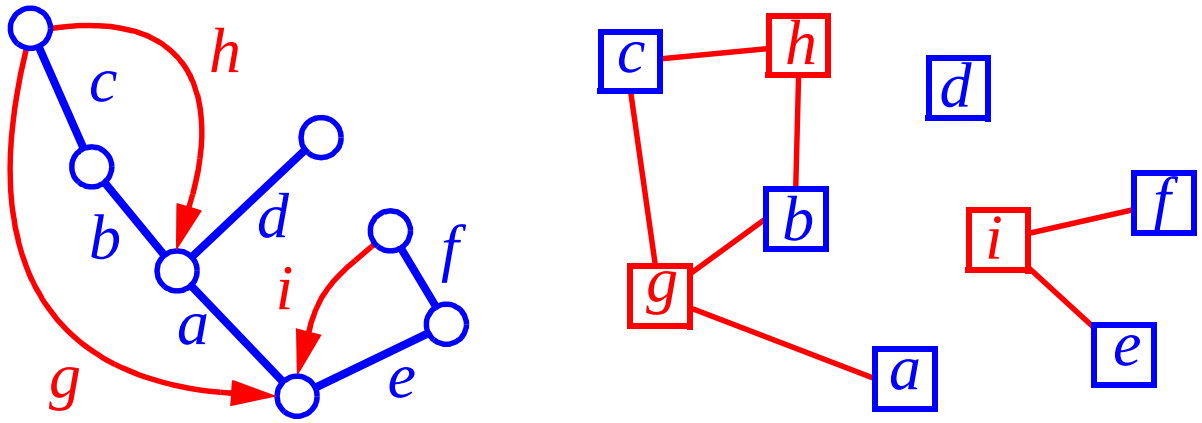
- ***Relation d'équivalence*** R sur les ***arcs*** de G: (e', e") ∈ R si il y a un cycle contenant à la fois e' et e"
- Preuve de la ***propriété transitive***



- Nous divisons les arcs de G en ***classes d'équivalence*** par rapport à R.
- Chaque classe d'équivalence correspond à:
  - une composante biconnexe de G
  - une composante connexe d'un graphe H dont les sommets sont les ***arcs*** de G et dont les arcs sont les ***paires*** dans la relation R.

DFS et composantes biconnexes

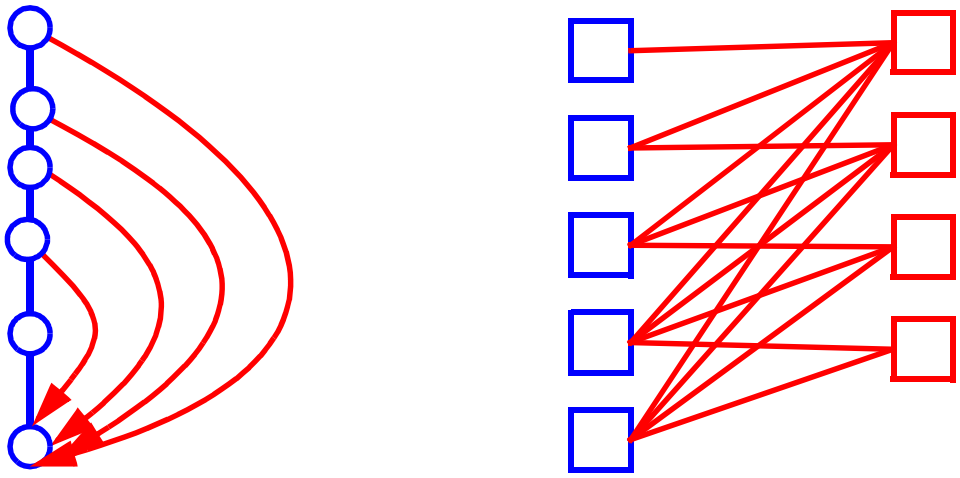
- Le graphe H a $O(m^2)$ arcs dans le pire des cas.
- Au lieu de calculer le graphe H tout entier, nous utilisons un graphe **mandataire** (*proxy*) K, qui est plus petit.
- Débutons avec un graphe K vide dont les sommets sont les arcs de G.
- Étant donné une DFS sur G, considérez les $(m - n + 1)$ cycles de G induits par les arcs.
- Pour chacun de ces cycles C = (e_0, e_1, ... , e_p) ajoutez les arcs (e_0, e_1) ... (e_0, e_p) à K.



- Les composantes connexes de K sont les mêmes que celles de H!

Un algorithme à temps linéaire

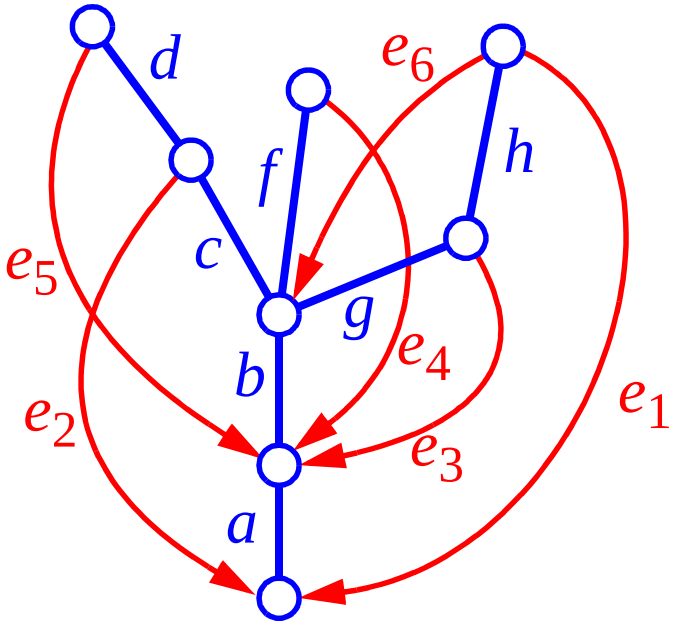
- La taille de K est $O(mn)$ dans le pire des cas.



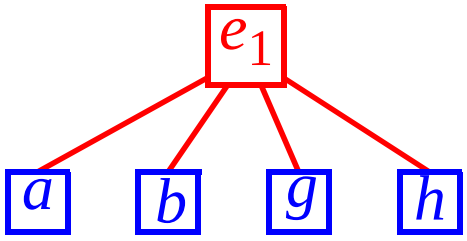
- Nous pouvons encore réduire la taille du graphe mandataire à $O(m)$
- Traitez les arcs selon une ***visite pré-ordre*** de leur sommet de destination dans l'arbre DFS
- Annotez les arcs de découverte formant les cycles
- Arrêter d'ajouter des arcs au graphe mandataire après avoir rencontré le premier arc annoté
- Le graphe mandataire résultant est une forêt!
- Cet algorithme requiert un temps $O(n+m)$.

Exemple

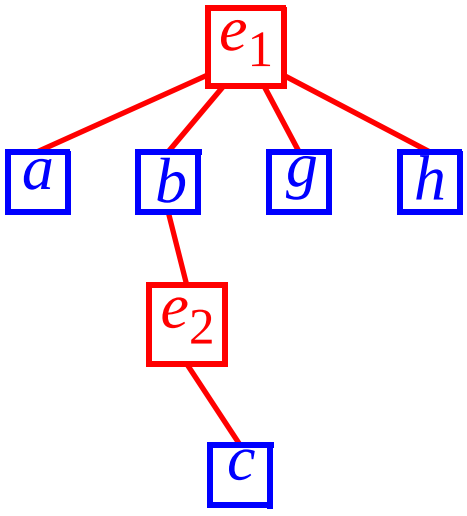
- Arcs arrières étiquetés selon la visite pré-ordre de leur sommet de destination dans l'arbre DFS



- Traitement de e_1

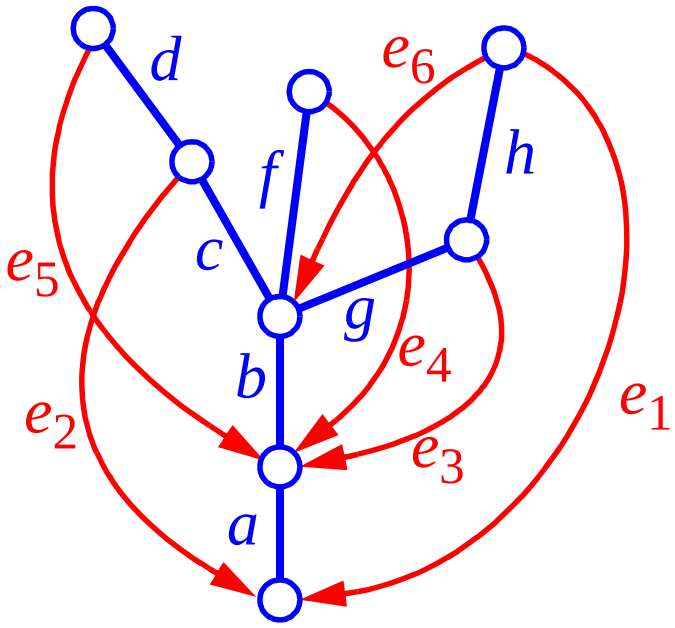


- Traitement de e_2

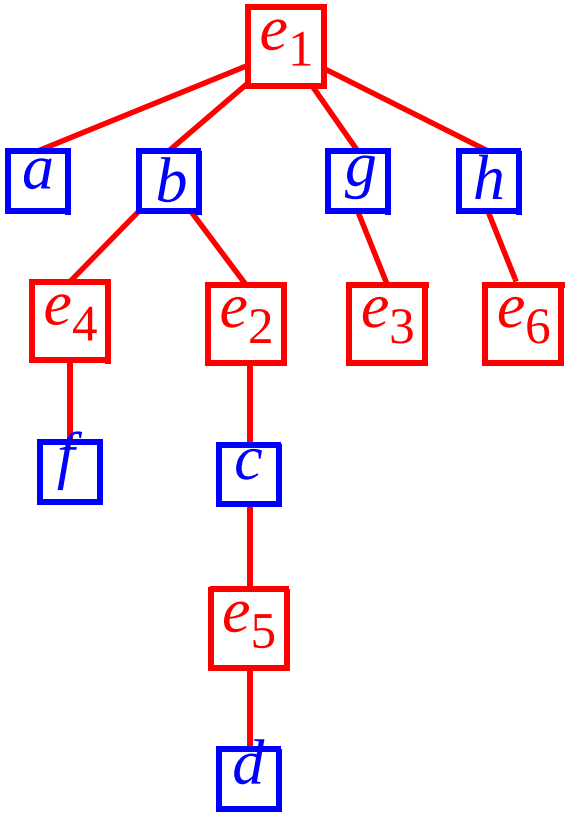


Exemple (suite)

- arbre DFS

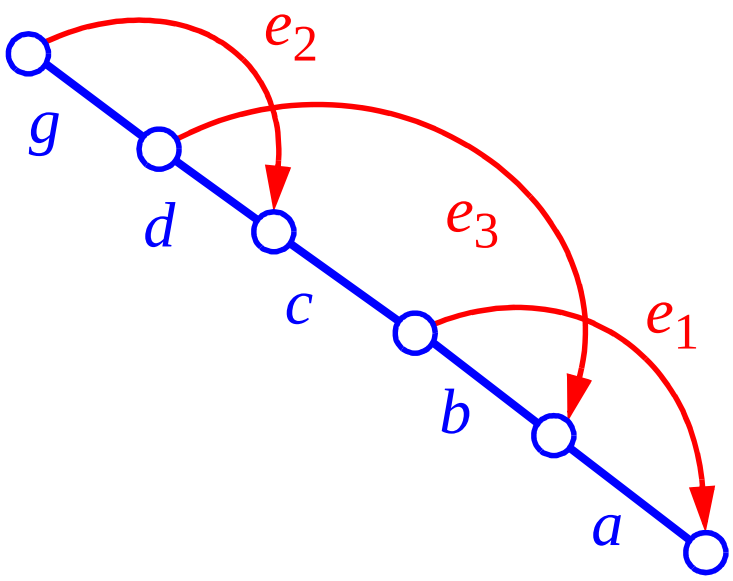


- graphe mandataire final (un arbre puisque le graphe est biconnexe)

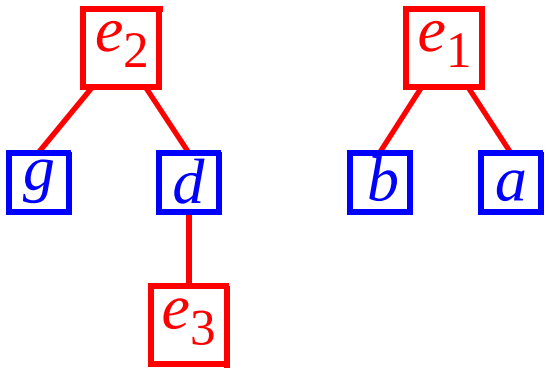


Pourquoi pré-ordre?

- L'ordre dans lequel les arcs arrières sont traités est essentiel pour la rectitude de l'algorithme
- L'utilisation d'un ordre différent...



- ... mène à un graphe qui contient des informations incorrectes.



Essayez l'algorithme sur ce graphe!