U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

NOTIONS DE BASE SUR LE GRAPHE

I- NOTION DE GRAPHE

Beaucoup de données traitées dans la vie courantes sont des **structures relationnelles**.

1- Pourquoi les graphes ?

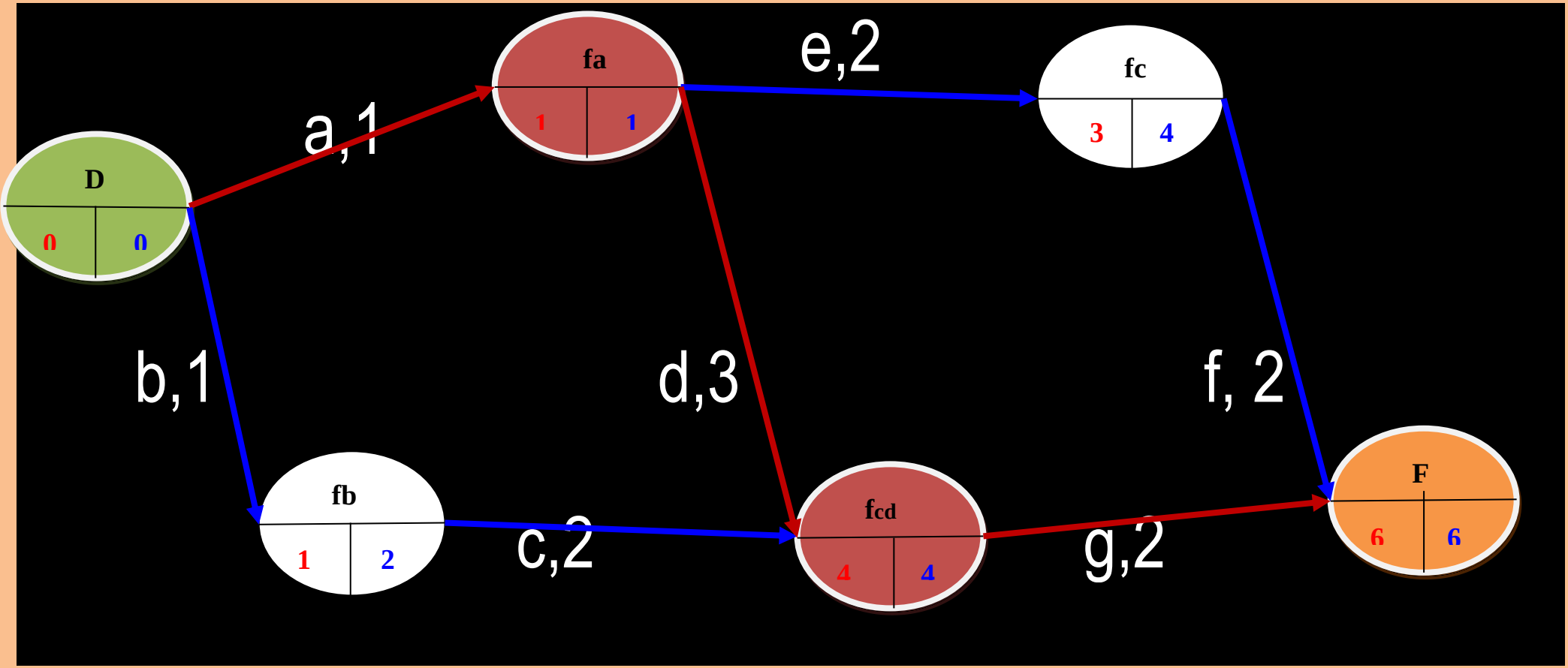
Dans un programme, les structures relationnelles sont modélisées à l'aide des **graphes**.

C’est le cas notamment des:

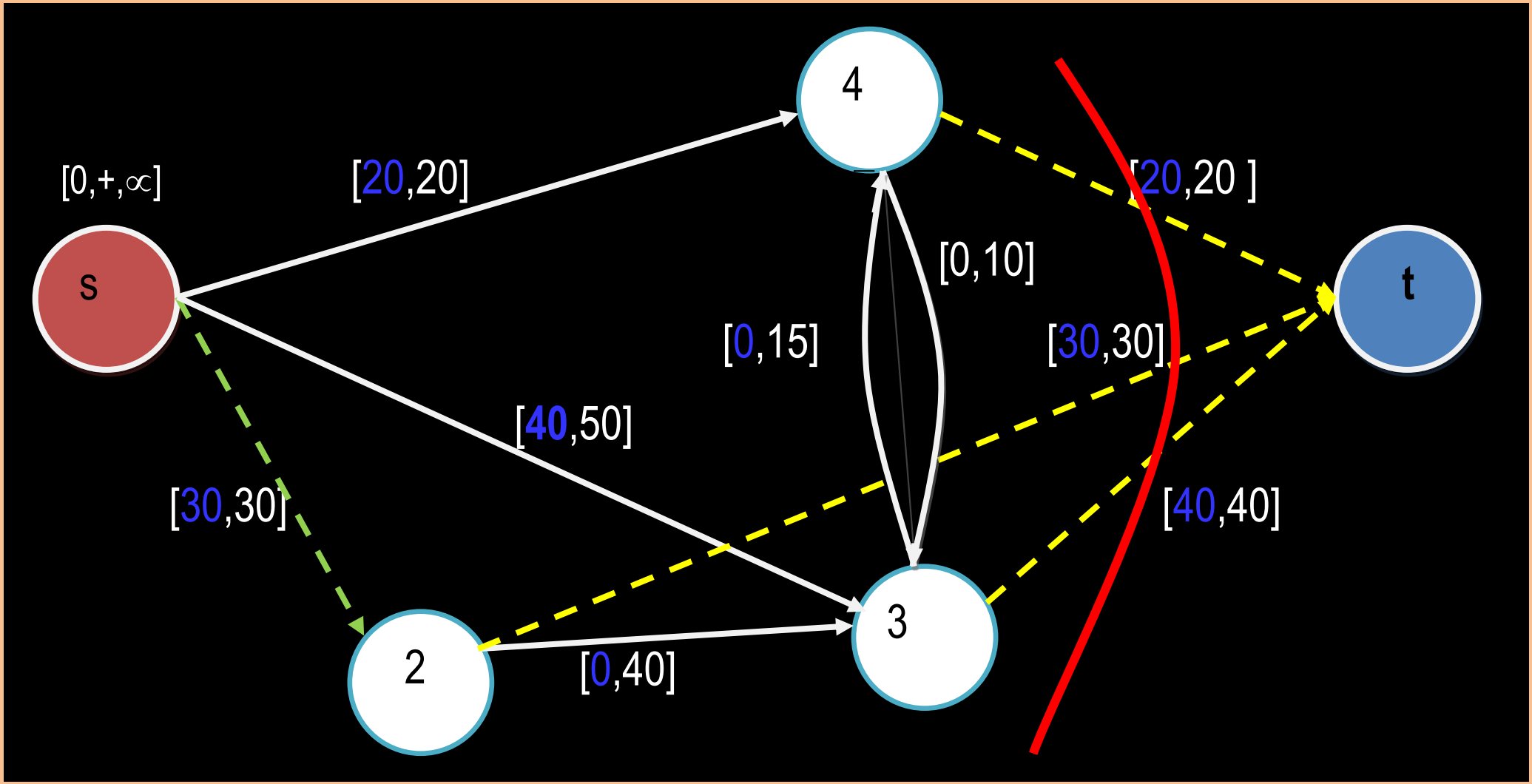
-les **réseaux de communication**,
-l’**ordonnancement des tâches** d’un robot,
- l’analyse et le **contrôle d’exécution** d’un programme,
- les **contrôleurs aériens**,
-...

qui sont des **structures relationnelles.**

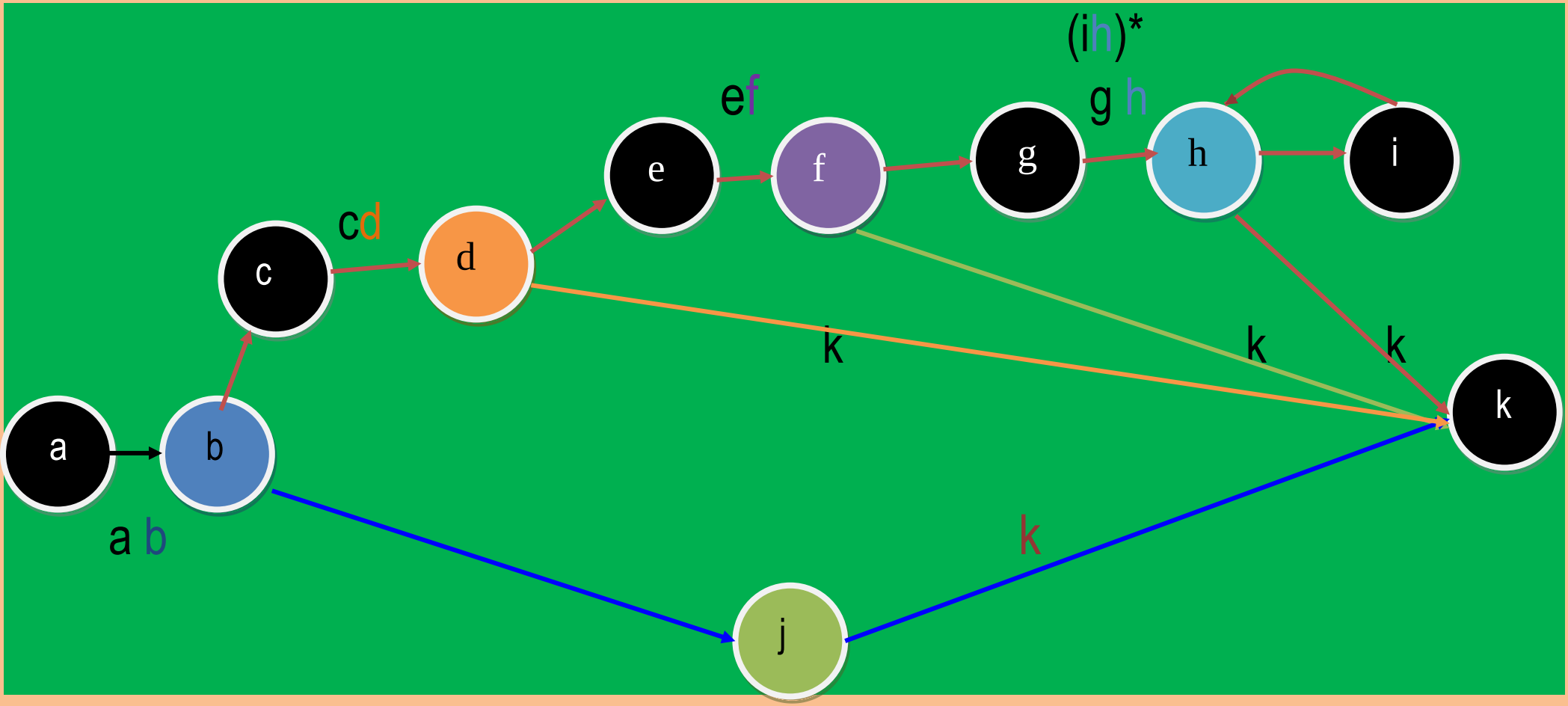
2- Quelques exemples d'application



Ordonnancement des tâches d'un robot

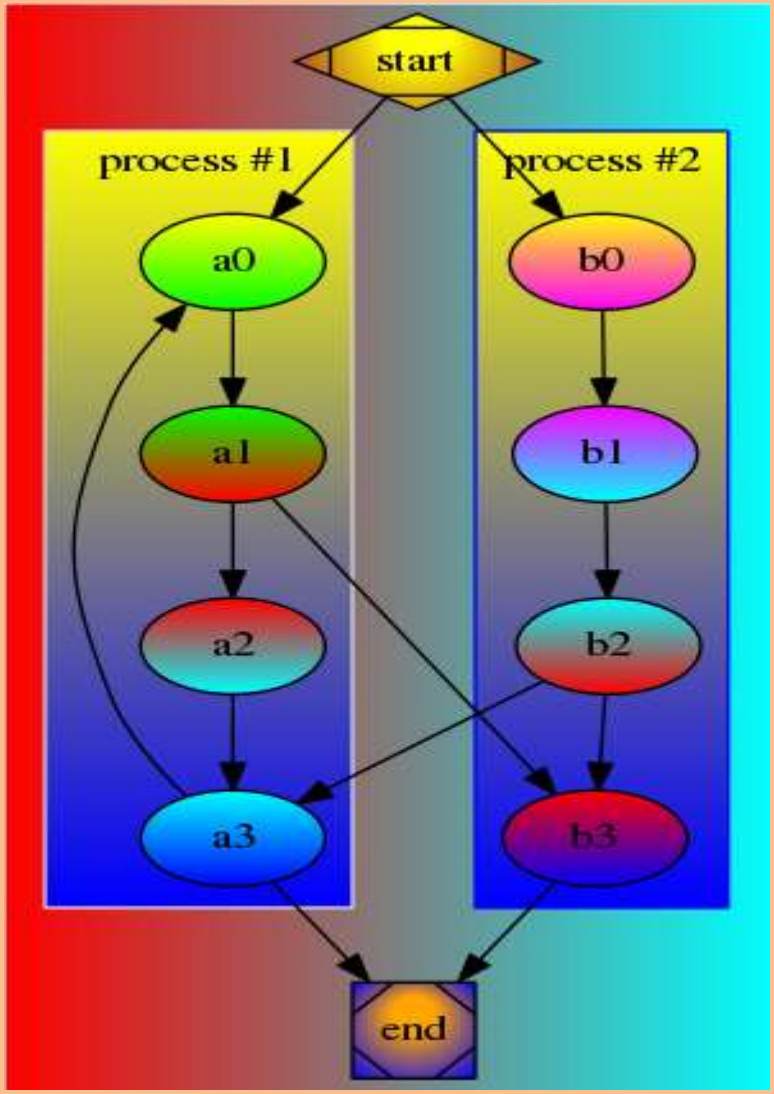


Flot sur un réseau de transport

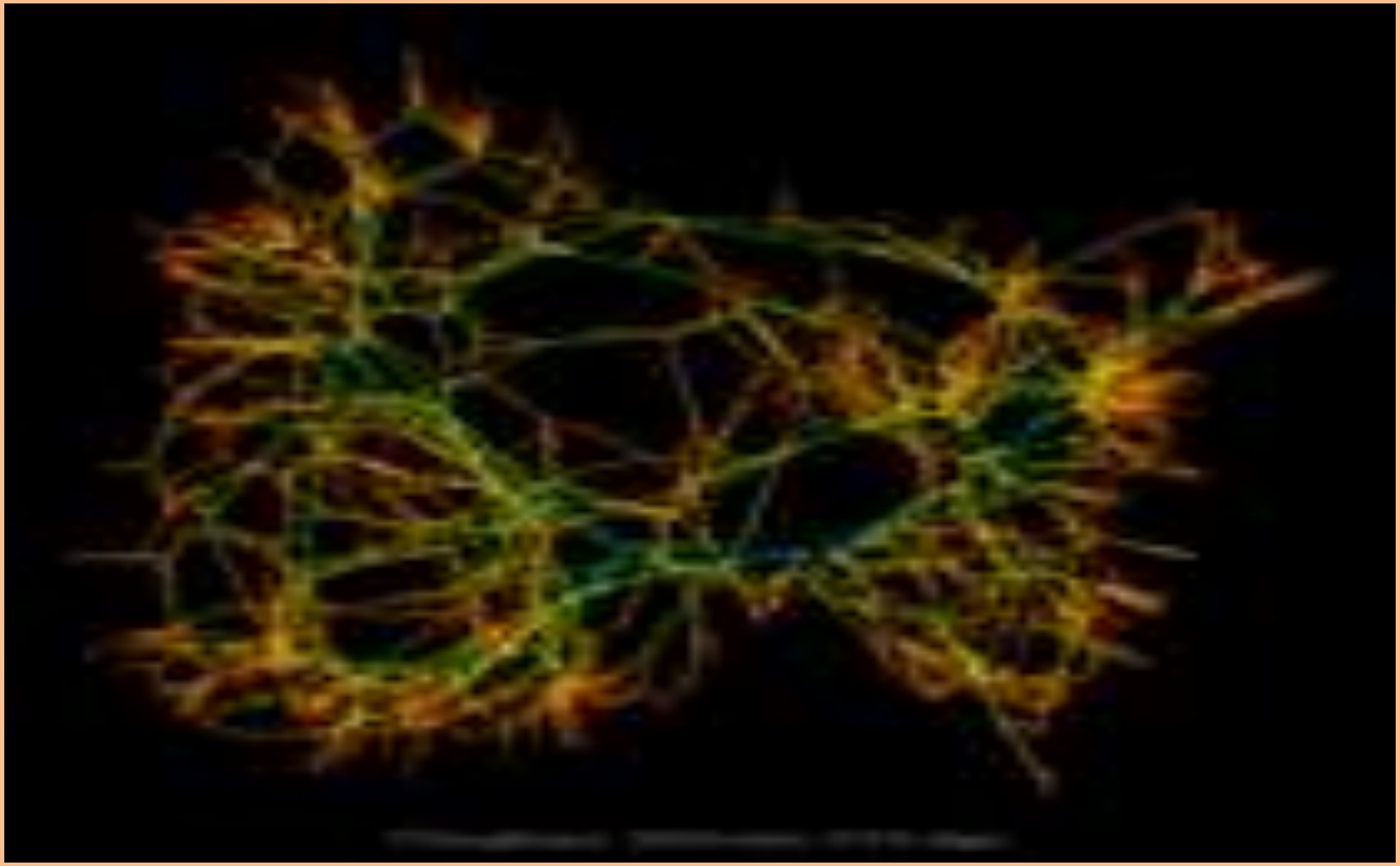


Graphe de contrôle d'un programme

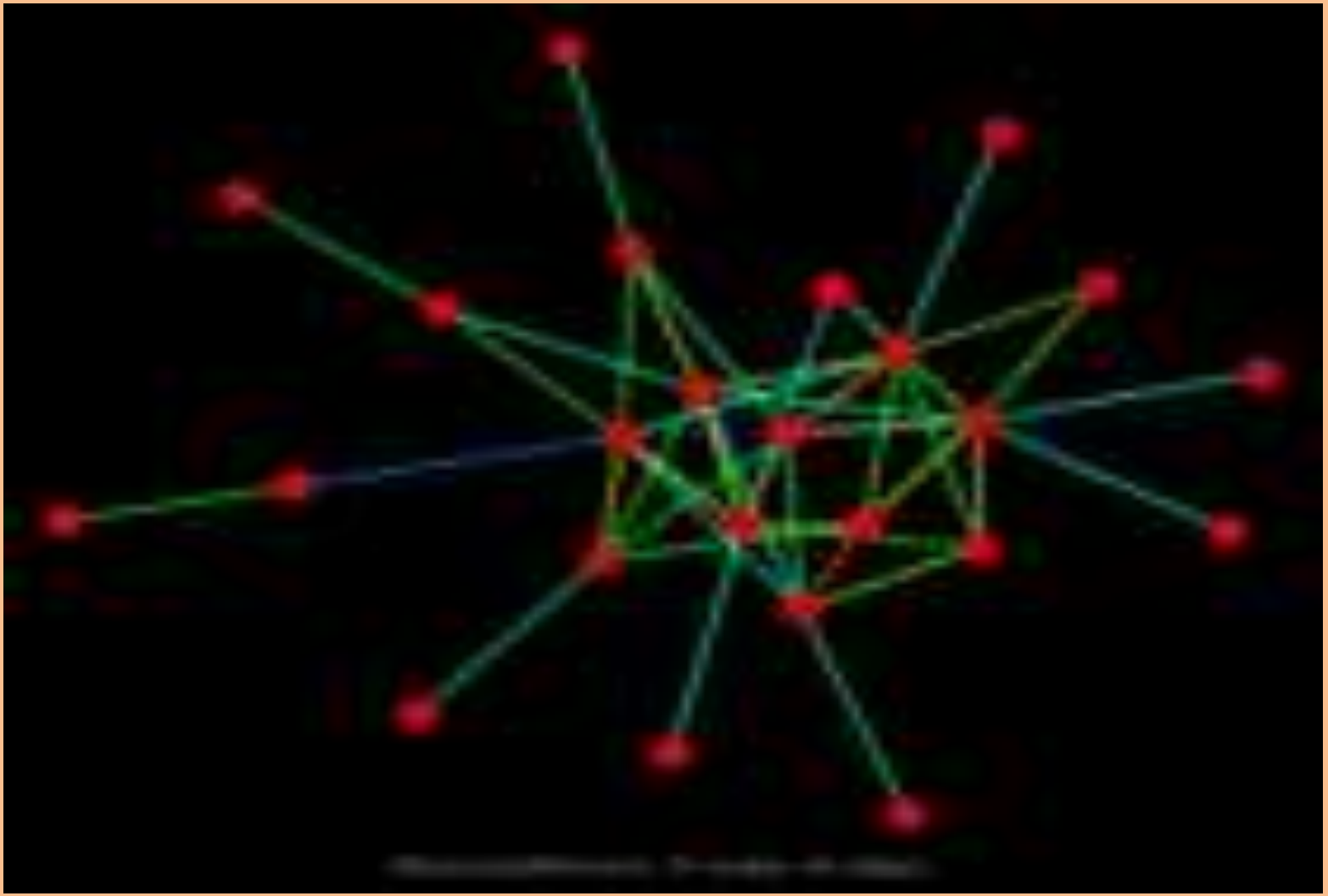
```
Unknown()                                                          
      begin                                                        
      read(b,c,x)                                                  a
      if b < c     then begin                                      b
                        d:=2*b ; f:=3*c                            c
                        if x>= 0    then begin                     d
                                        y:= x; e:= c               e
                                        if(y=0)     then begin     f
                                                    a:=f-e         g
                                                    while d<a  begin   h
                                                               d:=d+2  i
                                                               end     k
                                                    end            k
                                    end                            k
                  else begin                                       
                        b:=b-1                                     j
                        end                                        k
      end                                                          k
      end
```



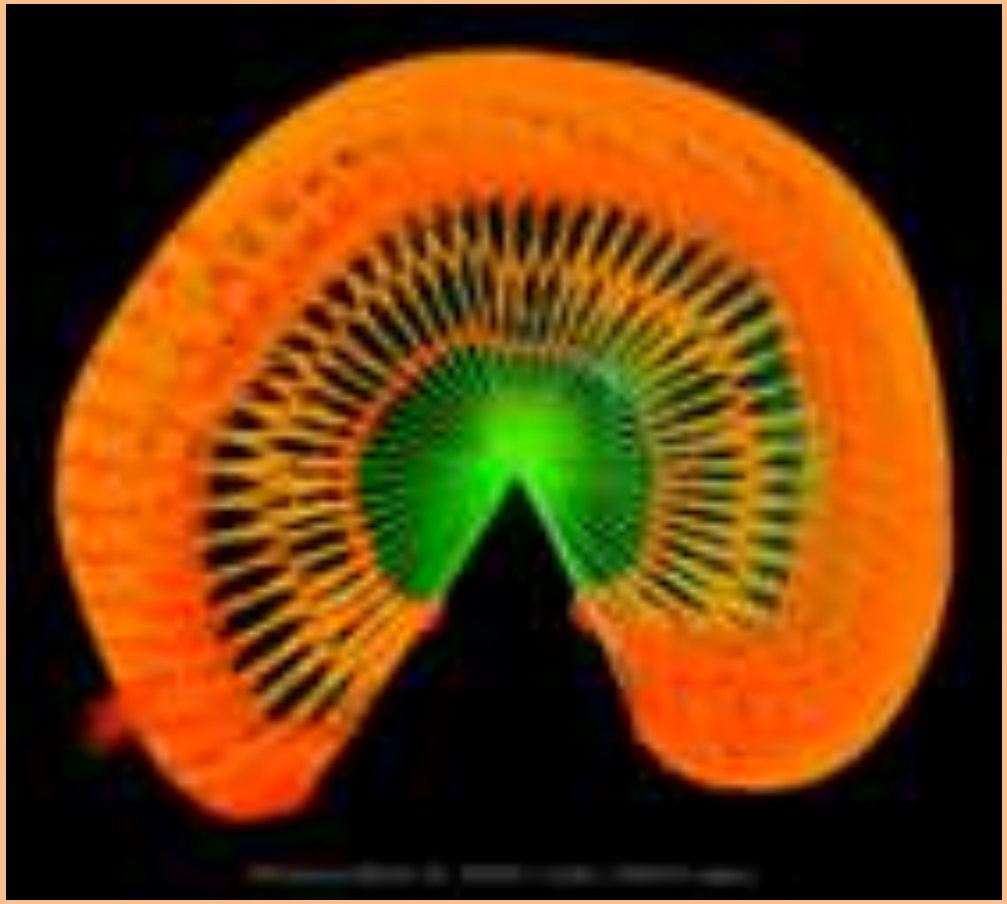
Graphe de processus concurrents



Graphe réseau de neurones



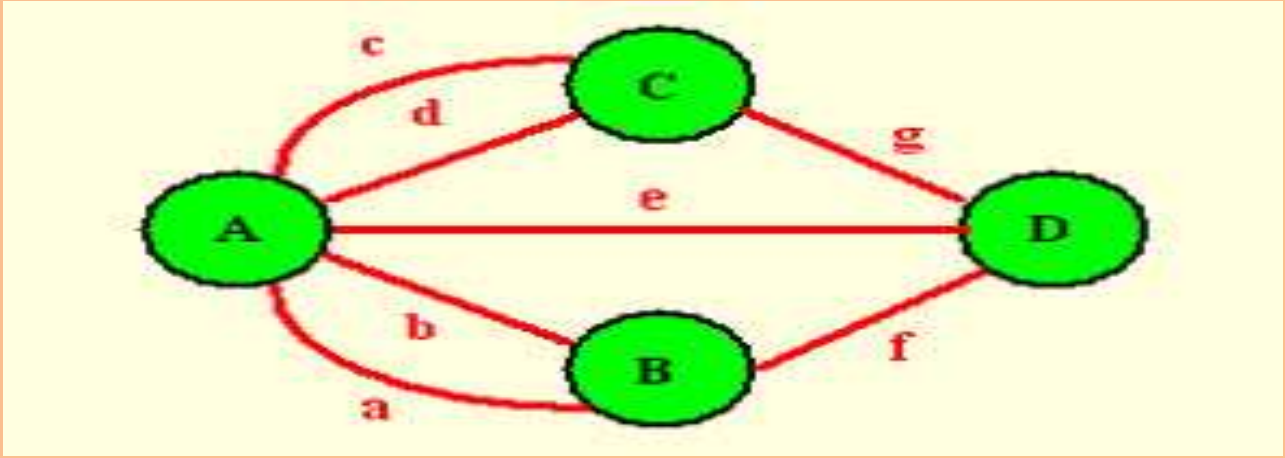
Visualisation d’un trafic aérien



Graphe de numérisation d’une cellule

Les 7 ponts de Königsberg

Modèle d'Euler (1707-1783)



3- Qu’est-ce qu’un graphe ?

F**ormellement**, on appelle graphe G :

1- un ensemble **S** d’**objets** appelés **sommets**,

2- un ensemble **A** de **relations** entre ces sommets.

On note habituellement:

G = (S, A)

Deux hypothèses se présentent:

1- les relations sont **symétriques**: on parle alors de **graphe non orienté**,

2- les relations ne sont **pas symétriques**: on parle alors de **graphe orienté**.

Graphe orienté

Un graphe orienté G est un couple :

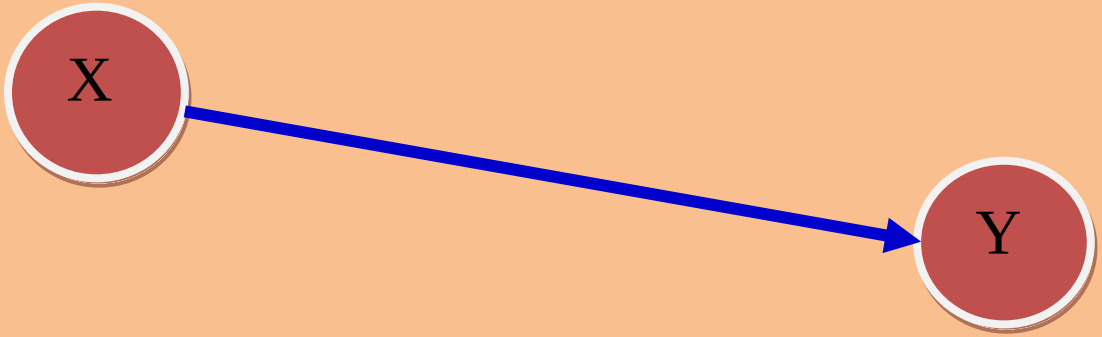
G= (S,A)

Où :

- S ensemble fini de **sommets**,
- A, un ensemble fini de **couples** de sommets, appelées **arcs**.

On note x$\rightarrow$ y l'arc (x,y):

- x désigne l'**extrémité initiale**,
- y désigne l'**extrémité terminale**.



On dit que :

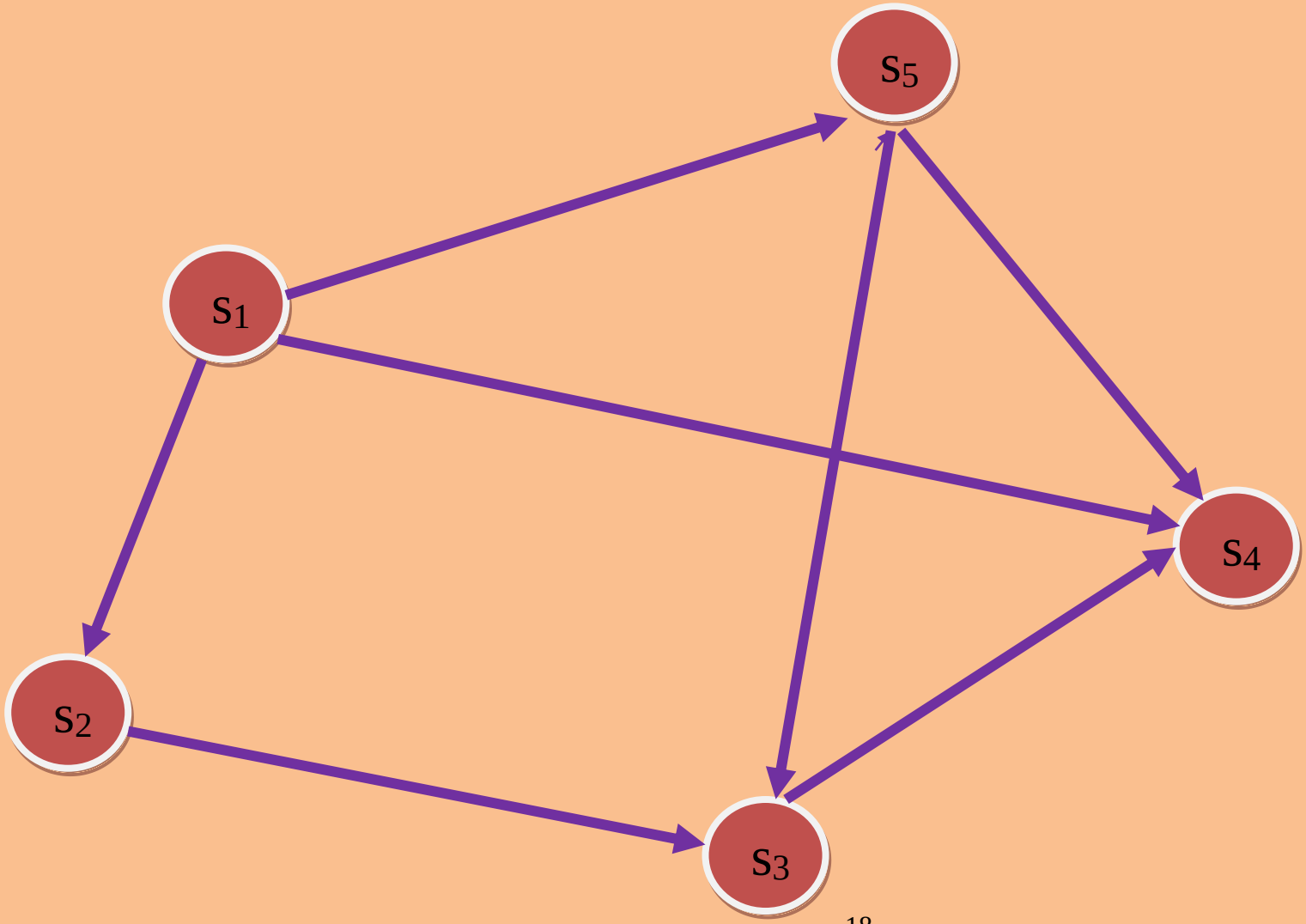
- y est le **successeur de** x,
- x est le **prédécesseur de** y.

Le sommet y est dit **adjacent** à x s'il existe un arc

$x \rightarrow y$ ou $y \rightarrow x$

Paires de sommets adjacents:

{ (s_1,s_2), (s_1,s_4), (s_1,s_5), (s_2,s_3), (s_3,s_4), (s_3,s_5), (s_4,s_5) }



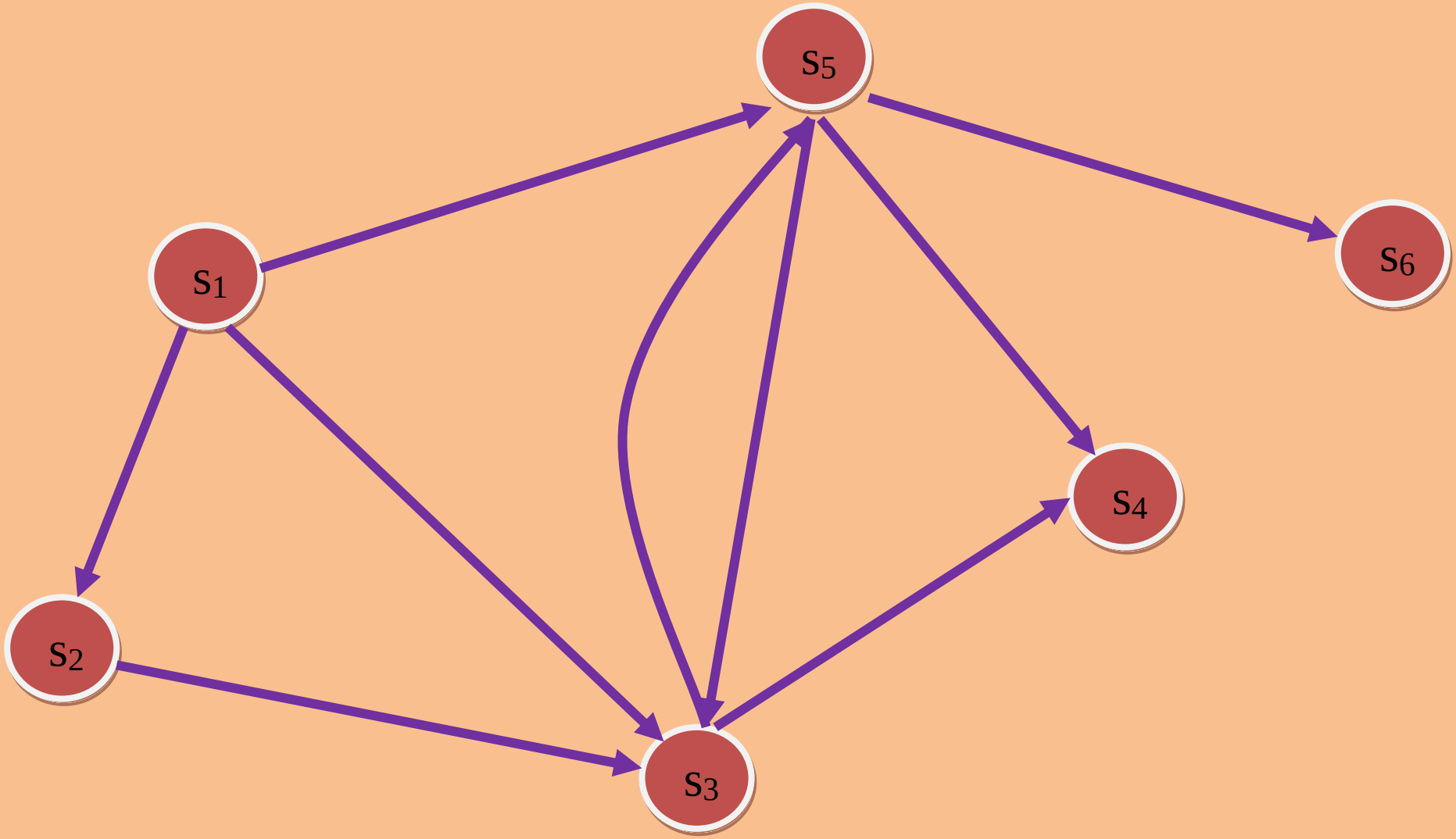
Exemple de graphe orienté

Soit le graphe G =(S,A) défini par :

$S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$

$A = \{ (s_1 \rightarrow s_2), (s_1 \rightarrow s_3), (s_1 \rightarrow s_5), (s_2 \rightarrow s_3), (s_3 \rightarrow s_4), (s_3 \rightarrow s_5), (s_5 \rightarrow s_3), (s_5 \rightarrow s_4), (s_5 \rightarrow s_6) \}$

Visualisation du graphe G



Graphe non orienté

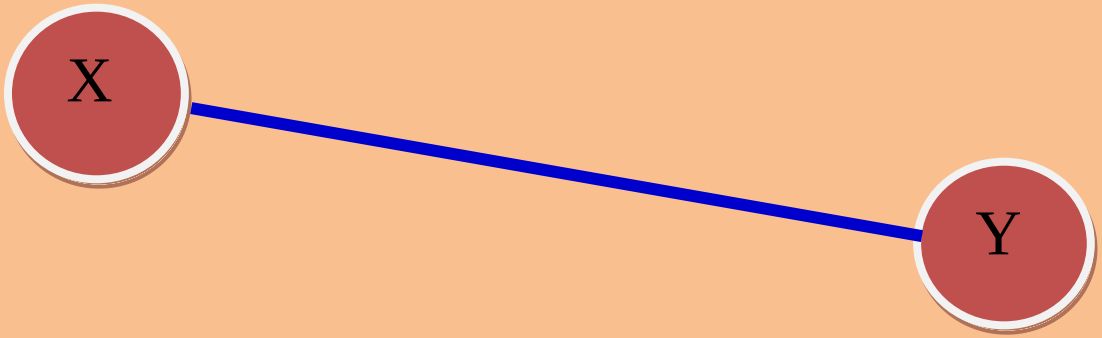
Un graphe **non orienté** G est un couple :

$$G= (S,A)$$

Où :

- S ensemble fini de **sommets**,
- A, un ensemble fini de **paires** de sommets, appelées **arêtes**.

On note x - y l'**arête** joignant x et y.

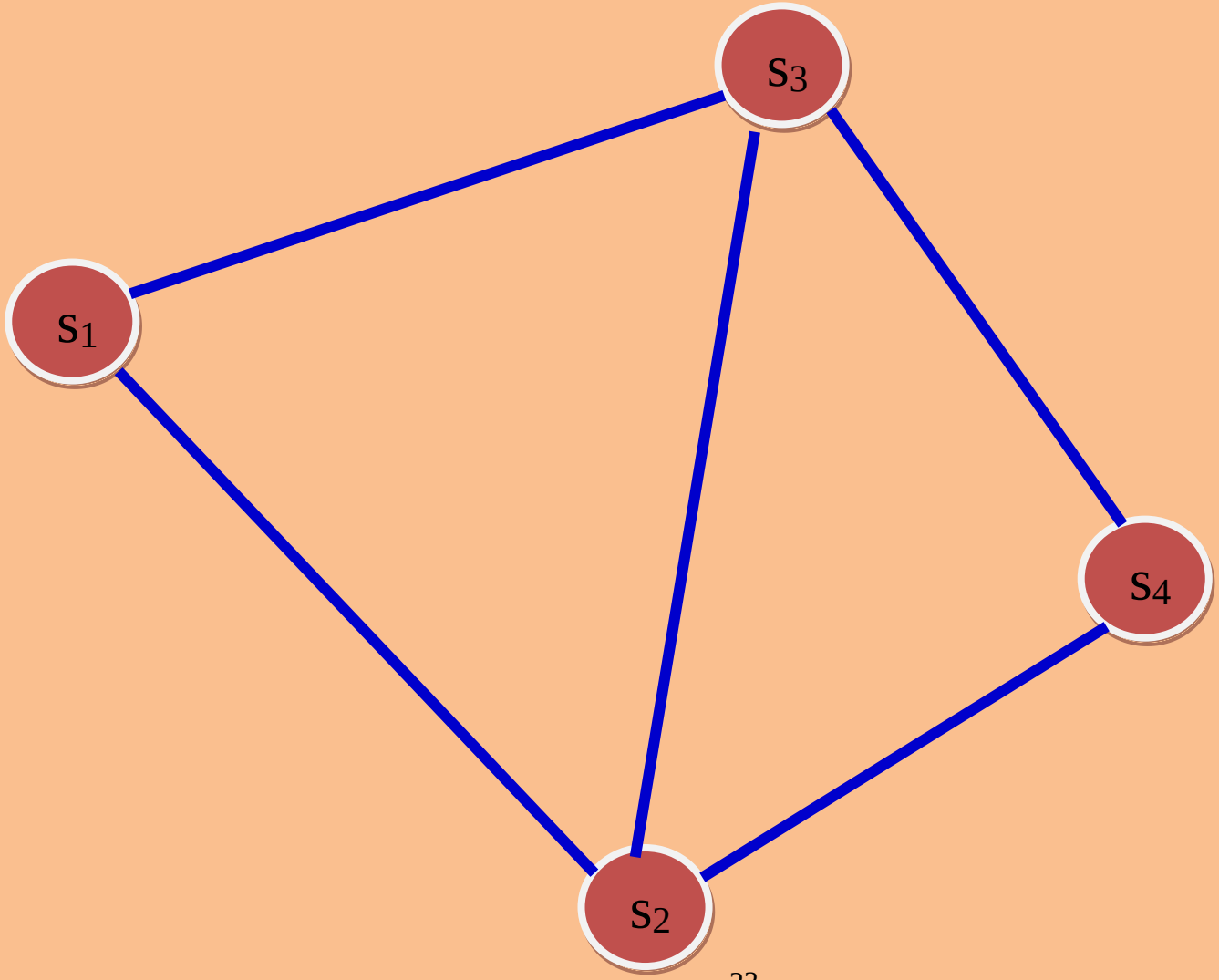


Les sommets x et y sont les deux **extrémités** de l'arête x-y.

Deux sommets sont dits **adjacents** s'il existe une arête les **joignant**.

Paires de sommets adjacents:

$\{ (s_1,s_2), (s_1,s_3), (s_2,s_3), (s_2,s_4), (s_3,s_4) \}$



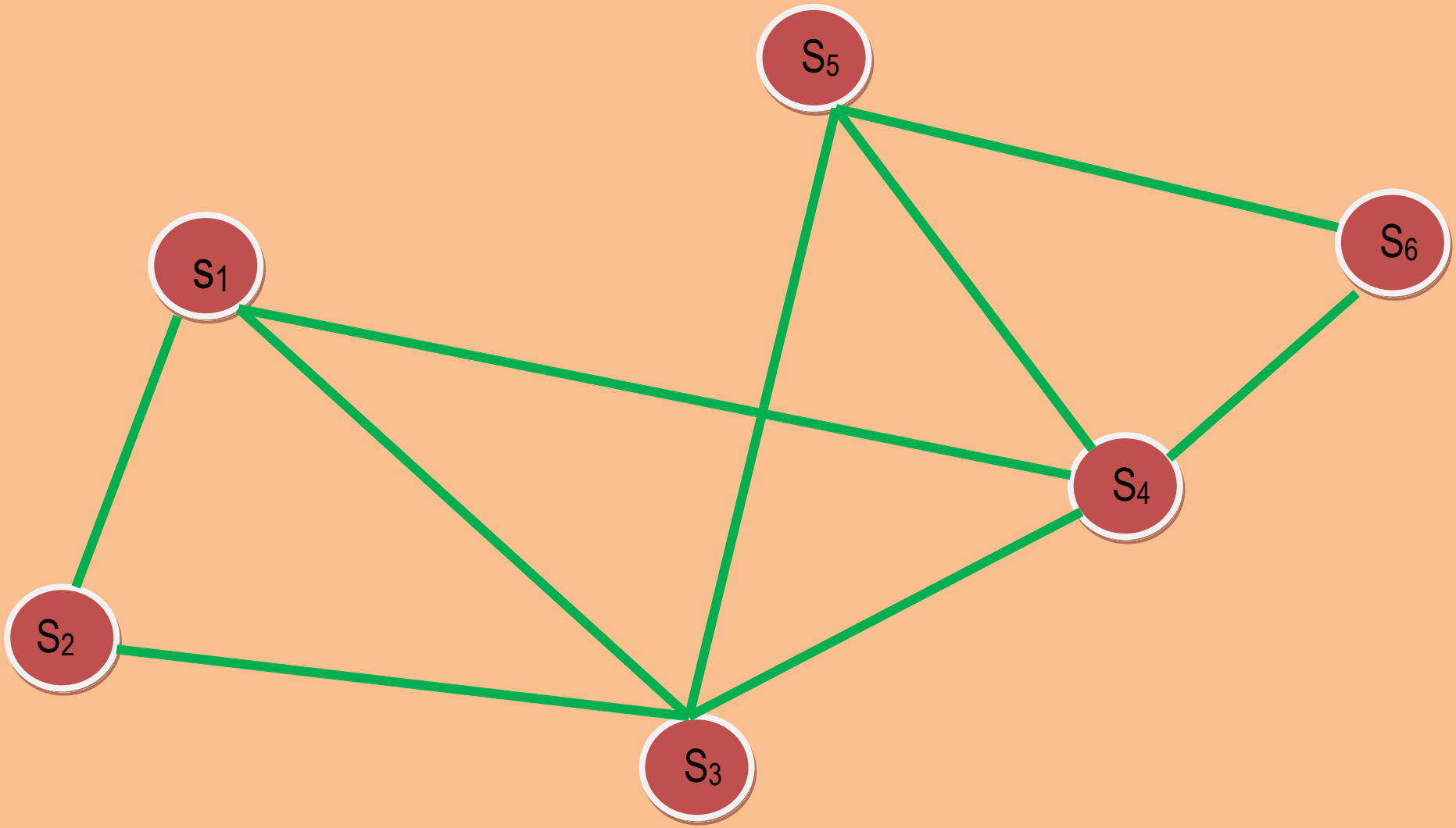
Exemple de graphe non-orienté

Soit le graphe G =(S,A) défini par :

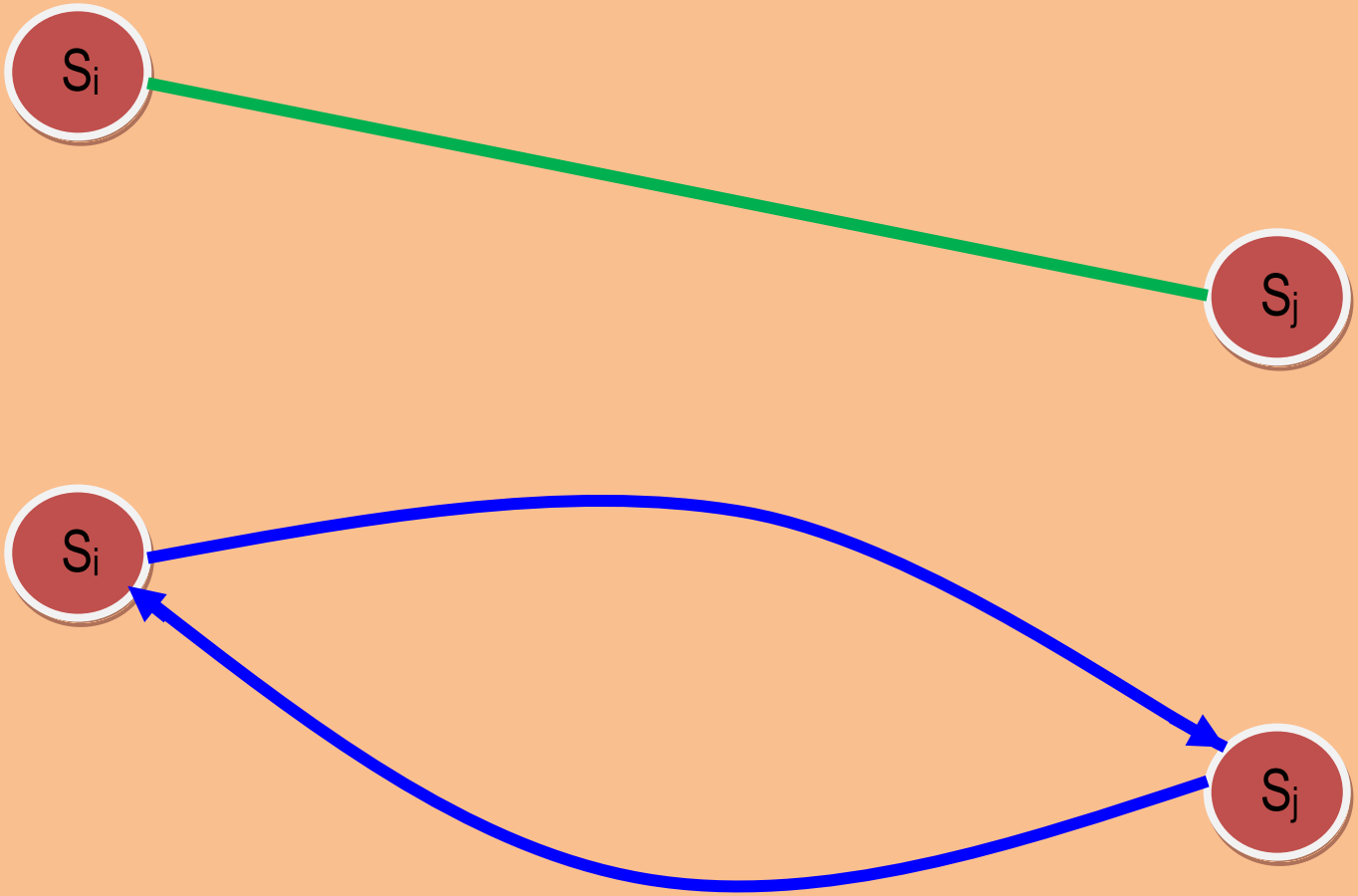
$S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$

$A = \{ (s_1\text{-}s_2),(s_1\text{-}s_3),(s_1\text{-}s_4),(s_2\text{-}s_3),(s_3\text{-}s_4),(s_3\text{-}s_5),(s_4\text{-}s_5),$
$(s_4\text{-}s_6), (s_5\text{-}s_6) \}$

Visualisation du graphe G



Arc - Arête



Graphe valué

Un graphe **valué** G est un triplet:

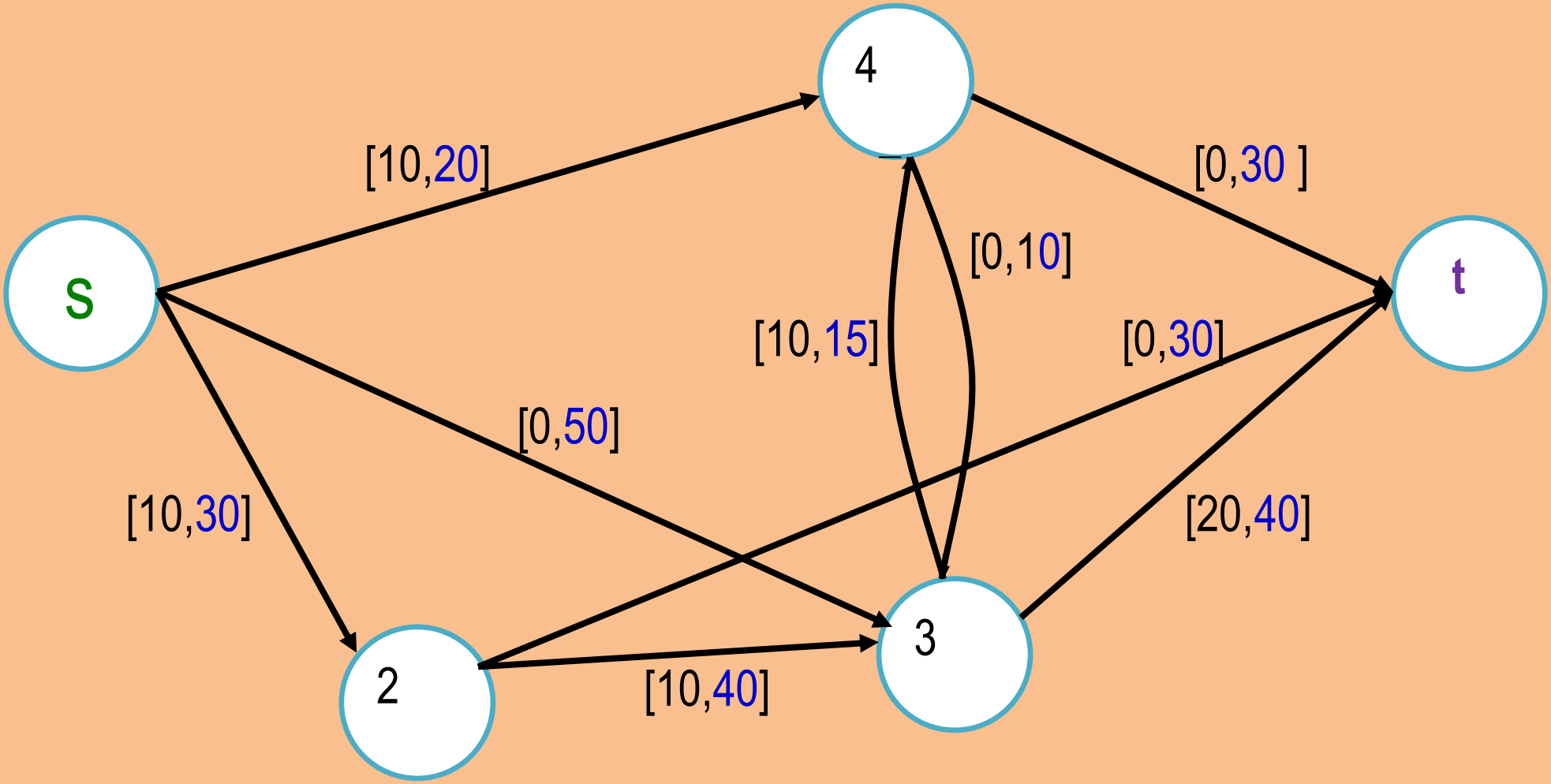
G= (**S**,**A**, F)

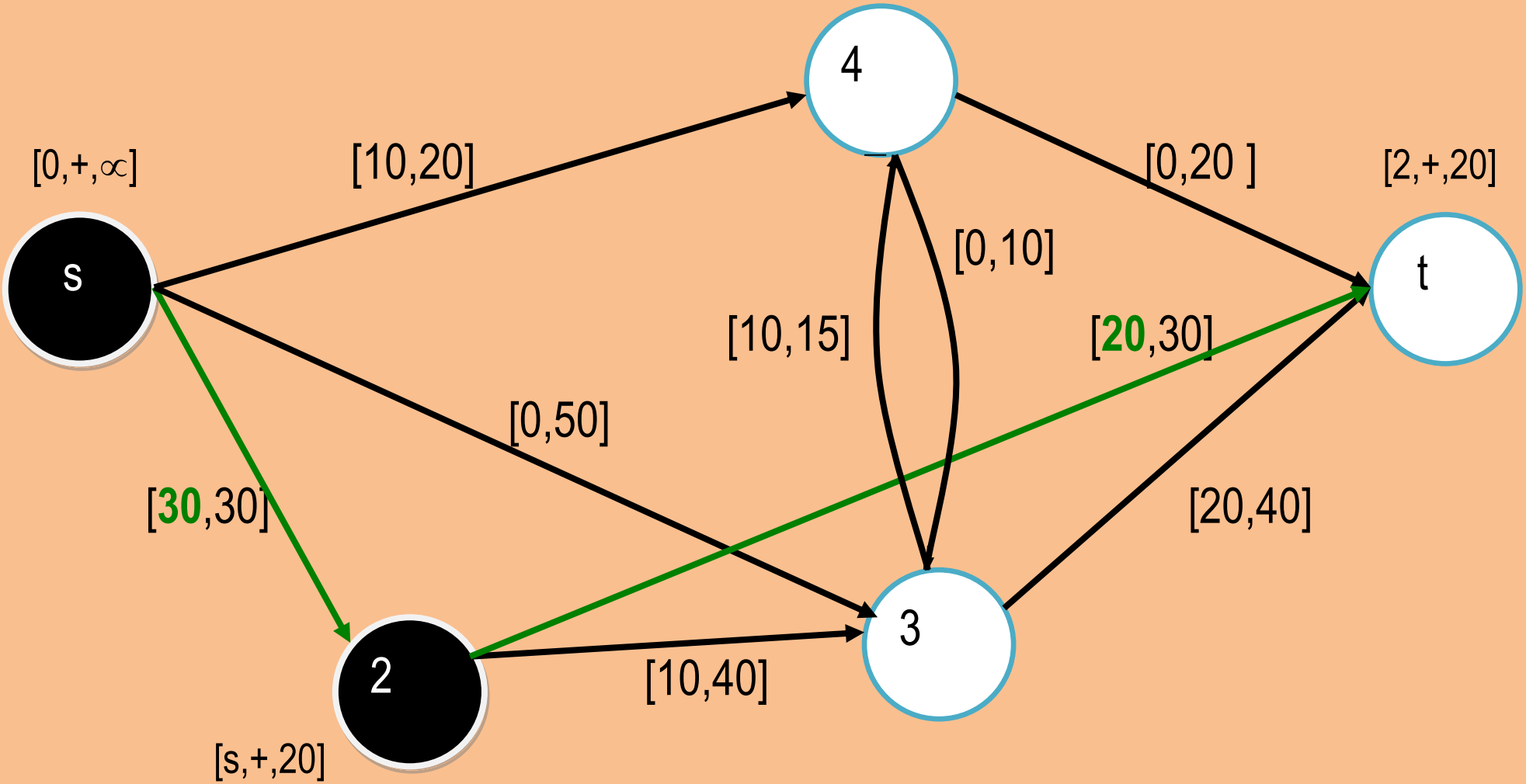
Où :

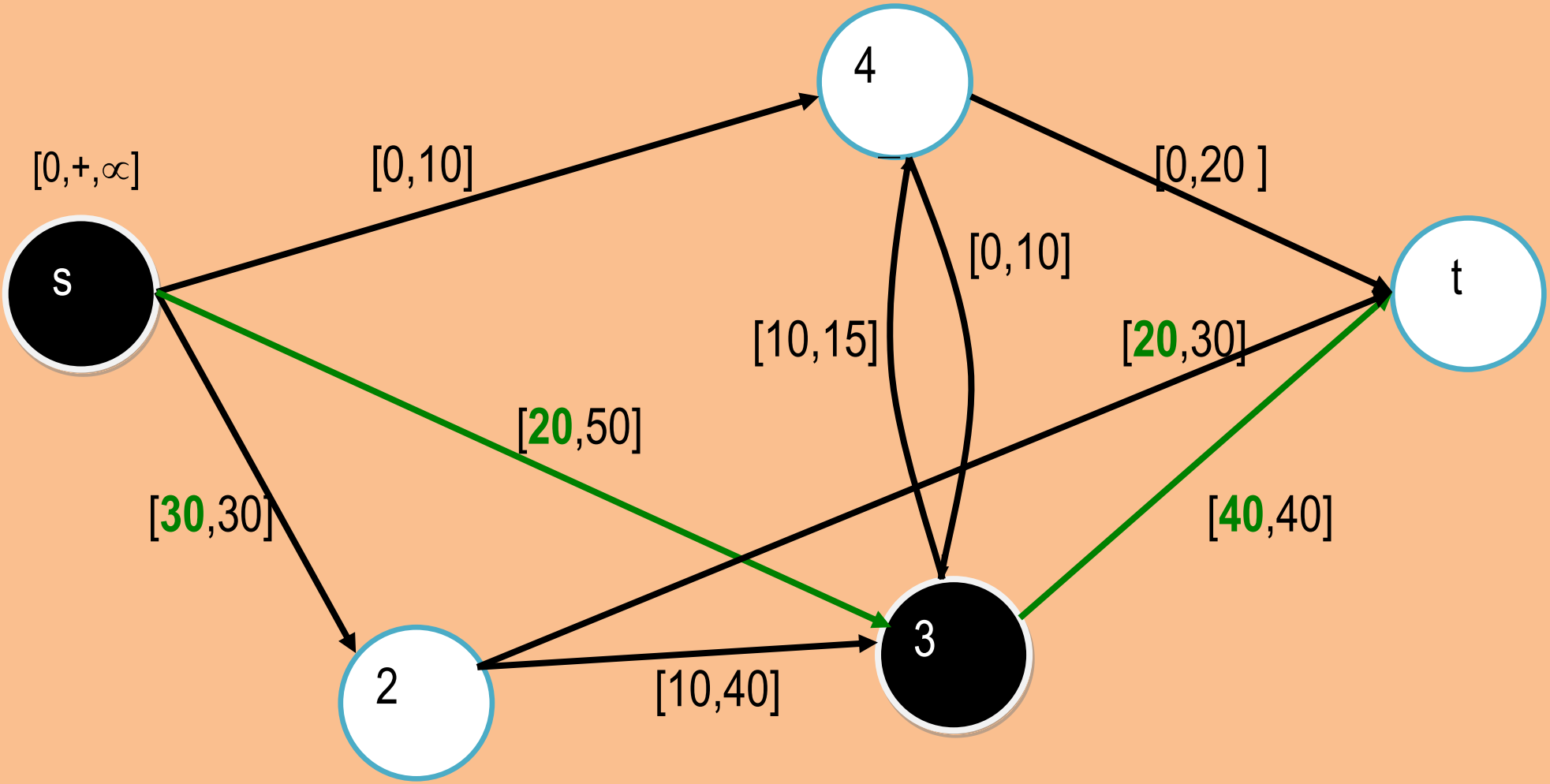
- S ensemble fini de **sommets**,
- A, un ensemble fini de **d'arcs** ou **d'arêtes,**
- F une **fonction** de **coût** :

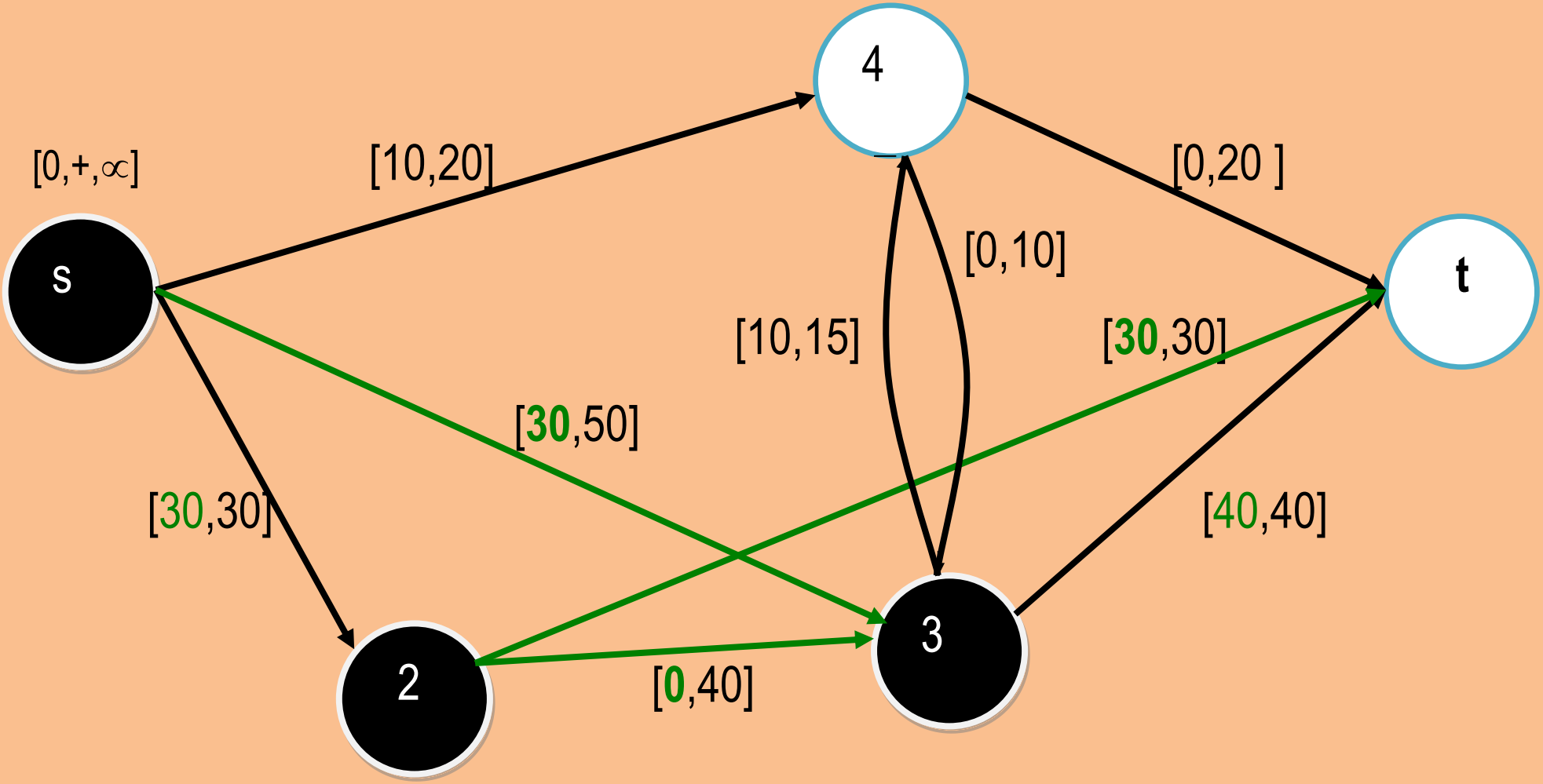
$$F : A \rightarrow \mathbb{R}^n$$

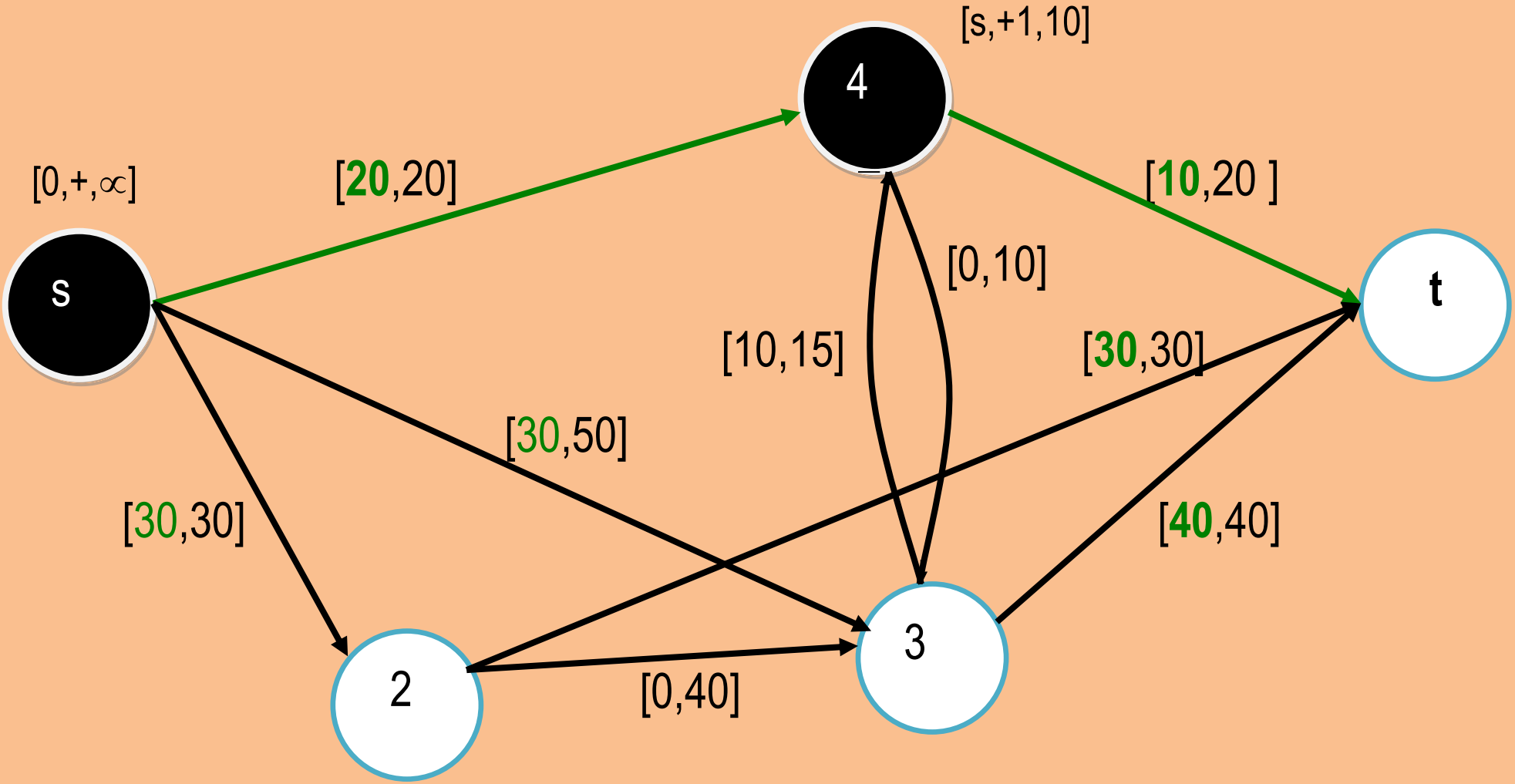
Exemple de graphe valué

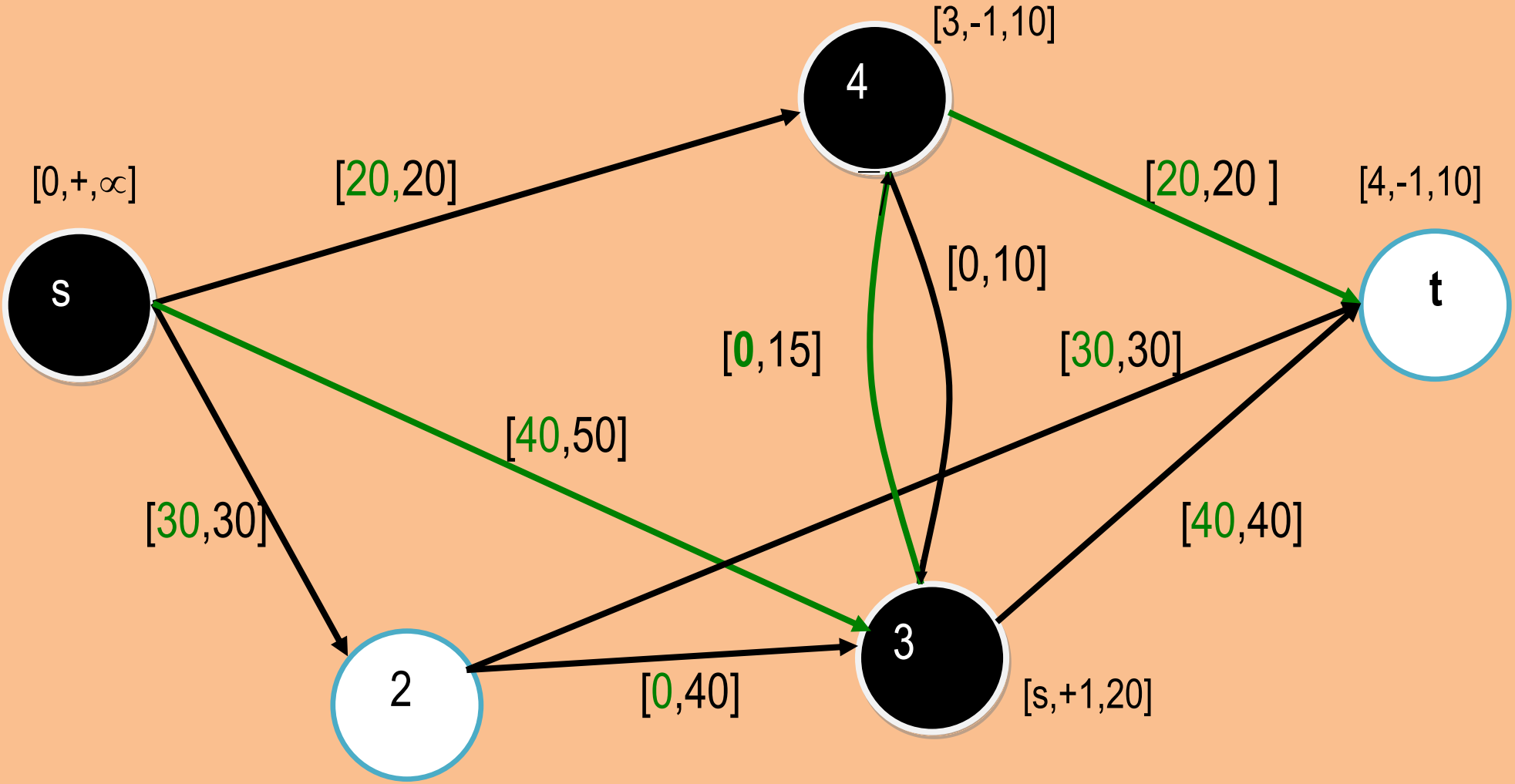


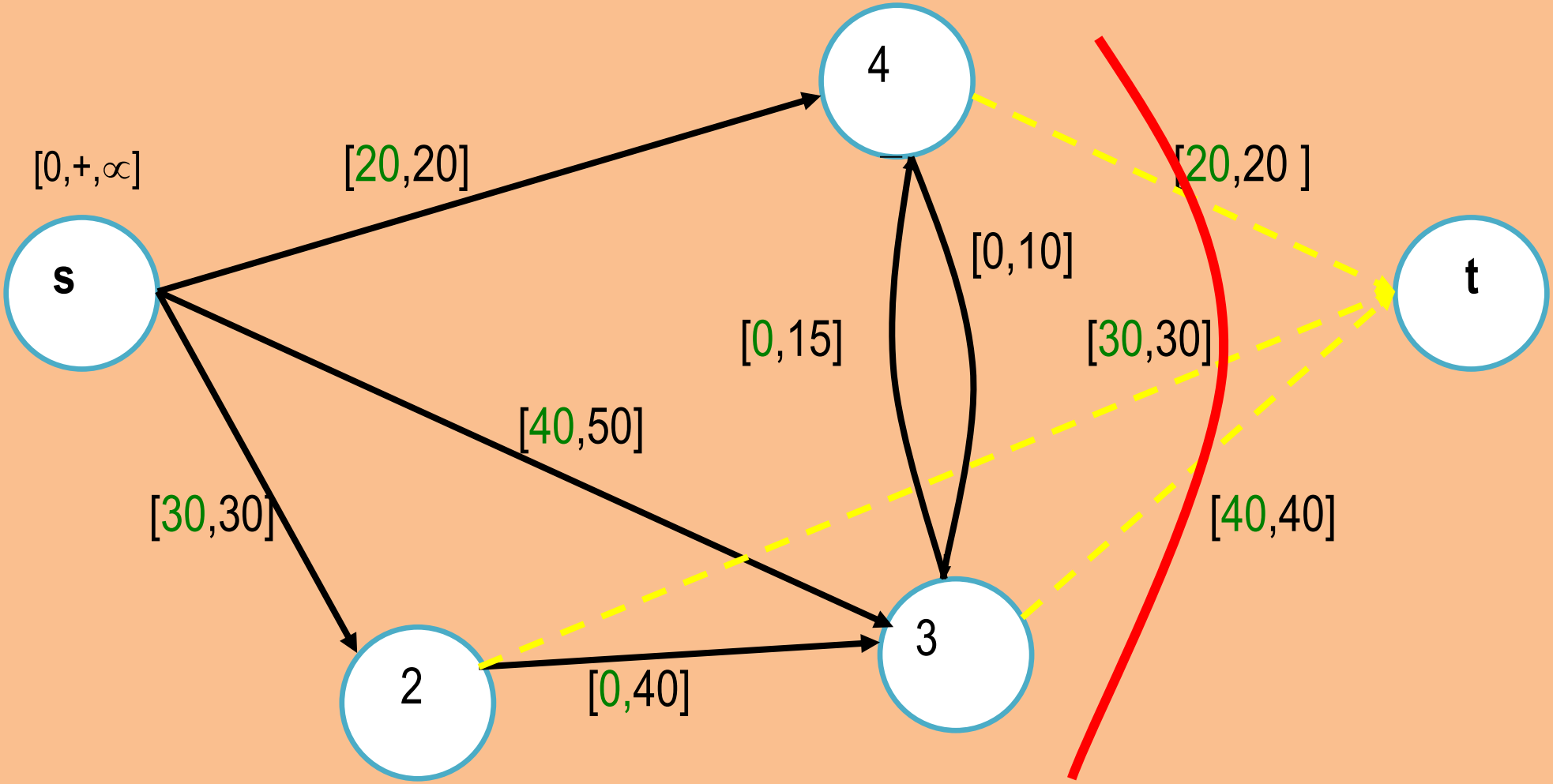

4
[0,+,∞]
[10,20]
[0,20]
[2,+,20]
s
[0,10]
t
[10,15]
[20,30]
[0,50]
[30,30]
[20,40]
3
2
[10,40]
[s,+,20]


4
[0,+,∞]
[0,10]
[0,20]
s
[0,10]
t
[10,15]
[20,30]
[20,50]
[30,30]
[40,40]
3
2
[10,40]

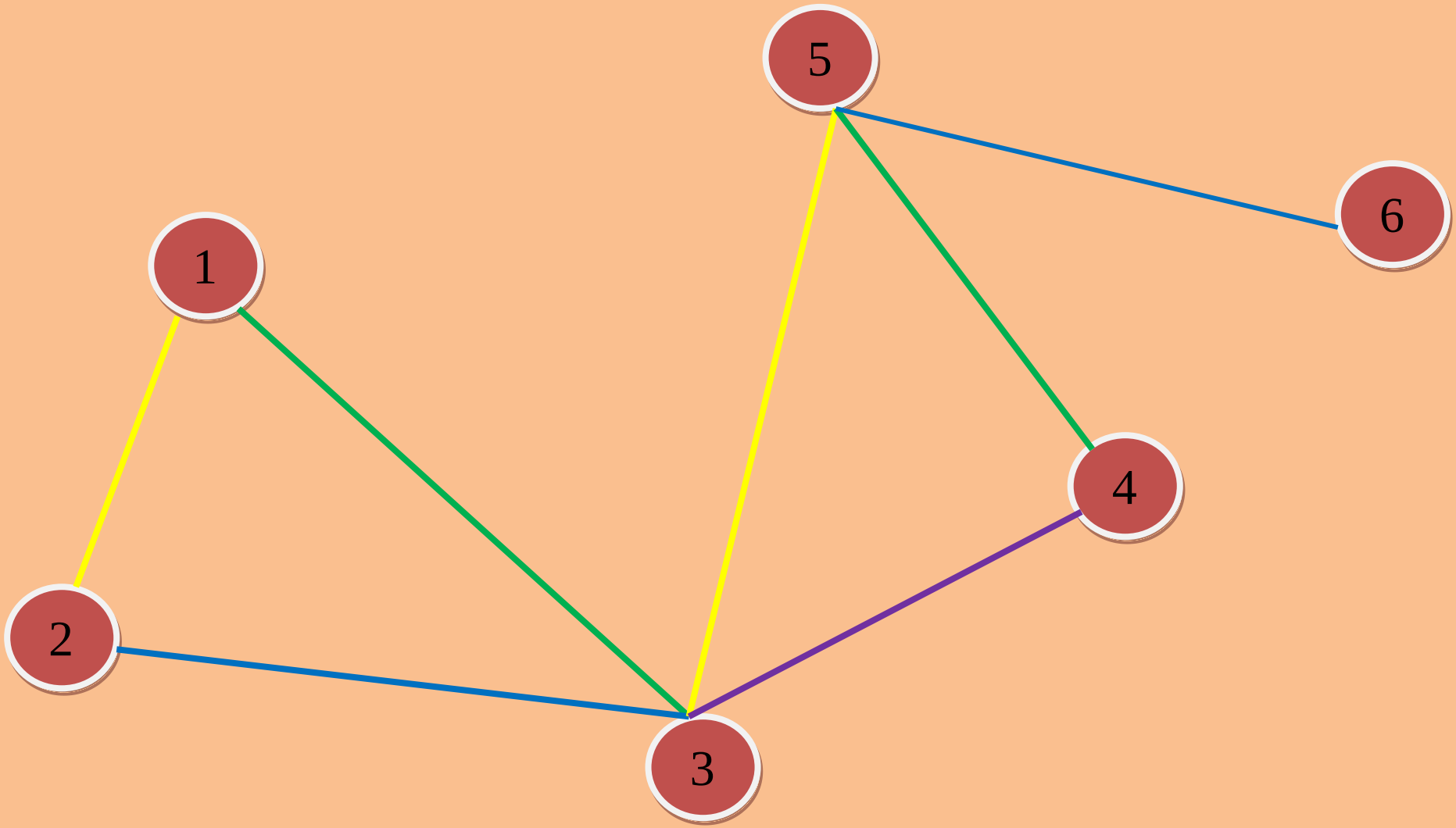

4
[0,+,∞]
[10,20]
[0,20]
s
t
[0,10]
[10,15]
[30,30]
[30,50]
[30,30]
[40,40]
3
2
[0,40]


[s,+1,10]
4
[0,+,∝]
[20,20]
[10,20]
s
[0,10]
t
[10,15]
[30,30]
[30,50]
[30,30]
[40,40]
3
2
[0,40]

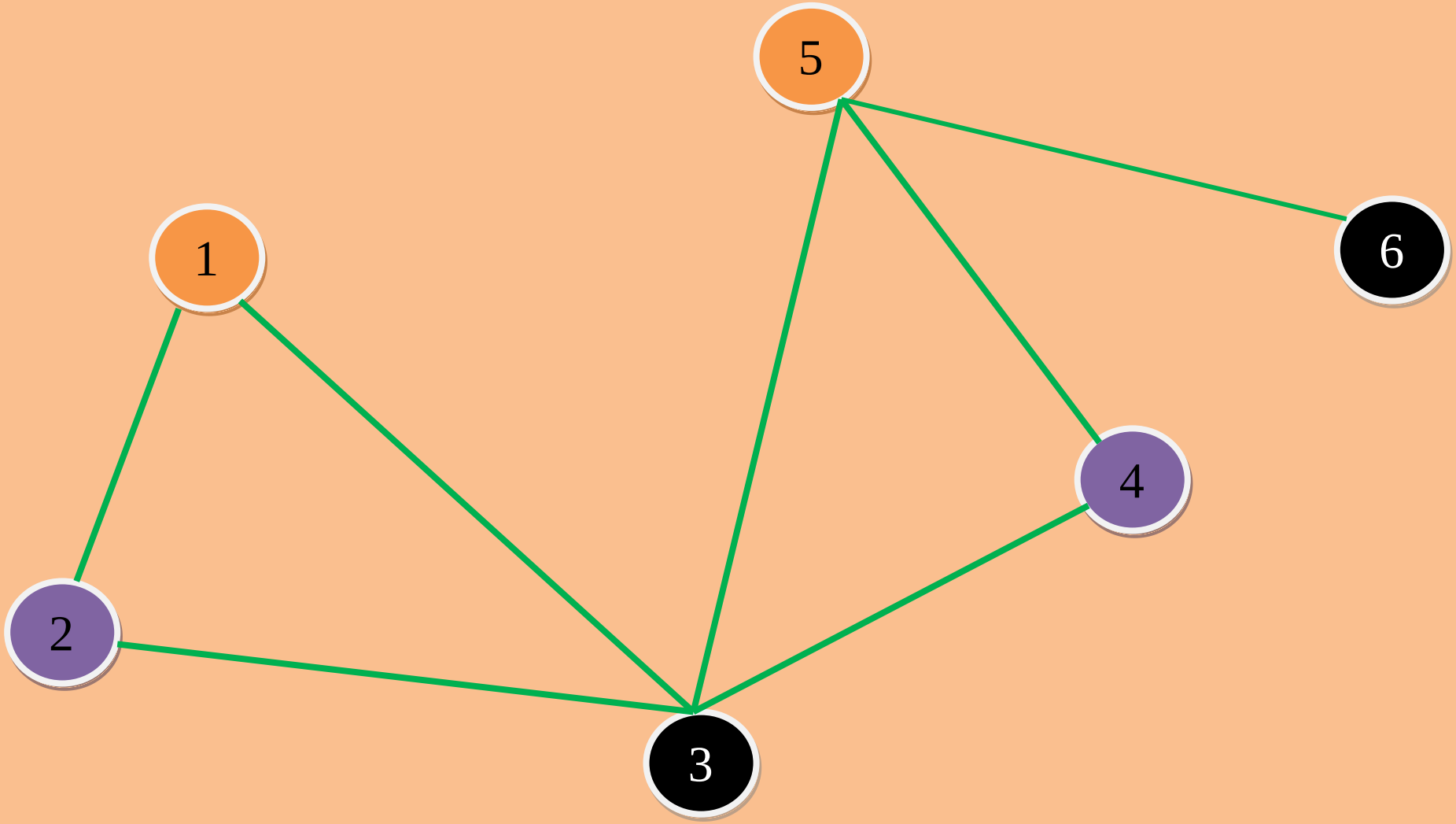

[3,-1,10]
4
[0,+,∝]
[20,20]
[20,20]
[4,-1,10]
s
t
[0,10]
[0,15]
[30,30]
[40,50]
[30,30]
[40,40]
3
2
[0,40]
[s,+1,20]


[0,+,∞]
s
[20,20]
4
[0,10]
[0,15]
[20,20]
t
[30,30]
[40,50]
[30,30]
[40,40]
3
2
[0,40]

Graphe coloré : coloration des arêtes



Graphe coloré : coloration des sommets



3- Relation entre les graphes

Soit **G= (S, A)**, un graphe.

Sous-graphe

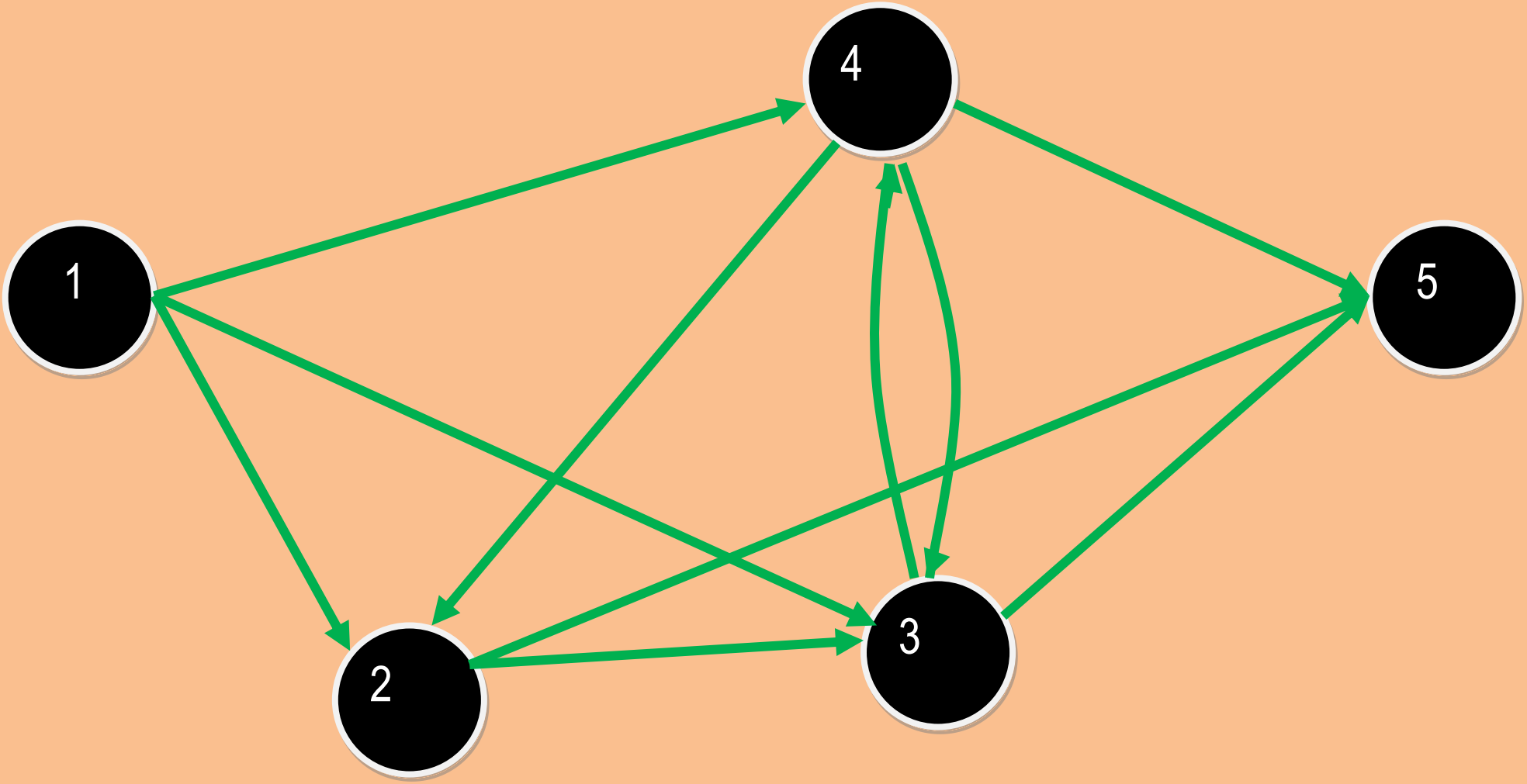
Le **sous-graphe** de **G** engendré par **S'** tel que **S' ⊆ S** est le graphe:

G'= (S',A')

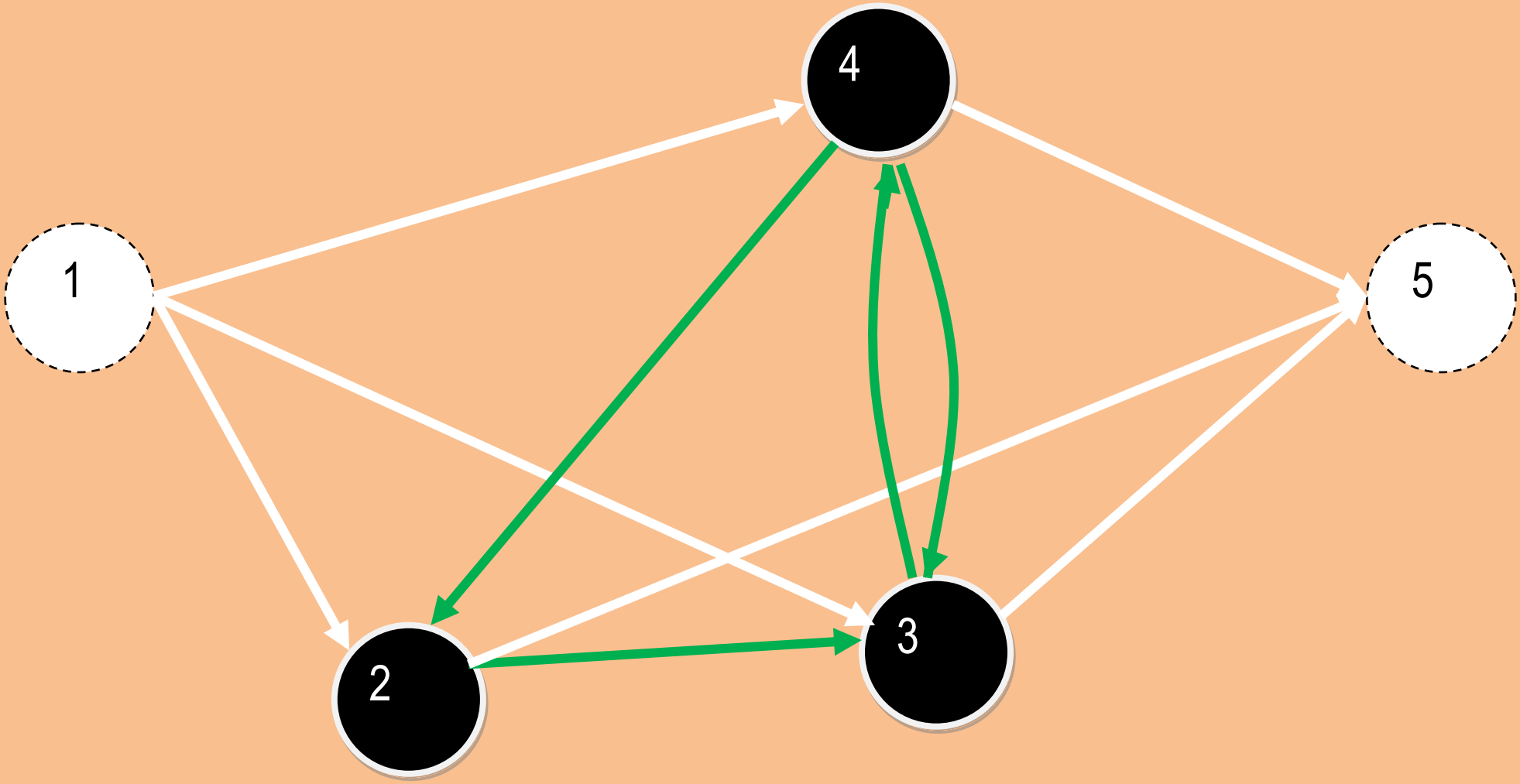
Où **A'** désigne le sous-ensemble des arcs (ou arêtes) :

- appartement à **A**,
- ayant leurs deux extrémités dans **S'**.

Soit le graphe G=(S,A) représenté ci-dessous:



Le **sous-graphe** G’ généré par S’= {2, 3, 4 } est représenté par:

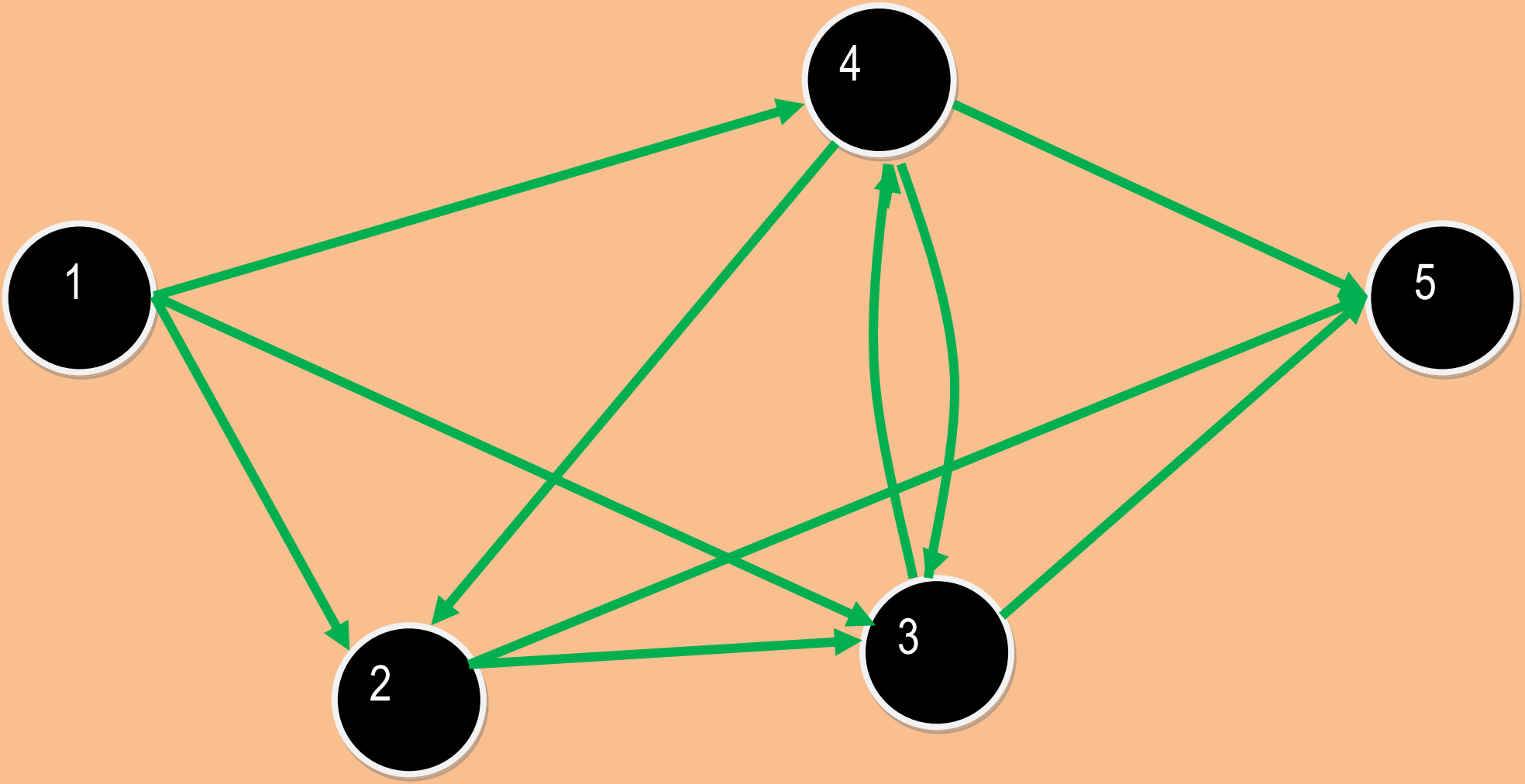


Comment construire un sous-graphe ?

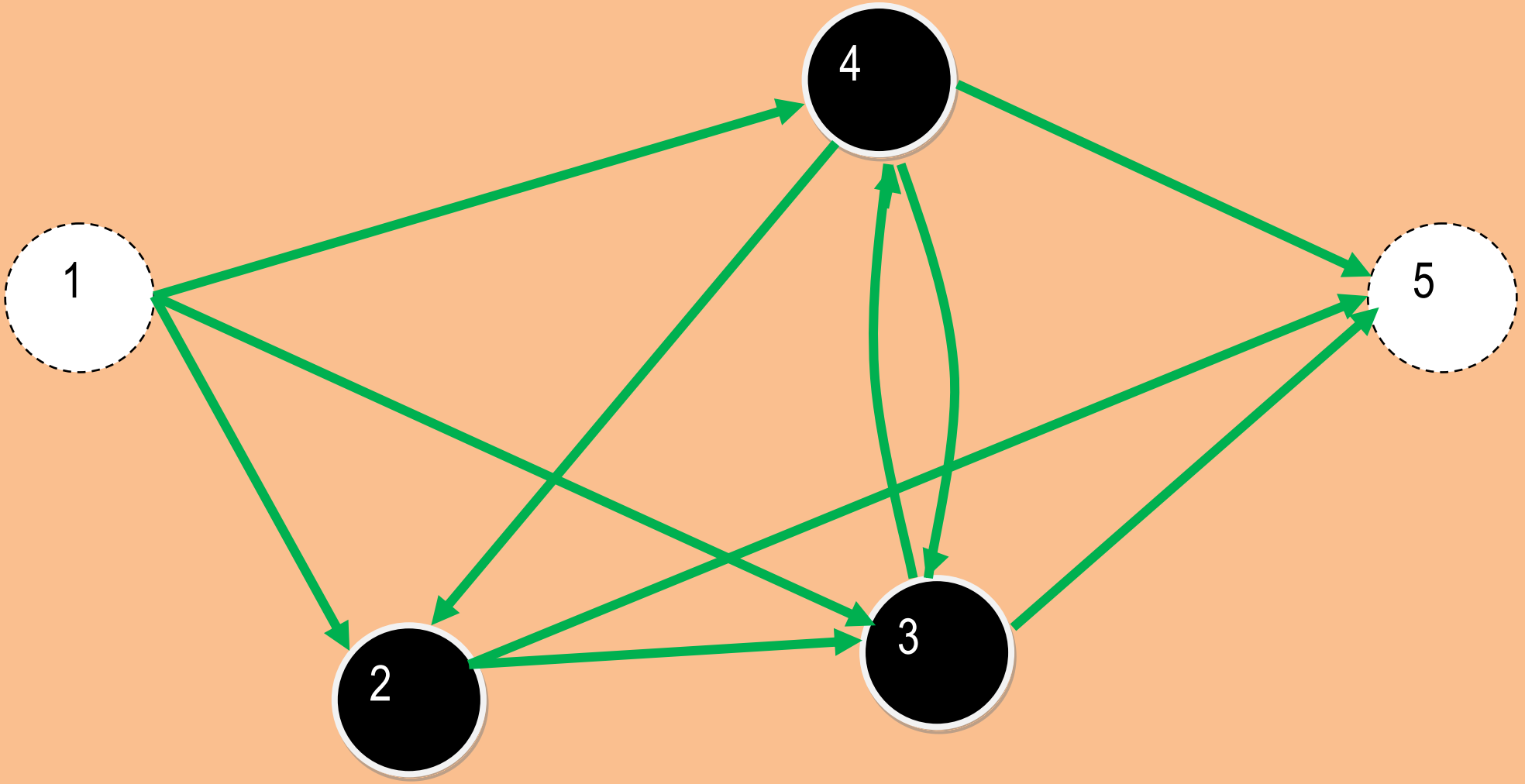
Partant du graphe G= (S,A), on ignore:

- les sommets appartenant à **S-S'**,
- les arcs ou arêtes ayant au moins une extrémité dans S-S'.

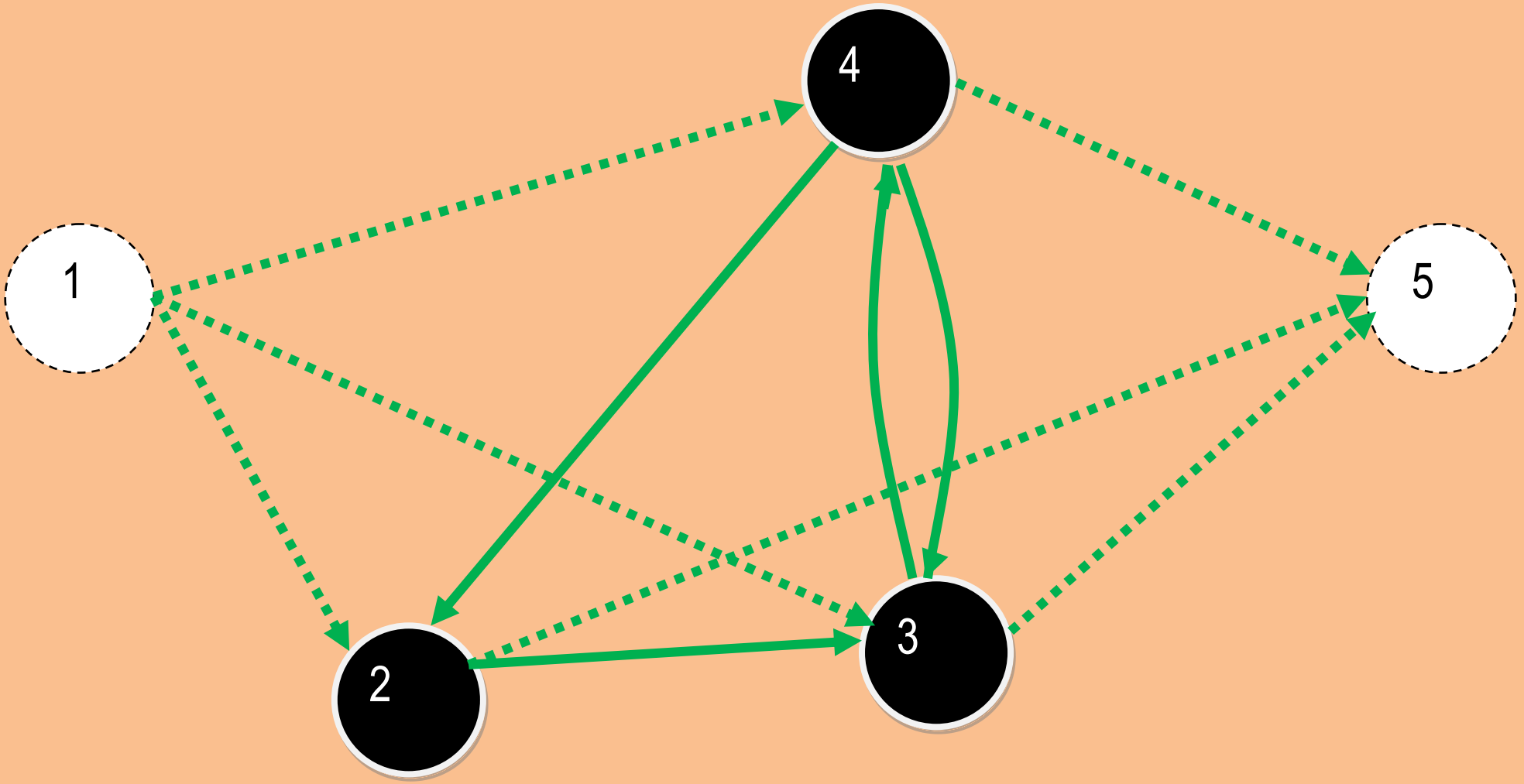
On part du graphe original G=(S,A) ci-dessous :



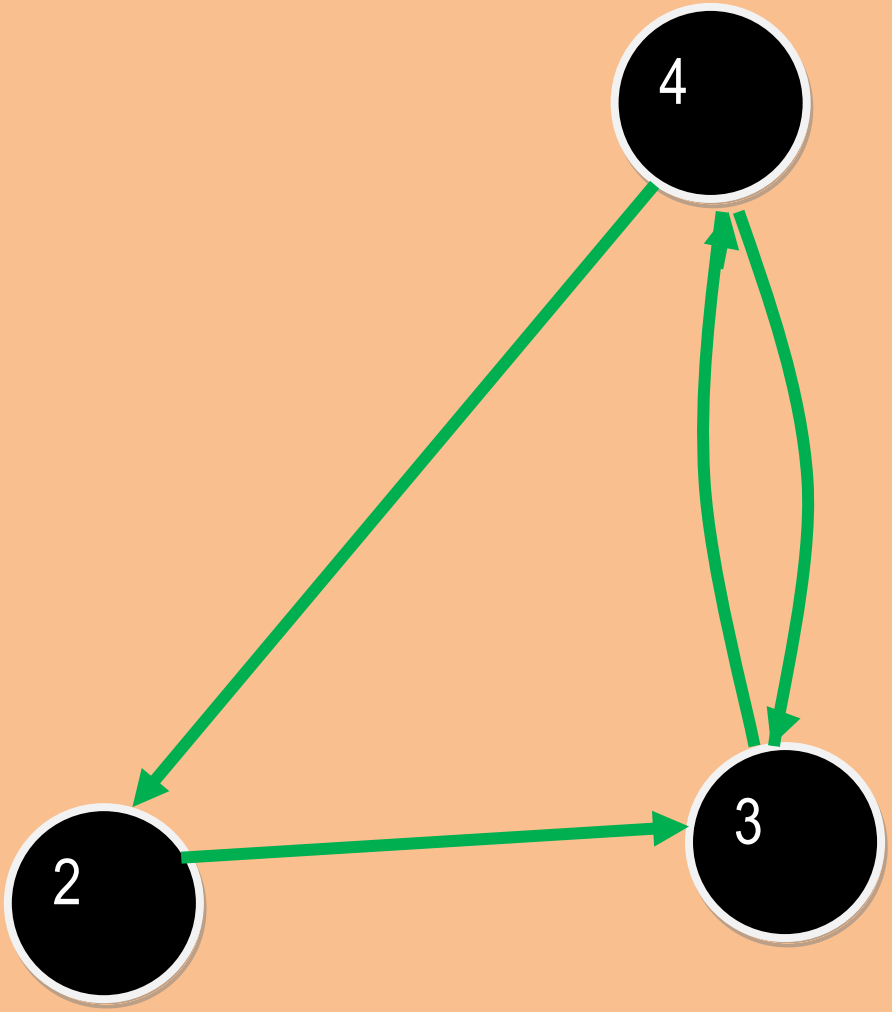
1- on ignore les sommets appartenant à **S-S'** :



2- on ignore les arcs ayant au moins une extrémité dans S-S':



On obtient le sous-graphe G’ ci-dessous:



Graphe partiel

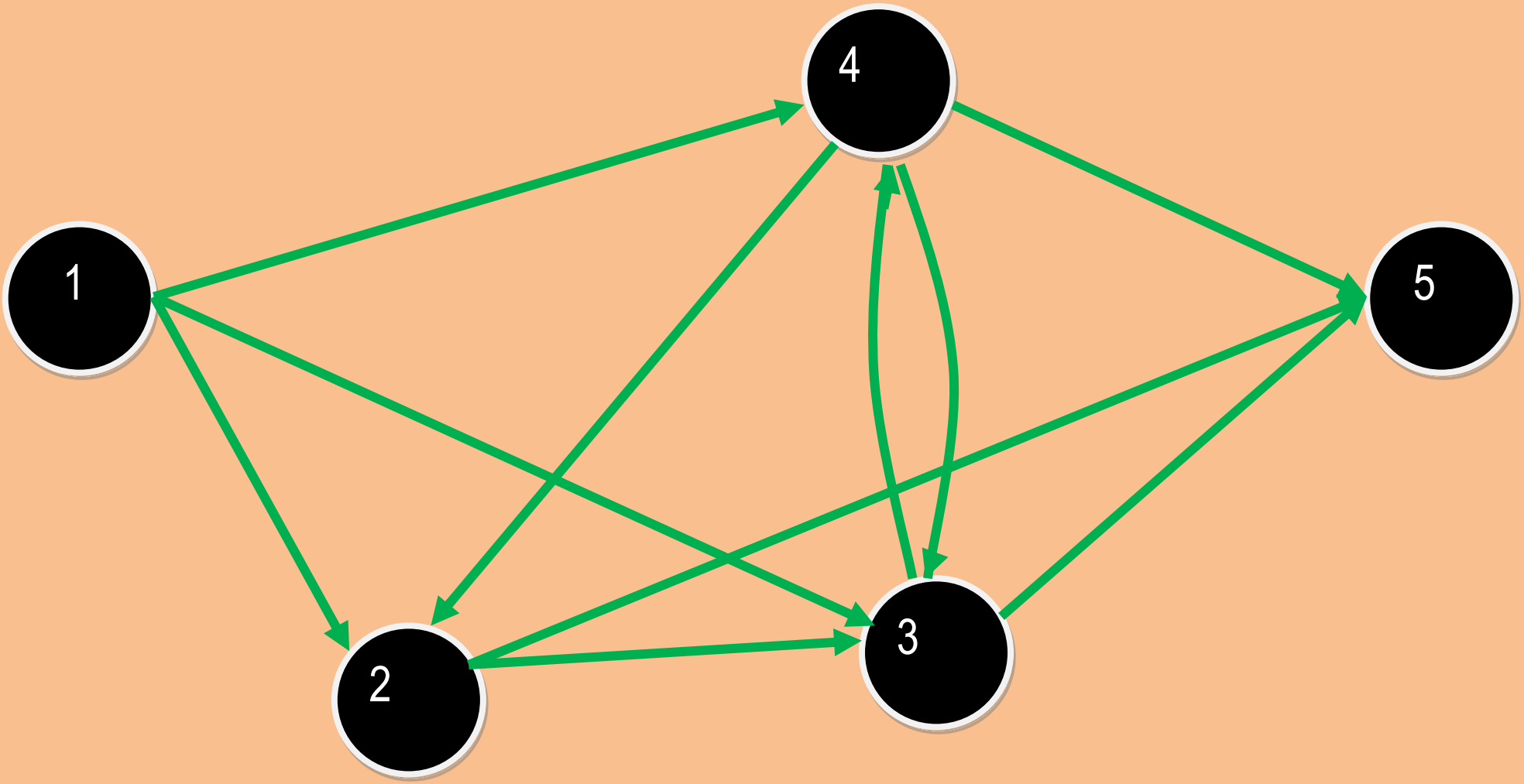
Le **graphe partiel** de **G** engendré par **A'** tel que :

$$\mathbf{A'} \subseteq \mathbf{A}$$

est le graphe:

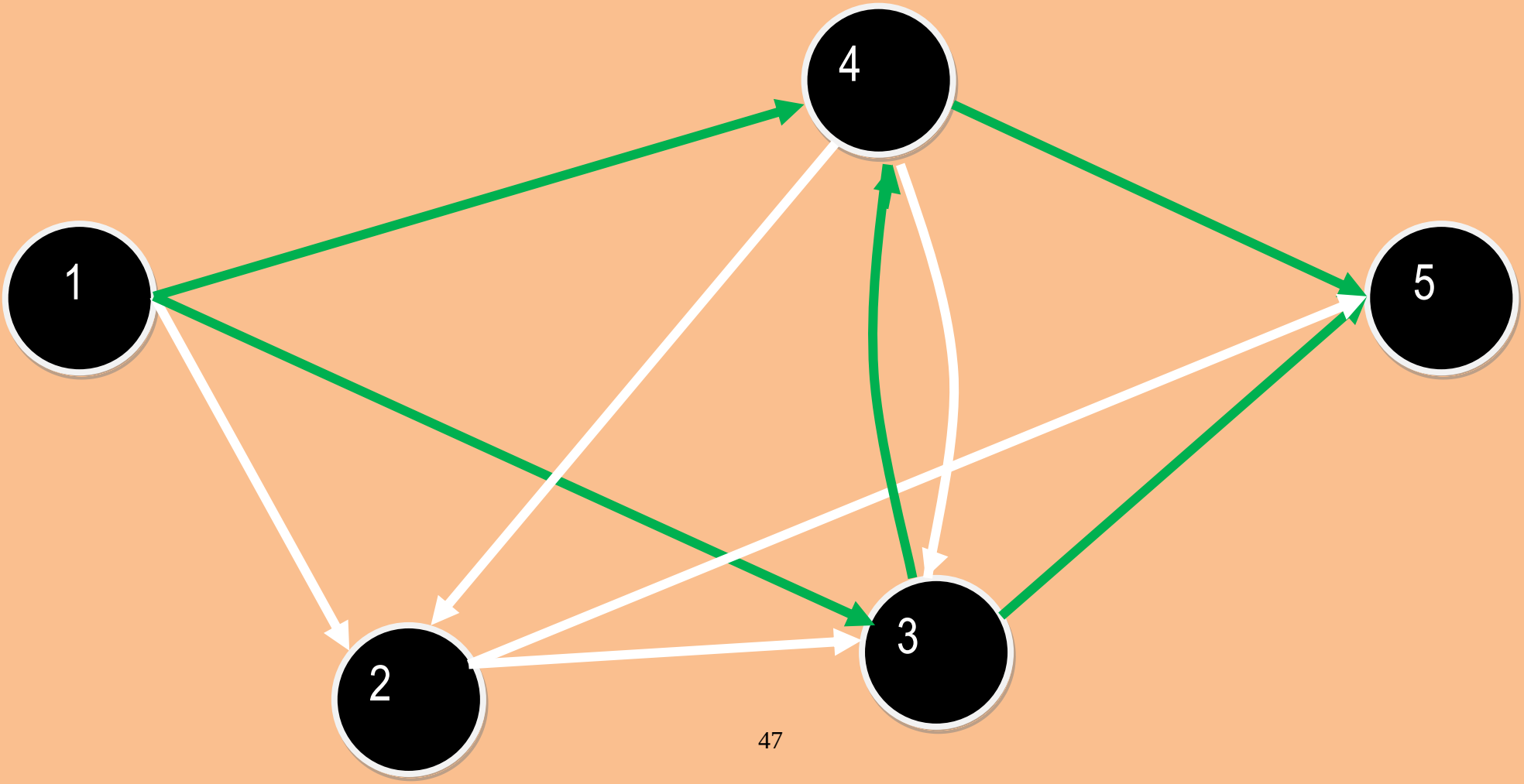
$$G' = (S, A')$$

Soit le graphe G=(S,A) représenté ci-dessous:

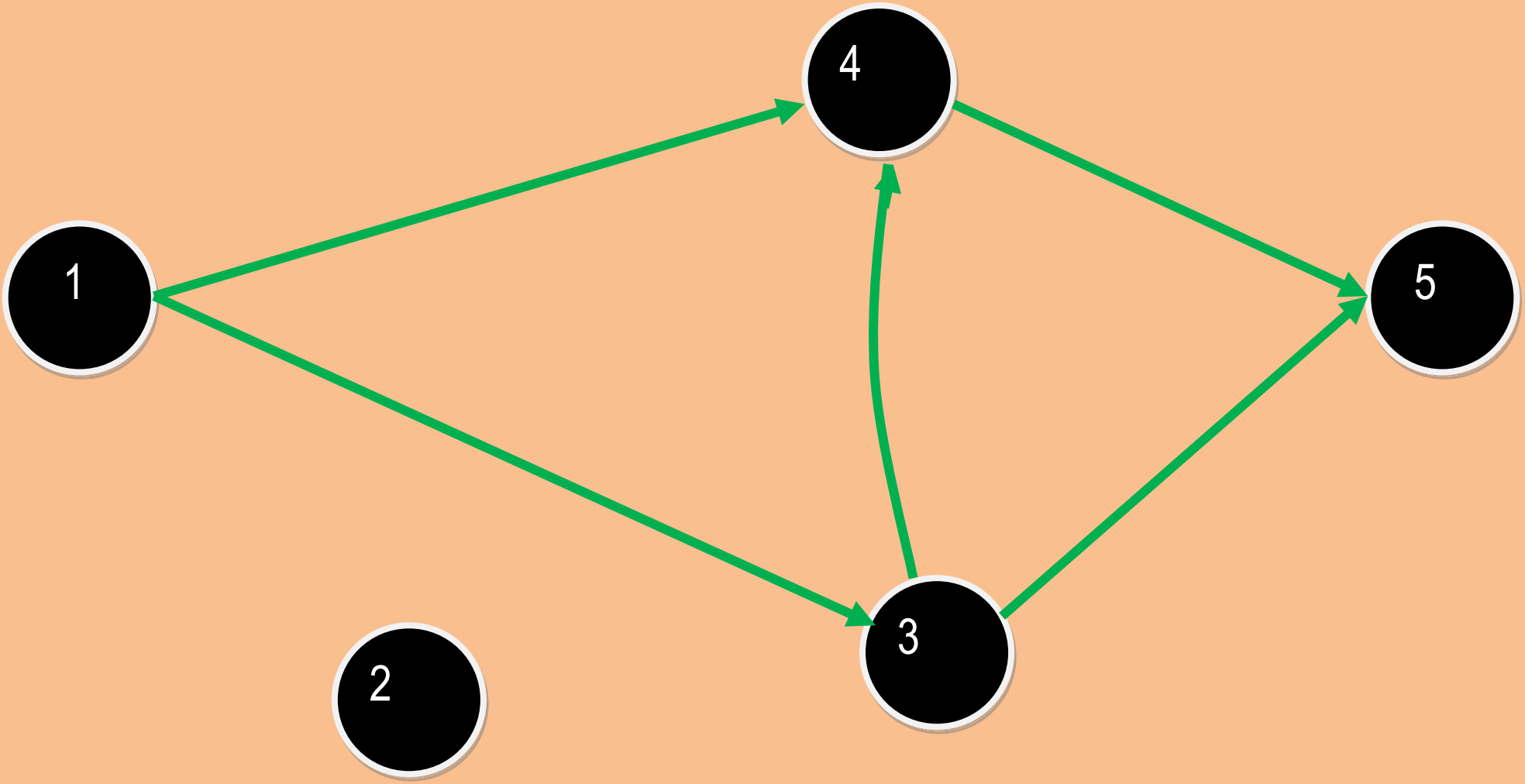


Le **graphe partiel** G' généré par:

A'={(1→3),(1→4),(3→4),(3→5),(4→5) }



est représenté ci-dessous:

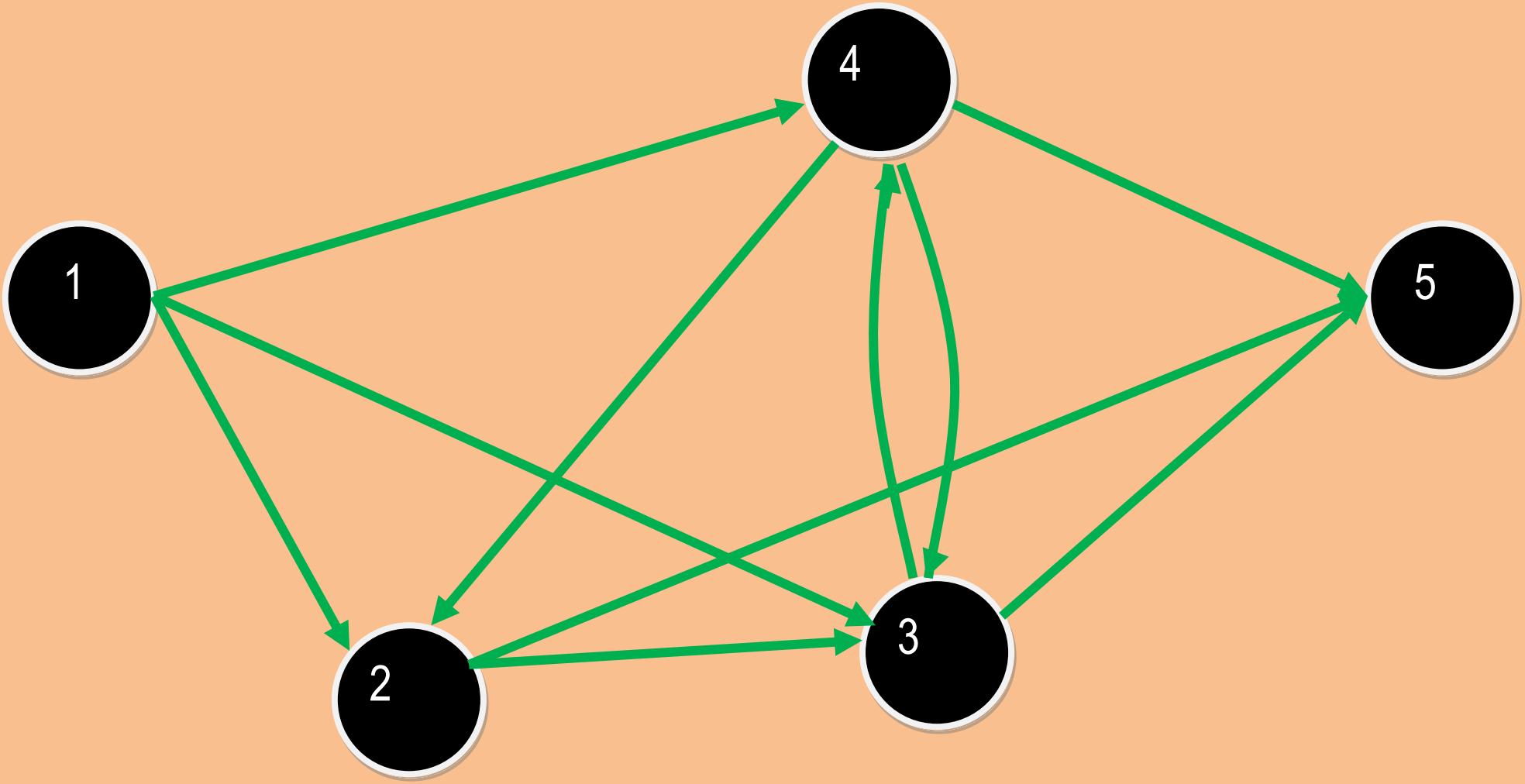


Comment construire un graphe partiel ?

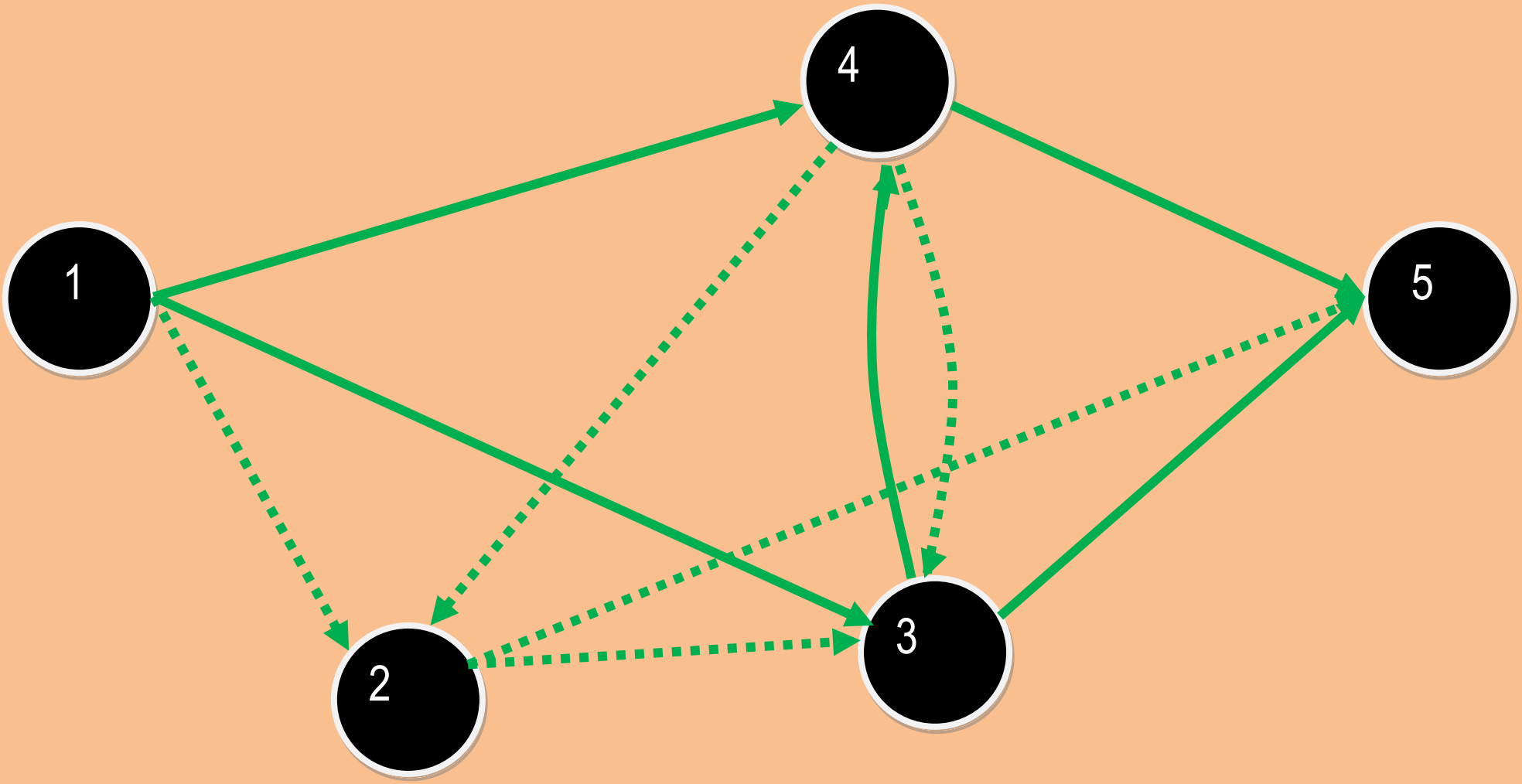
Partant du graphe G = (S,A) :

- S reste **inchangé**,
- on ignore tous les arcs ou arêtes appartenant à A-A'.

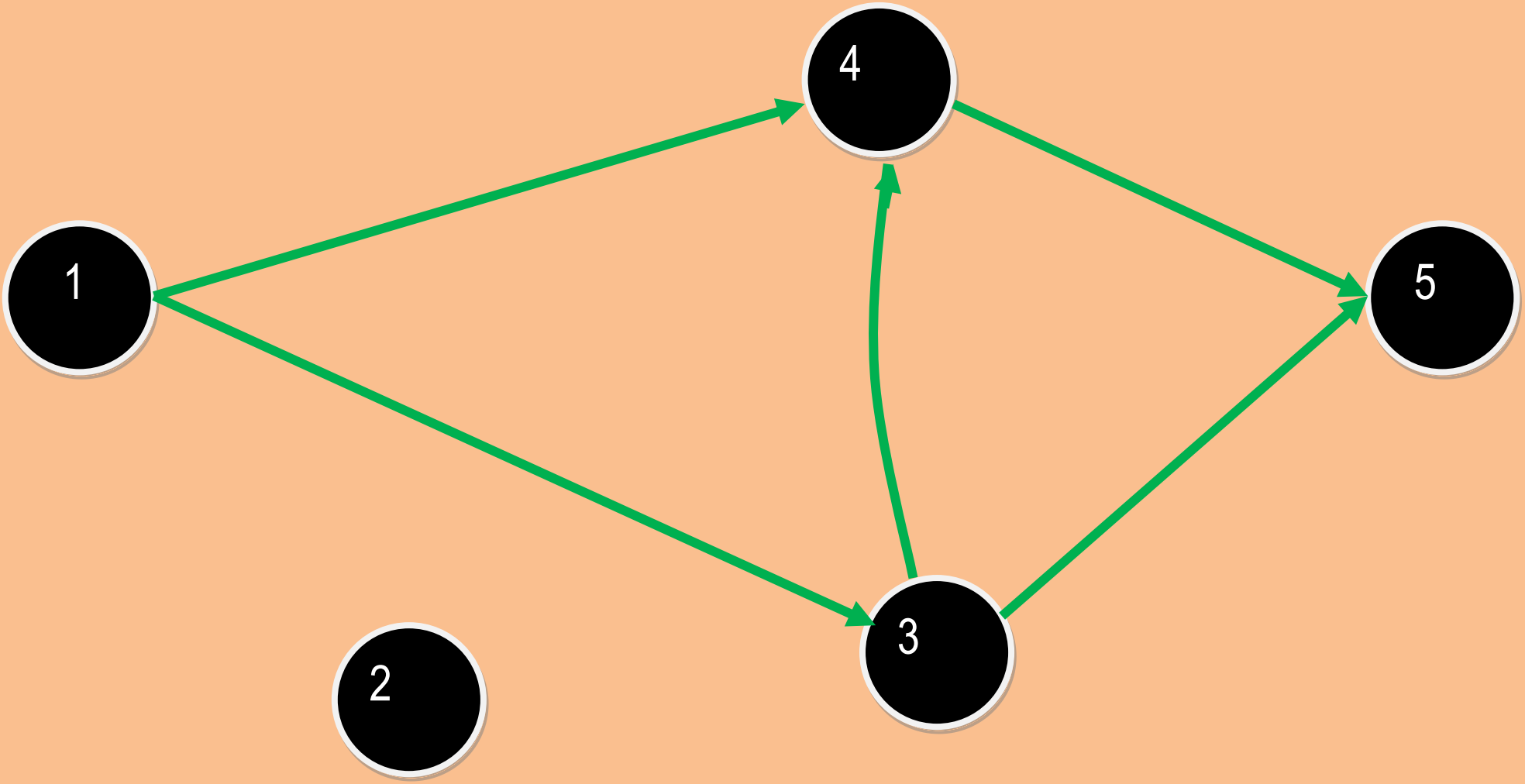
Partant du graphe original G=(S,A) représenté ci-dessous:



1-on ignore tous les arcs appartenant à A-A’.



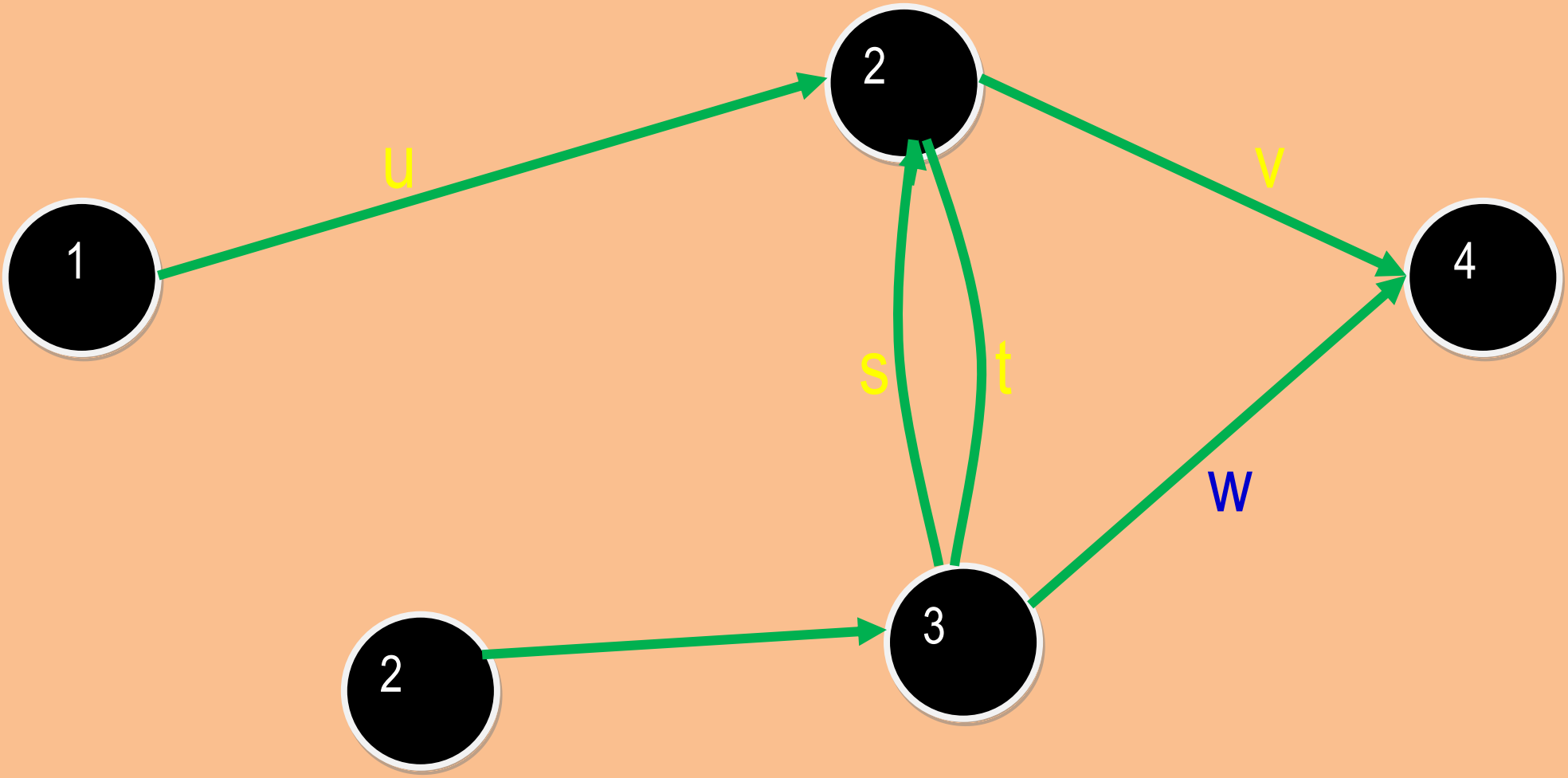
2- S reste inchangé



4- Relation entre arcs/arêtes et sommets

Arcs/arêtes adjacents

Deux arcs (arêtes) sont **adjacents** s'ils ont au moins une **extrémité commune**.



u,s,t,v sont des arcs **adjacents**

u et **w** sont des arcs **non adjacents**

Incidence Arc/sommet

Si le sommet x est l'extrémité initiale d'un arc u :

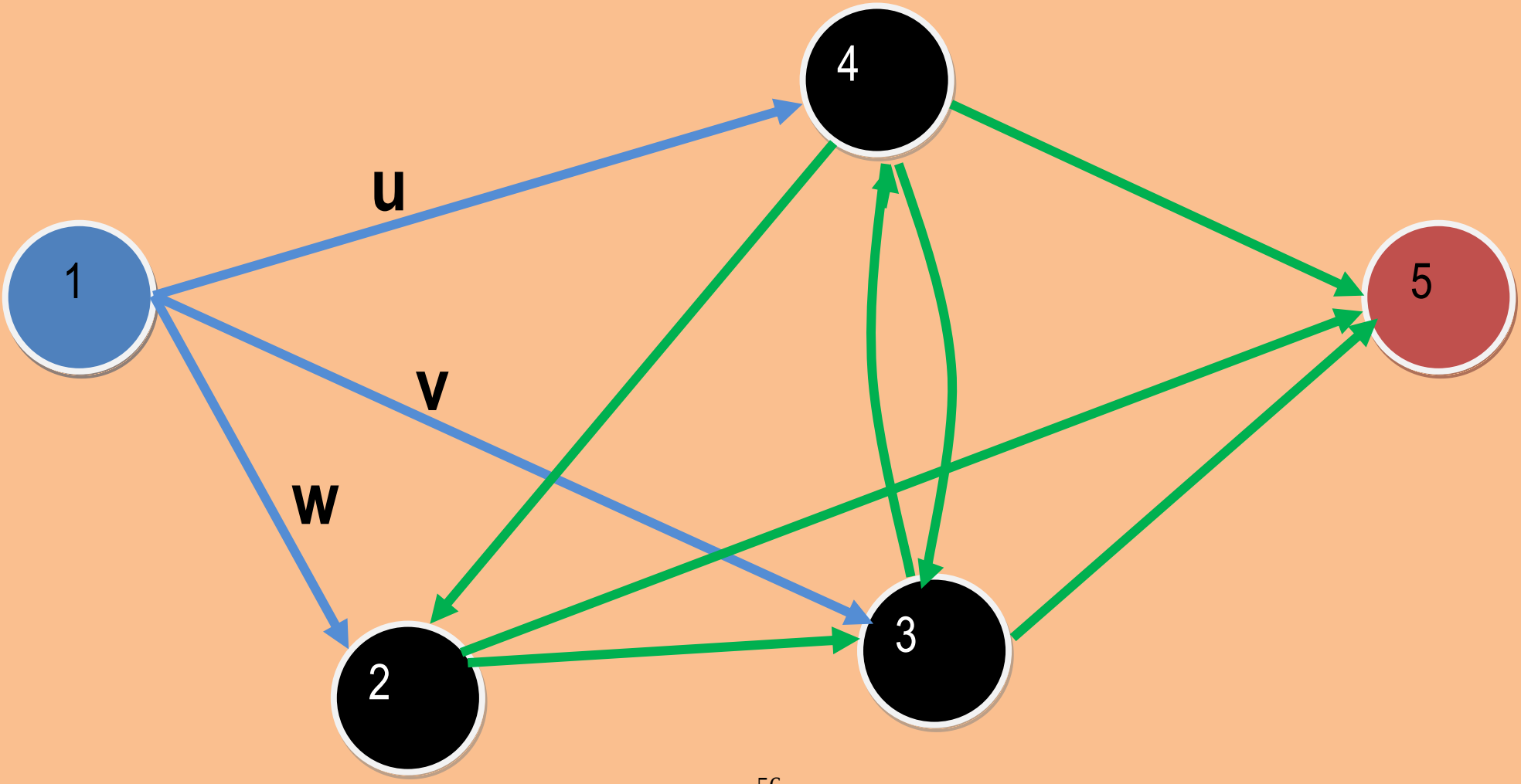
$u = x \rightarrow y$

on dit que u est **incident à x vers l'extérieur**.

Le **nombre d'arcs** ayant leur extrémité initiale en x est appelé **demi-degré extérieur** ou **demi-degré positif**.

On note $d^{\circ +}(x)$ le demi-degré positif de x.

Les arcs **incidents** au sommet 1 vers l'**extérieur sont : u,v** et **w**.



On écrira :

$$d^{\circ+}(1) = 3$$

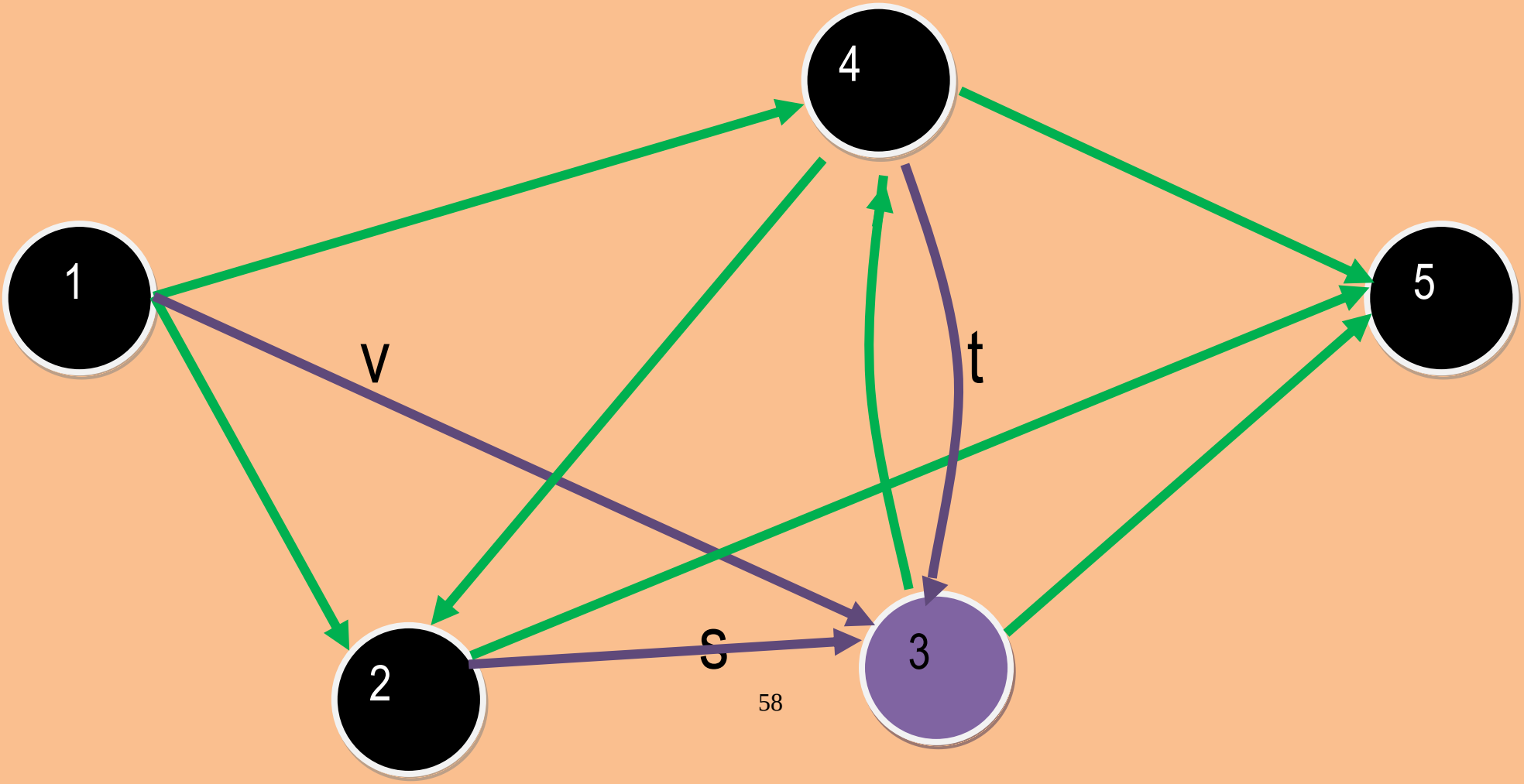
Aucun arc **n'est incident** au sommet 5 vers l'**extérieur**.

On écrira :

$$d^{\circ+}(5) = 0$$

On définit de même les notions :

- d'arc **incident vers l'intérieur**,
- de demi-degré **négatif**, noté **d°-(x).**



Les arcs incidents à 3 vers l'**intérieur** sont: s,t,v

$$\mathbf{d^{\circ -}(x) = 3}$$

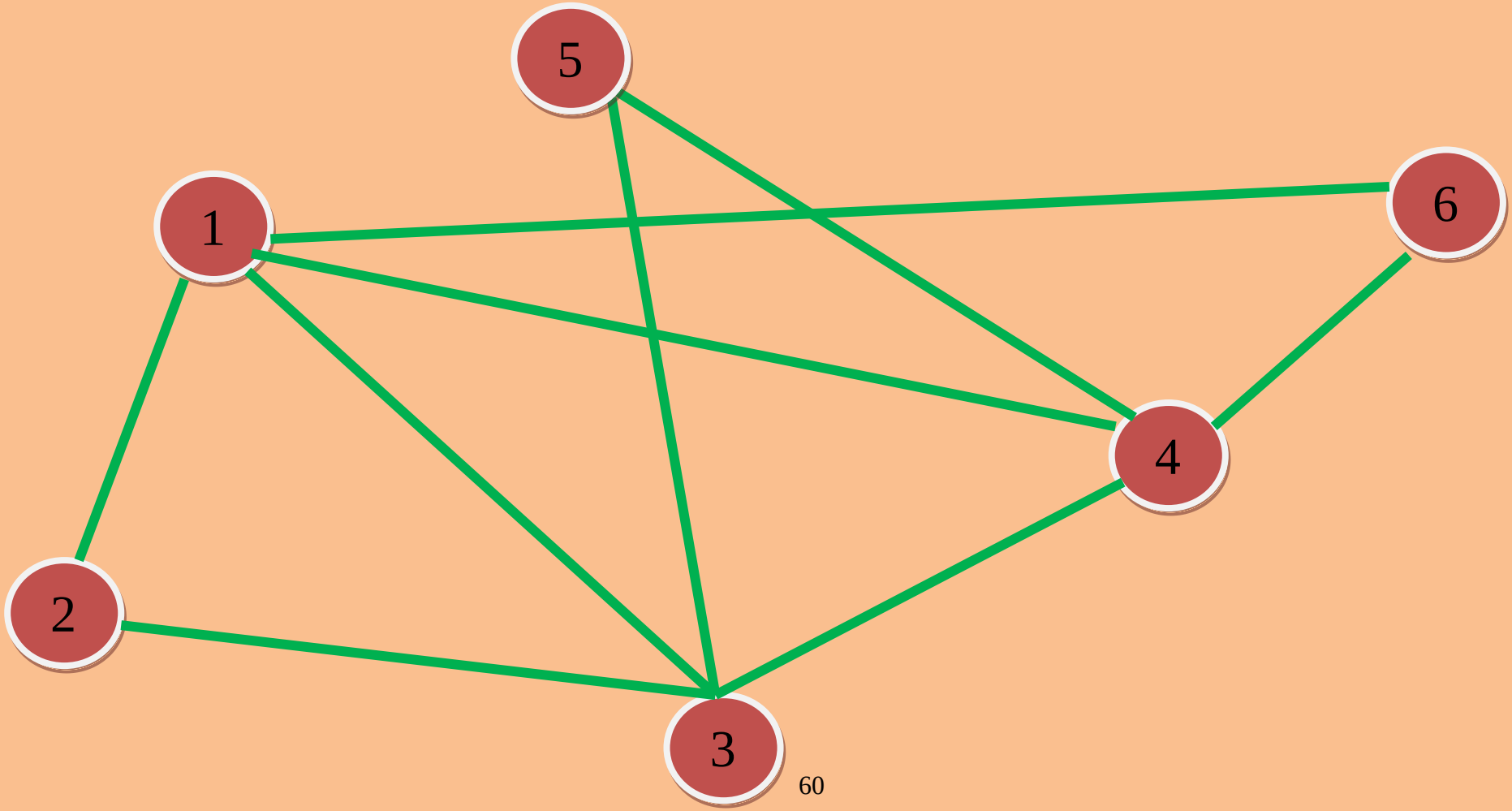
Dans le cas d'un **graphe orienté**, on :

$$\forall x \in S \quad \mathbf{d^{\circ}(x) = d^{\circ +}(x) + d^{\circ -}(x)}$$

d°(x) est appelé **degré** du sommet x.

Cas de graphe non orienté

Dans un graphe non orienté, le **degré** d'un sommet **x** est égal au **nombre d'arêtes** ayant pour extrémité **x**.



Calcul des degrés des sommets du graphe

x	**d°(x)**
1	4
2	2
3	4
4	4
5	2
6	2

5- Chemin et circuit

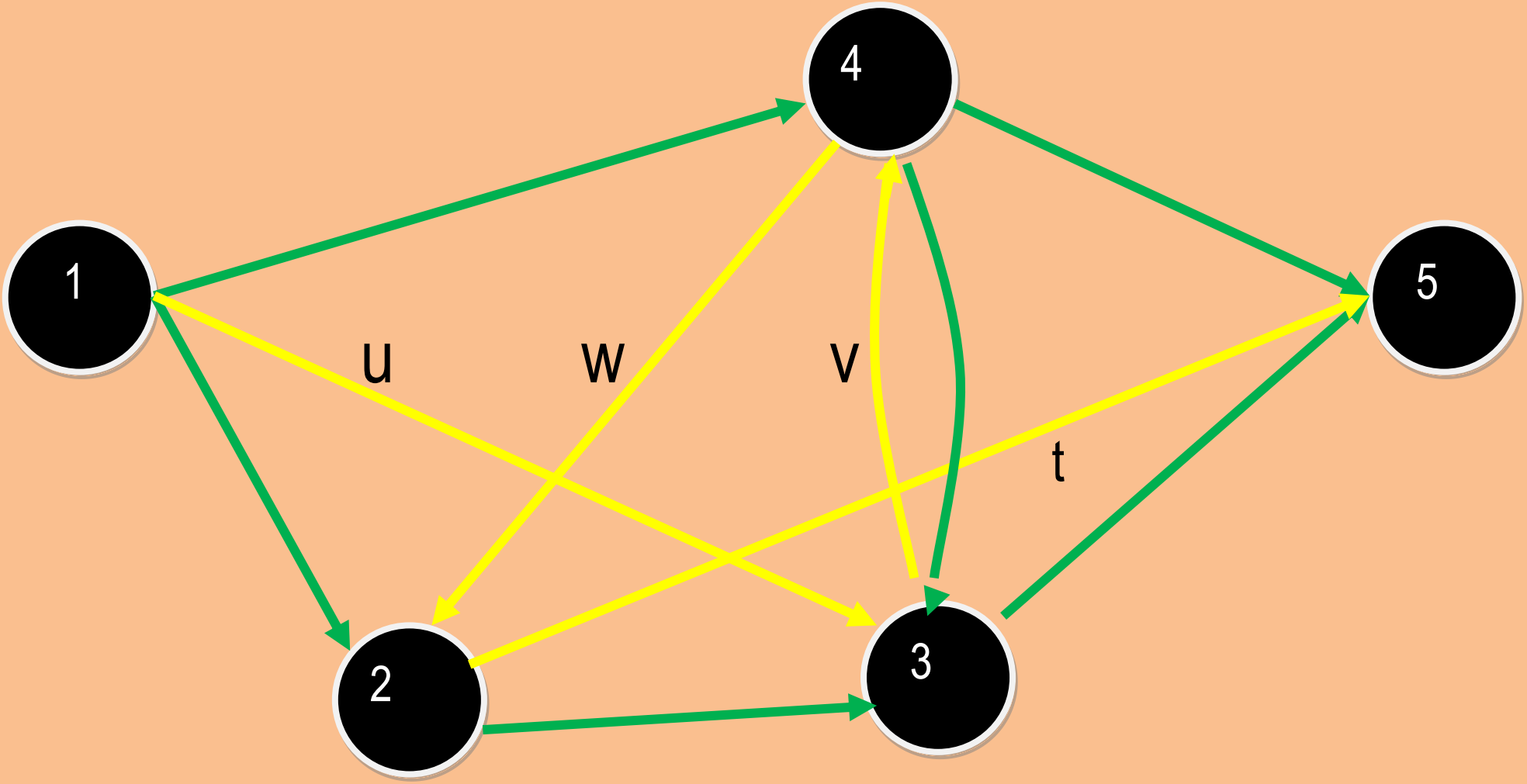
Dans un graphe orienté G, on appelle **chemin** de longueur λ, une **suite** de $(\lambda+1)$ sommets :

$$(S_0, S_1 , \ldots, S_\lambda)$$

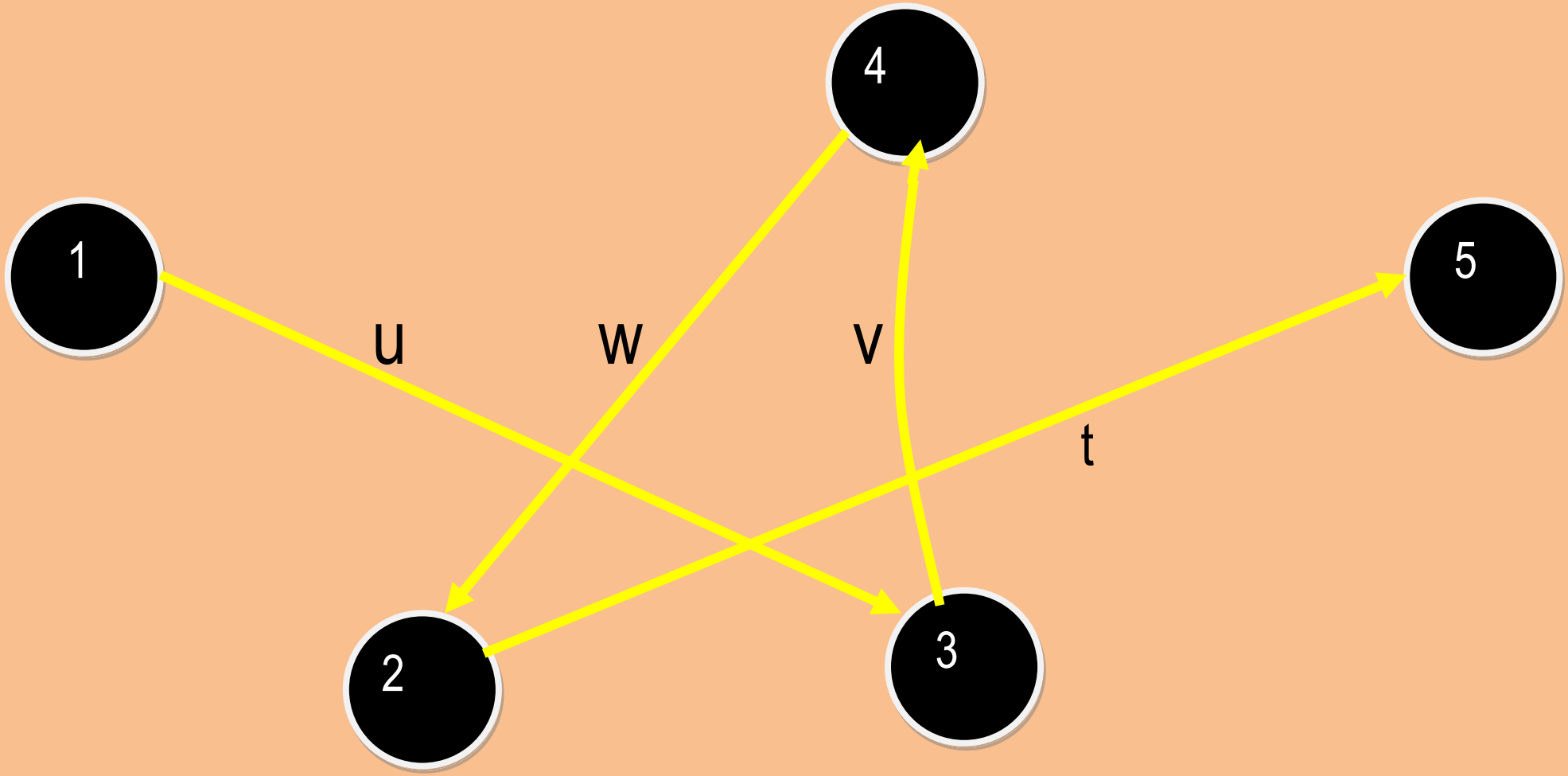
telle que :

$$\forall i \in [0, \lambda-1] \quad S_i \rightarrow S_{i+1} \in A$$

Le **chemin** (1,3,4,2,5) est de longueur 4 .

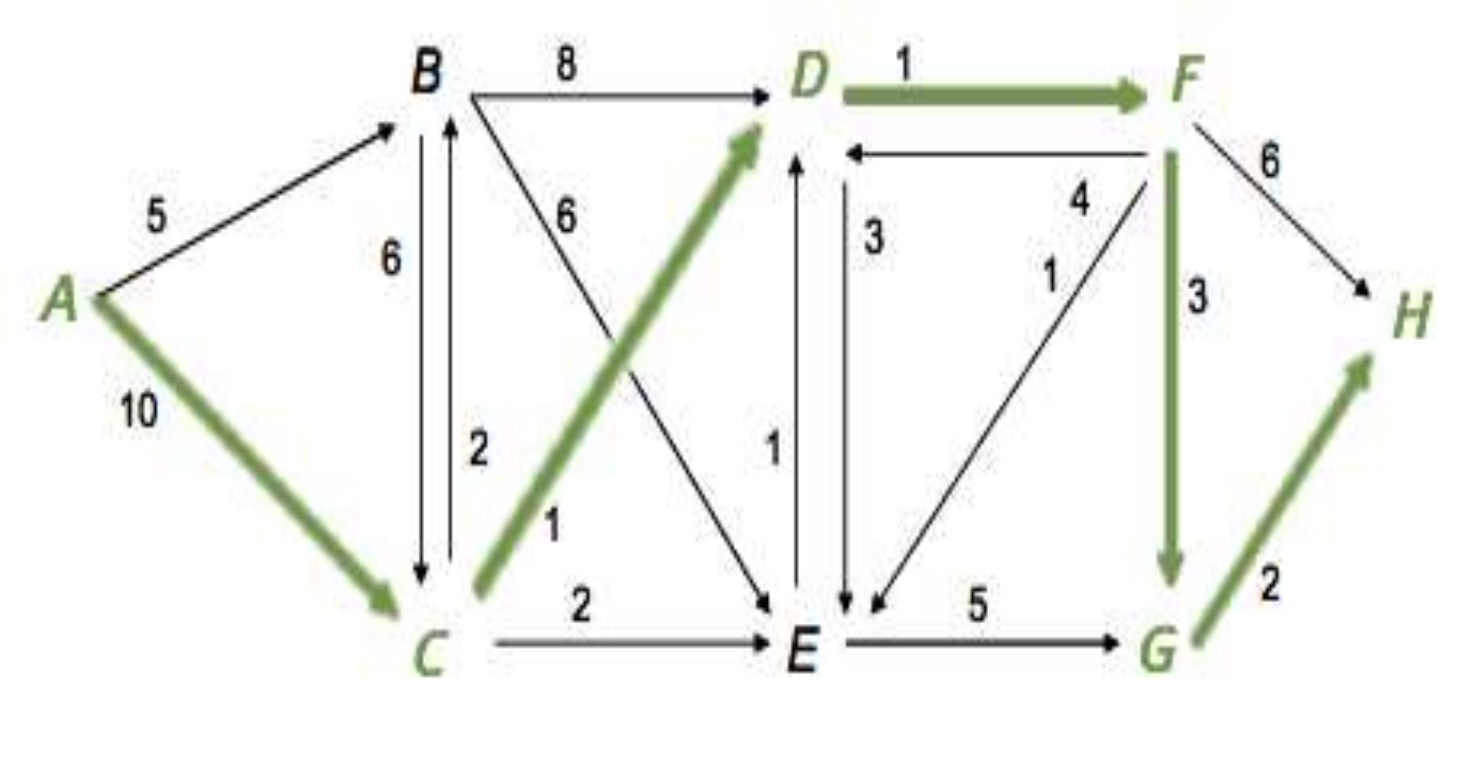


Ici, la **longueur** du chemin est exprimée en nombre d'arcs : u,v,w, t



Attention ce n’est pas le cas général !

Le chemin coloré ci-dessous est de **longueur** 17.



C’est le chemin **optimal** pour aller de **A** à **H** .

Son tracé est :

A→ C→ D→ F→G→ H

est appelé **routage**.

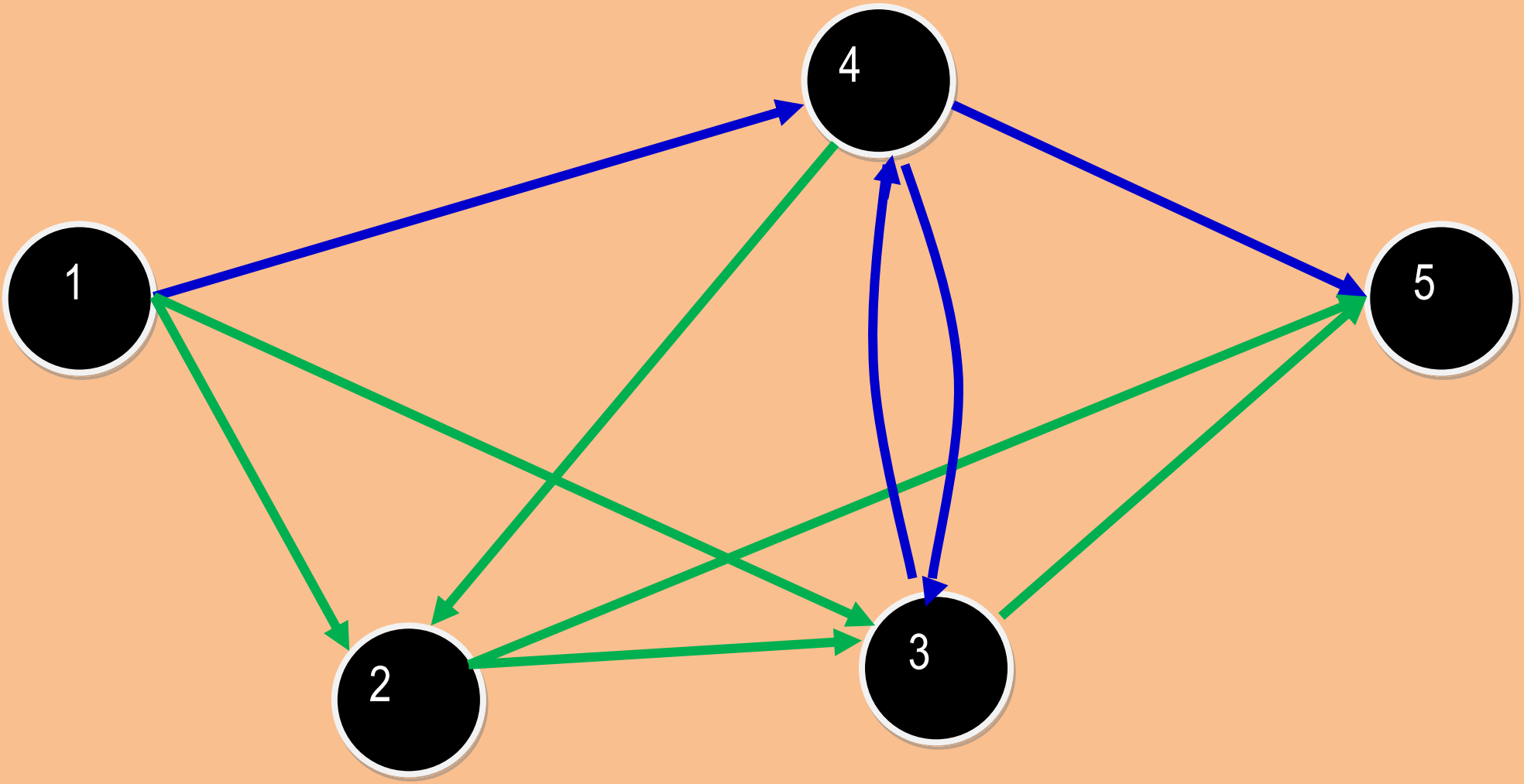
Chemin élémentaire

Un chemin est dit **élémentaire** s'il ne contient pas plusieurs fois le **même sommet**.

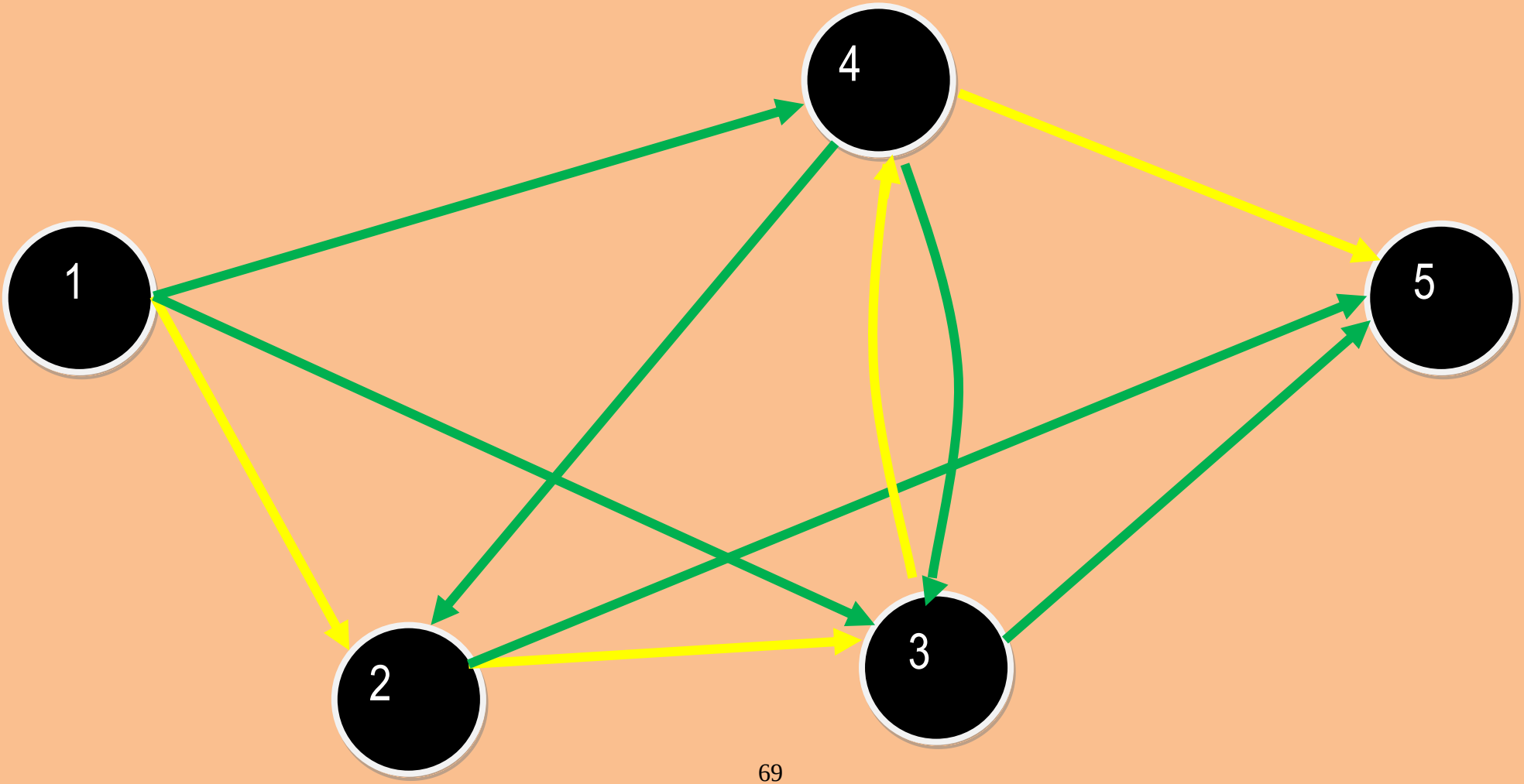
Chemin simple

Un chemin est dit **simple** s'il ne contient pas plusieurs fois le **même arc**.

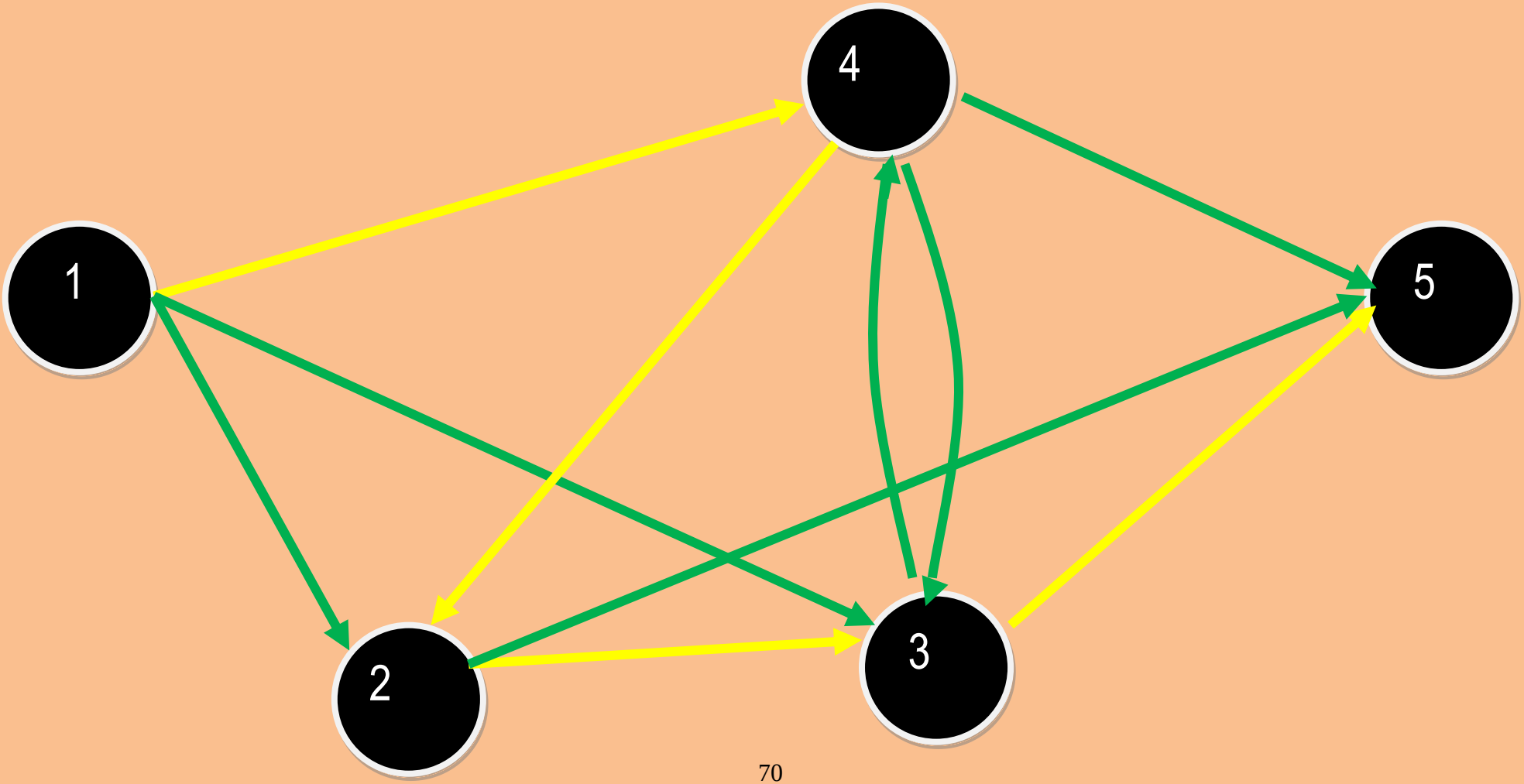
Chemin simple non élémentaire



Chemin simple élémentaire



Chemin élémentaire



Circuit d'un graphe

Dans un graphe orienté G, un chemin

$(S_0, S_1 , \ldots, S_\lambda)$

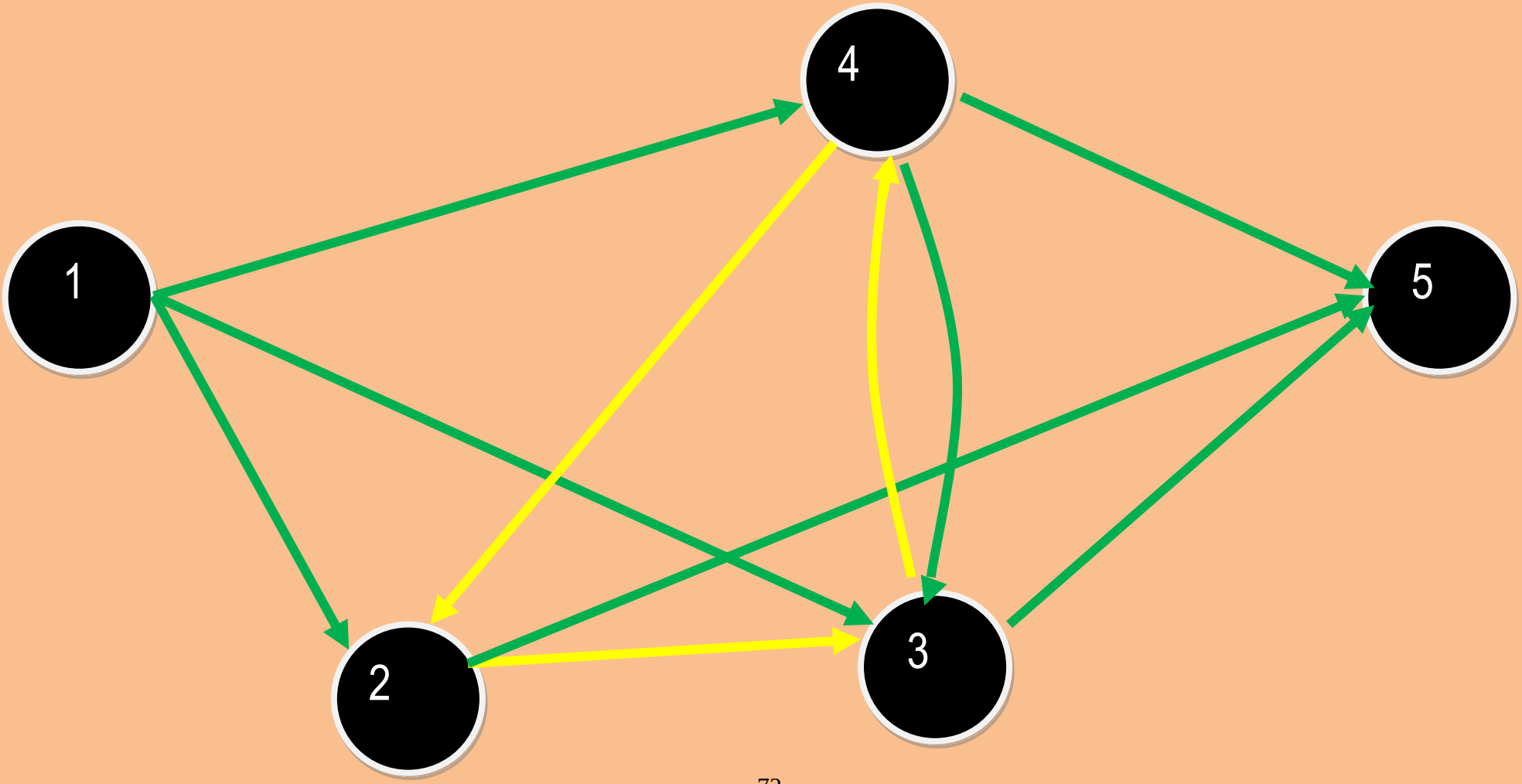
dont:

-les λ arcs sont distincts deux à deux,

- les deux sommets S_0 et S_λ coïncident,

est un **circuit**.

Circuit (4,2,3,4)



6- Chaîne et cycle

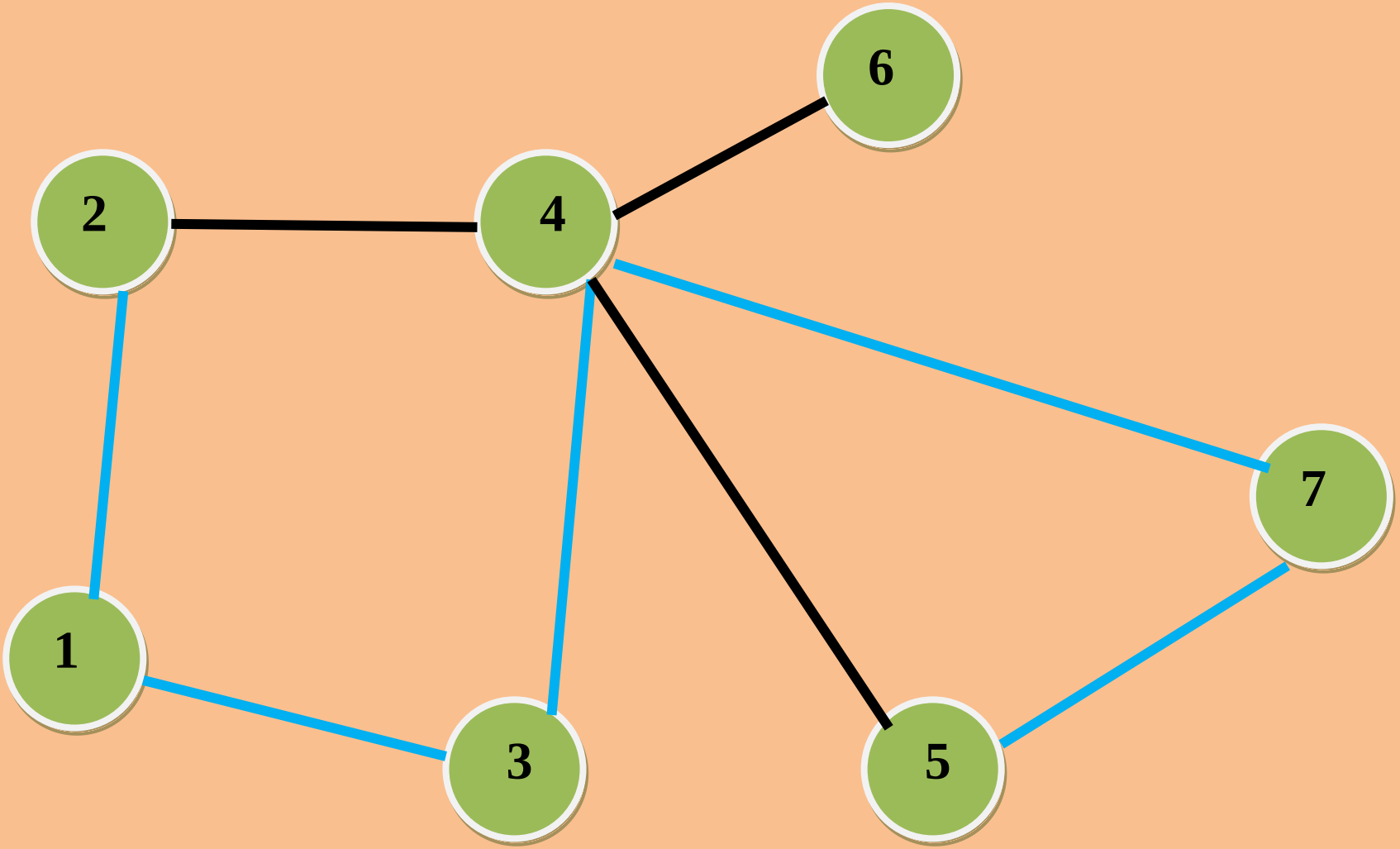
Dans un graphe non orienté G, on appelle **chaîne** de longueur λ, une **suite** de $(\lambda+1)$ sommets :

$$(S_0, S_1, \ldots, S_\lambda)$$

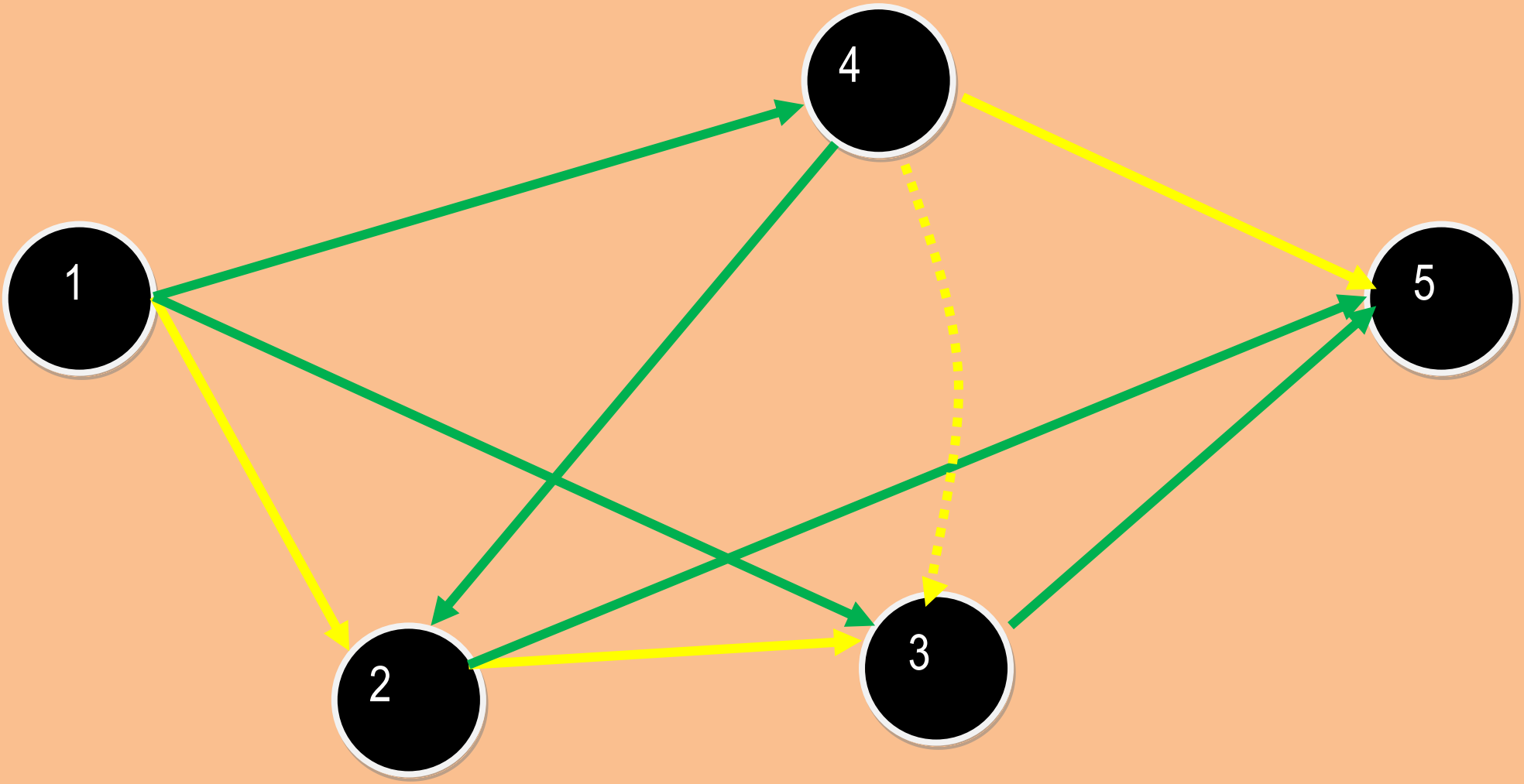
telle que:

$$\forall i \in [0, \lambda-1] \quad S_i - S_{i+1} \in A$$

Exemple : chaîne (2,1,3,4,7,5)



Chaîne (1,2,3,4,5) dans un graphe orientée



Chaîne élémentaire

Une chaîne est dite **élémentaire** si elle ne contient pas plusieurs fois le **même sommet**.

C’est une propriété **hamiltonienne**

Chaîne simple

Une chaîne est dite **simple** si elle ne contient pas plusieurs fois la **même arête**.

C’est une propriété **eulerienne.**

Cycle d'un graphe

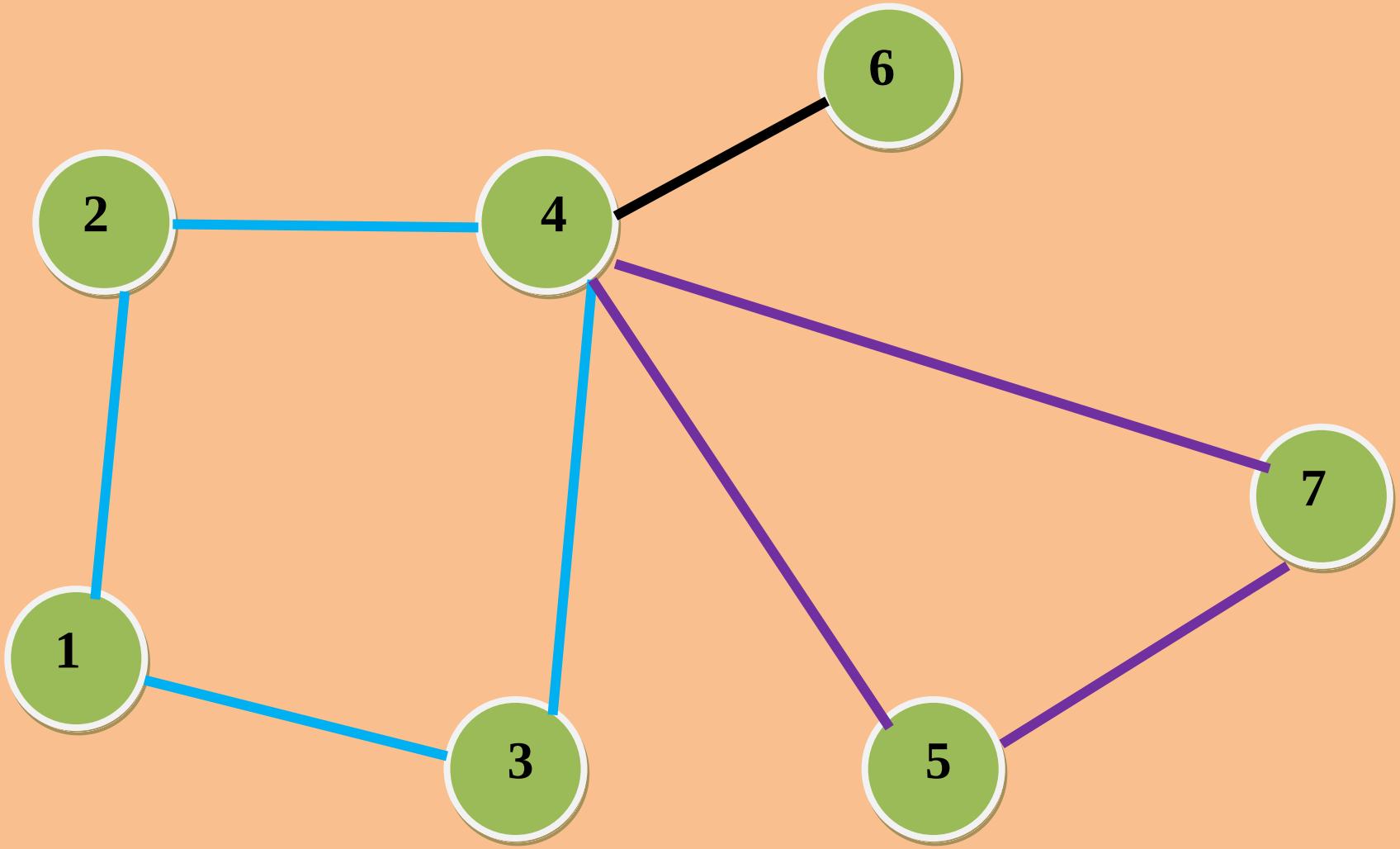
Dans un graphe non orienté G, une chaîne

$(S_0, S_1, \ldots, S_\lambda)$

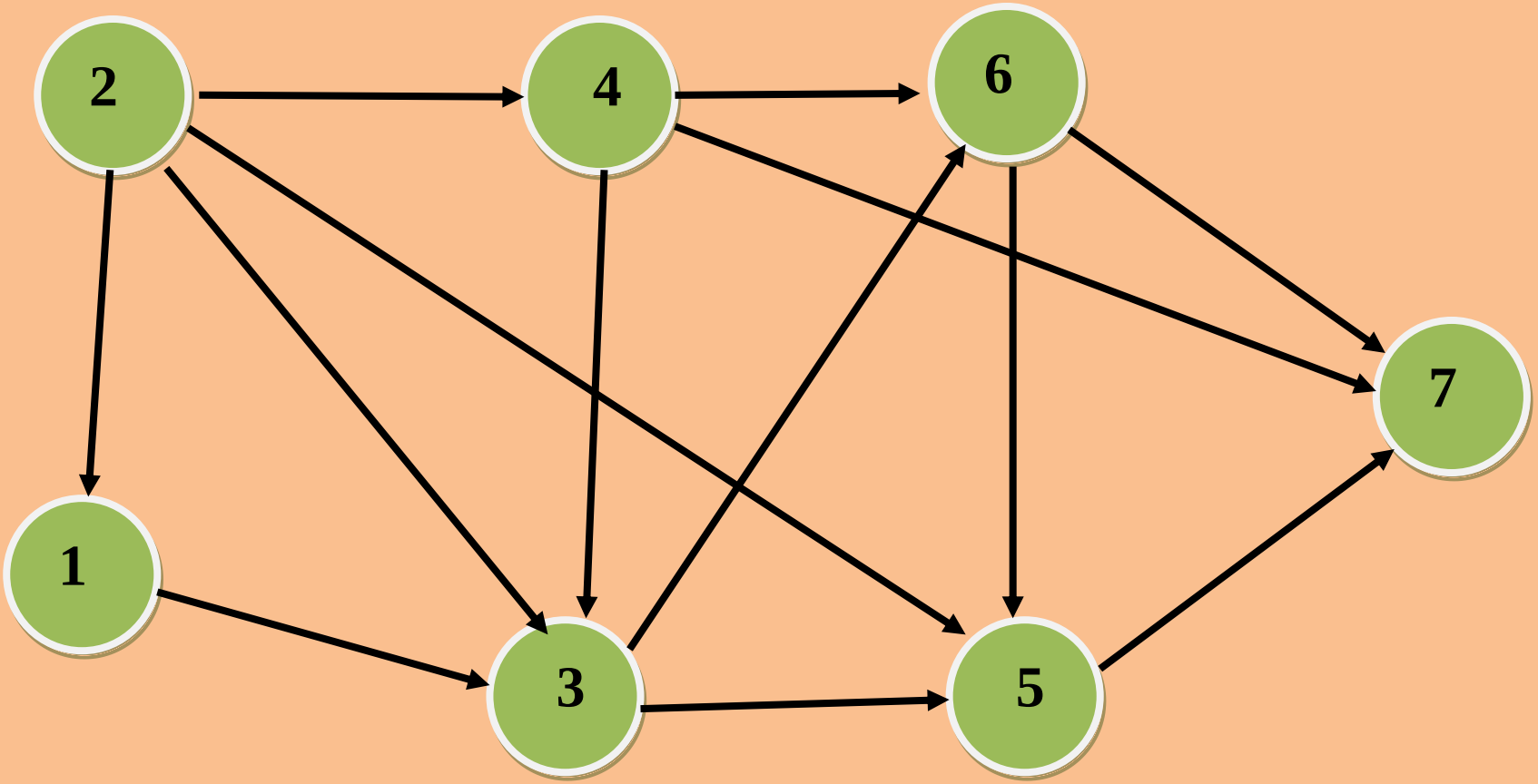
dont:

- les λ arêtes sont distinctes deux à deux,
- les deux sommets S_0 et S_λ coïncident,

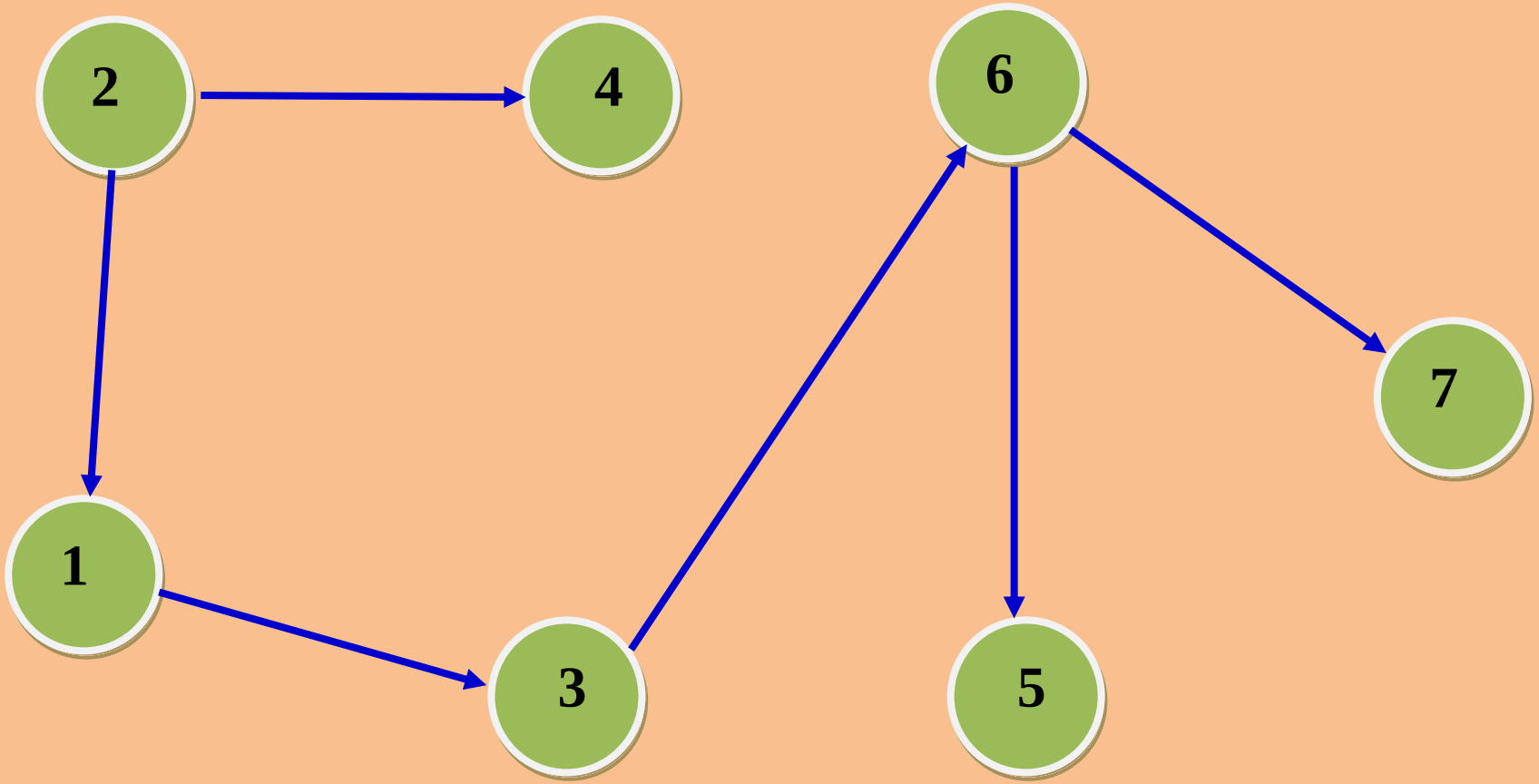
est un **cycle**.


6
2
4
7
1
3
5

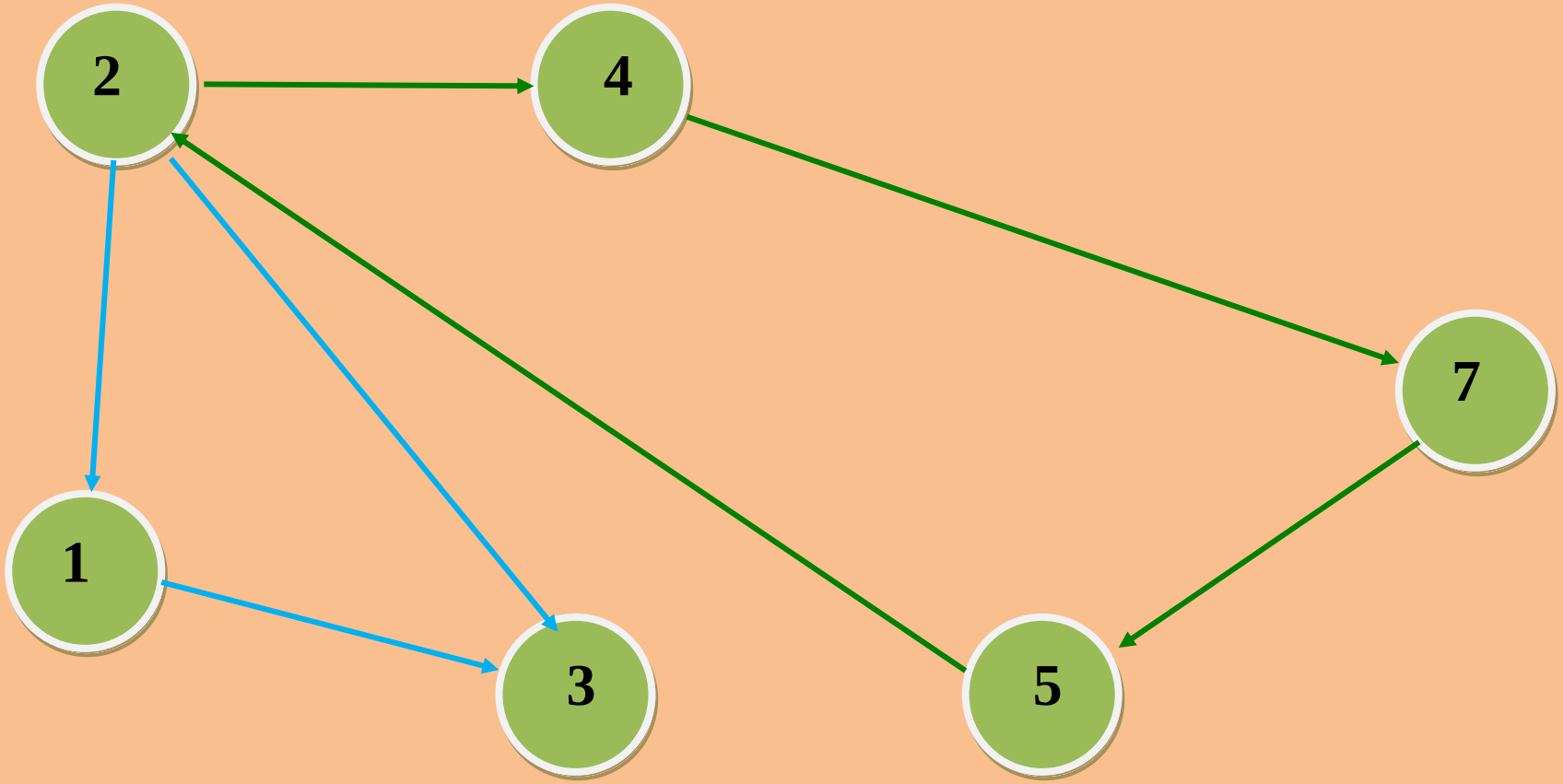
Soit le graphe orienté **visualisé** comme suit :



On peut en extraire les 2 graphes suivants :



Graphe sans cycle



Graphe compportant 2 cycles

7. Mesures sur un graphe

En plus du nombre de sommets et d'arcs/arêtes, deux mesures élémentaires sont utilisées pour définir les **attributs structurels** d'un graphe:

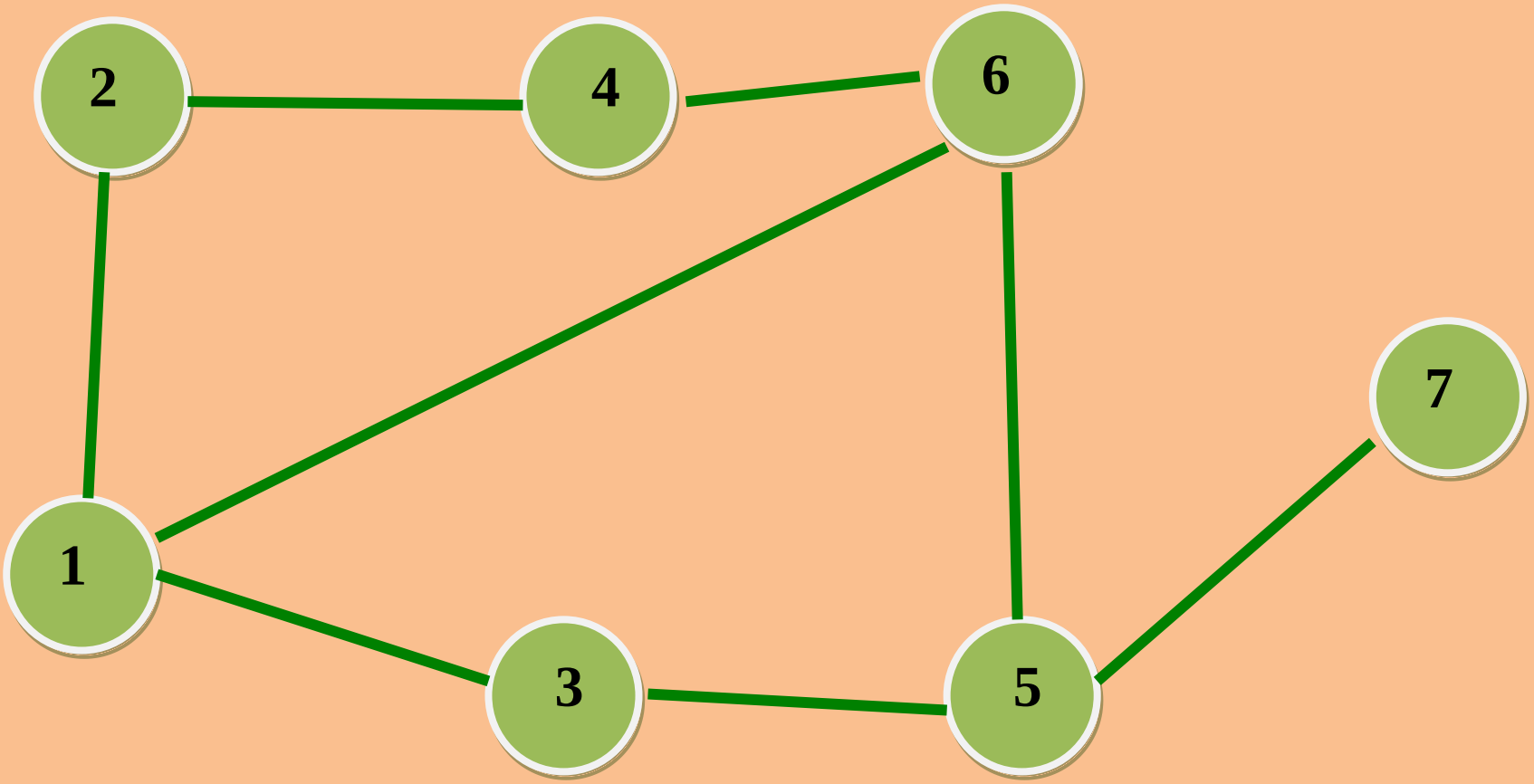
-son **diamètre**,
-son **nombre cyclomatique**

Diamètre d’un graphe

La longueur du **plus court chemin** entre deux sommets d’un graphe mesure la **distance topologique** entre ces deux sommets.

La distance topologique **d** entre les sommets les plus distants d’un graphe est appelé **diamètre** du graphe.

Plus le diamètre est élevé, plus faible sera le **degré de connexité** du graphe.



Dans ce graphe, le nombre d’arêtes entre les sommets les plus distants (2 et 7) en termes topologiques est **4**.

Le diamètre peut être obtenu grâce à une matrice de distances topologiques.

v	1	2	3	4	5	6	7
1	0	1	1	2	2	1	3
2	1	0	2	1	3	2	4
3	1	2	0	3	1	2	2
4	2	1	3	0	2	1	3
5	2	3	1	2	0	1	1
6	1	2	2	1	1	0	2
7	3	4	2	3	1	2	0

Nombre cyclomatique

Soit **n** la taille d'un graphe non orienté G = (S,A),

n = |S|.

Soit **m** le nombre d'arêtes de *G*,

m = |A|.

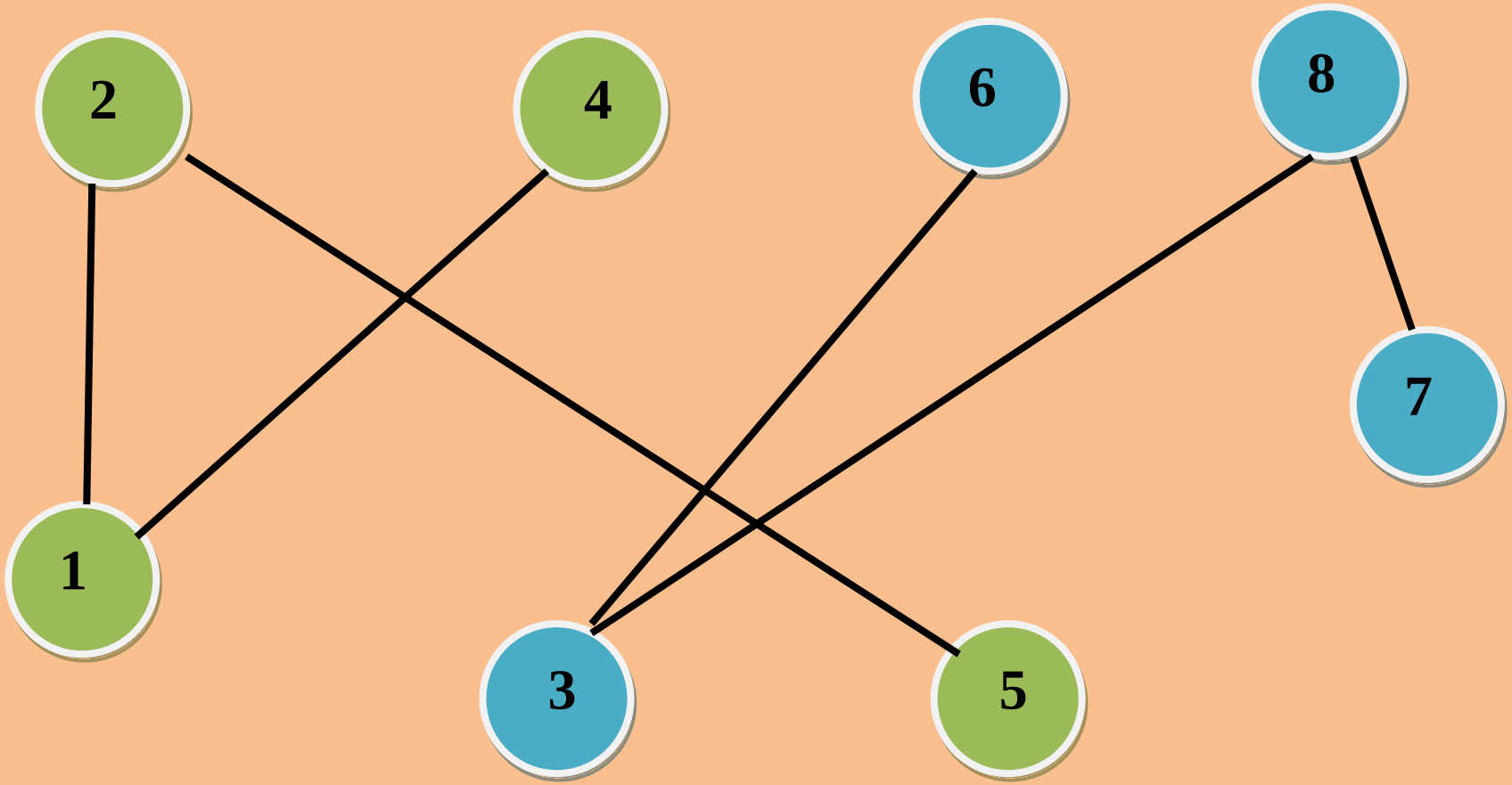
Le nombre **cyclomatique** de G est estimé à partir de **n**, **m** et du nombre **p** de ses composantes connexes:

$$\nu(G) = m-n+p$$

$\nu(G)$ estime le nombre de **cycles** que possède le graphe :

$$\nu(G) \geq \mathbf{0}$$

Calculer le nombre cyclomatique du graphe suivant:



$\nu(G) = m-n+p$

$= 6-8+2 = 0$

Un **pseudo-cycle** est une **chaîne :**

- dont les deux extrémités coïncident,
- le même arc (arête) pouvant être utilisé plusieurs fois.

Un **cycle** est une **chaîne** telle que :

- les deux extrémités coïncident,
- le même arc (arête) ne figure pas deux fois,

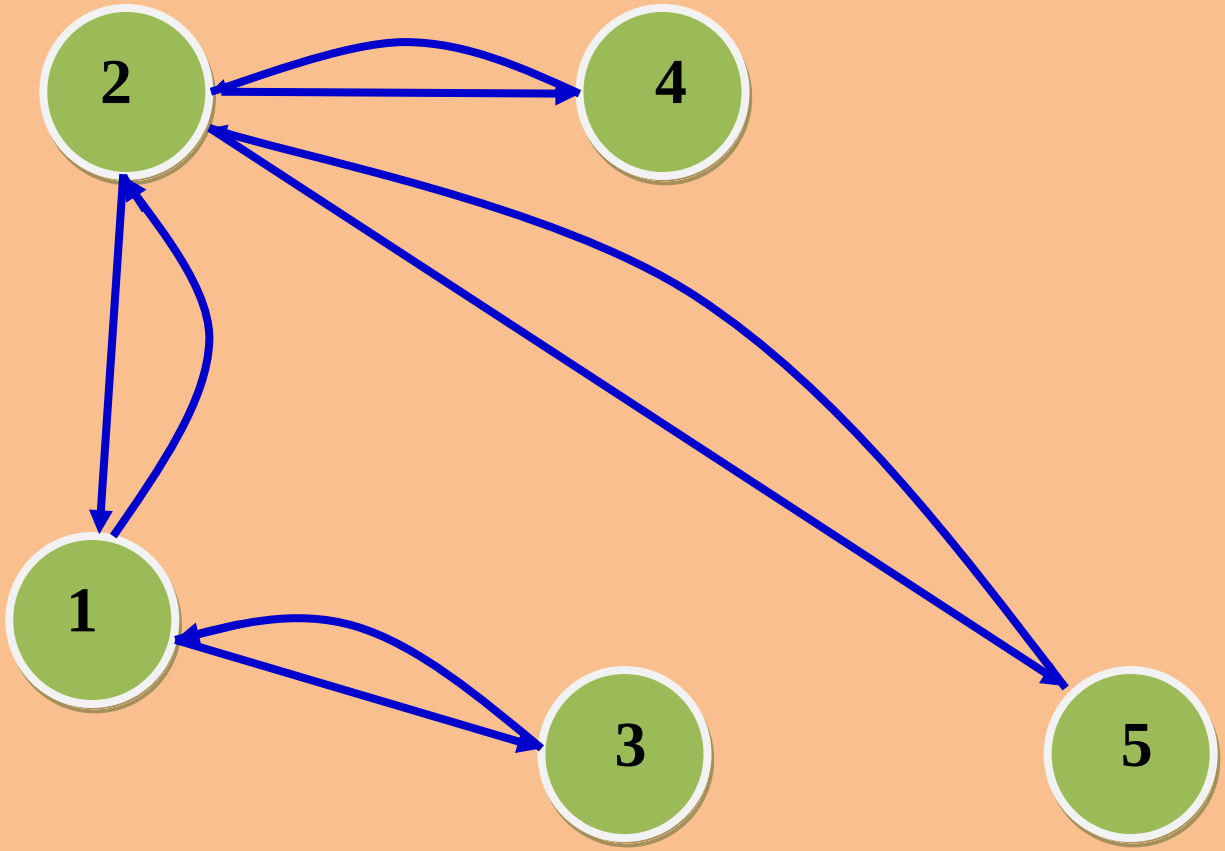
Un **cycle élémentaire** est un **cycle** ne rencontrant pas deux fois le même sommet (excepté le sommet initial qui coïncide nécessairement avec le sommet final).

8-Graphes particuliers

Graphe symétrique

Un graphe orienté G= (S,A) est dit **symétrique** si :

$$\forall x ; y \in S \bullet x \rightarrow y \in A \Rightarrow y \rightarrow x \in A$$



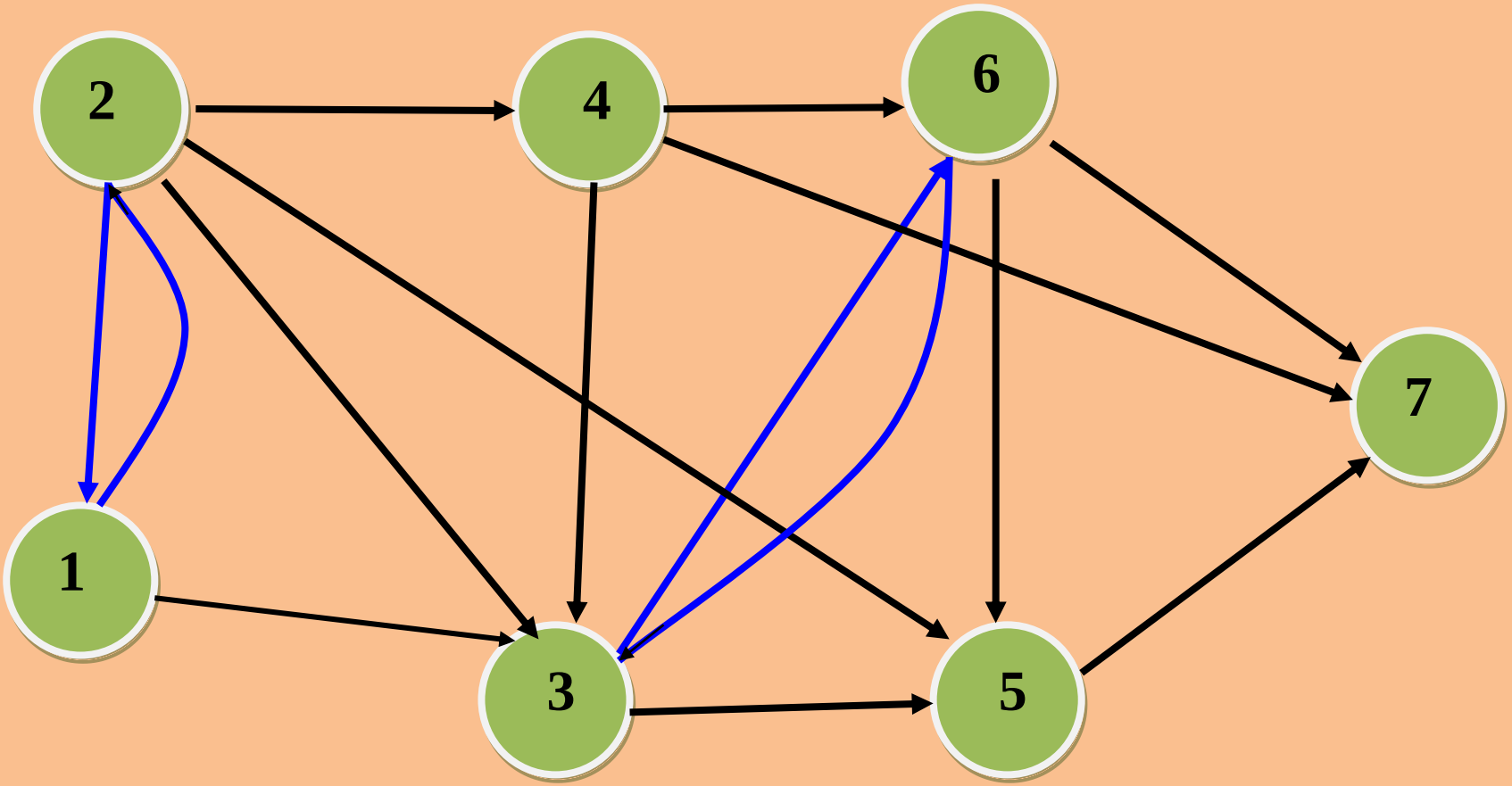
Graphe symétrique

Graphe antisymétrique

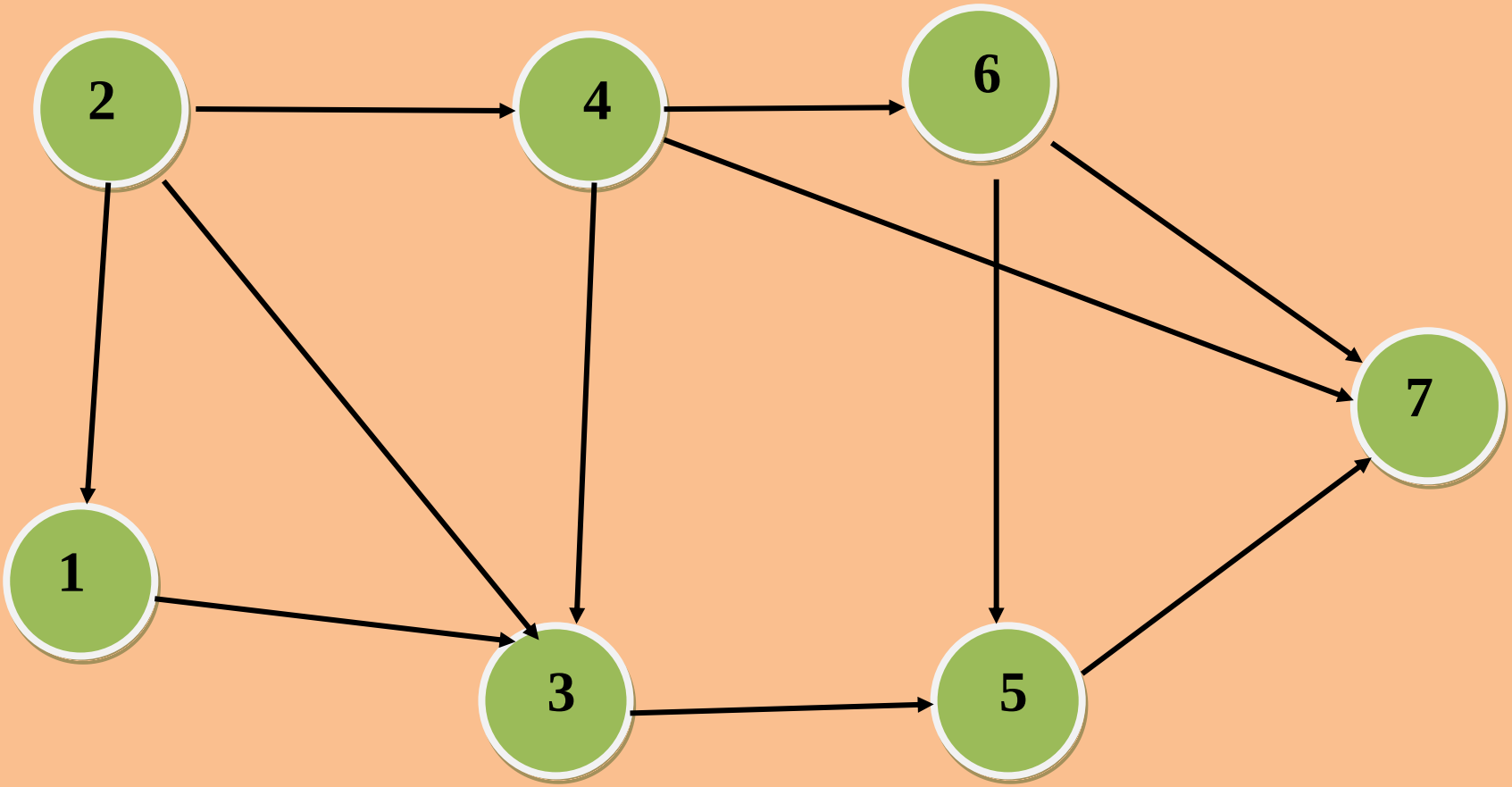
Un graphe orienté G= (S,A) est dit **antisymétrique** si :

$$\forall x ; y \in S \bullet x \rightarrow y \in A \Rightarrow y \rightarrow x \notin A$$

Graphe non antisymétrique



Graphe antisymétrique



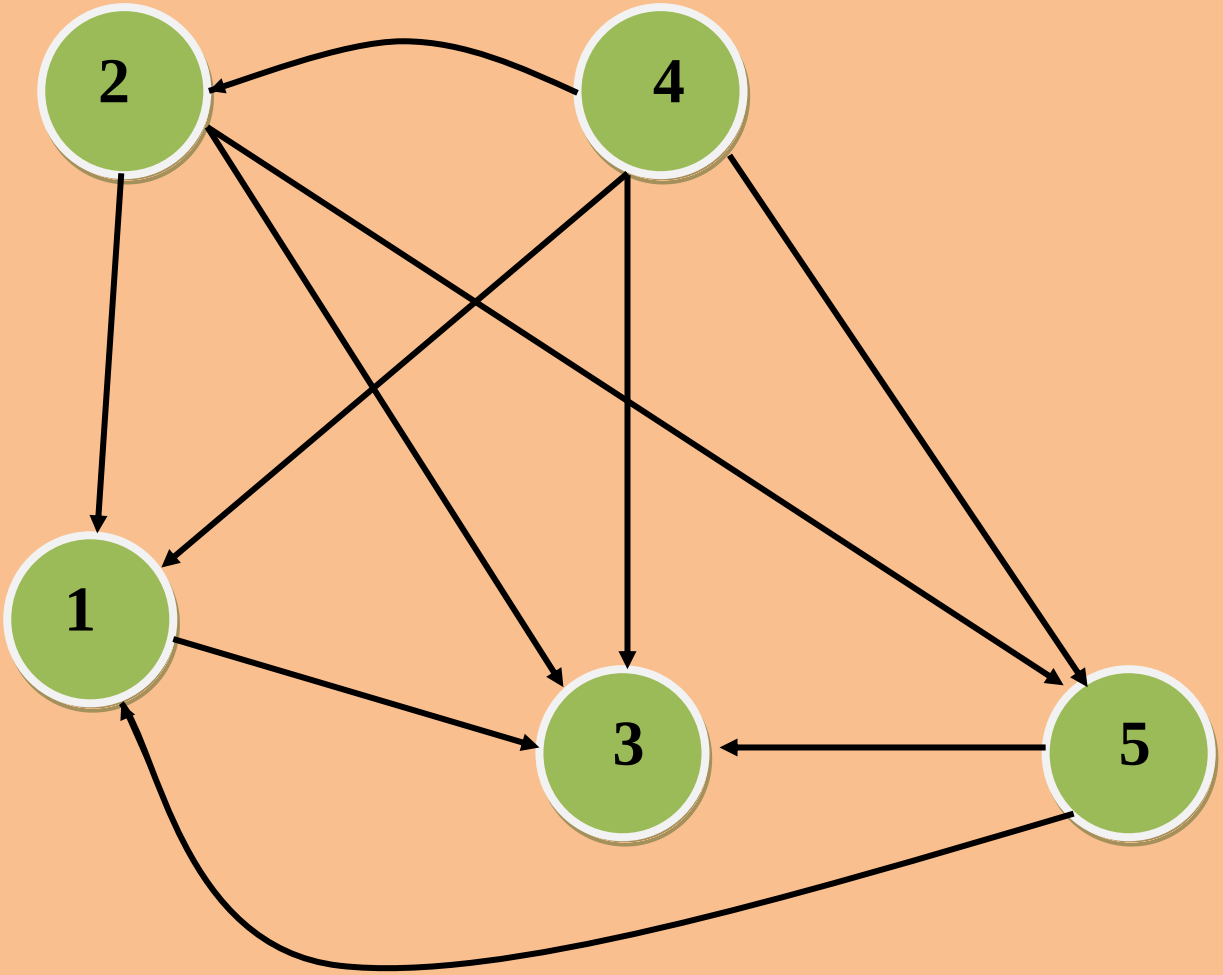
Graphe simplement complet

Un graphe orienté G= (S,A) est dit **simplement complet** si :

$$\forall x ; y \in S \bullet x \rightarrow y \in A \vee y \rightarrow x \in A$$

Un **graphe complet** à n sommets s'appelle une **n-clique**.

Exemple de 5-clique

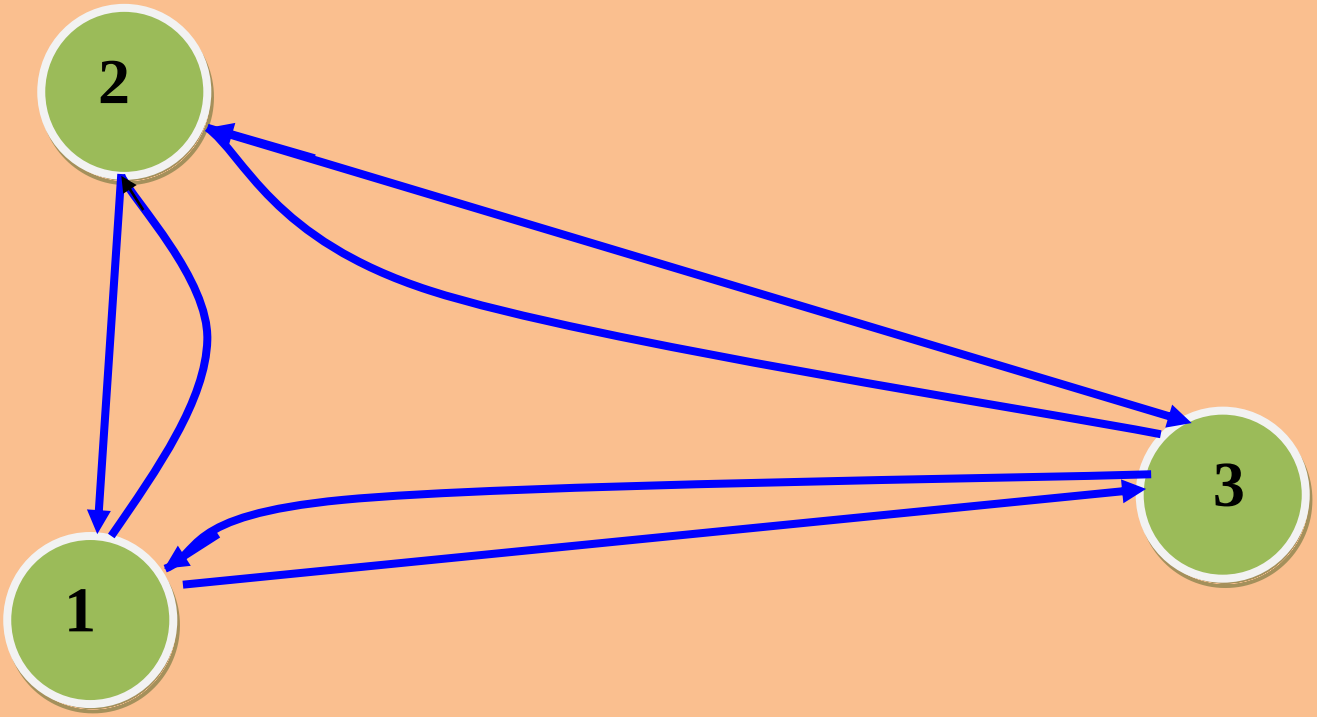


Graphe fortement complet

Un graphe orienté G= (S,A) est dit **fortement complet** si :

$$\forall x ; y \in S \bullet x \rightarrow y \in A \ \wedge \ y \rightarrow x \in A$$

Exemple



Graphe fortement complet

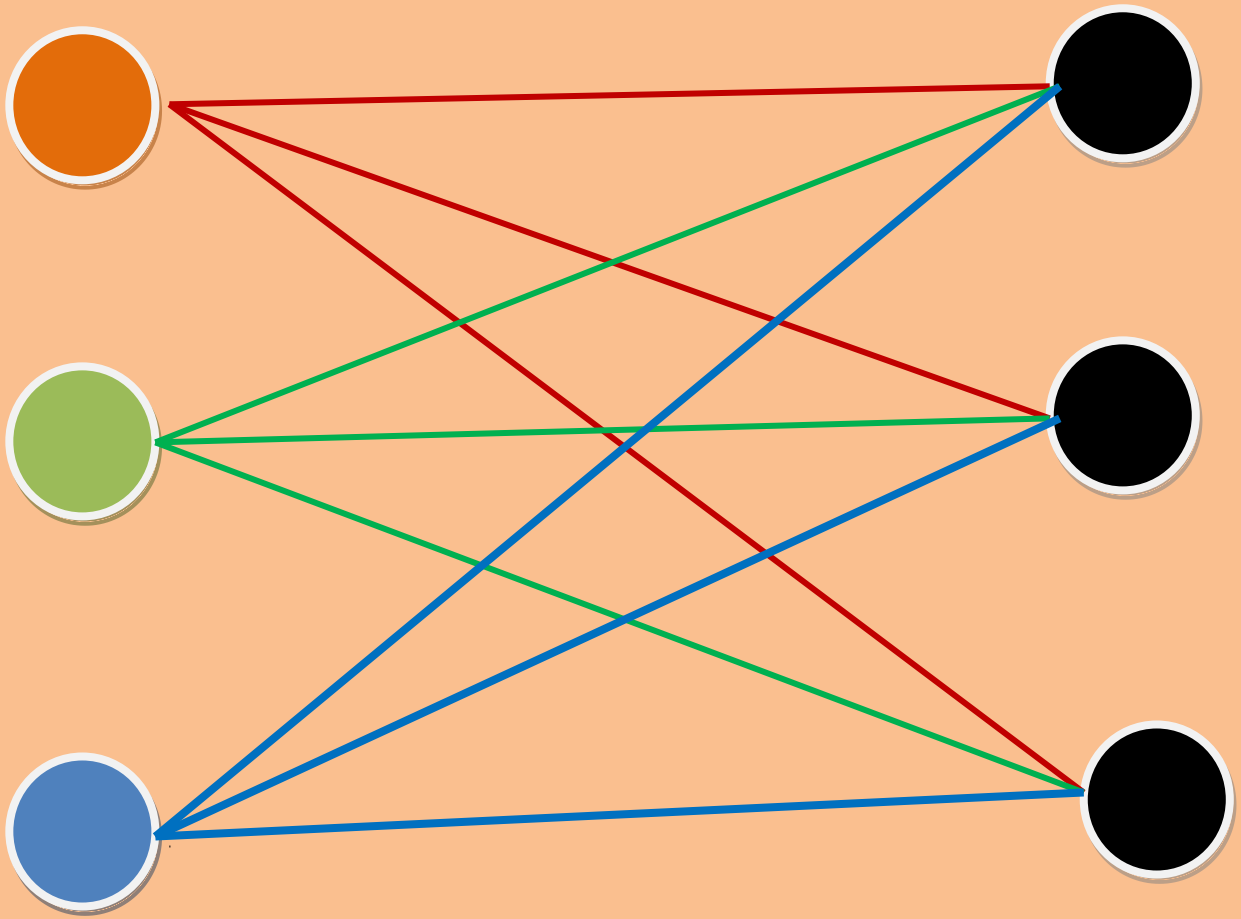
Graphe biparti

Un graphe est **biparti** si :

- l'ensemble de ses sommets peut être partitionné en **deux classes**

- deux sommets de la même classe ne sont **jamais adjacents**.

Graphe biparti complet : K_{33}



Le problème de l'**emploi du temps** se ramène à un problème de **coloration des arêtes** d'un biparti.

Le problème consiste à chercher le **plus petit** nombre de **créneaux** en lequel il est possible d'établir l'emploi du temps.

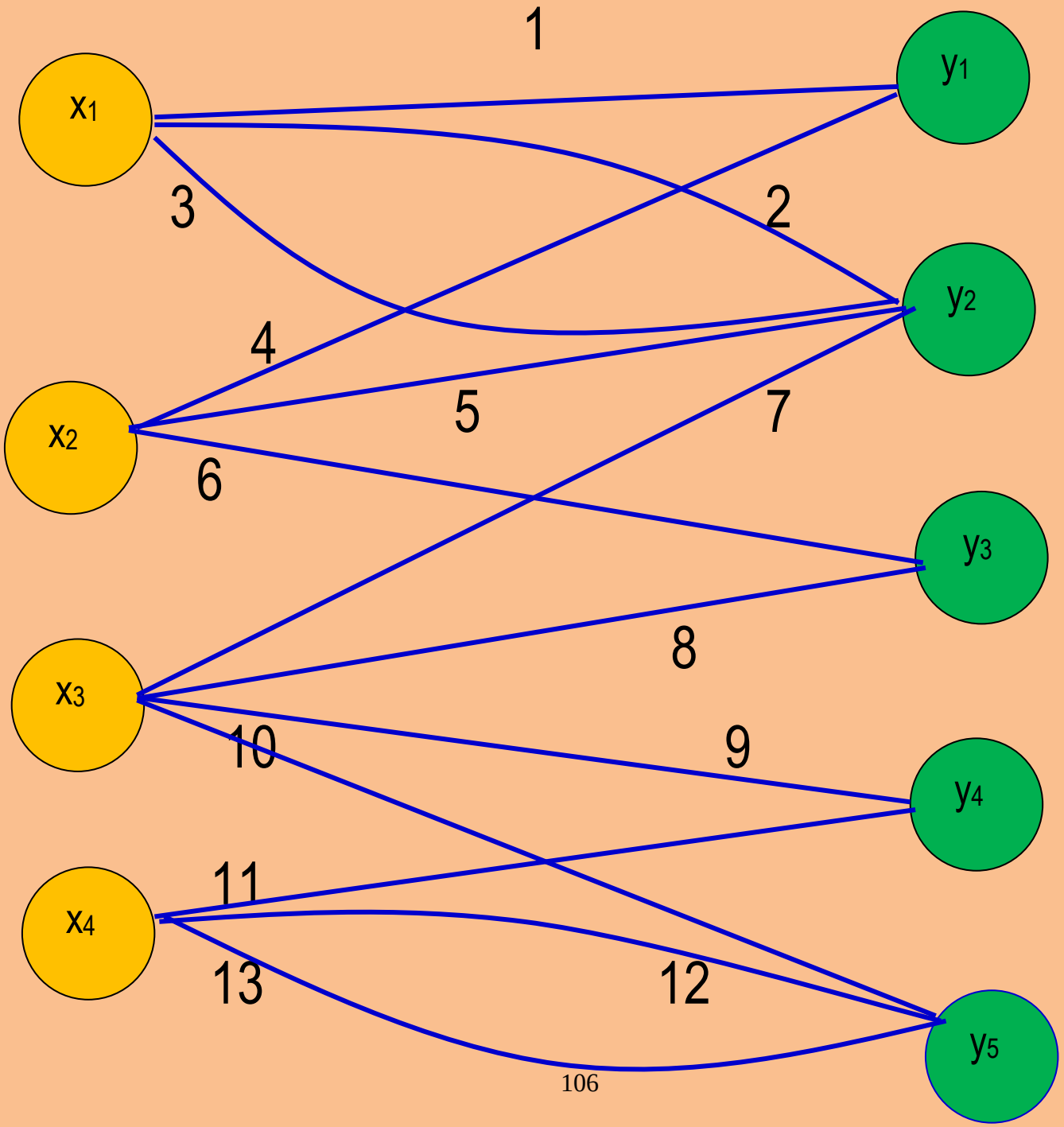
Exemple:

- 4 **enseignants** : 4 sommets

  $\{ x_1, x_2, x_3, x_4 \}$

- dispensent des séances de cours à : les **arêtes**

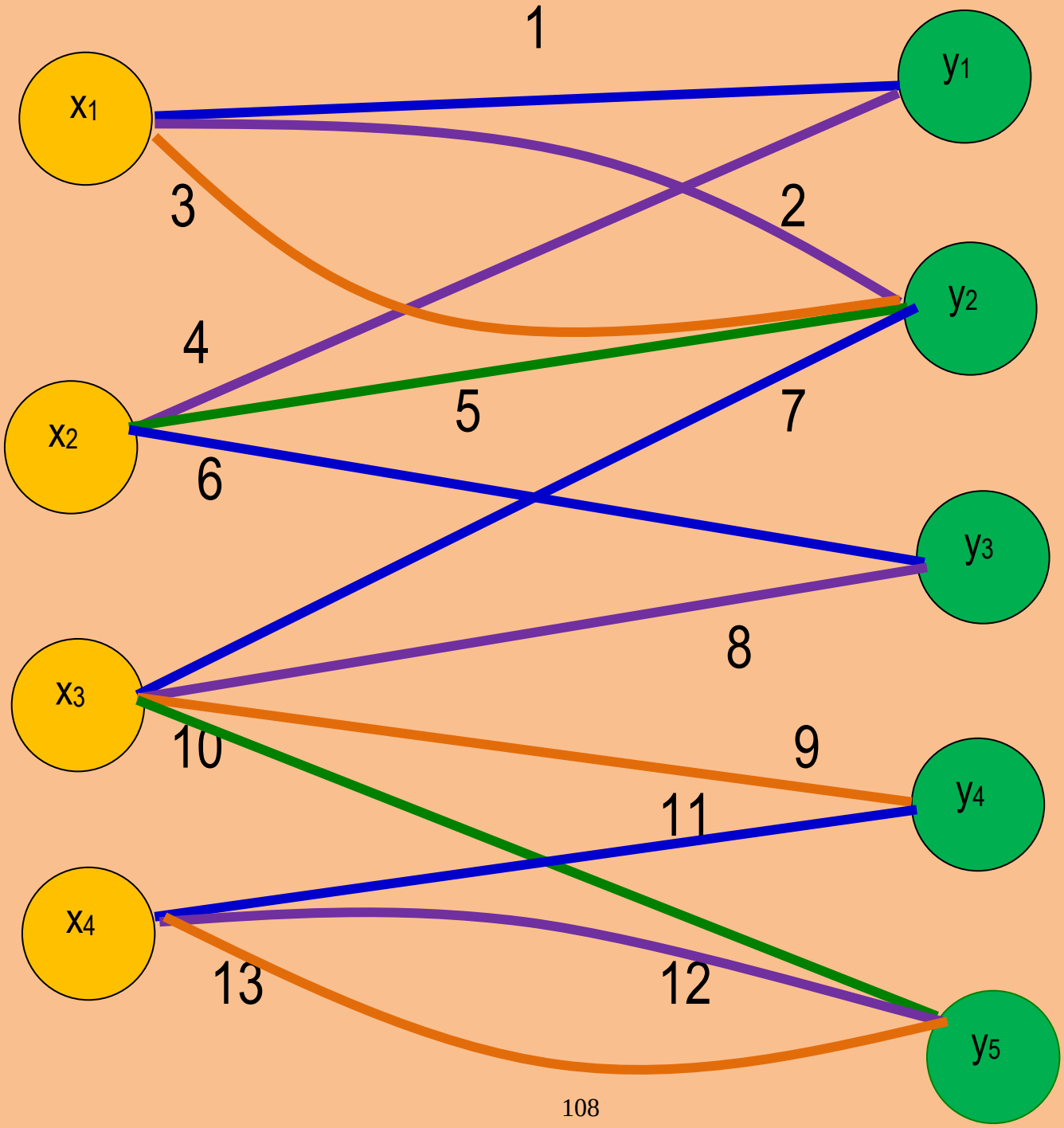
- 5 **classes** : 5 sommets $\{ y_1, y_2, y_3, y_4, x_5 \}$


1
x1
y1
3
2
y2
4
x2
5
7
6
y3
8
x3
10
9
y4
11
x4
13
12
y5

Une solution :

Ici, **quatre** couleurs au **minimum** sont nécessaires pour colorier le biparti :

{1,6,7,11} , {5,10} , {2,4,8,12} , {3,9,13}


1
x1
y1
3
2
y2
4
x2
5
7
6
y3
8
x3
10
9
y4
11
x4
13
12
y5

Connexité d'un graphe

Un graphe orienté est dit **fortement connexe** si :

$\forall$ (x,y) $\in$ S x S, il existe :

- chemin reliant x vers y,
- chemin reliant y vers x.

Composante fortement connexe

On appelle **composante fortement connexe** d'un graphe orienté un sous-graphe fortement connexe **maximal**.

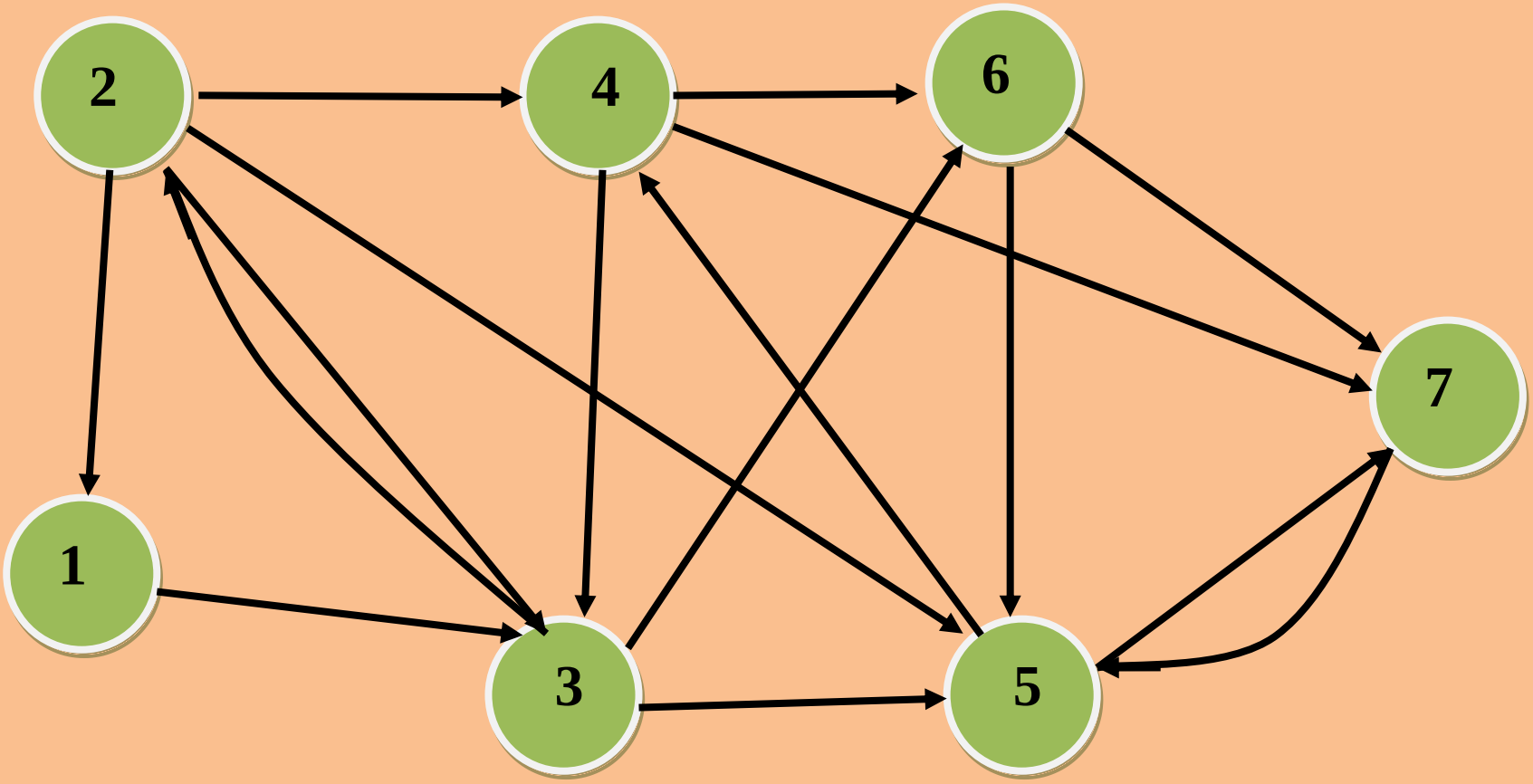
Un sous-graphe $G_i = (S_i, A_i)$ fortement connexe est dit **maximal** si

$\forall\ G_k = (S_k, A_k)$ on ait :

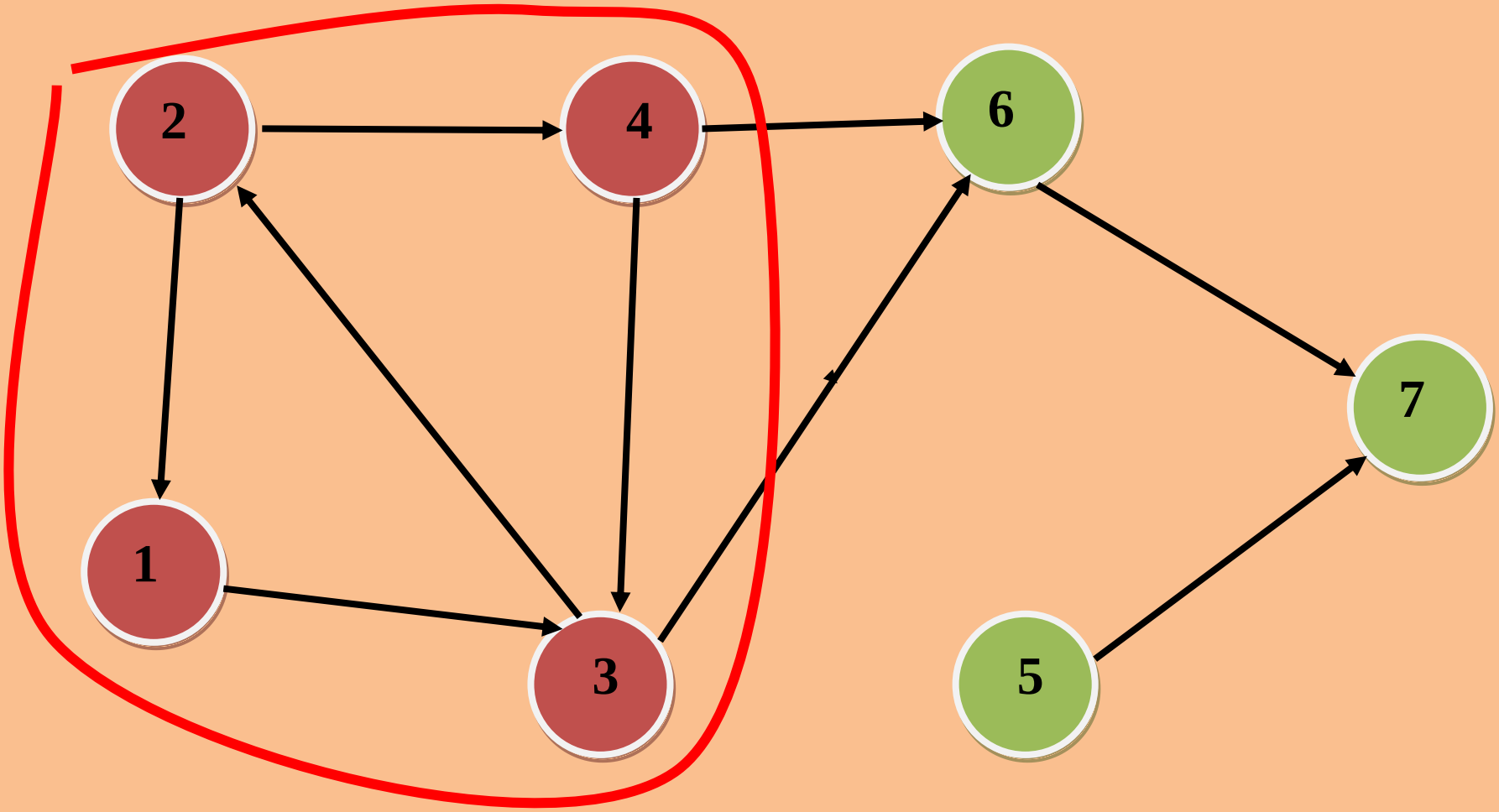
G_k est une sous graphe fortement connexe de G

$$\Rightarrow \quad S_k \subseteq S_i$$

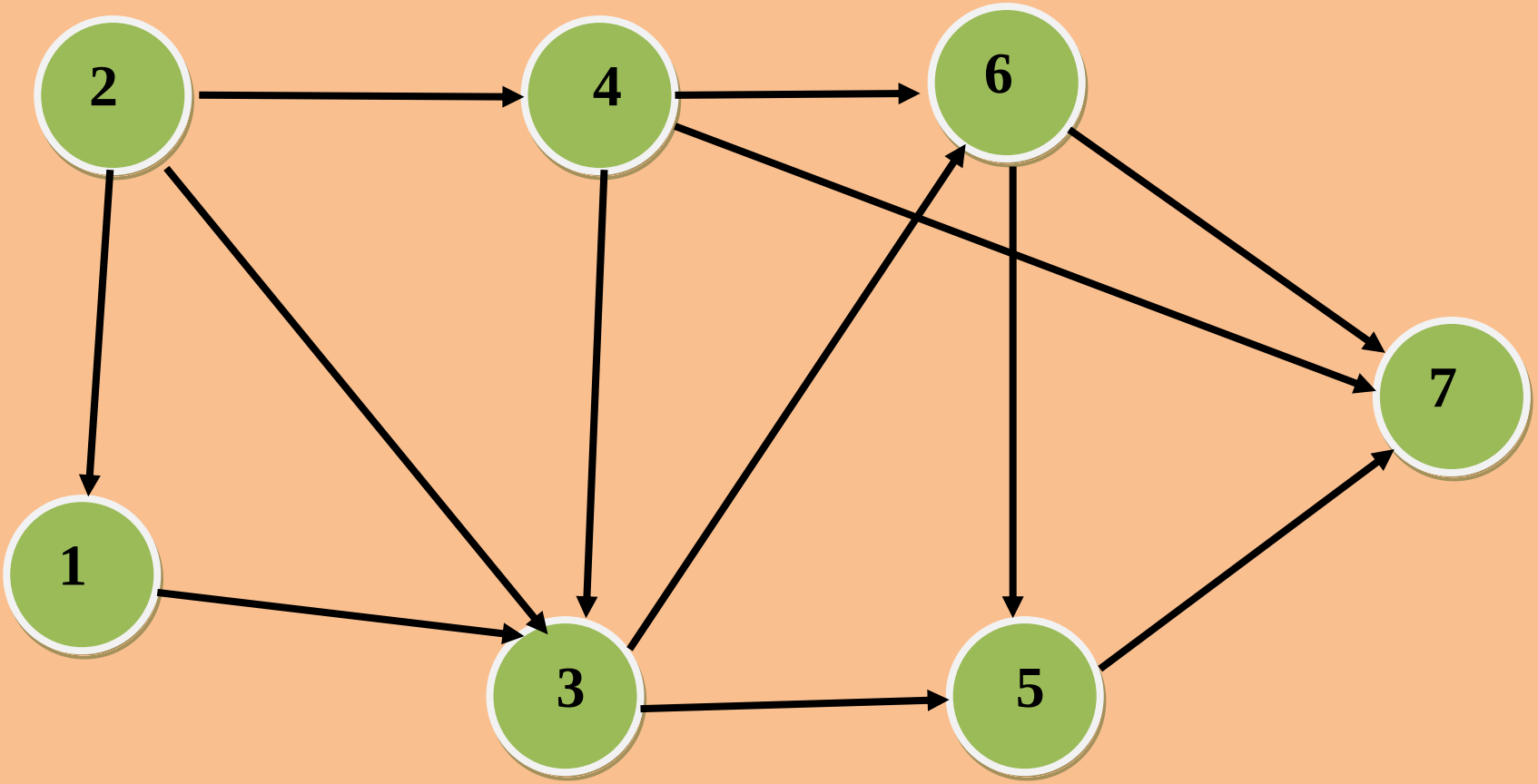
Un graphe orienté **fortement connexe**



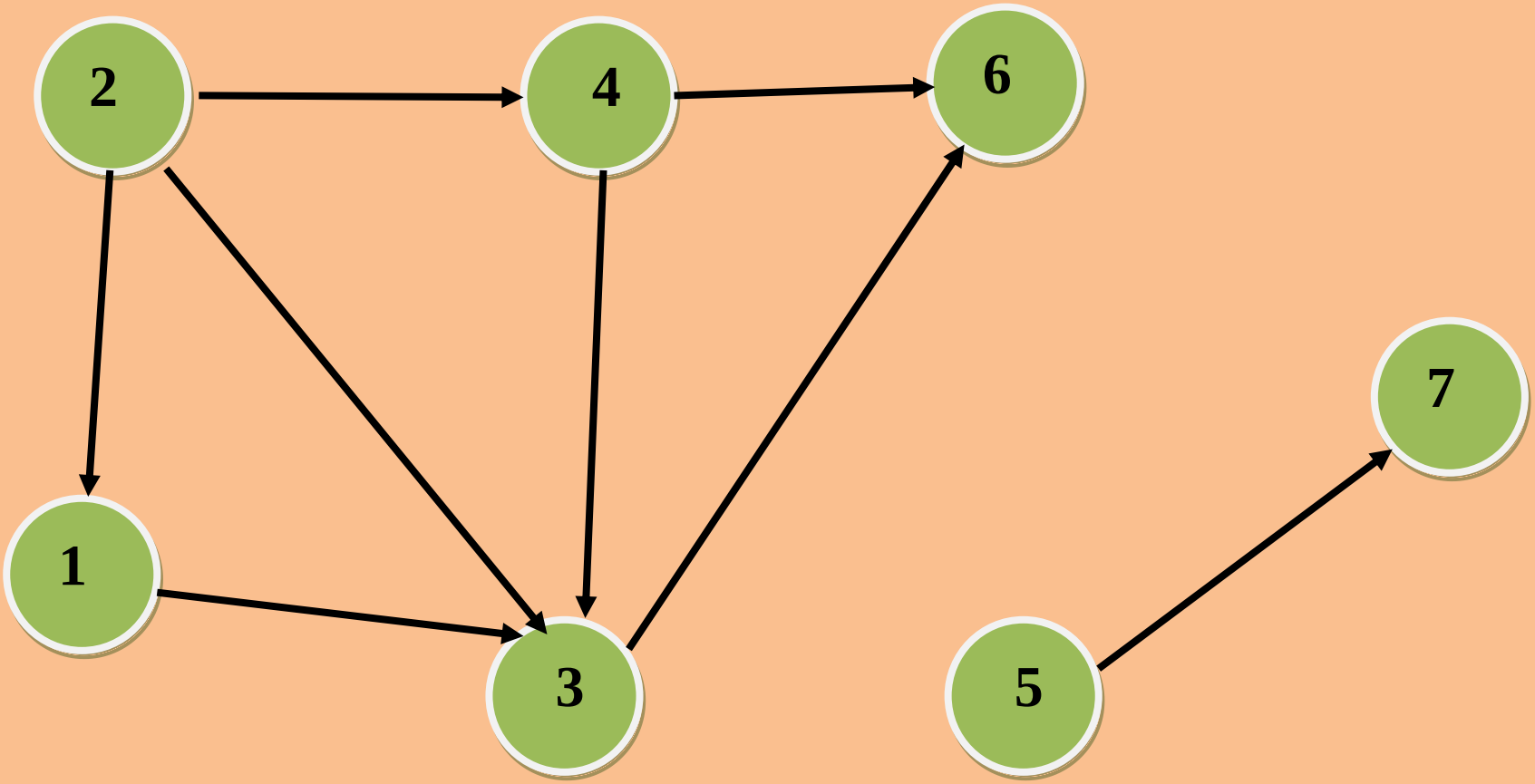
Composante fortement connexe



Un graphe orienté **connexe**



Un graphe orienté **non connexe**



Graphe non orienté connexe

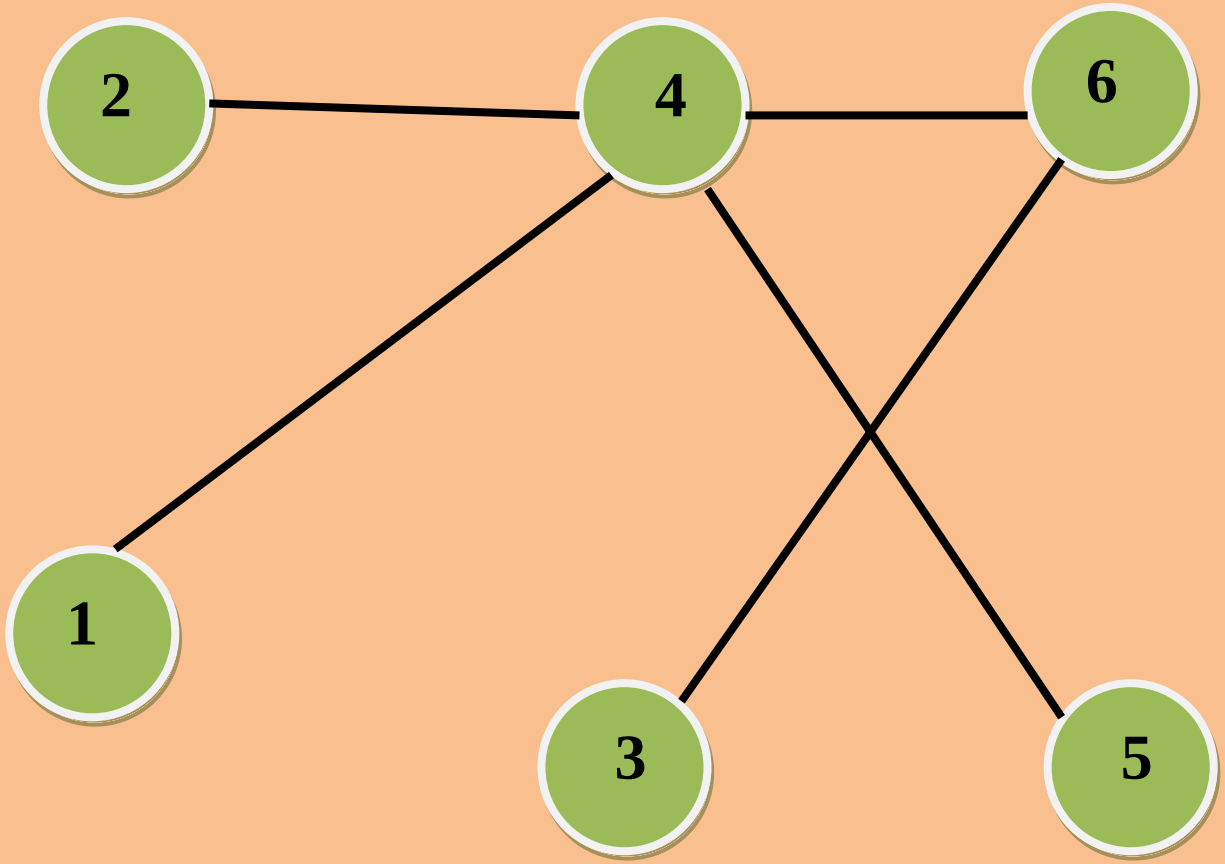
Un graphe non orienté est dit **connexe** si :

$\forall x, y \in S$, il existe une **chaîne** reliant x vers y.

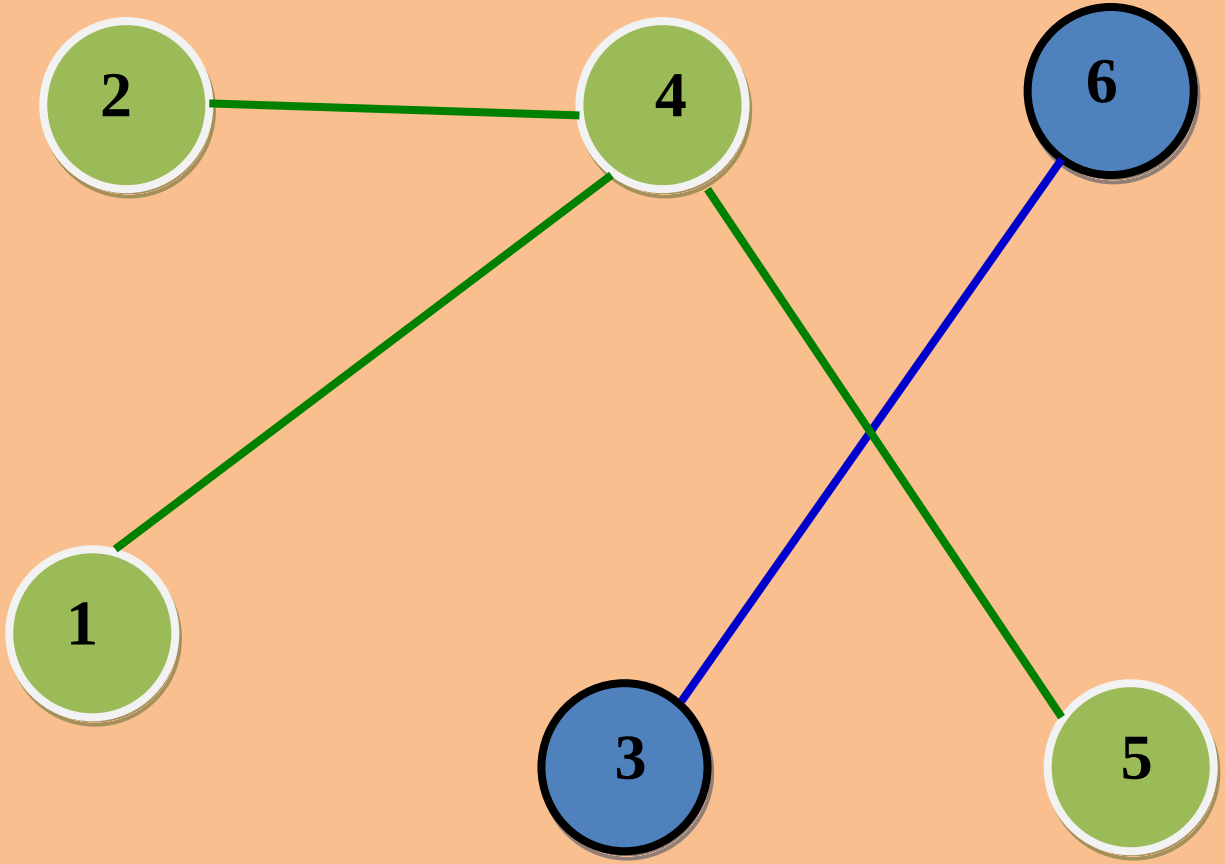
Composante connexe

De même, dans un graphe non orienté, on appelle **composante connexe** un sous-graphe connexe **maximal**.

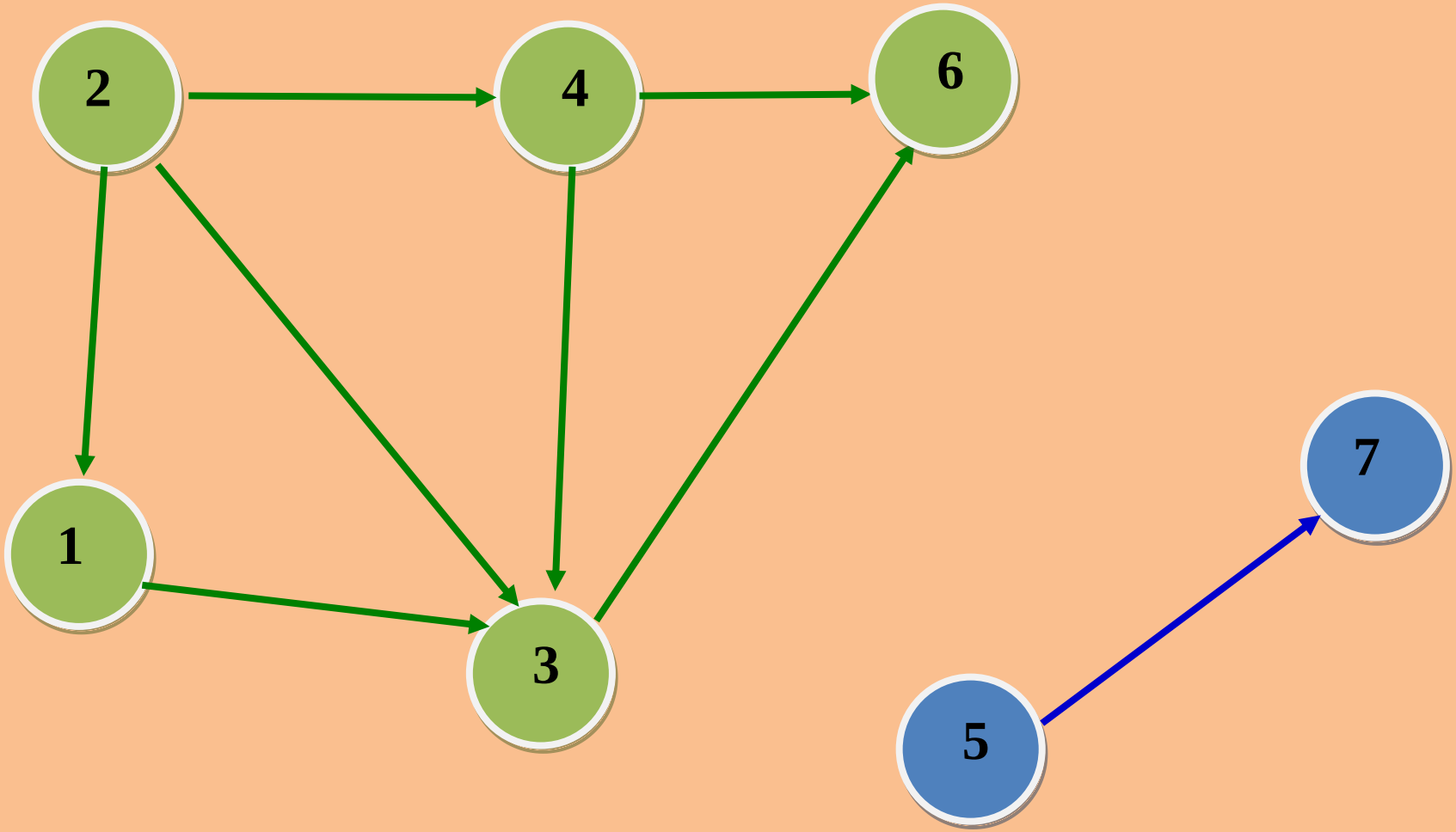
Graphe non orienté **connexe**



Graphe non orienté **non connexe**



Graphe orienté **non connexe**



Graphe eulérien

On dit qu'un graphe non orienté est **eulérien :**

- s'il est possible de trouver un **cycle**
- cycle passant une et une seule fois par toutes les **arêtes**.

Graphe semi-eulérien

Un graphe non orienté est **semi-eulérien :**

-s'il est possible de trouver une **chaîne**,

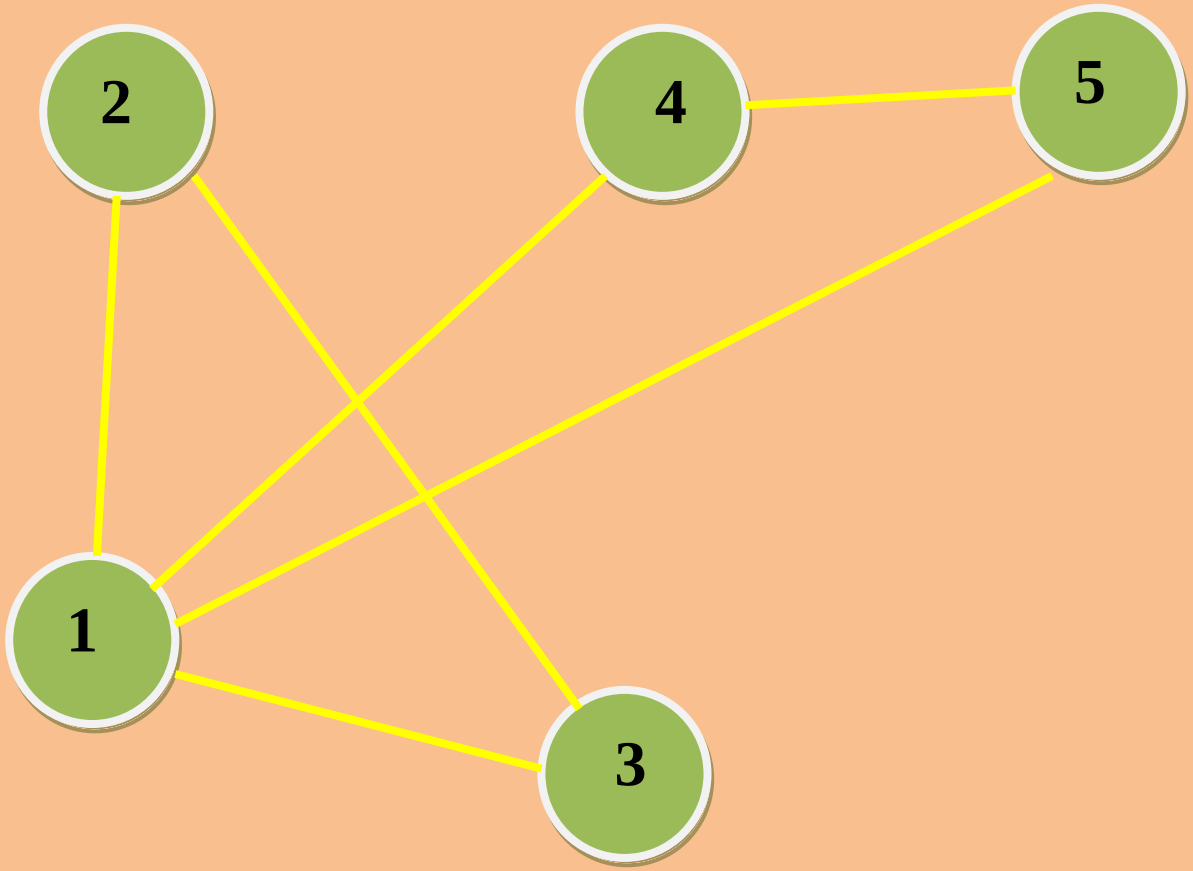
-chaîne passant une et une seule fois par toutes les **arêtes**.

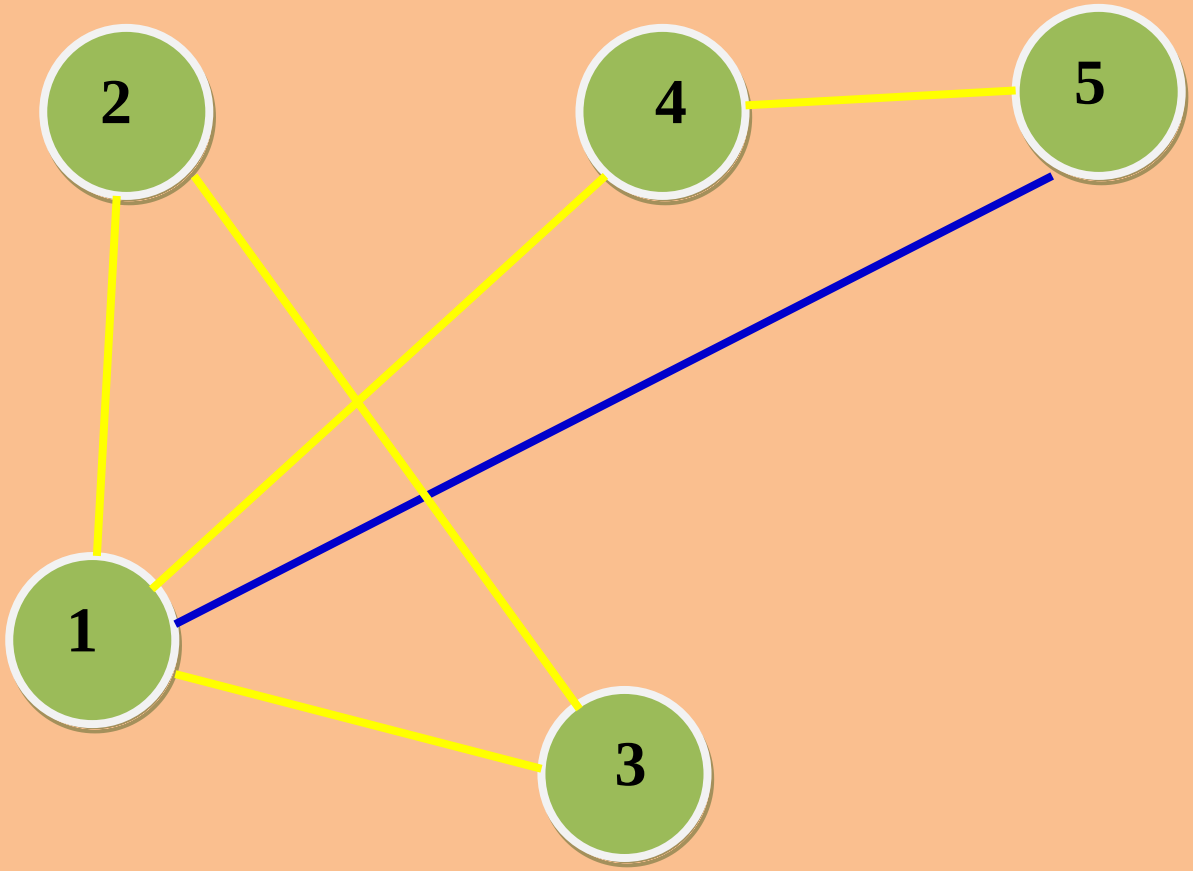
Deux théorèmes importants

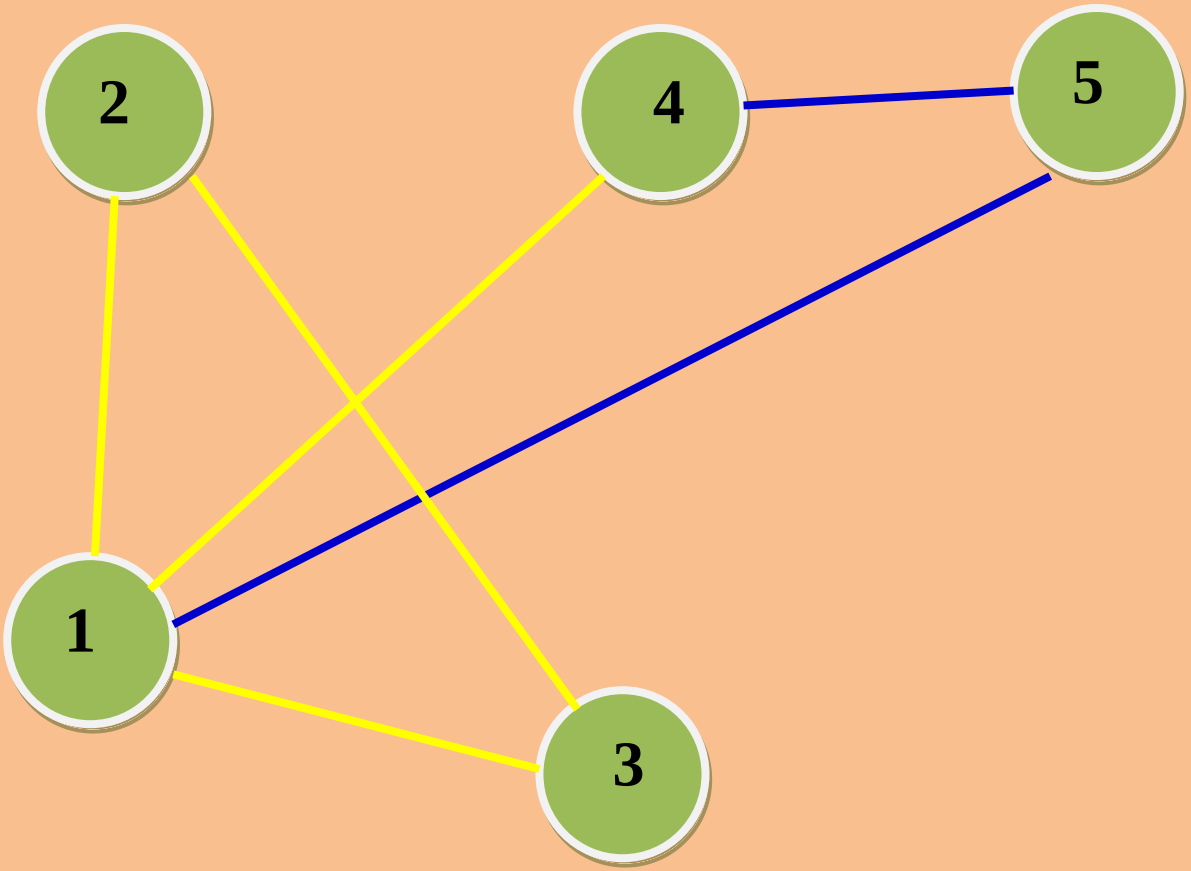
Théorème 1

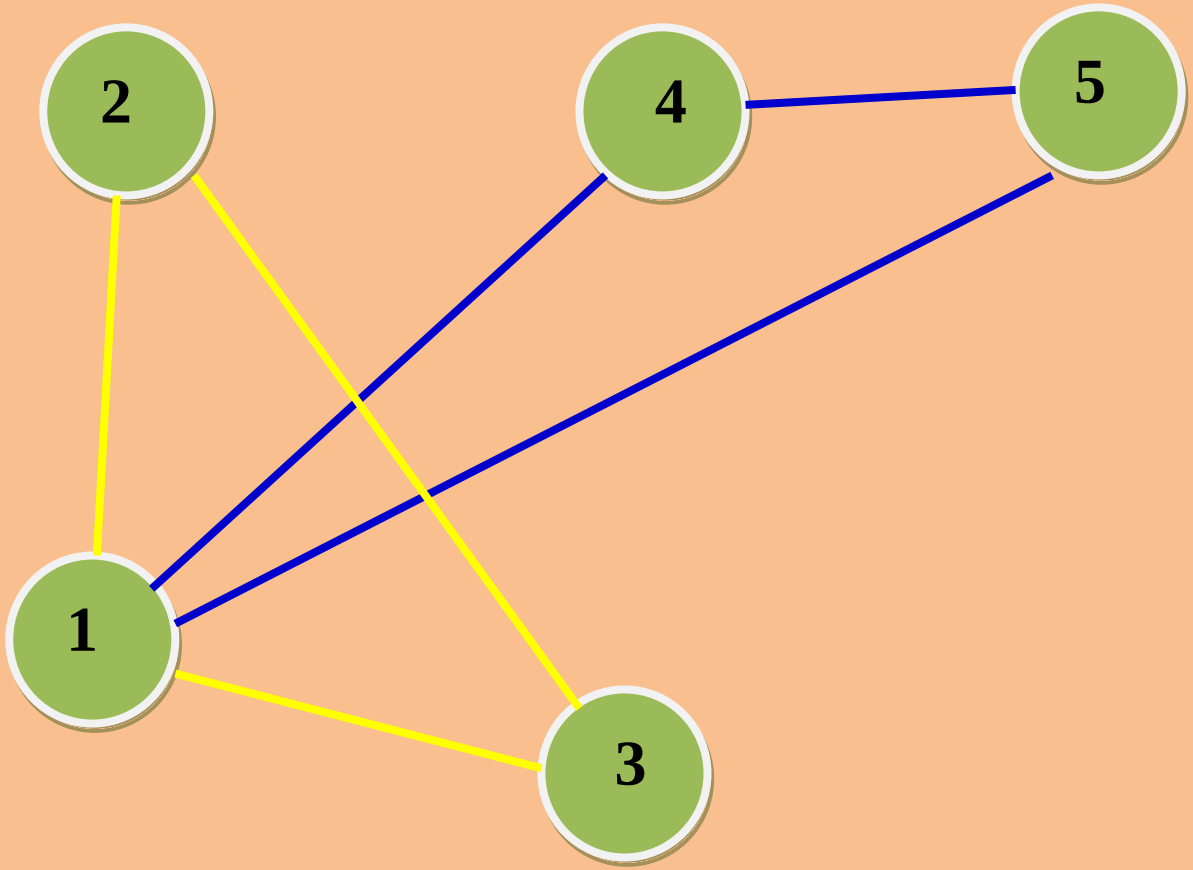
Un graphe connexe admet un **cycle eulérien** si et seulement si **tous** ses sommets sont de **degré pair**.

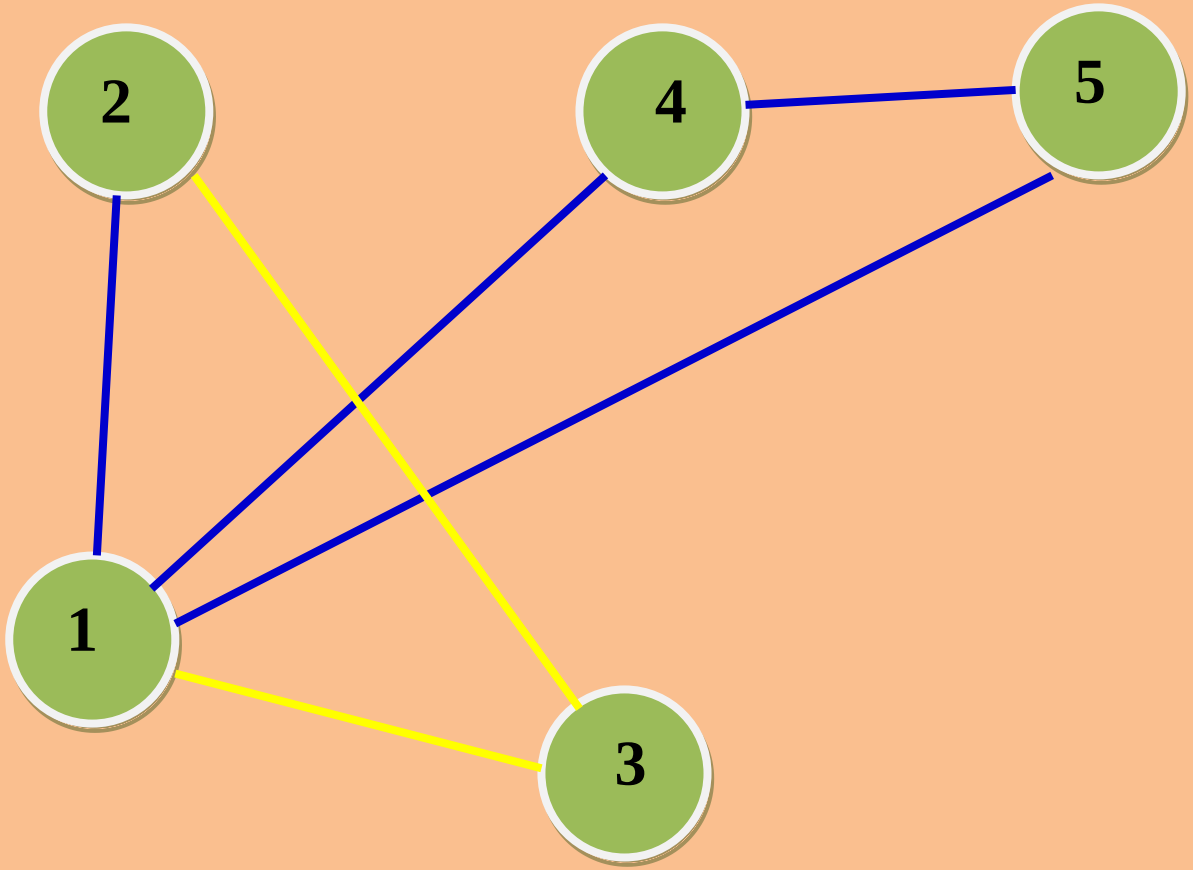
Exemple

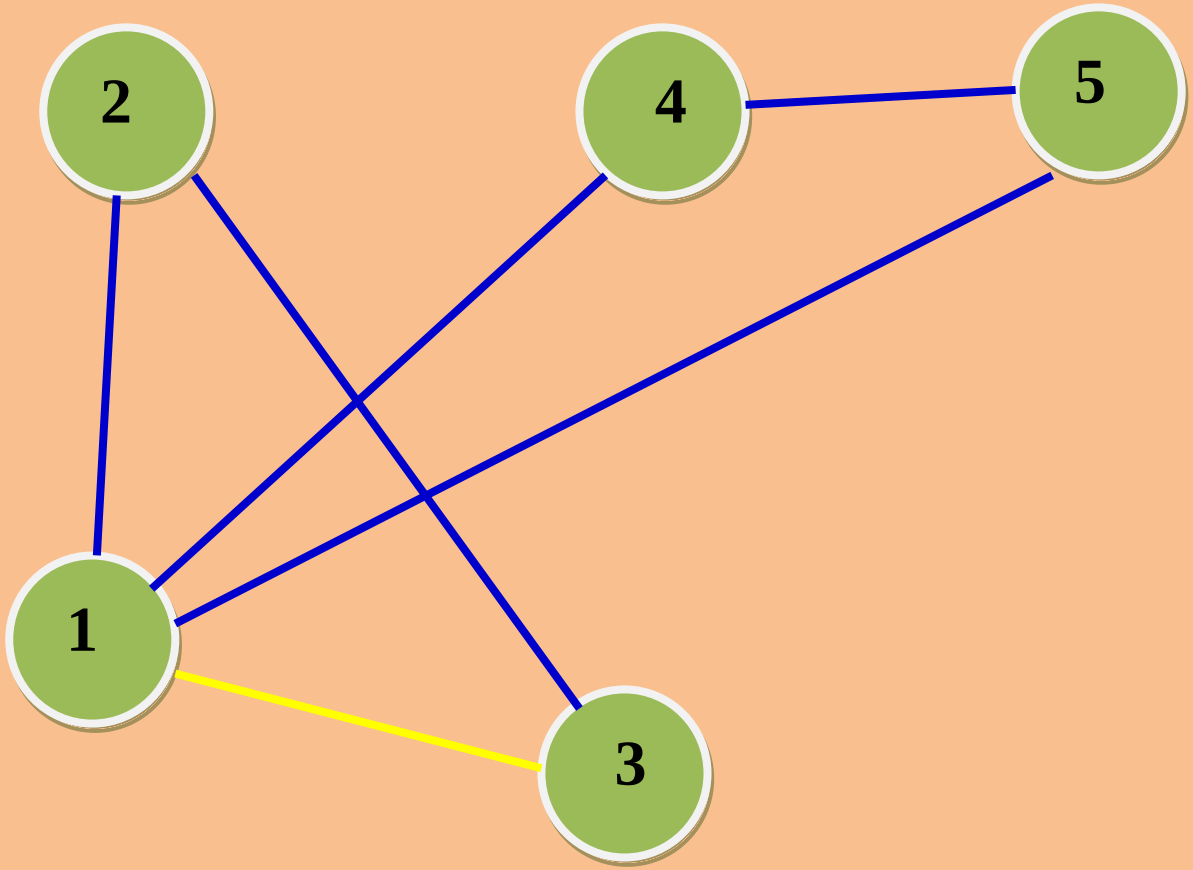



2
4
5
1
3

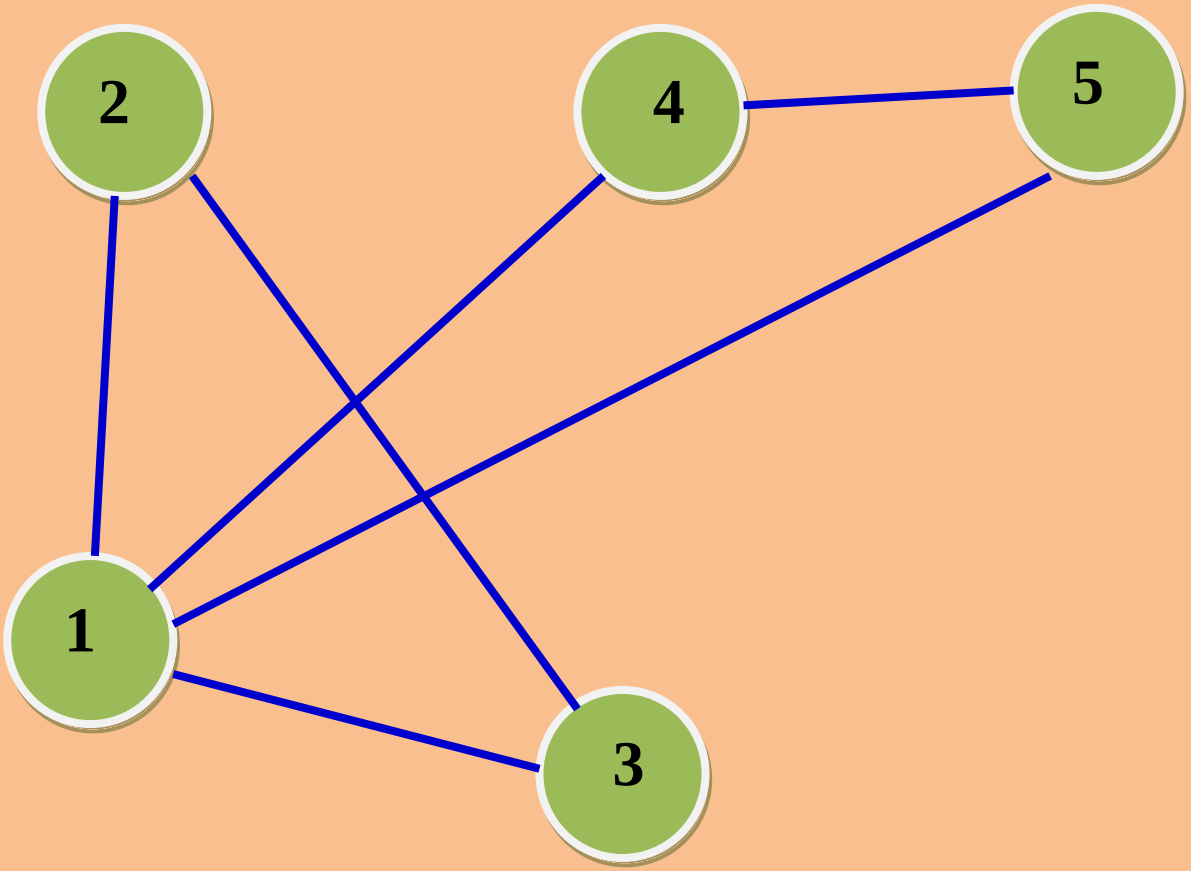

2
4
5
1
3


2
4
5
1
3


2
4
5
1
3


2
4
5
1
3

Cycle eulérien (1- 5 - 4 - 1- 2 - 3 - 1)

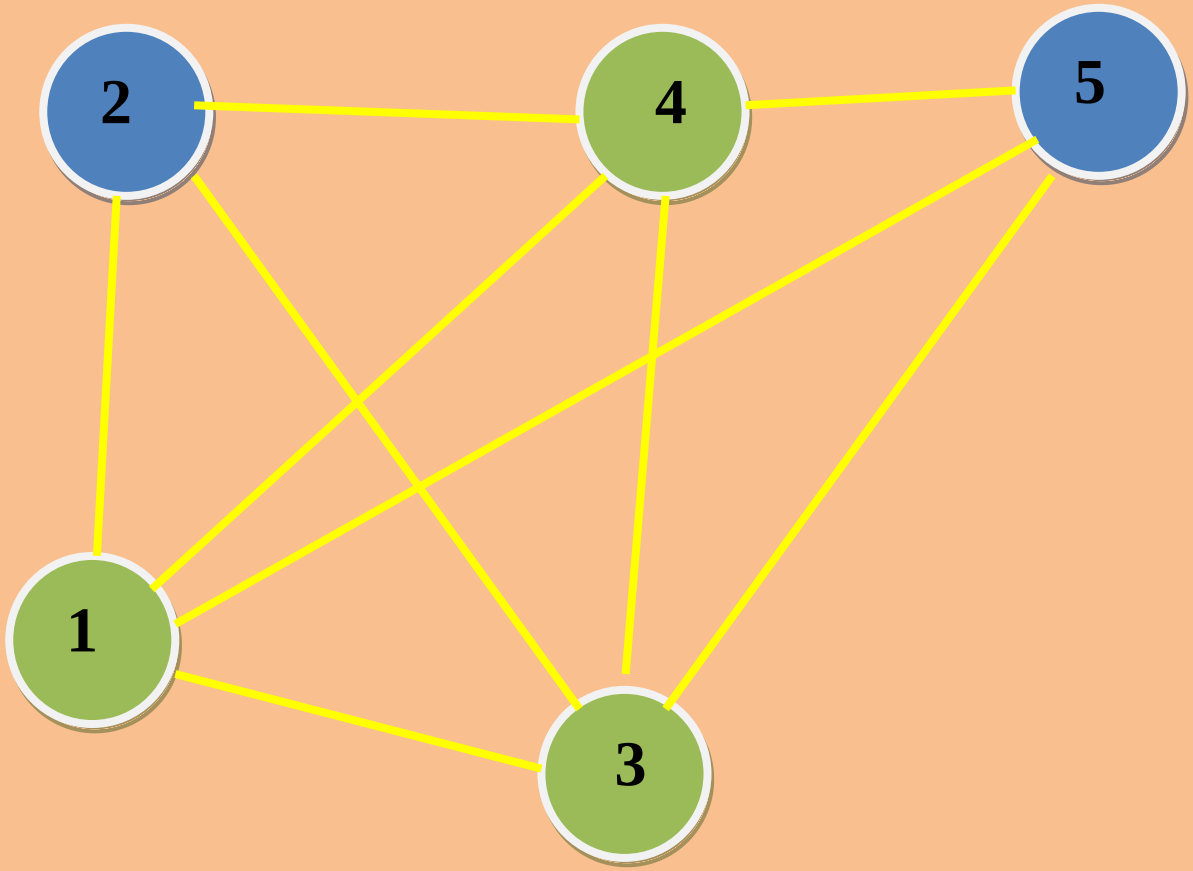


Théorème 2

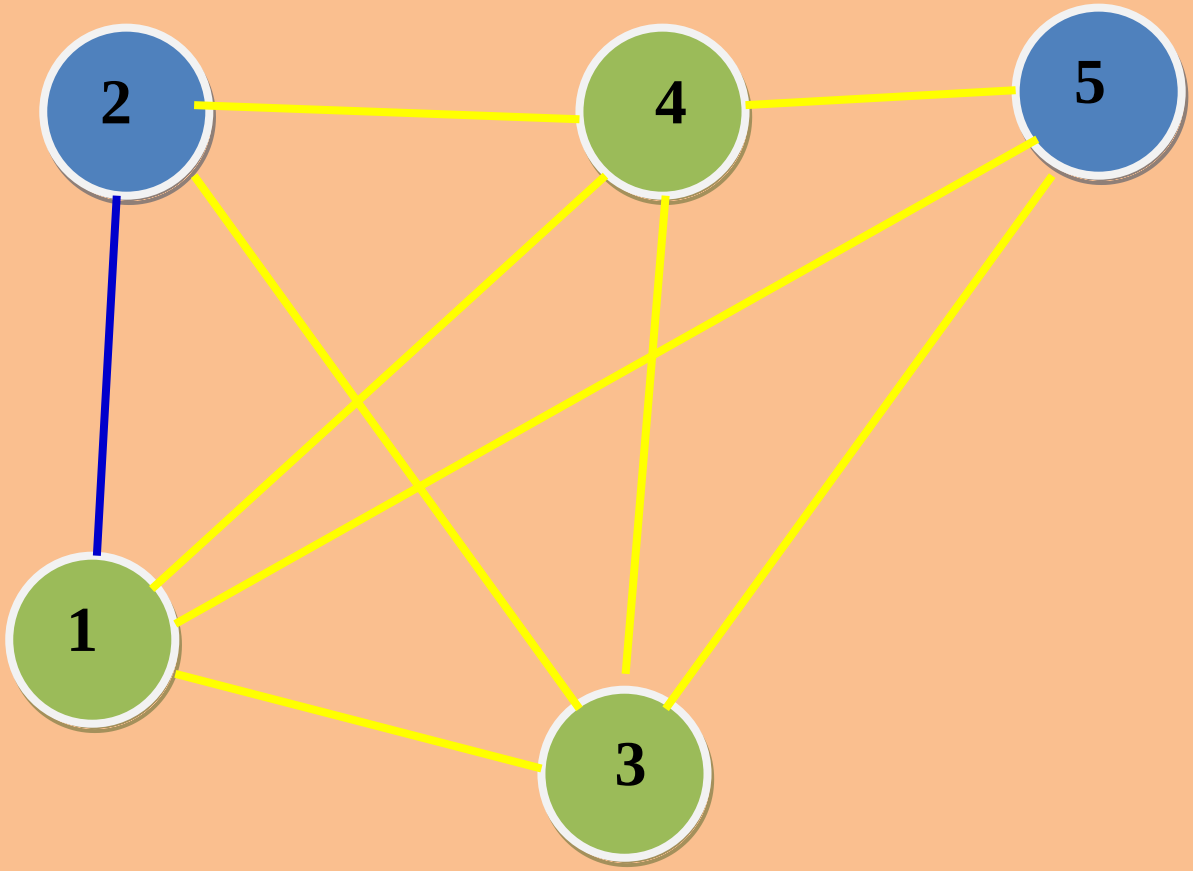
Un graphe connexe *G* admet **une chaîne eulérienne** si et seulement si le nombre de sommets de degré impair est égal à **2**.

Les deux sommets de degré impair, sont les extrémités de la chaîne eulérienne.

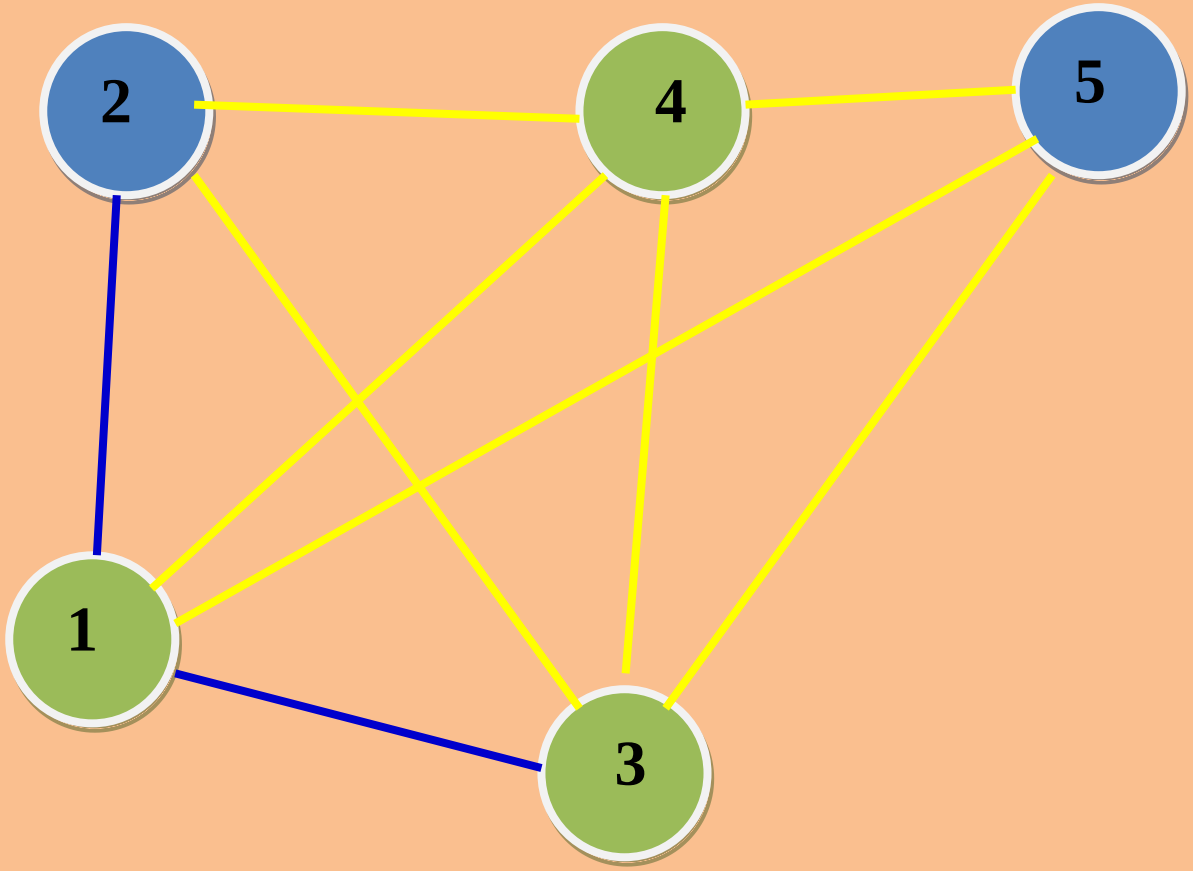
Exemple



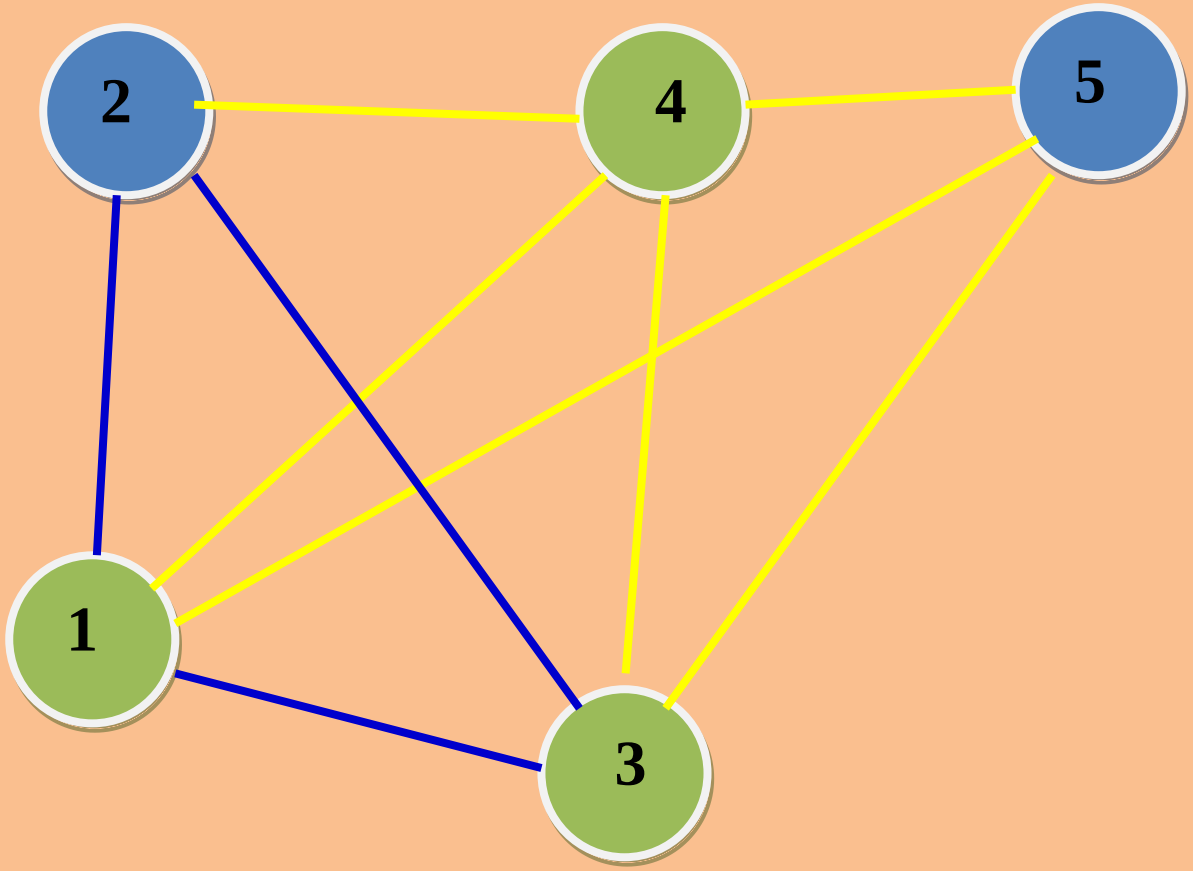
Chaîne eulérienne : (2-1-



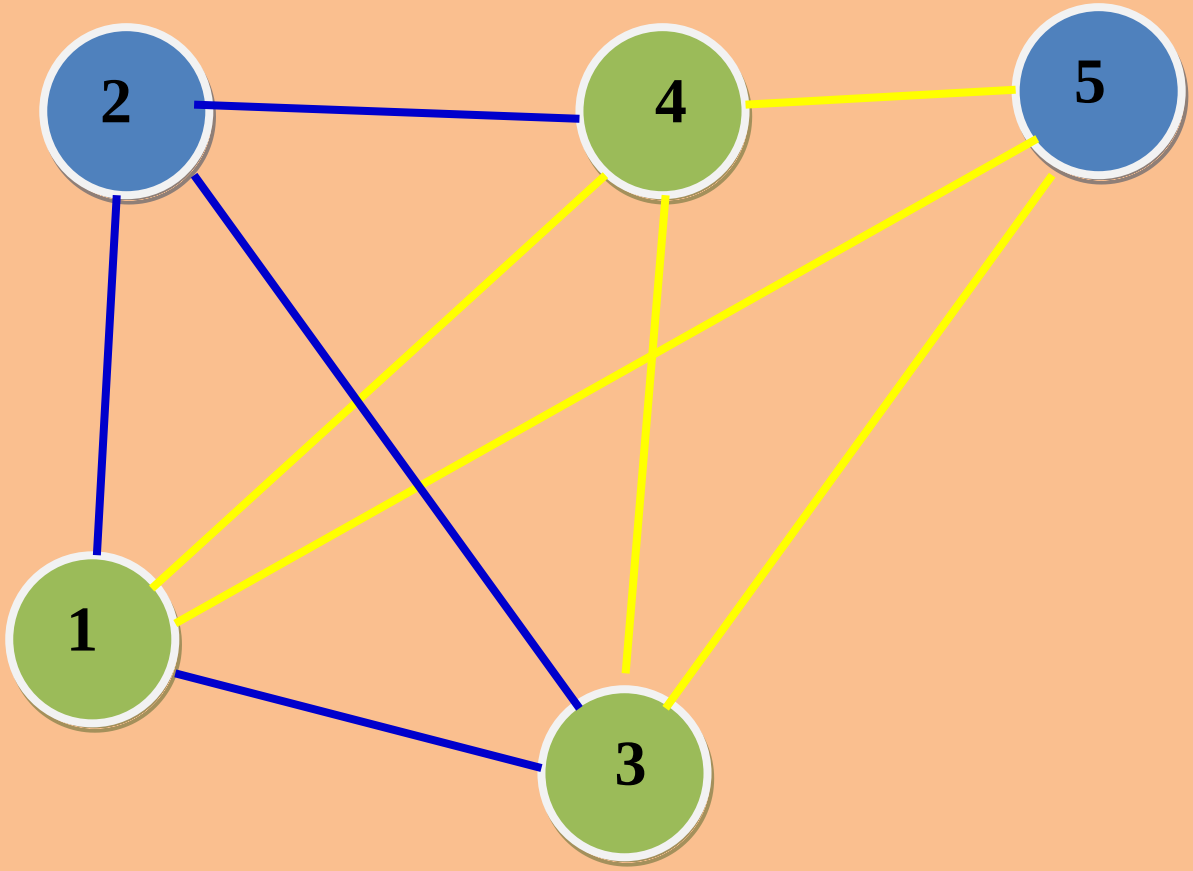
Chaîne eulérienne : (2-1-3-



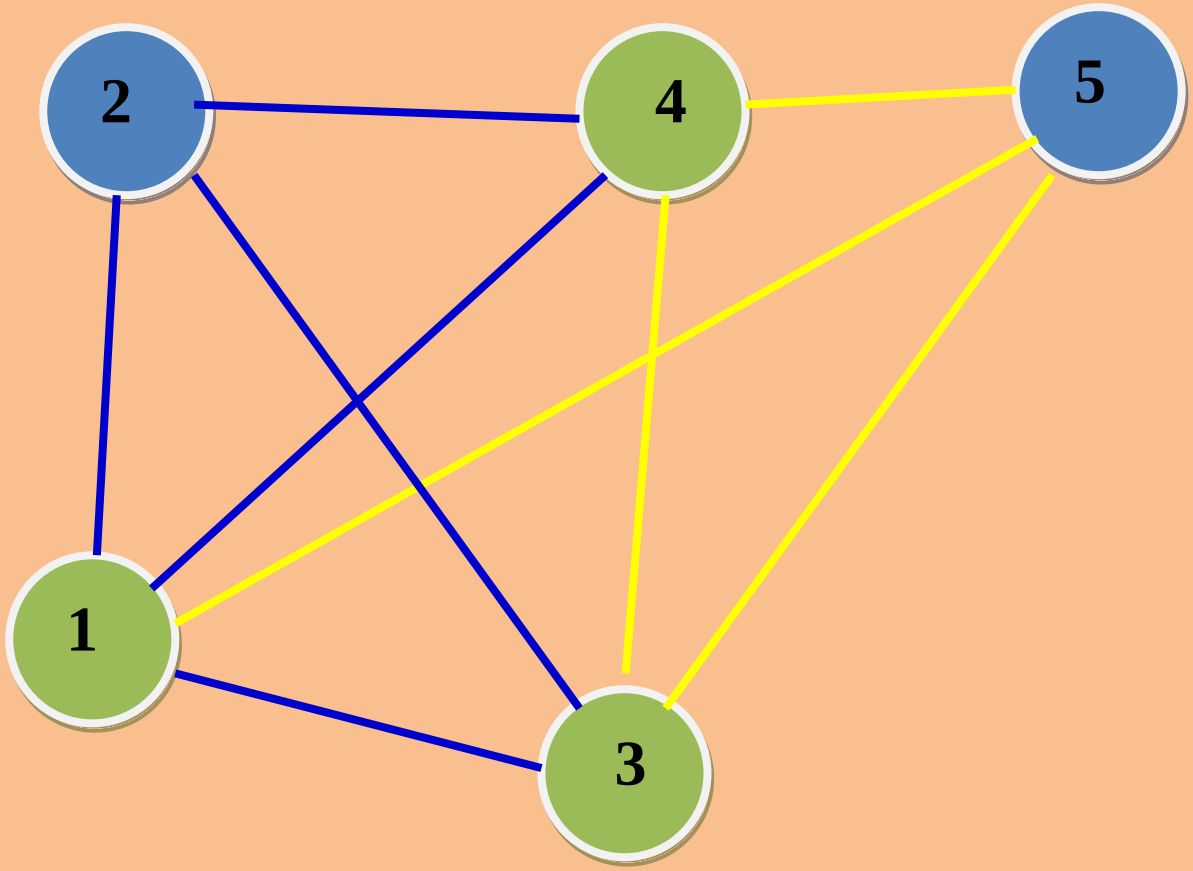
Chaîne eulérienne : (2-1-3-2-



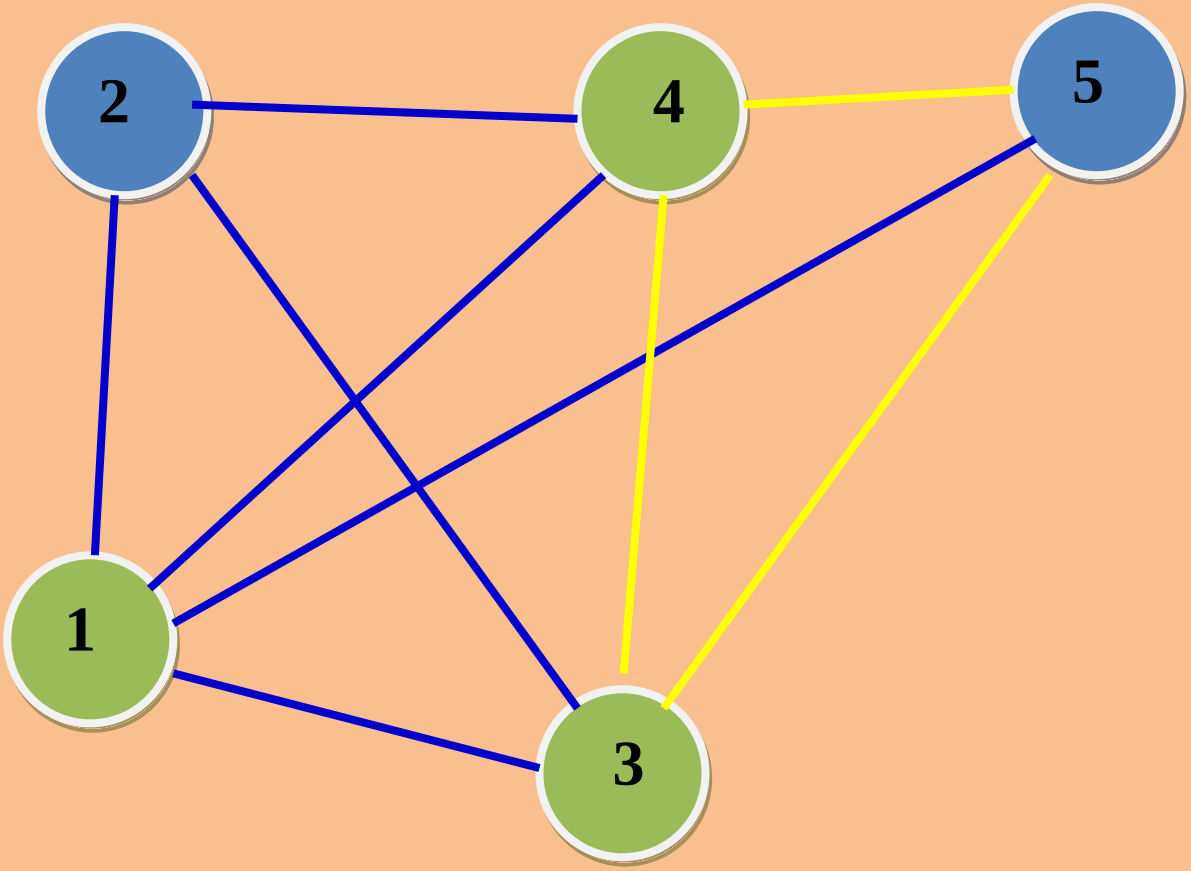
Chaîne eulérienne : (2-1-3-2-4-



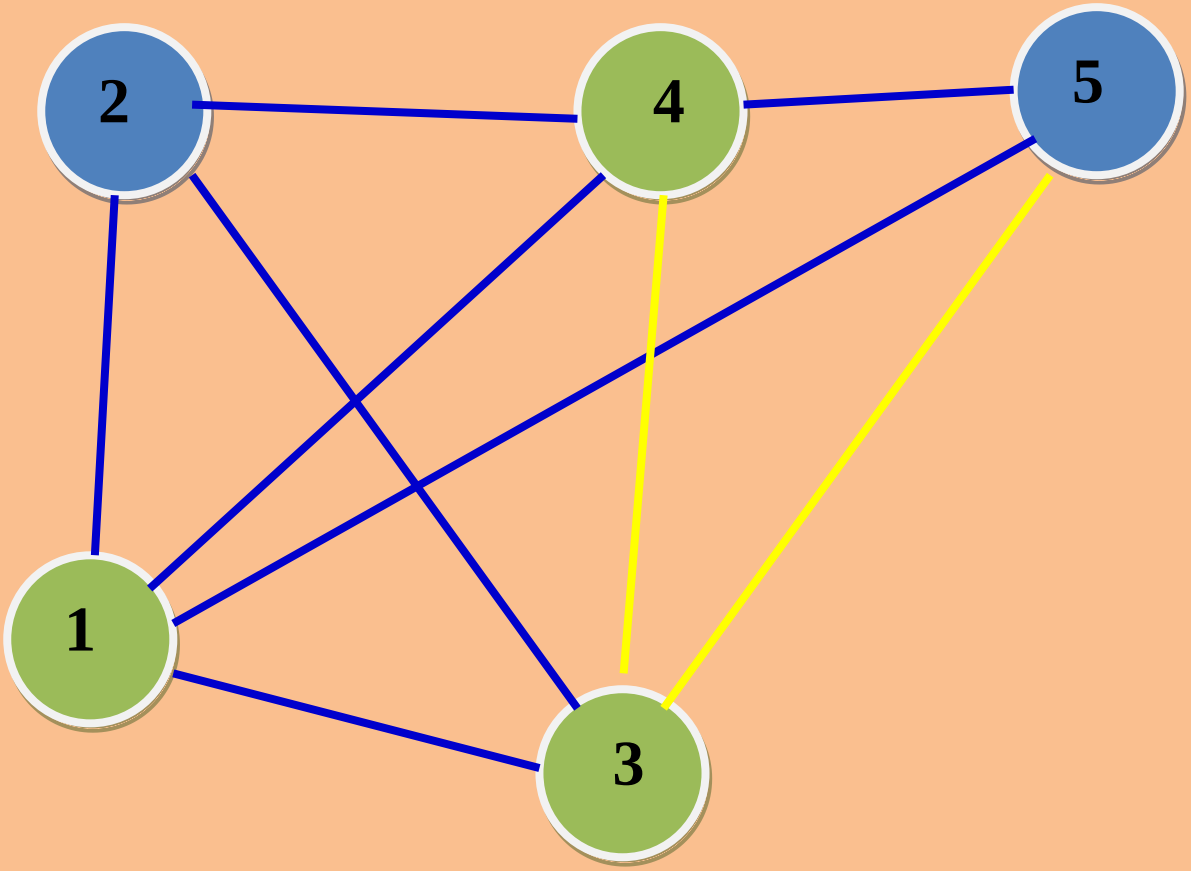
Chaîne eulérienne : (2-1-3-2-4-1-



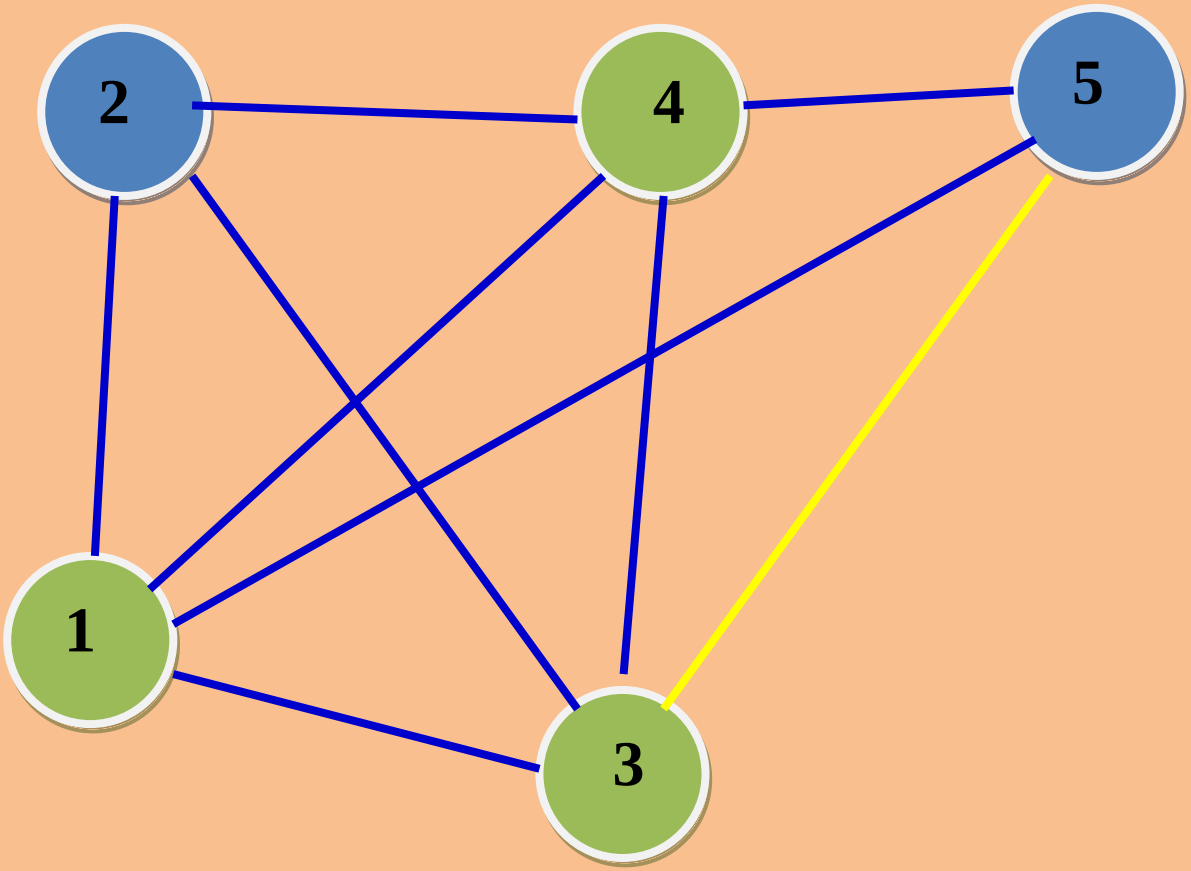
Chaîne eulérienne : (2-1-3-2-4-1-5-



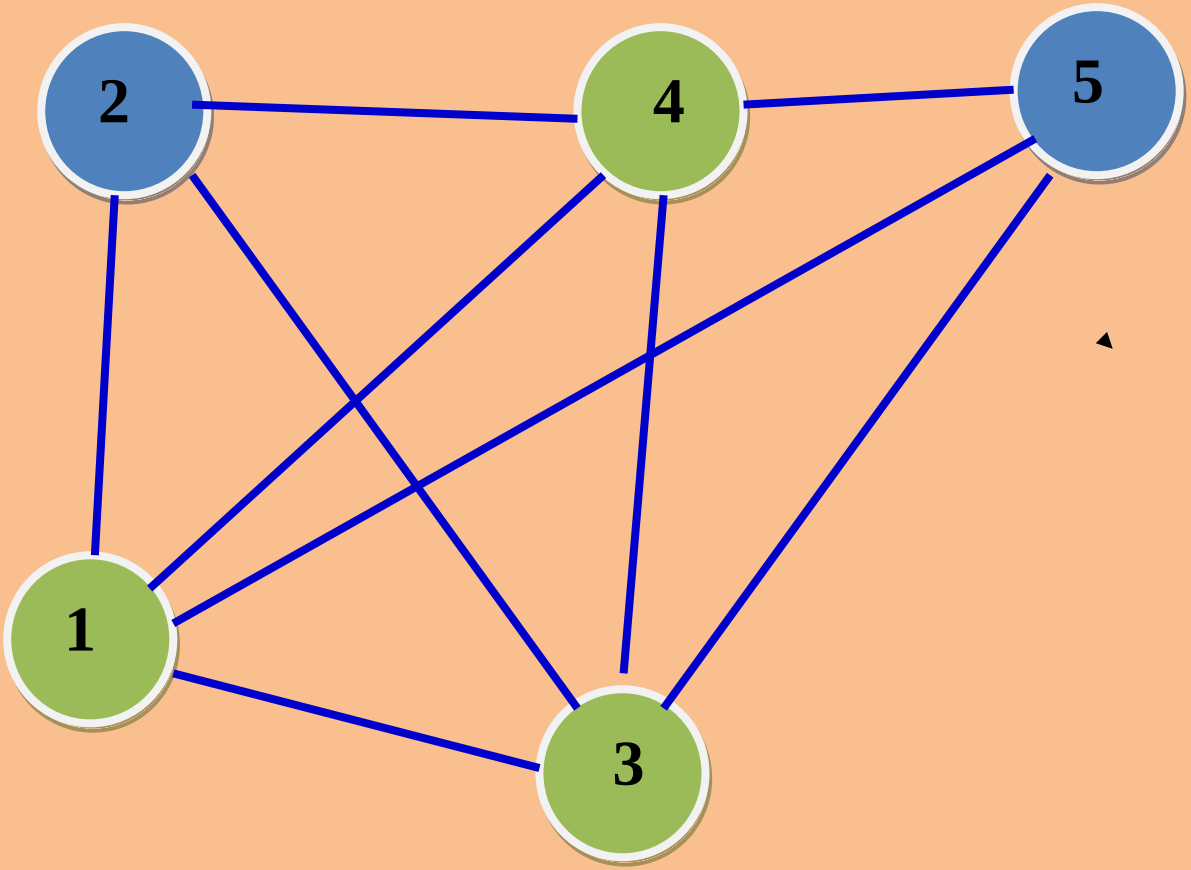
Chaîne eulérienne : (2-1-3-2-4-1-5-4-



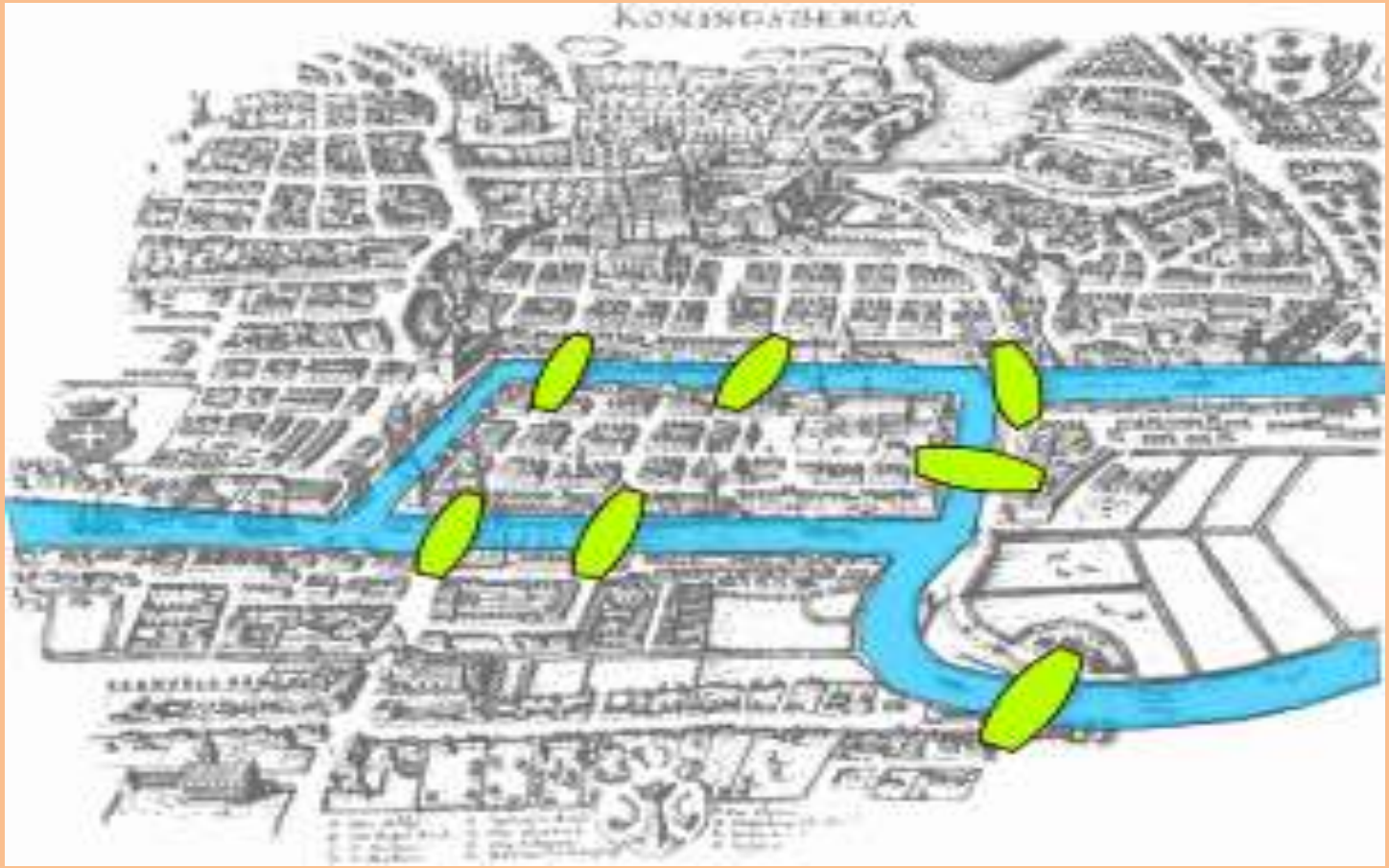
Chaîne eulérienne : (2-1-3-2-4-1-5-4-3-



Chaîne eulérienne : (2-1-3-2-4-1-5-4-3-5)



Problème des 7 ponts de Königsberg

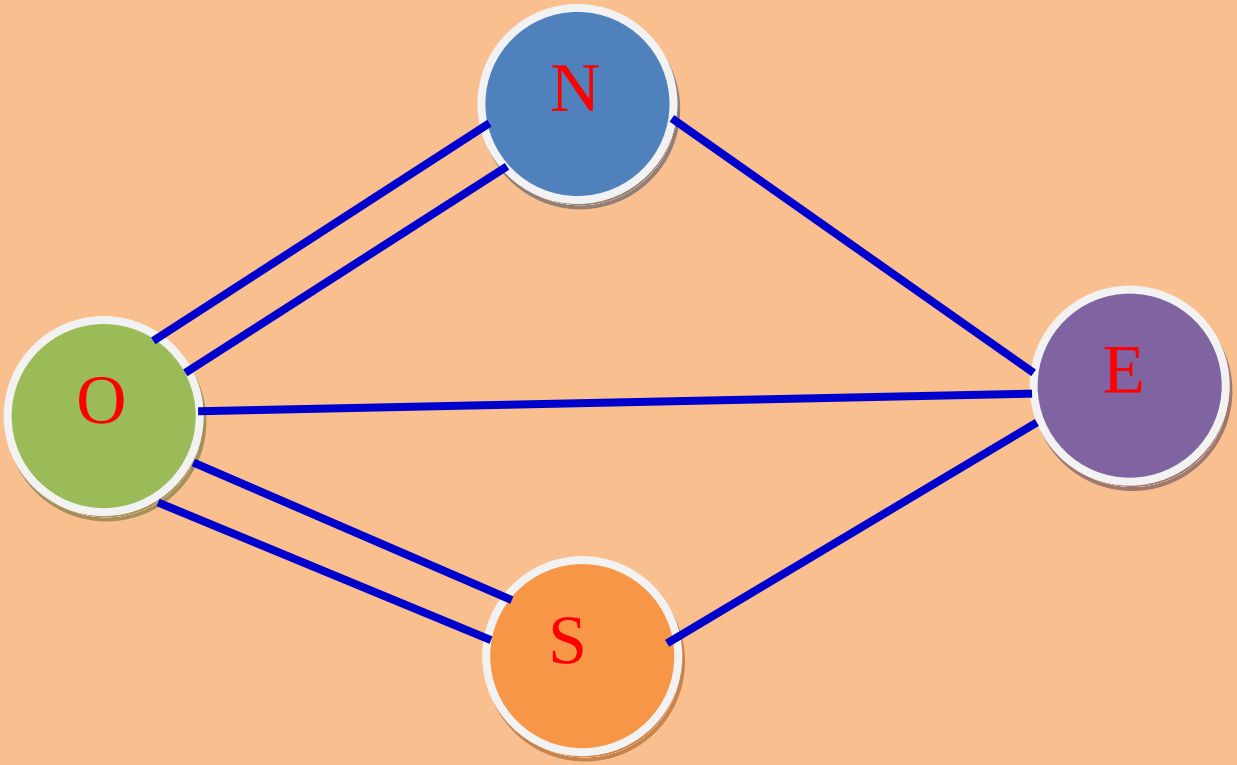


Position du problème :

1-un promeneur doit revenir à son quartier de départ à la fin de son parcours : il décrit un **cycle**.

2-chaque pont doit être traversé une fois exactement : **cycle eulérien**

Graphe d'Euler



Conclusion d’Euler

De chaque sommet partent un **nombre impair** d'arêtes

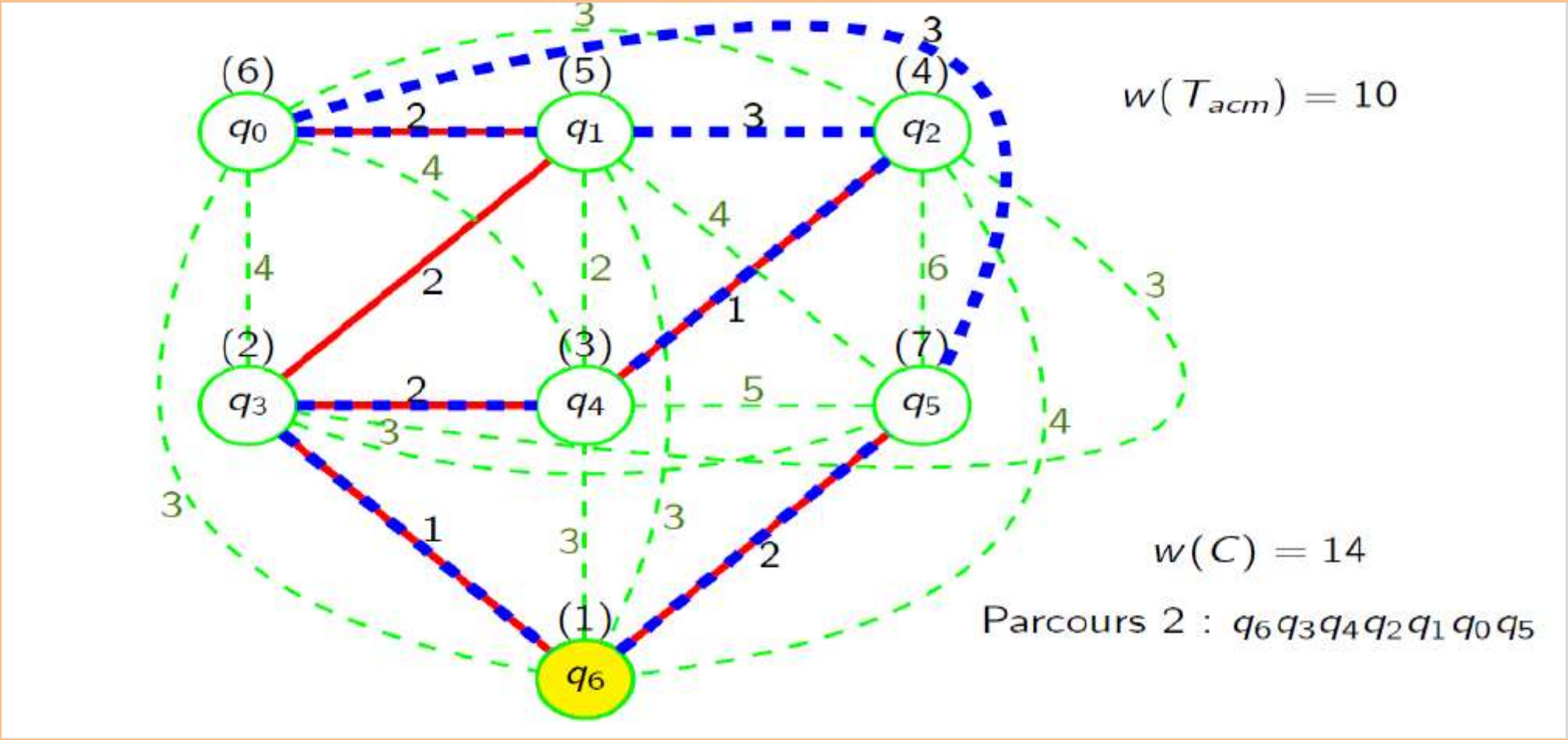
Donc, le graphe ne possède ni **cycle** eulérien ni **chaîne** eulérienne.

Graphe hamiltonien

Un graphe non orienté est **hamiltonien:**

-s'il est possible de trouver un **cycle**,
-cycle passant **une et une seule fois** par **tous** les **sommets**.

Exemple de cycle hamiltonien



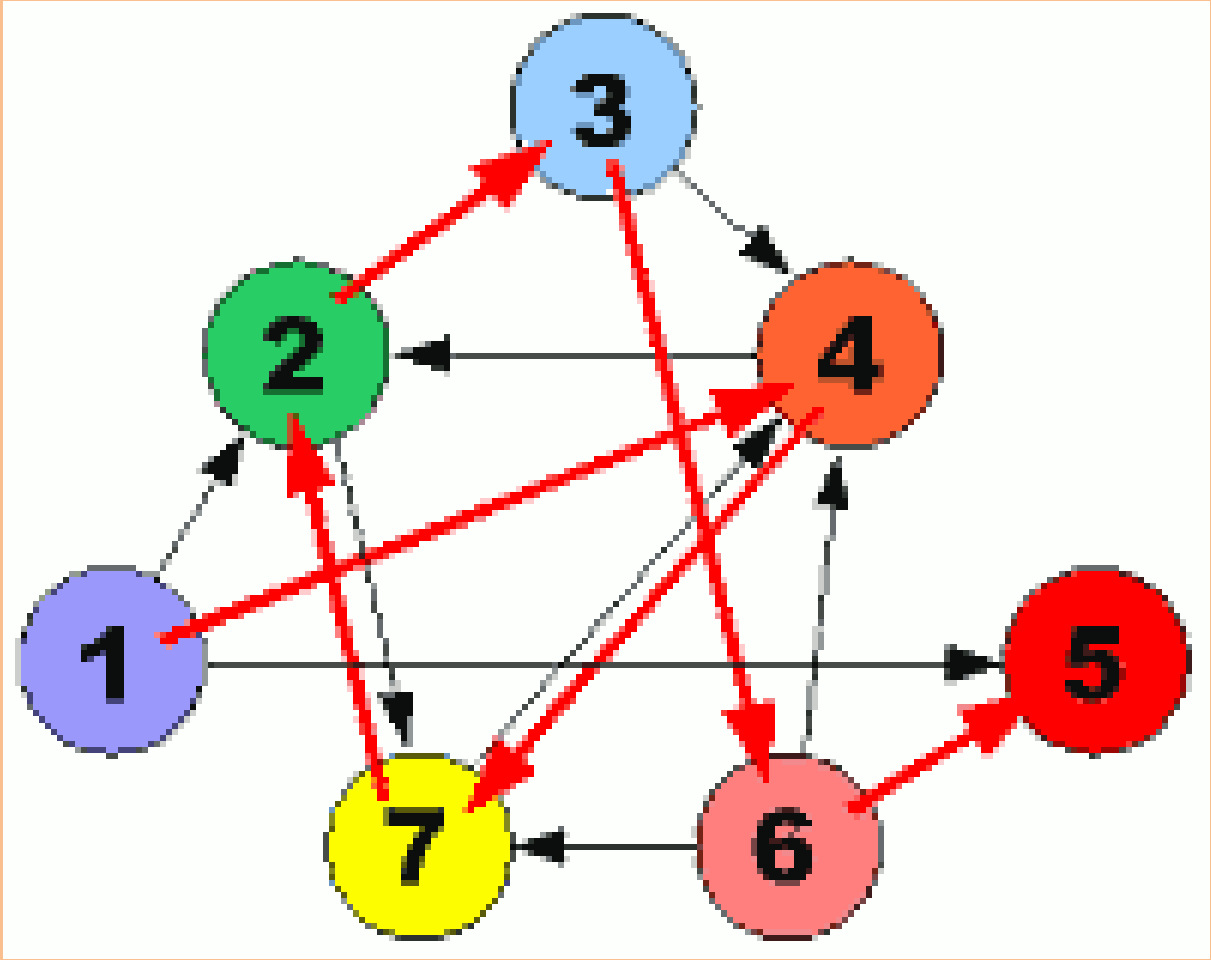
Graphe semi-hamiltonien

Un graphe non orienté est **semi-hamiltonien**:

-s'il est possible de trouver une **chaîne**,

-chaîne passant **une et une seule fois** par **tous** les **sommets**.

Exemple de chaîne hamiltonienne



Théorème d'Ore

Soit *G* un graphe non orienté simple d'ordre *n* .

Si pour toute paire (*x*, *y*) de sommets **non adjacents**, on a

$$d(x) + d(y) \geq n$$

alors *G* est **hamiltonien**.

Corollaire de Dirac

Soit *G* un graphe non orienté simple d'ordre $n \geq 3$.

Si pour tout sommet *x* de *G*, on a :

$$d(x) \geq n/2$$

alors *G* est **hamiltonien**.

En effet, un tel graphe vérifie les conditions du théorème précédent.

Si x et y ne sont pas adjacents, on a bien :

$$d(x) \geq n/2$$
$$d(y) \geq n/2$$
$$d(x) + d(y) \geq n/2 + n/2 = n$$

Graphe planaire

On dit qu'un graphe non orienté est **planaire**:

- si on peut le dessiner **dans le plan**
- de sorte que ses arêtes ne **se croisent pas**.

Une face est une région du plan limitée par des arêtes et dont l'ensemble constitue la frontière.

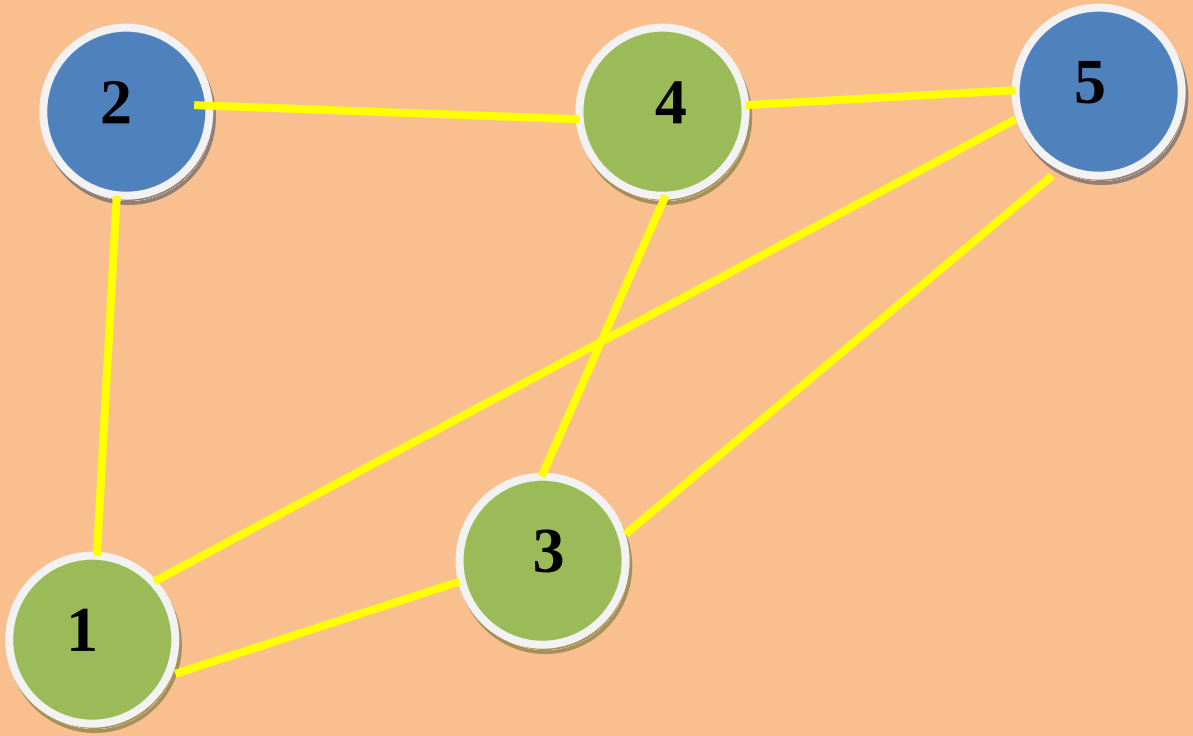
Théorème d'Euler

Si un graphe planaire connexe possédant **n** sommets, **m** arêtes, a une représentation planaire à **f** faces, alors, on a la relation :

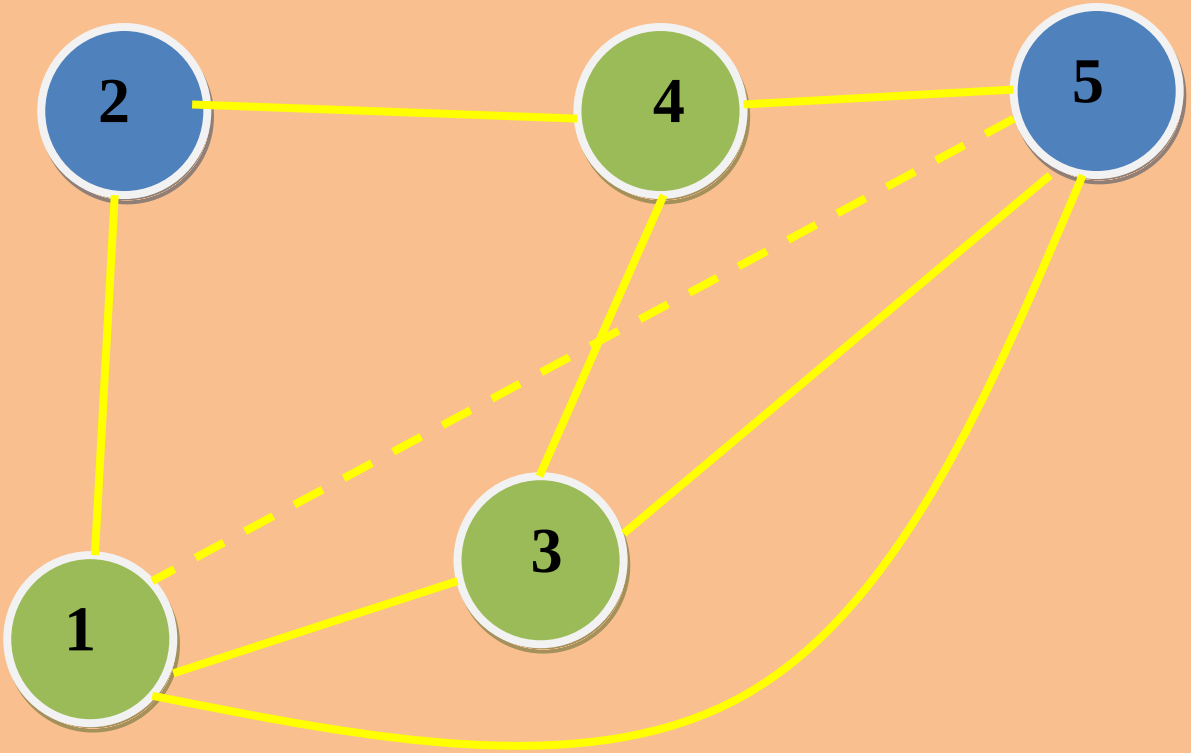
$$n - m + f = 2$$

Ce théorème permet de donner une preuve du fait que K_{33} et K_5 ne sont **pas planaires**.

Exemple

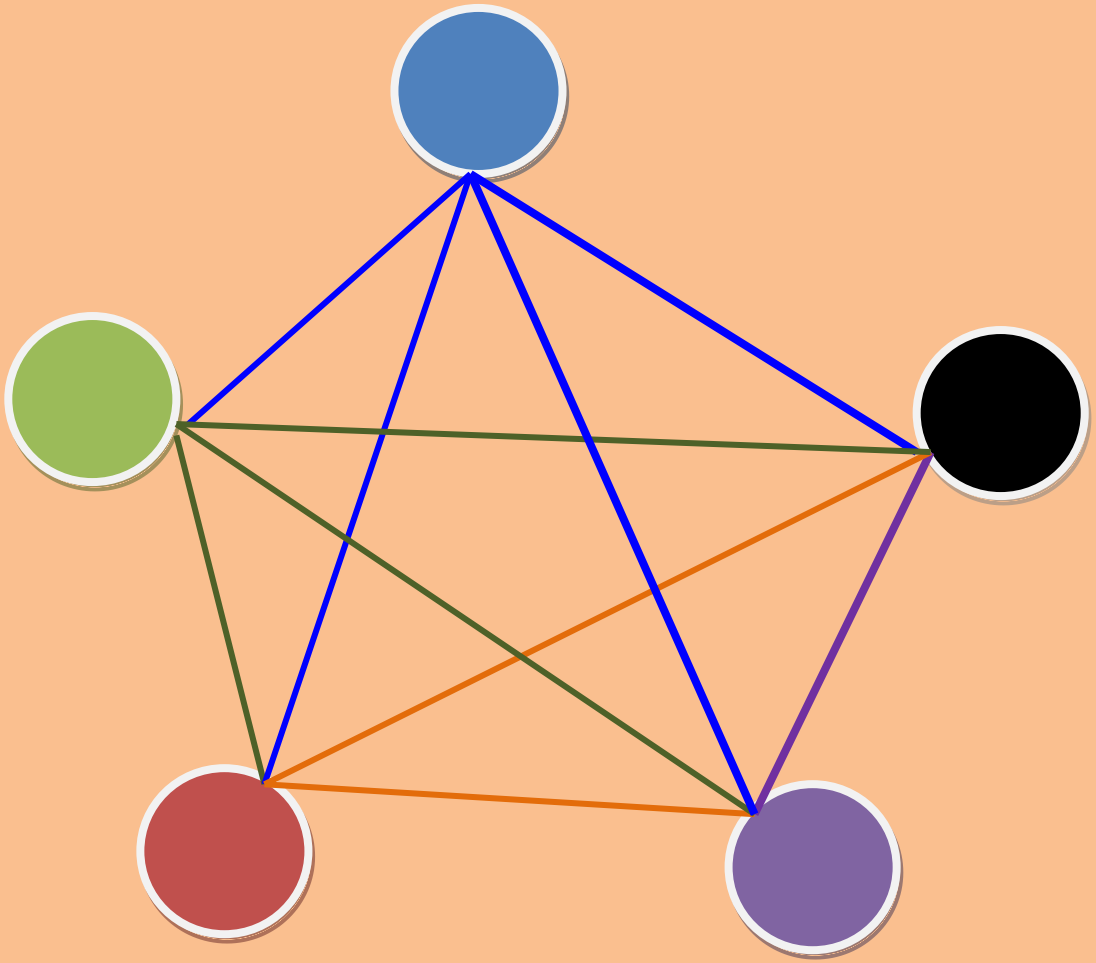


n=5 ; m=7 ; f=5 ; n - m+ f = 5-7+5= 3

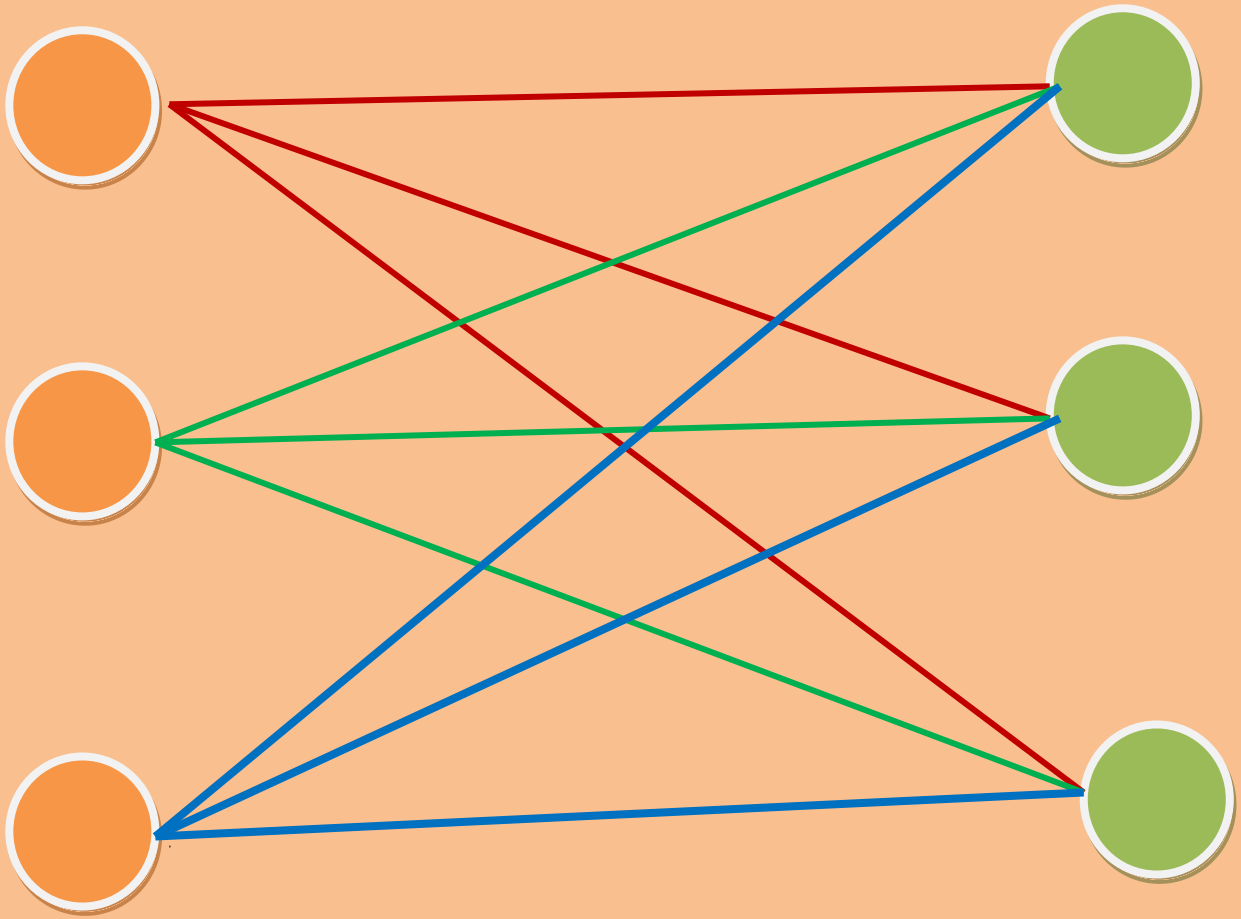


n=5 ; m=7 ; f=4 n - m + f = 5 – 7 + 4 = 2

Clique K_5



Graphe biparti K_{33}



Qu'est-ce que le ruban de Möbius ?

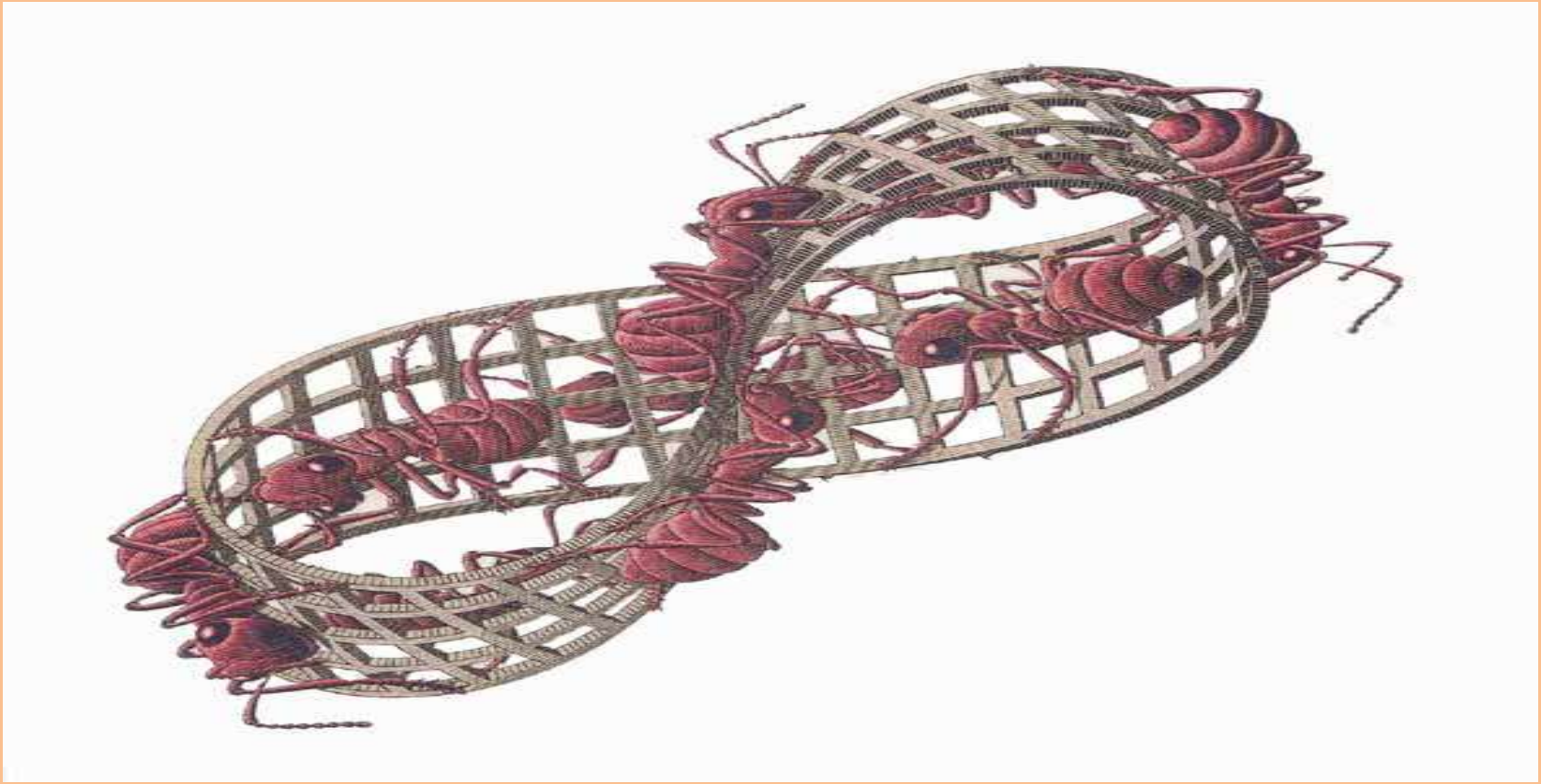
Le ruban de Möbius est une surface **fermée non orientable.**

C'est à dire que c'est un ruban refermé sur lui-même mais qui n'a qu'une seule face.

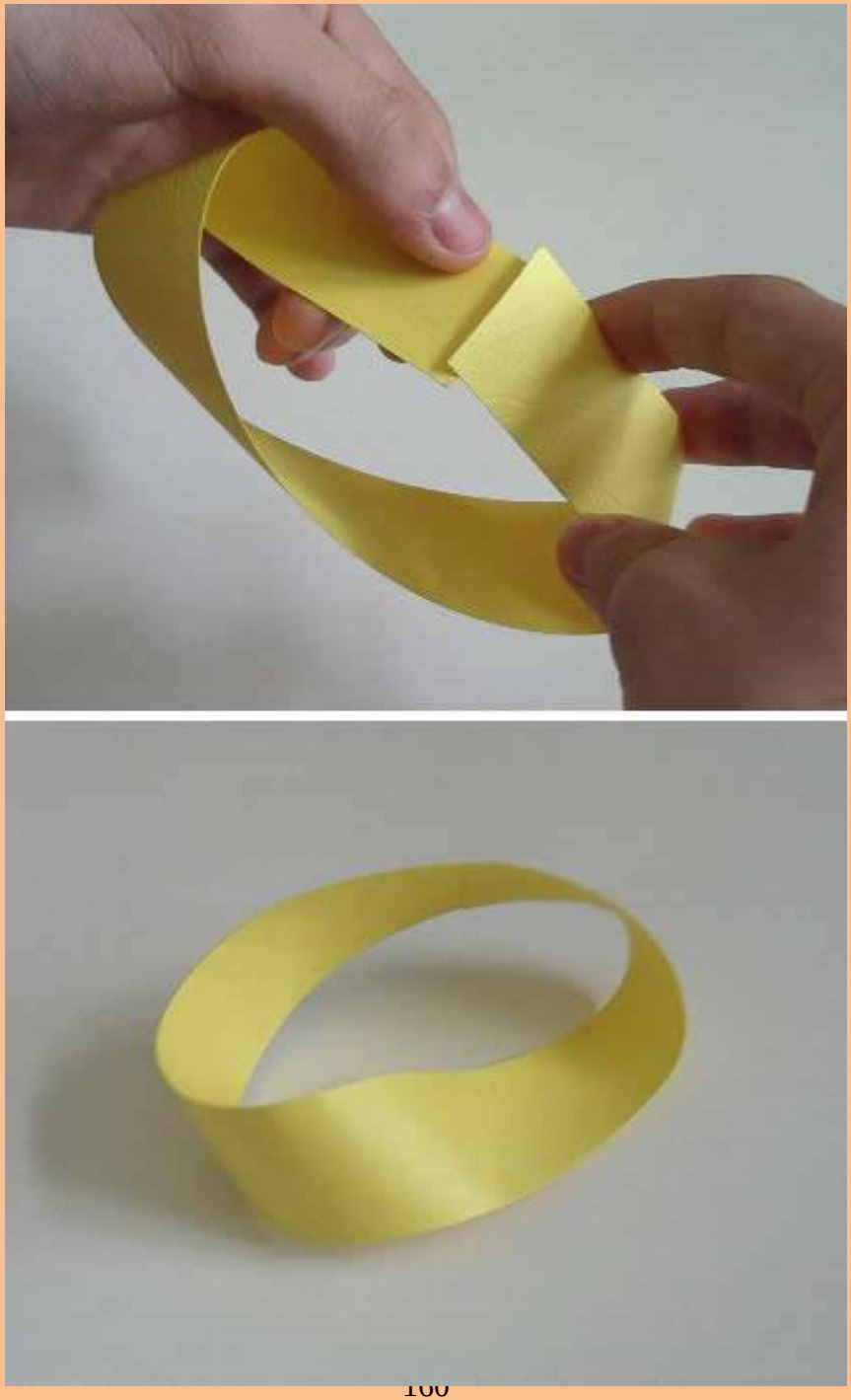
Impossible?

Non.

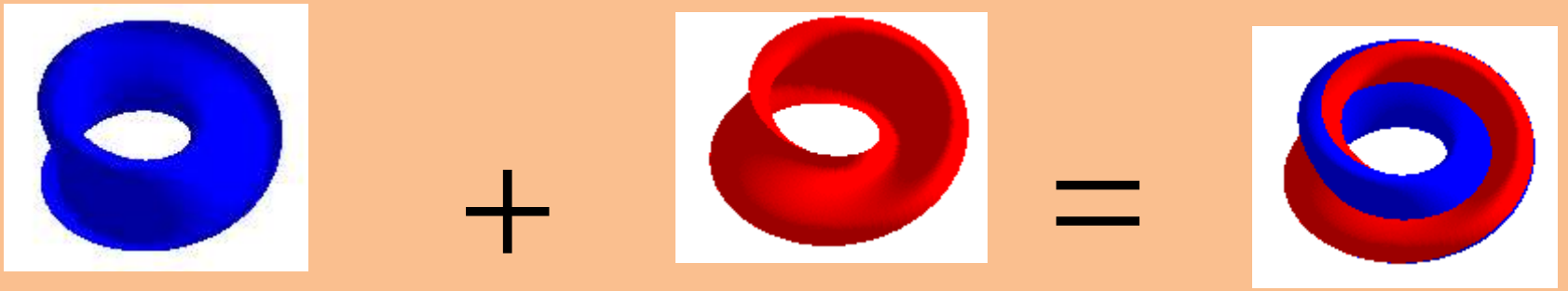
Il faut définir ce que veut dire «avoir une seule face» : pourvoir atteindre n'importe quelle partie de la surface sans passer par le bord.



Cet objet est très important puisque c'est l'exemple typique de surface **non orientable**.



Ruban de Möbius : composition



Ruban de Möbius : paramétrage

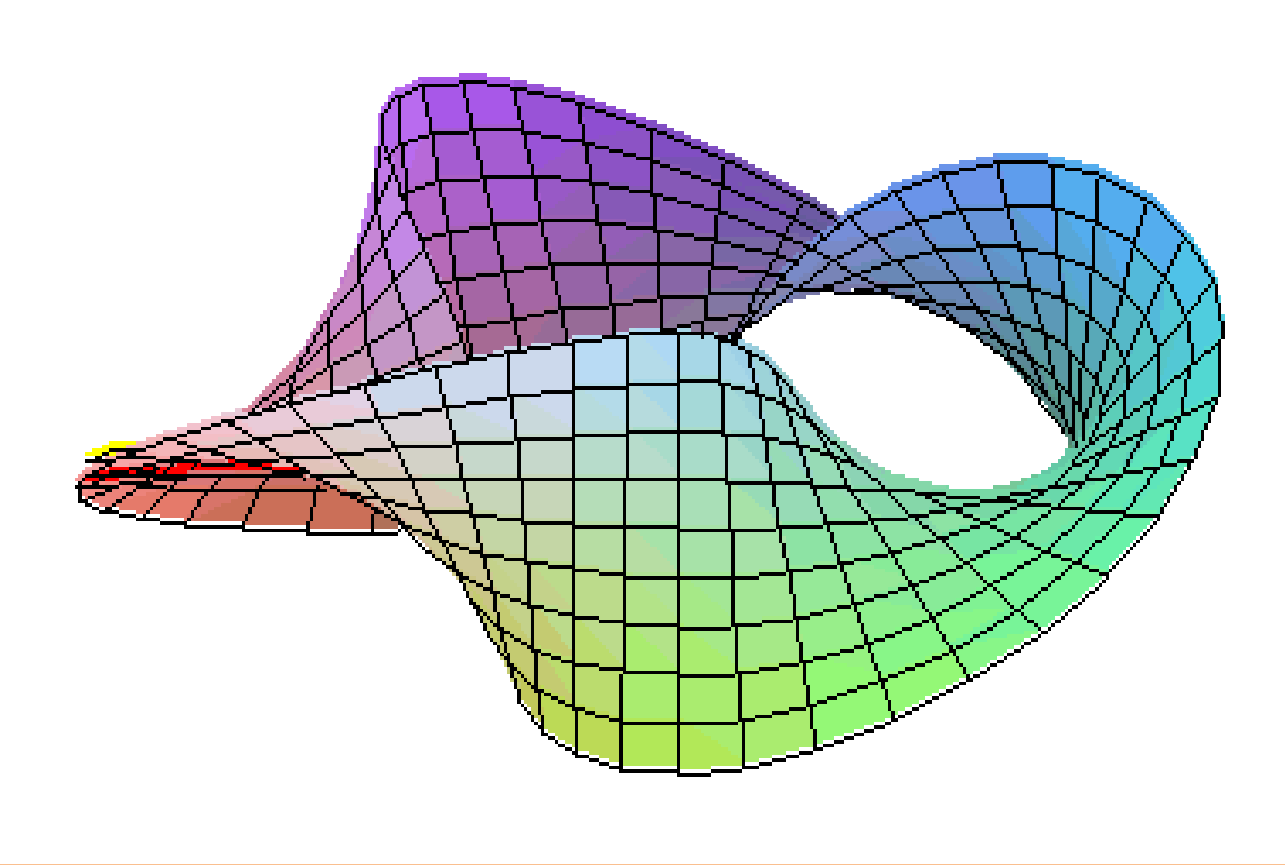
Le ruban de Möbius peut être engendré par un segment pivotant dont le centre décrit un **cercle fixe**.

Un paramétrage correspondant est

$$\begin{cases} x = \left(1 + \frac{t}{2}\cos\frac{v}{2}\right)\cos v \\ y = \left(1 + \frac{t}{2}\cos\frac{v}{2}\right)\sin v \\ z = \frac{t}{2}\sin\frac{v}{2} \end{cases} \qquad \begin{matrix} -1 \leq t \leq 1 \\ 0 < v \leq 2\pi \end{matrix}$$

Ou l’ensemble des solutions de l’équation suivante :

$$x^2y + yz^2 + y^3 - y - 2xz - 2x^2z - 2y^2z = 0.$$



Anneau de Möbius: selon un designer en RO

Ruban de Möbius à 3 demi-tours - Université de Flensburg

Ruban de Möbius: tressé avec un seul brin par J-P Baudry

II- TYPE GRAPHE

Le **type abstrait** graphe est spécifié en utilisant le langage CASL :**C**ommon **A**lgebraic **S**pecification **L**anguage

On retiendra la spécification proposée aux étudiants en 3ième année à l'Université de Brême.

1- Cas de graphe orienté

```
library libraryGrapheOriente
%%*************************************************************
****
%%          SPECIFICATION EN CASL DU TYPE ABSTRAIT GRAPHE
%%
%%   Spécification prouvée: Prouveur Hets/Isabelle
%%                  Auteur:K. OURIACHI, Professeur des universités
%%                              Université de PAU (France)
%%                  Date: 12/04/2005
%%*************************************************************
```

```
from Basic/StructuredDatatypes get SET, LIST
from Basic/Numbers get NAT
```

```
%%-------------------------------------------------------
%%              spécification canonique
%%-------------------------------------------------------


spec GRAPHE0[sort Sommet] [sort Arc] =
  generated type Graphe::= grapheVide  |
                           addSommet (Sommet; Graphe) |
                           addArc(Sommet;Sommet; Arc; Graphe) ?



preds
%% prédicats exprimant les propriétés pertinentes
        estSommetDe :   Sommet * Graphe;
        estArcDe :      Arc    * Graphe

 ops
         source: Arc * Graphe ->? Sommet;
         cible:  Arc * Graphe ->? Sommet
```

```
forall
    n0, n1, s0,t0,s1,t1,s2,t2: Sommet;
    e0, e1, e2 : Arc;
    g0, g1: Graphe

  . def addArc(s0,t0,e0,g0)    <=> not estArcDe(e0,g0)
  . def source(e0, g0)         <=> estArcDe(e0,g0)
  . def cible(e0, g0)          <=> estArcDe(e0,g0)



.not estSommetDe(n0,grapheVide)
.estSommetDe(n0,addSommet (n1,g0)) <=>
                                n0=n1 \/  estSommetDe (n0,g0)
.estSommetDe (n0, addArc(s0, t0, e0, g0)) <=>
                        n0=s0 \/ n0=t0 \/ estSommetDe(n0,g0)
```

```
. not estArcDe(e0,grapheVide)

. estArcDe(e0,addSommet(n0,g0))     <=>   estArcDe(e0,g0)
. estArcDe(e1,addArc(s2,t2,e2,g0)) <=>  e1=e2 \/ estArcDe(e1,g0)



. source(e0, addSommet(n0,g0)) = source(e0,g0)
. source(e1, addArc(s0, t0, e2, g0)) =
                            s0 when e1=e2 else source(e1,g0)


. cible(e0, addSommet(n0,g0)) = cible(e0,g0)
. cible(e1, addArc(s0,t0,e2,g0)) =
                             t0 when e1=e2  else cible(e1,g0)
```

```
. g0 = g1 <=>
    (forall n0: Sommet . estSommetDe(n0,g0) <=>
     estSommetDe(n0,g1)) /\
    (forall e0: Arc . estArcDe(e0,g0) <=> estArcDe(e0,g1)) /\
    (forall e0: Arc . source(e0, g0) = source(e0, g1)) /\
    (forall e0: Arc . cible(e0, g0) = cible(e0, g1) )

end
```

```
%% La spécification précédente peut être enrichie par extension.


%%L'extension ajoute les opérations de suppression de sommet  et
des arcs dans un gaphe:
%%
%%  supSommet : Sommet * Graphe -> Graphe;
%%
%%  supArc: Arc * Graphe -> Graphe
```

```
spec GRAPHE[sort Sommet] [sort Arc] =
                  GRAPHE0 [sort Sommet] [sort Arc]
    then
%% extension de la spécification GRAPHE %
    ops
        suppSommet: Sommet  * Graphe -> Graphe;
        suppArc:    Arc   * Graphe  -> Graphe
```

```
 forall n0,n1,n2:Sommet; e0,e1,e2:Arc;  g0,g1:Graphe

. suppSommet(n0, grapheVide) = grapheVide
. suppSommet(n0, addSommet(n1,g0))=
                g0  when n0 = n1
               else addSommet(n1,suppSommet(n0,g0))

. suppSommet(n0, addArc(n1,n2,e0,g0))=
            suppSommet(n0,g0) when n0= n1 \/ n0 =n2
           else addArc(n1,n2,e0,suppSommet(n0,g0))

. suppArc(e0, grapheVide) = grapheVide
. suppArc (e0, addSommet(n1,g0)) = addSommet(n1,suppArc(e0,g0))
. suppArc (e0, addArc(n1,n2,e1,g0)) =
                suppArc(e0,g0) when e0=e1
                else addArc(n1,n2,e1,suppArc(e0,g0))
```

end

Opérations de base sur le graphe

- Toujours commencer par le **graphe vide**

```
grapheVide()
```

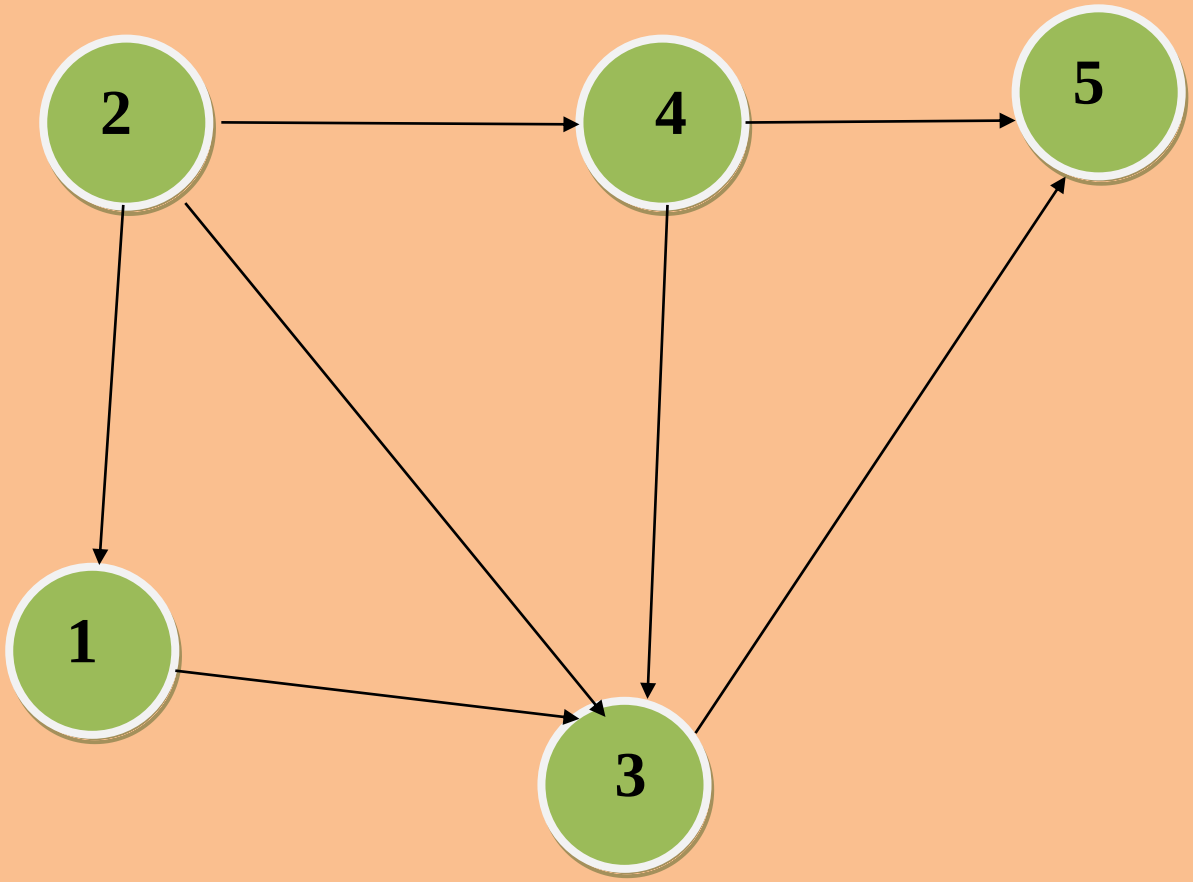
- ensuite **ajouter** les **sommets** et **arcs** nécessaires

```
addSommet (Sommet,Graphe)
addArc(Sommet,Sommet,Arc,Graphe)
```

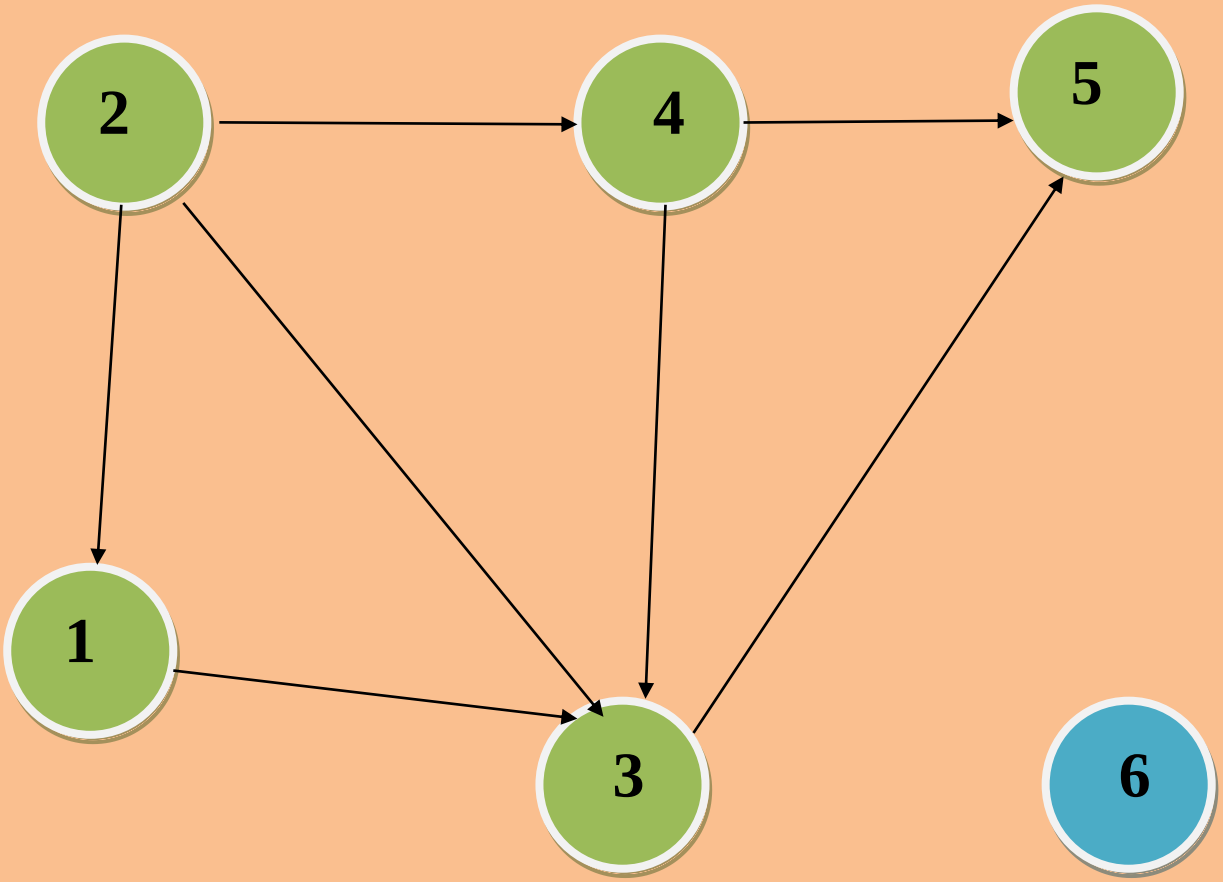
- pour la mise à jour, on peut **supprimer** des **sommets** ou **arcs**

```
suppSommet(Sommet,Graphe)
suppArc(Arc,Graphe)
```

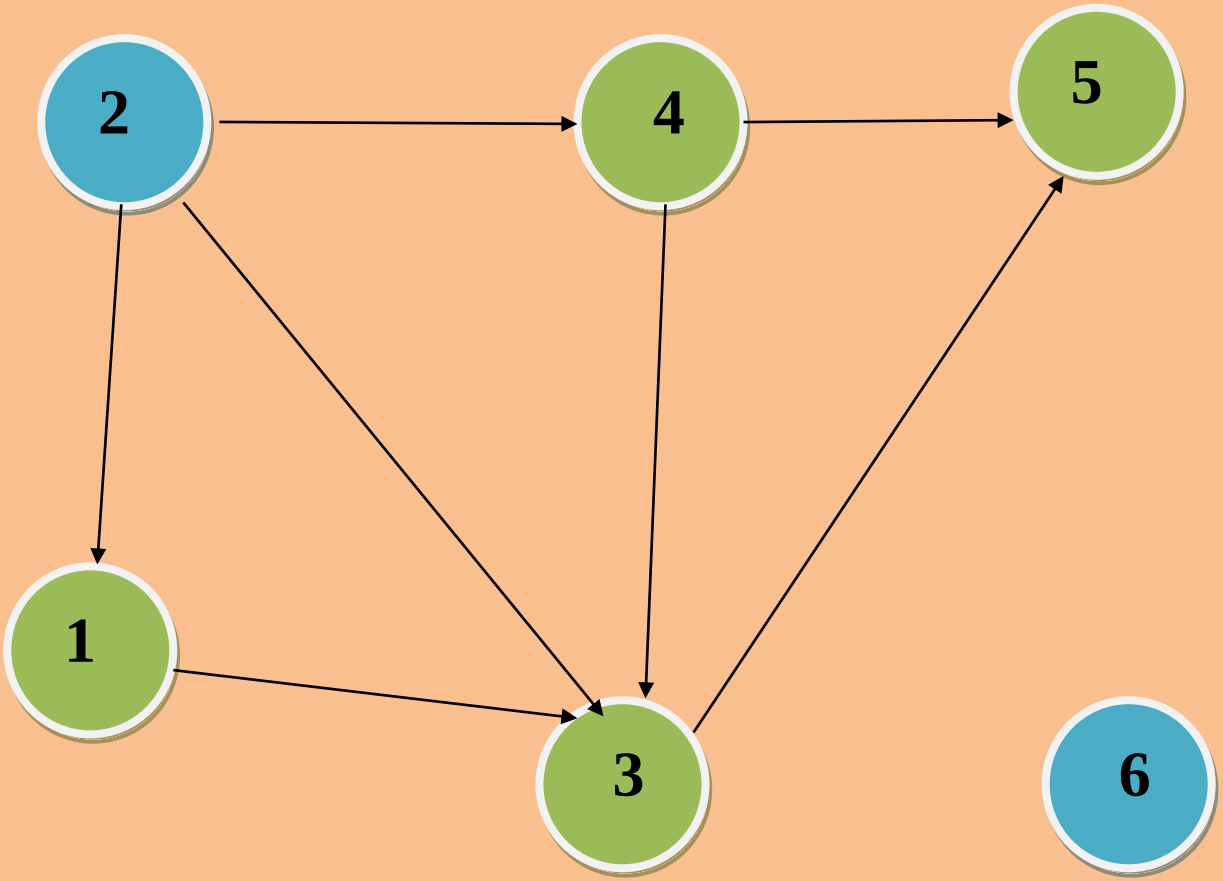
Graphe de départ



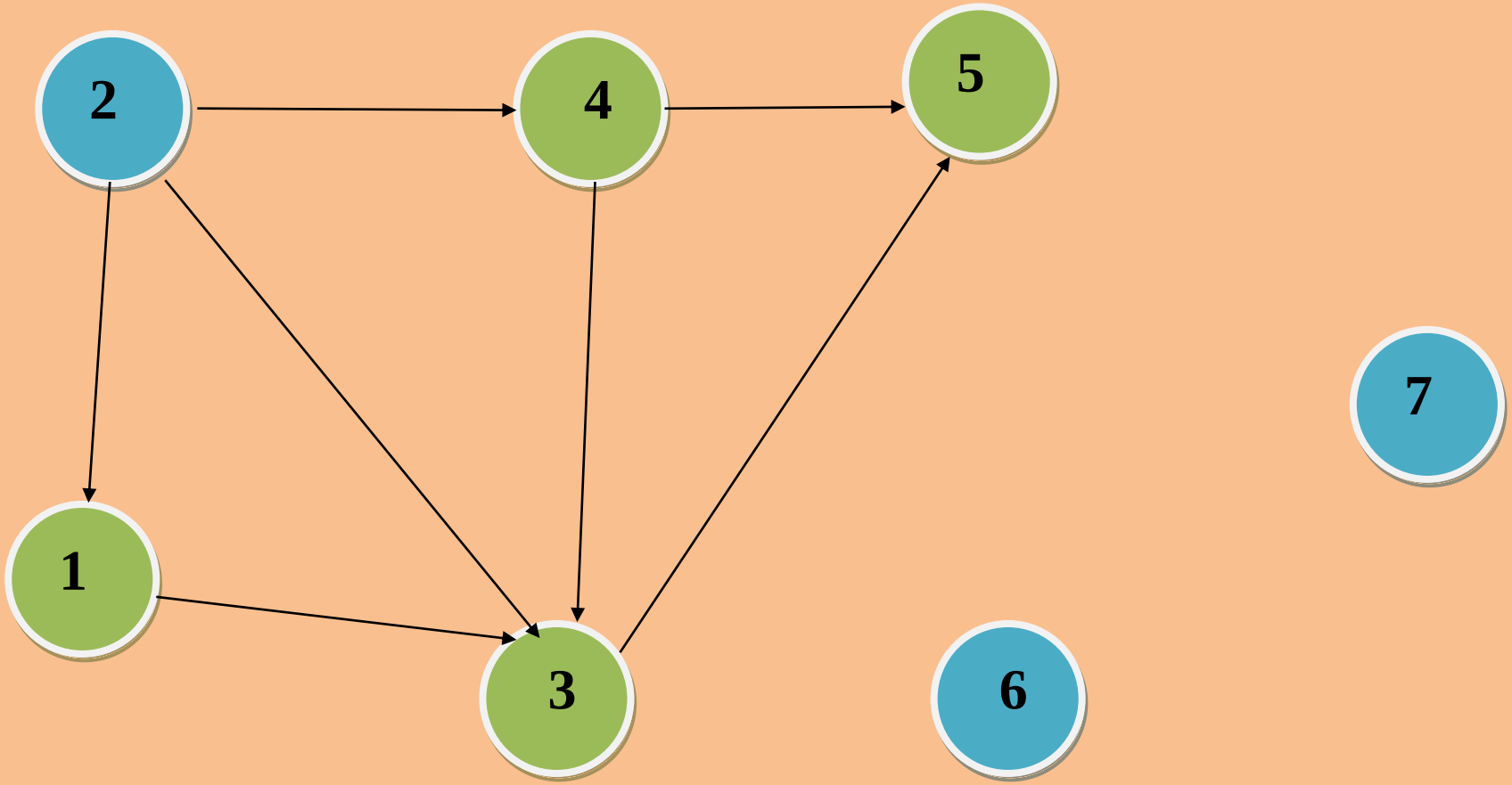
addSommet(**6**, G)



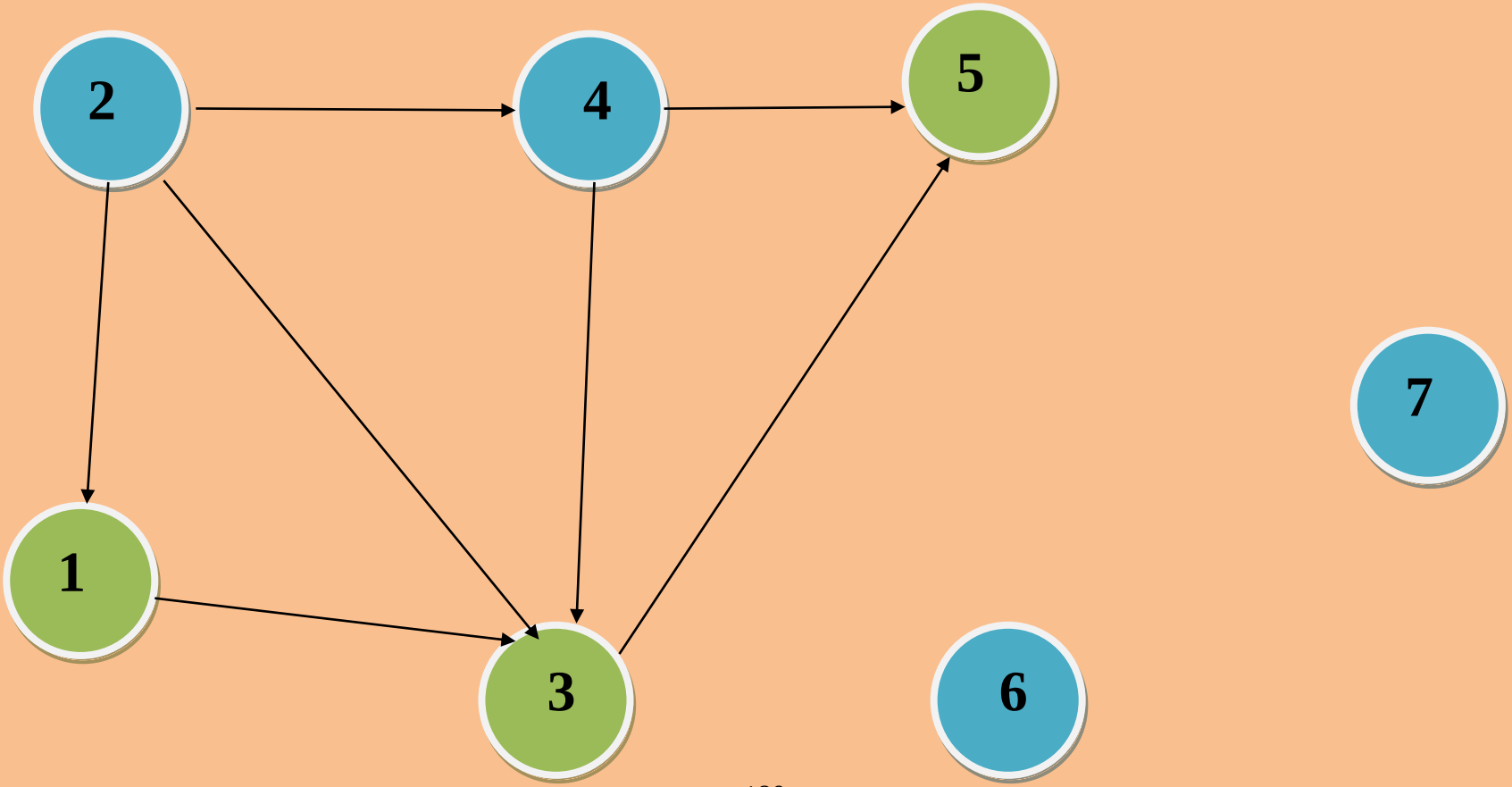
addSommet(**2**, G)



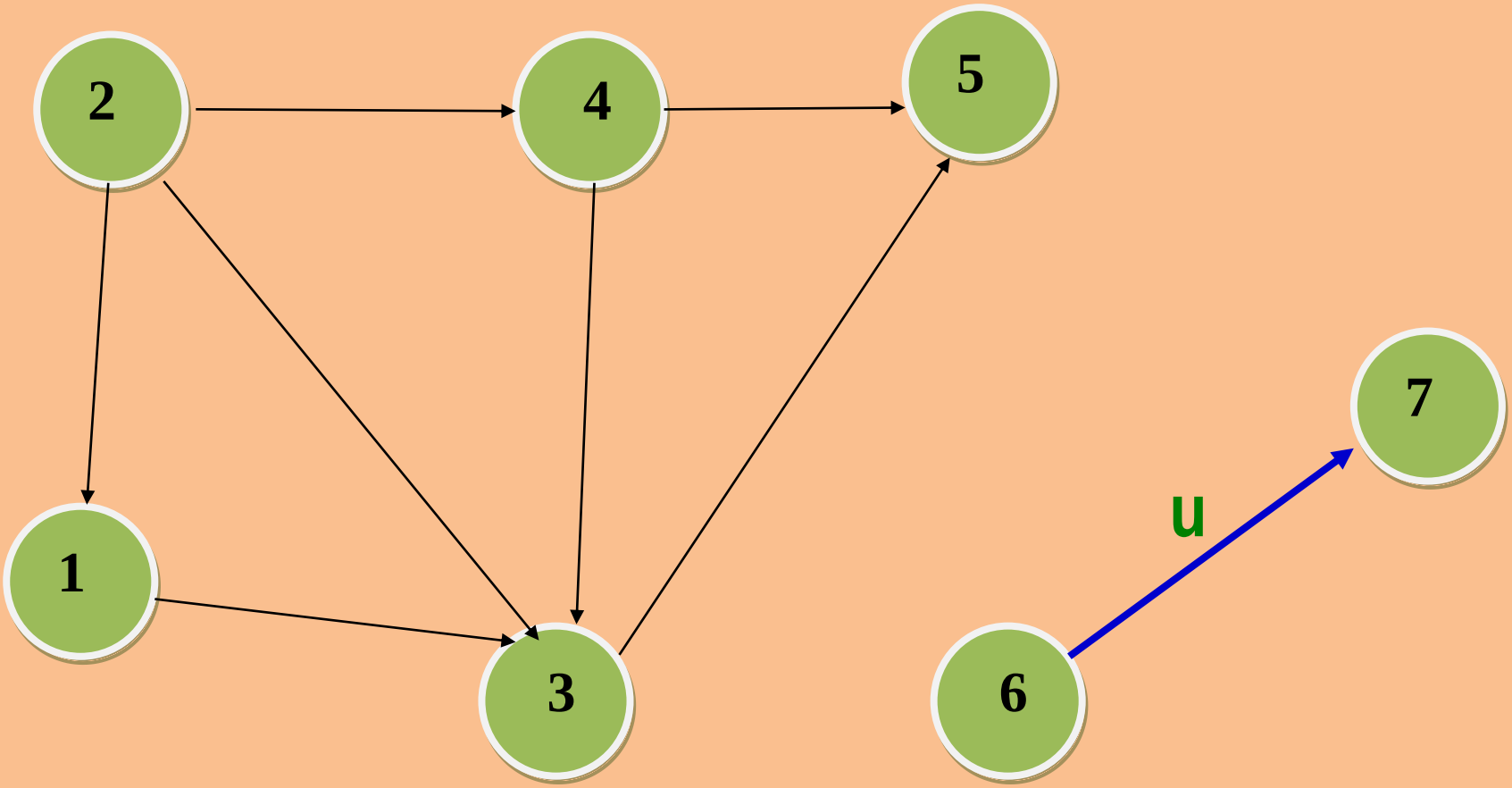
addSommet(**7**, G)



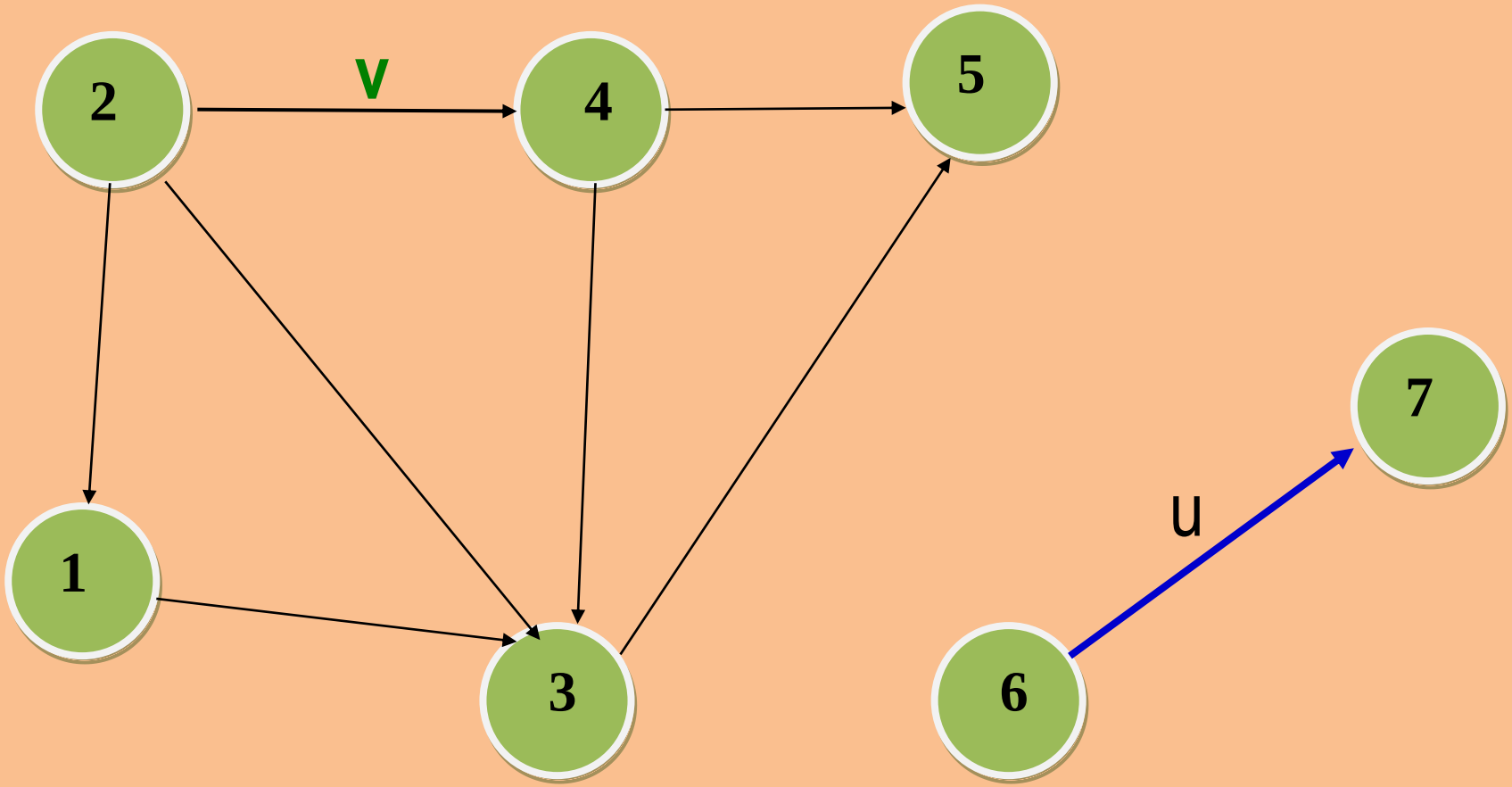
addSommet(**4**, G)



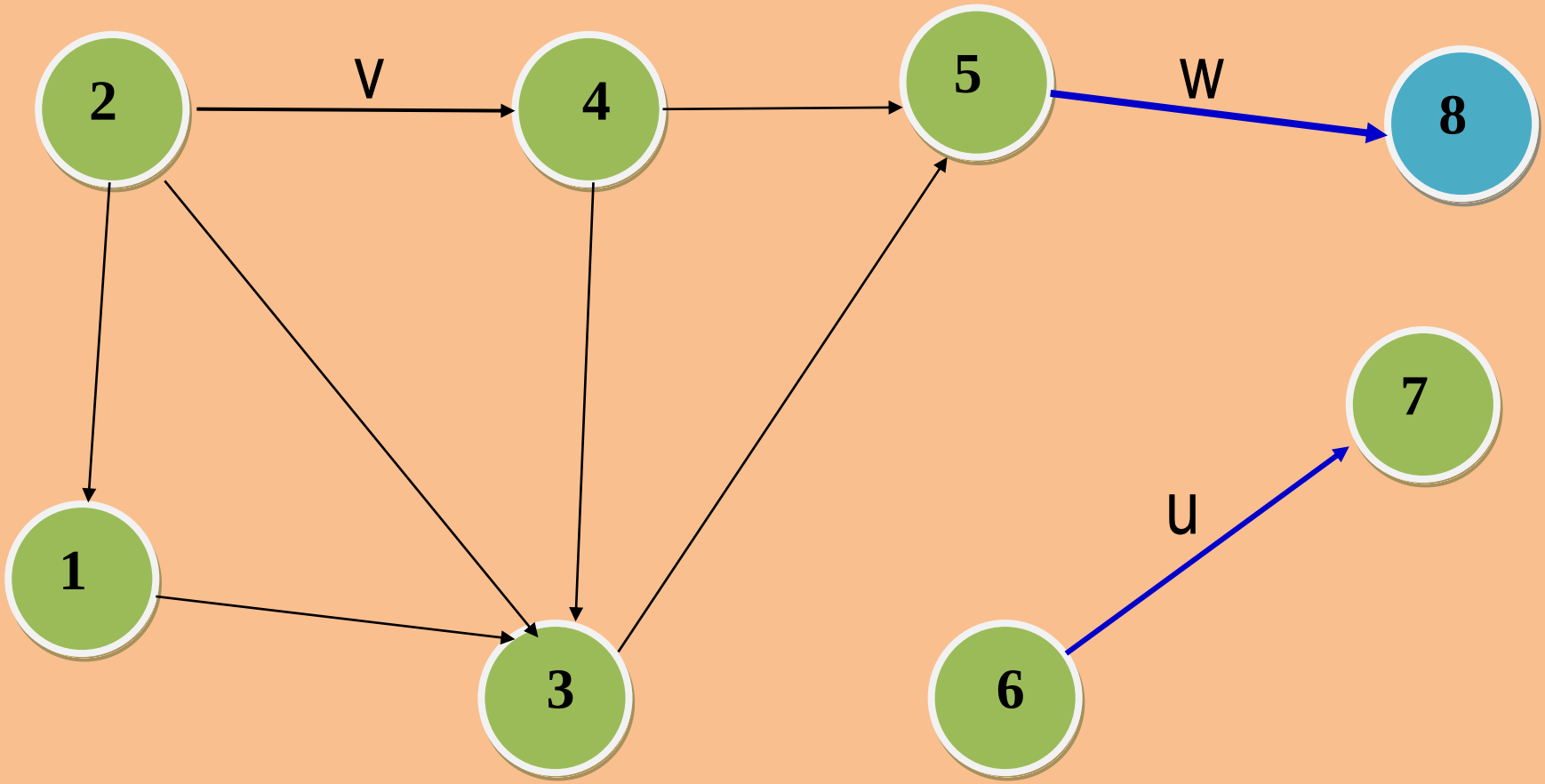
addArc(6,7,u, G)



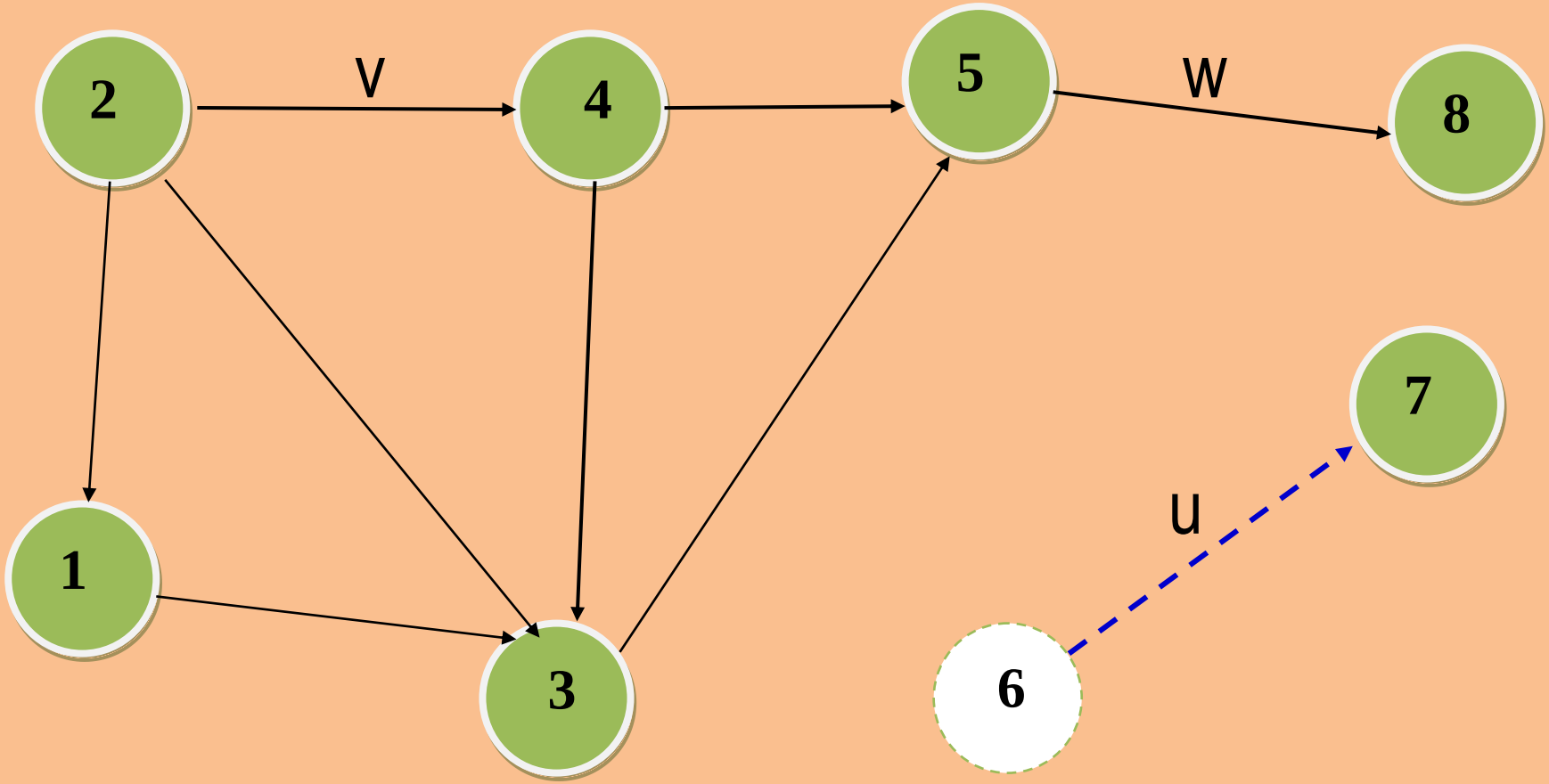
addArc(2,4,**v**, G): opération non autorisée



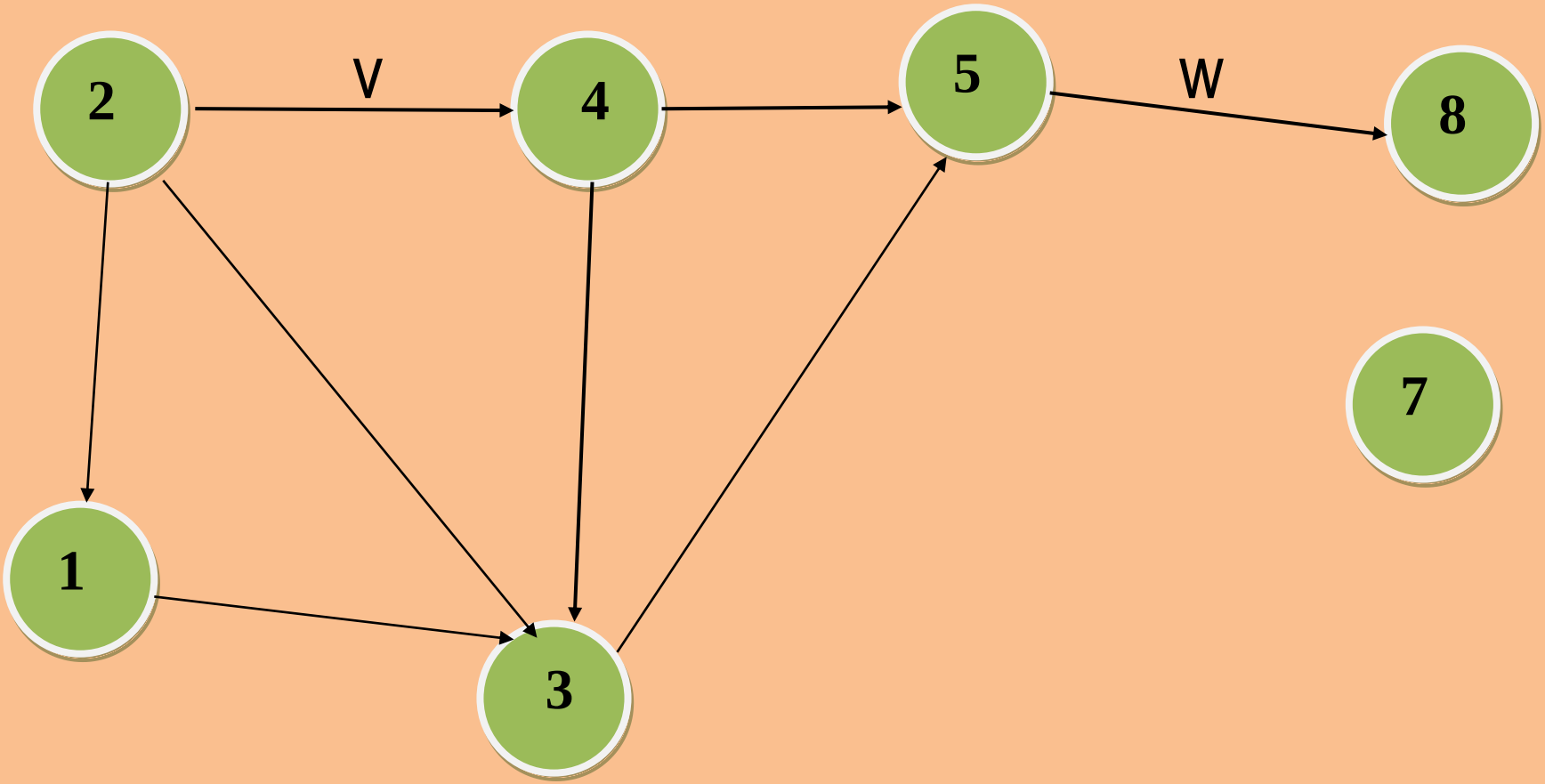
addArc(5,**8**,w, G)



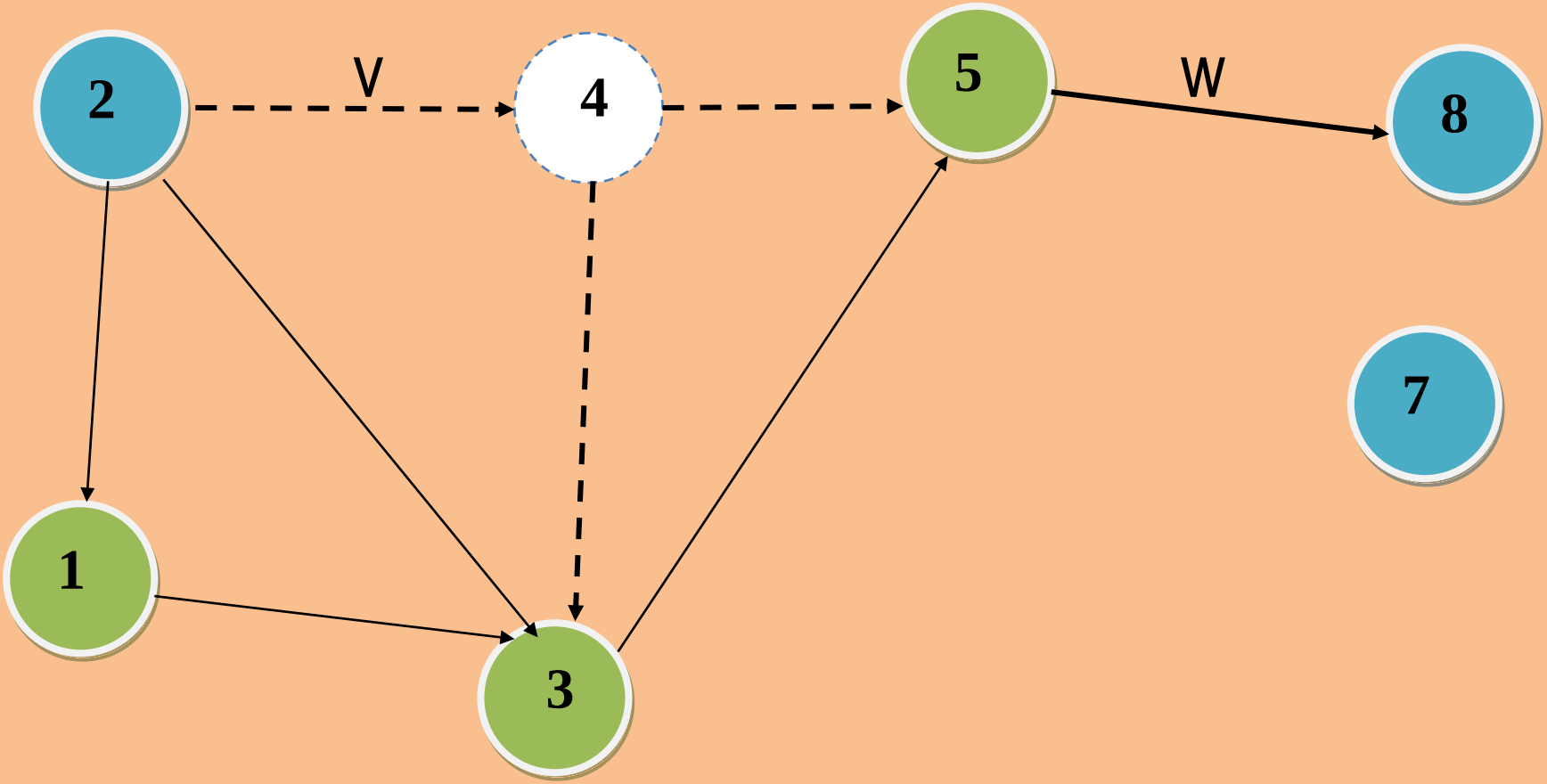
suppSommet(**6**, G)



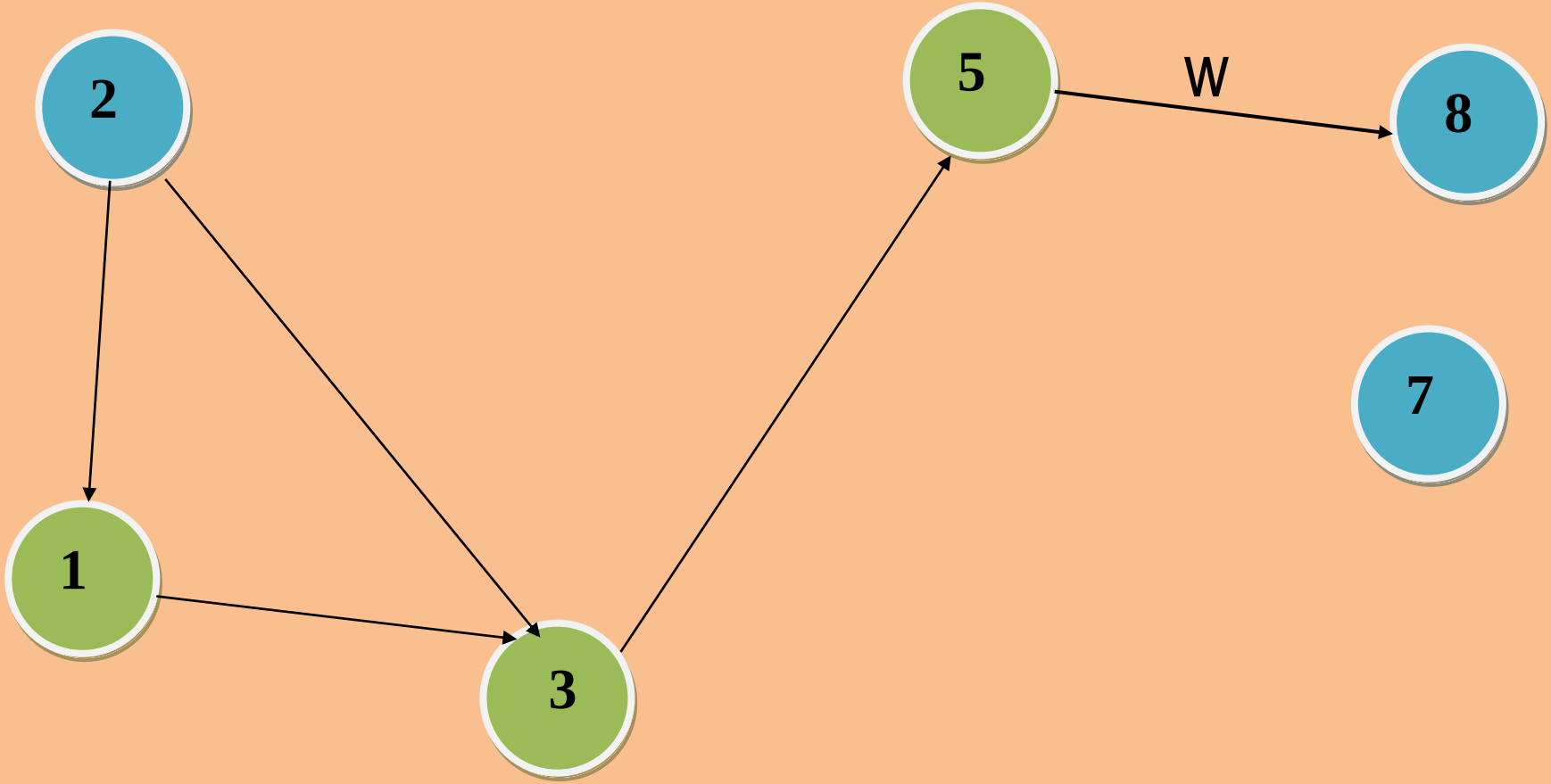
suppSommet(**9**, G)



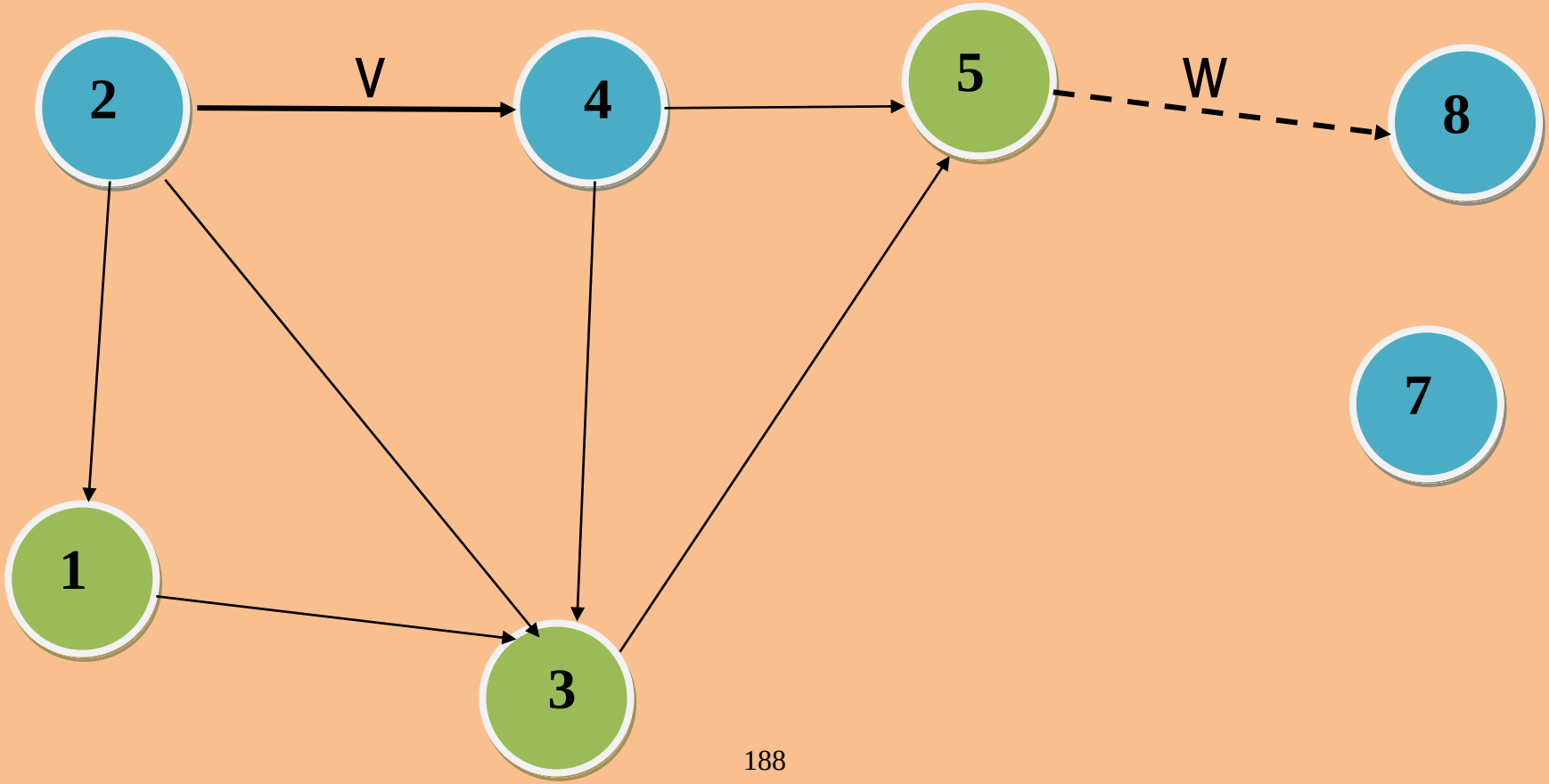
suppSommet(**4**, G)



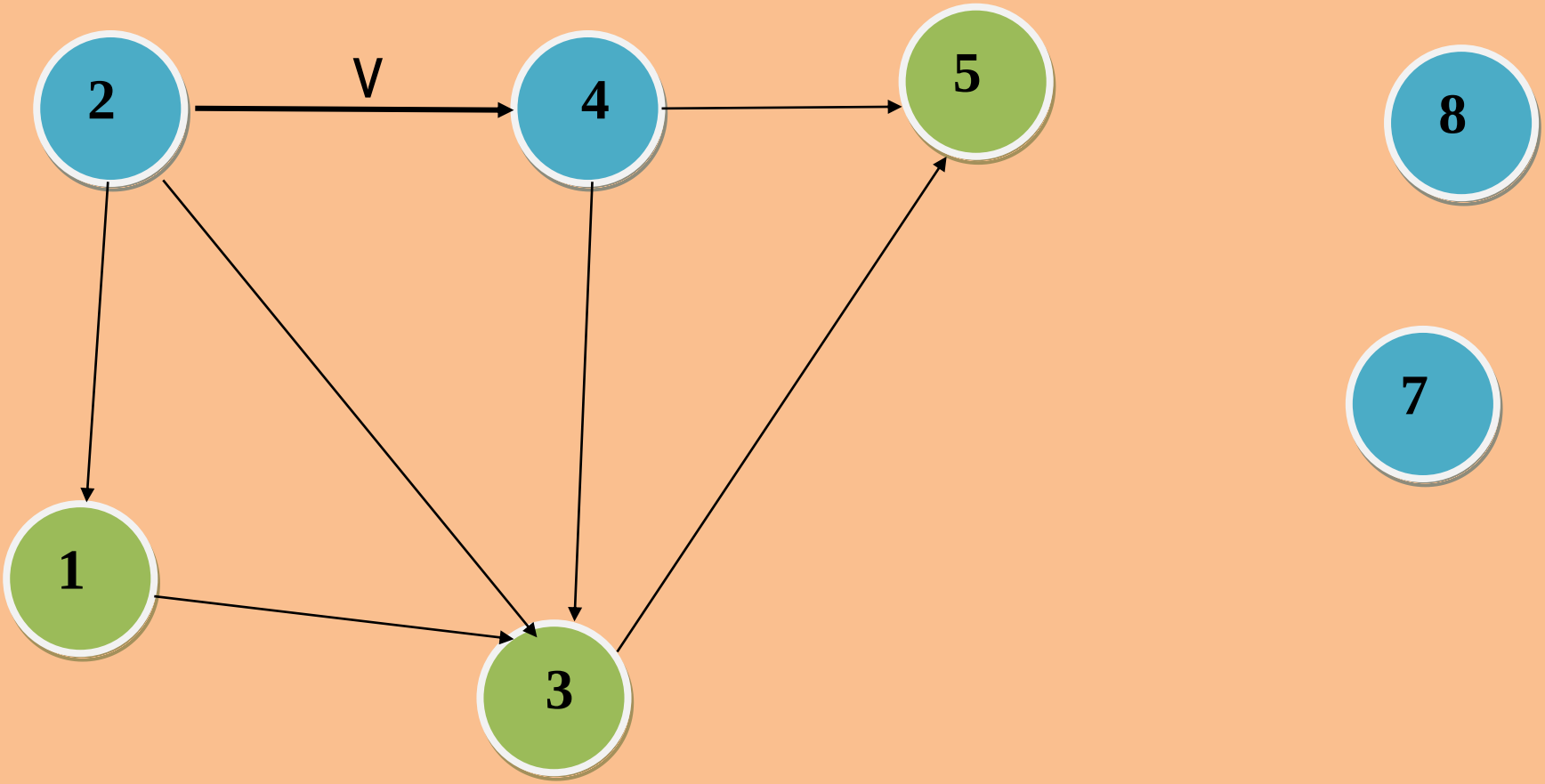
Graphe résultant



suppArc(5,8,w, G)



suppArc(7,8, x, G)



2- Cas de graphe non orienté

Le type abstrait graphe non orienté est spécifié par **extension** du type abstrait graphe orienté.

```
spec SYM_GRAPHE [sort Nœud ] [sort Arc] =
        GRAPHE [sort Nœud  ] [sort Arc]
```

```
then

sort  Sym_Graphe = {g: Graphe • ∀ s, t: Nœud
; e,e': Arc
          • estArcDe(e,g)∧ source(e,g)=s ∧
cible(e,g)=t
           ⇔   estArcDe(e',g)∧ source(e',g)= t
∧ cible(e',g)=s}

type  Sym_Graphe ::=
     empty_Graphe
   | addNoeud (n: Noeud  ; g:Sym_Graphe)
   | addArcSym(s, t: Noeud  ; e: Arc; g:
Sym_Graphe)?
End
```

Librairie BGL

Que propose Boost Graph ?

Boost Graph Library (BGL) propose une **interface standard** pour manipuler des graphes.

Il est donc possible d'utiliser les **algorithmes spécifiques** pour les graphes fournis par Boost Graph.

I. Comment créer un graphe générique ?

BGL propose de nombreux outils.

Cependant, il est possible d'utiliser la classe **adjacent_list** comme boîte à outils généraliste.

```
#include <boost/graph/adjacency_list.hpp>
```

Les informations relatives à chaque élément d'un graphe sont enregistrées dans des structures :

```
struct VertexProperties { ... };
struct EdgeProperties { ... };
struct GraphProperties { ... };
```

On peut alors créer un graphe générique basé sur **adjacent_list.**

Pour des raisons pratiques, il est préférable de créer un **typedef** .

Définition du graphe

```
typedef boost::adjacency_list<
   boost::vecS, boost::vecS, boost::bidirectionalS,
   boost::property<boost::vertex_bundle_t, VertexProperties>,
   boost::property<boost::edge_bundle_t, EdgeProperties>,
   boost::property<boost::graph_bundle_t, GraphProperties>
> Graph;
```

Création du graphe

```
Graph   g;
```

II-Comment manipuler les sommets ?

Le type correspondant à la description d'un sommet est accessible via la classe de traits **graph_traits** :

```
typedef boost::graph_traits<Graph>::vertex_descriptor vertex_t;
```

La fonction **add_vertex** permet d'ajouter un sommet dans un graphe.

Les informations attachées au sommet peuvent être spécifiées lors de la création :

```
struct VertexProperties
{
  std::string name;
  unsigned id;
  VertexProperties() : name(""), id(0) {}
  VertexProperties(std::string const& n, unsigned i) : name(n), id(i) {}
};
```

Appel du constructeur par défaut

```
vertex_t v1 = boost::add_vertex(g);
```

Appel du constructeur avec paramètres

```
vertex_t v2 = boost::add_vertex(VertexProperties("toto", 12), g);
```

L'opérateur **[ ]** permet de récupérer les informations d'un sommet.

Référence constante

```
VertexProperties const& vertexProperties = g[v1];
std::cout << "Vertex name : " <<  vertexProperties.name << std::endl;
```

Référence non constante

```
VertexProperties& vertexProperties = g[v2];
v2.id = 17;
```

La fonction **num_vertices** permet de connaître le nombre de sommet dans un graphe :

```
boost::graph_traits<Graph>::vertices_size_type s = num_vertices(g);
```

III-Comment manipuler les arcs ?

Le type correspondant à la description d'un arc est accessible via la classe de traits **graph_traits** :

```
typedef boost::graph_traits<Graph>::edge_descriptor edge_t;
```

La fonction **add_edge** permet de connecter deux sommets.

Les informations attachées à l'arc peuvent être spécifiées lors de la création :

```
struct EdgeProperties
{
   float weight;
   float distance;
   EdgeProperties() : weight(0.0), distance(0.0) {}
   EdgeProperties(float w, float d) : weight(w), distance(d) {}
};
```

Appel du constructeur par défaut

```
std::pair<Graph::edge_descriptor, bool> e1 = boost::add_edge(v1, v2, g);
```

Appel du constructeur avec paramètres

```
std::pair<Graph::edge_descriptor, bool> e2 =
boost::add_edge(v1, v2, EdgeProperties(1.0, 50.0), g);
```

L'opérateur **[ ]** permet de récupérer les informations d'un arc.

Il retourne une référence, ce qui permet de pouvoir modifier les informations :

Référence constante

```
EdgeProperties const& edgeProperties = g[e1];
std::cout << "Edge weight : " <<  edgeProperties. weight << std::endl;
```

Référence non constante

```
Référence non constante
EdgeProperties& edgeProperties = g[e2];
e2. distance = 103.8;
```

La fonction **num_edges** permet de connaître le nombre d'arcs dans un graphe :

```
boost::graph_traits<Graph>::edges_size_type s = num_edges(g);
```

Il est possible de récupérer les descripteurs des sommets associés à un arc à l'aide des fonctions **source()** et **target()** :

```
Graph::target_descriptor v1 = boost::target(e1, g);
Graph::target_descriptor v2 = boost::source(e5, g);
if (v1 == v2)
   std::cout << "Same vertex" << std::endl;
```

IV- Comment supprimer des sommets et des arcs ?

Boost Graph fournit plusieurs fonctions pour supprimer des éléments d'un graphe.

Suppression de tous les sommets et arcs d'un graphe

```
clear(g);
```

Suppression des arcs partant ou arrivant à un sommet

```
clear_in_edges(v1, g);
clear_out_edges(v2, g);
```

Suppression de tous les arcs partant et arrivant à un sommet

```
clear_vertex(v1, g);
```

Suppression d'un arc

```
remove_edge(v1, v2, g);
remove_edge(e3, g);
```

Suppression d'un sommet

// il est nécessaire de supprimer les arcs liés à un sommet...

```
clear_vertex(v1, g);
```

// ... avant de le supprimer

```
remove_vertex(v1, g);
```

III- REPRESENTATION DE GRAPHE

Il existe deux classes de représentations pour les objets de type GRAPHE:

-la **matrice d'adjacence**,
-les **listes d'adjacence**

1- Représentation par matrice d'adjacence

Soit un graphe orienté

$$G = (S,A)$$

tel $|S| = n$

La technique est centrée sur la représentation des **arcs** du graphe.

Elle permet de représenter G l'aide d'une matrice **M** appelée **matrice d'adjacence**.

1.1- Calcul de la matrice d'adjacence

Les lignes et les colonnes de la matrice M représentent les sommets du graphe G.

Notons M_{ij} l'élément appartement à la ligne **i** et à la colonne **j** de la matrice M.

$$M_{ij} = 1$$

signifie que :

$$\mathbf{i} \rightarrow \mathbf{j} \in A$$

$$M_{ij} = 0$$

signifie que :

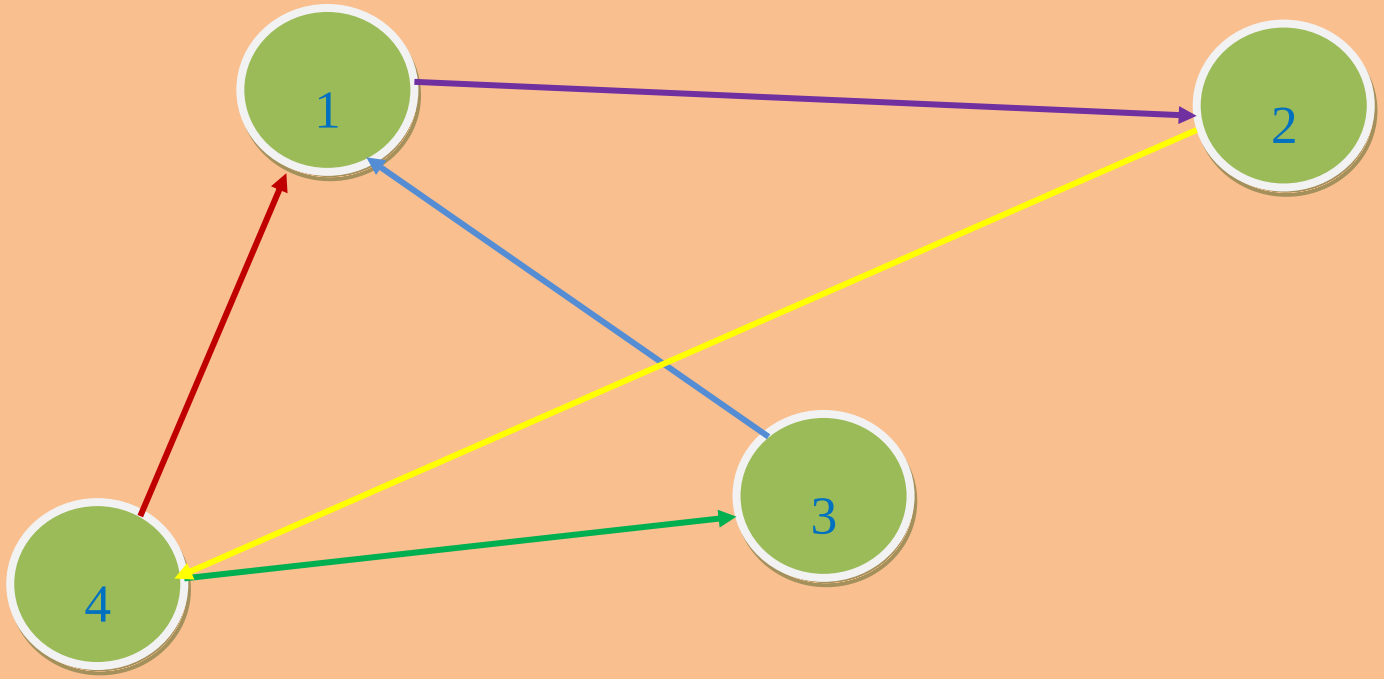
$$i \rightarrow j \notin A$$

La matrice d'adjacences est une matrice **binaire carrée** d'ordre n.

Il n'y a que des zéros sur la diagonale.

La présence d'un 1 sur la diagonale indiquerait une **boucle** : ce qui est interdit par convention.

Le graphe suivant :



est représenté par la matrice :

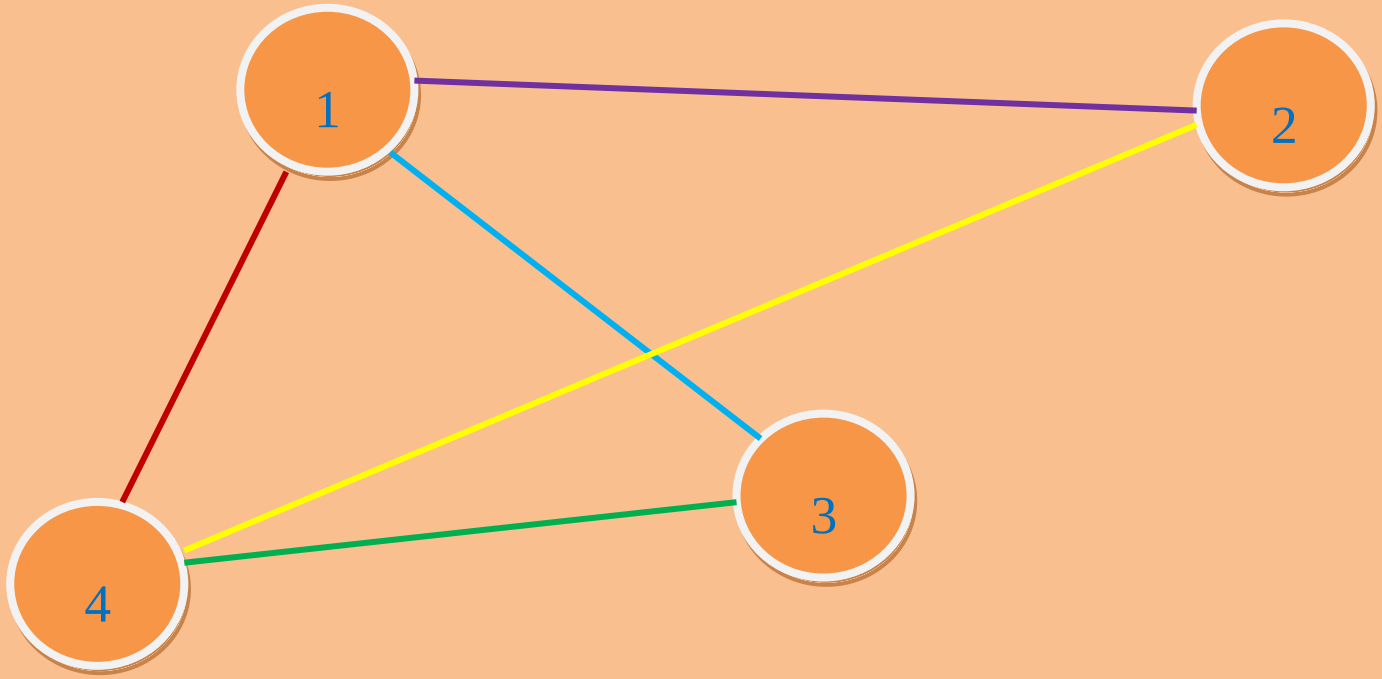
$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Remarque :

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

$$M_{ij} = M_{ji}$$

Le graphe suivant :



est représenté par la matrice :

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

1.2- Calcul du nombre de chemins

Le calcul des chemins du graphe G se ramène à un calcul des puissances successives de M :

$$M^2, M^3, \ldots, M^{p-1}, M^p, \ldots$$

Où M^i désigne la puissance i-ième de la matrice M.

Le nombre M^p_{ij} est égal au **nombre de chemins** de longueur **p** de G qui:

- partent du sommet **i**
- et arrivent au sommet **j**.

La preuve se fait *par récurrence* sur **p**.

- pour **p** = 1 le résultat est évident.

M^{1}_{ij} = 1 si arc ij $\in A$ ou M^{1}_{ij} = 0 sinon

- pour **p** >1 on a par hypothèse

M^{p-1}_{ik}

le **nombre** de chemins de longueur **p-1** allant de *i* à *k*

- pour obtenir les chemins de longueur **p** de *i* à *j*, on fait la somme de ces nombres pour tous les arcs de *k* vers *j* :

$M^{p-1}_{i1} \times M_{1j}$ + (pour k=1)

$M^{p-1}_{i2} \times M_{2j}$ + (pour k=2)

…

+

$M^{p-1}_{in} \times M_{nj}$ (pour k=n)

$= M^{p-1} \times M$

$= M^{p}$ (d'où le résultat)

Pour le calcul de M^p, la complexité d'un algorithme naïf est en $\Theta(p.n^3)$

On peut la réduire à $\Theta(\log p.n^3)$ grâce à l'algorithme suivant :

```
Puissance_p(M,p)
  begin
  if p=1  then return ( M )
          else
                  begin
                  J ←M^Ent(p/2)
                  K←J x J
                  if  pair(p)  then return (K)
                               else return  (K x M)
                  end
 end
```

1.4- Avantages

1- La représentation matricielle est pratique pour **tester l'existence** d'un arc ou arête.

2- Il est plus facile d'**ajouter** ou **retirer** un arc ou une arête.

3- Il est facile de **parcourir** les successeurs ou prédécesseurs d'un sommet.

1.5- Inconvénient

1- Il demande **n tests** pour détecter les successeurs ou prédécesseurs d'un sommet s quel que soit leur nombre.

2- Il en est de même du **calcul de $d°^+$ et $d°^-$** de s.

3- La **consultation complète** de la matrice de dimension **n** requiert un **temps** d'ordre $\mathbf{n^2}$.

4- La représentation matricielle exige un **espace mémoire** de $\Theta(\mathbf{n^2})$

5- Cela interdit d'avoir des algorithmes d'ordre inférieur à $\mathbf{n^2}$ pour des graphes à n sommets n'ayant que **peu d'arcs** (arêtes).

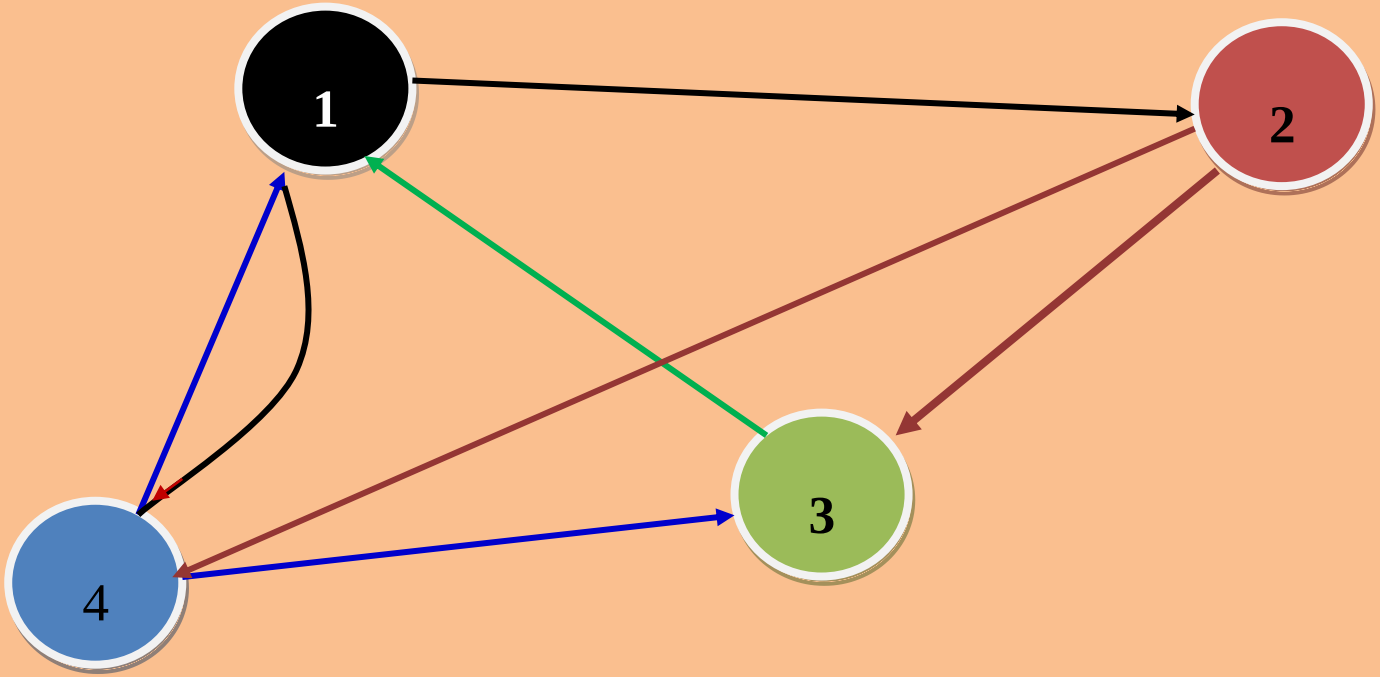
2- Listes d'adjacence

Cette technique est centrée sur la représentation **des sommets**.

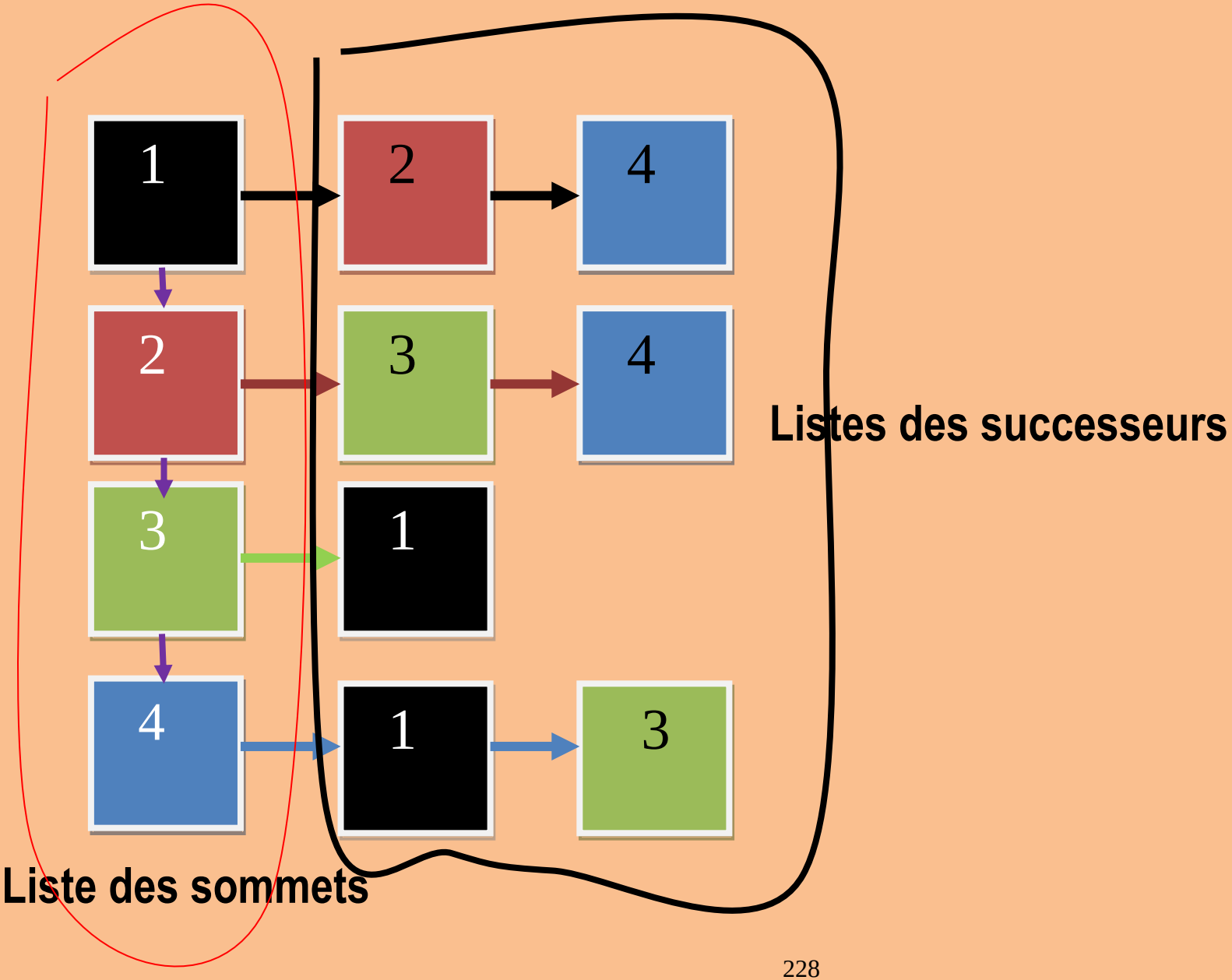
Elle consiste à :

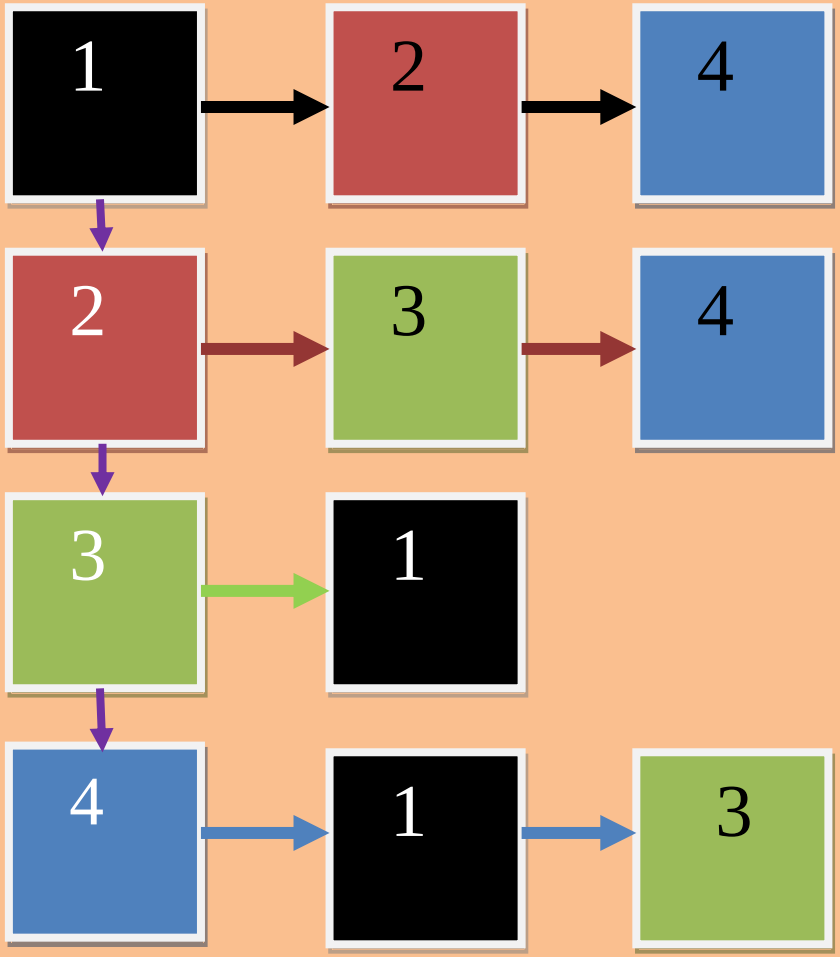
- représenter l'ensemble des sommets,
- associer à chaque sommet la liste de ses **successeurs** rangés dans un **ordre** arbitraire.

Le graphe suivant



est représenté par les listes suivantes :




1
2
4
2
3
4
3
1
4
1
3

Les listes générées par cette représentation sont appelées **listes d'adjacences**.

Si le nombre de sommet n'évolue pas, les **listes des successeurs** sont accessibles :

- à partir d'un **tableau**,
- tableau qui contient, pour chaque sommet, un pointeur vers sa **tête** de liste.

2.1-Avantages

1- L’espace mémoire utilisé pour un graphe **orienté** avec **n** sommets et **p** arcs est en O(**n+p**).

2- Dans le cas d’un graphe **non orienté** avec **n** sommets et **p** arêtes, l’espace mémoire utilisé est en O(**n+2p**).

3- Pour un traitement sur les **successeurs** d'un sommet **s**:

nombre de sommets parcourus = $\mathbf{d^{\circ +}(s)}$

4- Un algorithme qui traite tous les arcs d'un graphe de **p** arcs peut donc être d'ordre **p**.

2.2- Inconvenients

1-Pour tester s'il existe un arc **x** $\rightarrow$**y** (arête **x-y**), la représentation exige :

- un temps d'ordre **n**
- dans le pire des cas.

Le pire des cas :

- la liste d'adjacence est de longueur **n-**1,
- **y** est en fin de liste

2- Il en va de même pour **ajouter** un arc ou une arête (avec test de non répétition).

3- Elle ne permet pas de calculer facilement les opérations relatives aux **prédécesseurs**:

d°-(s **) , ième_pred(**i,s,g**)**

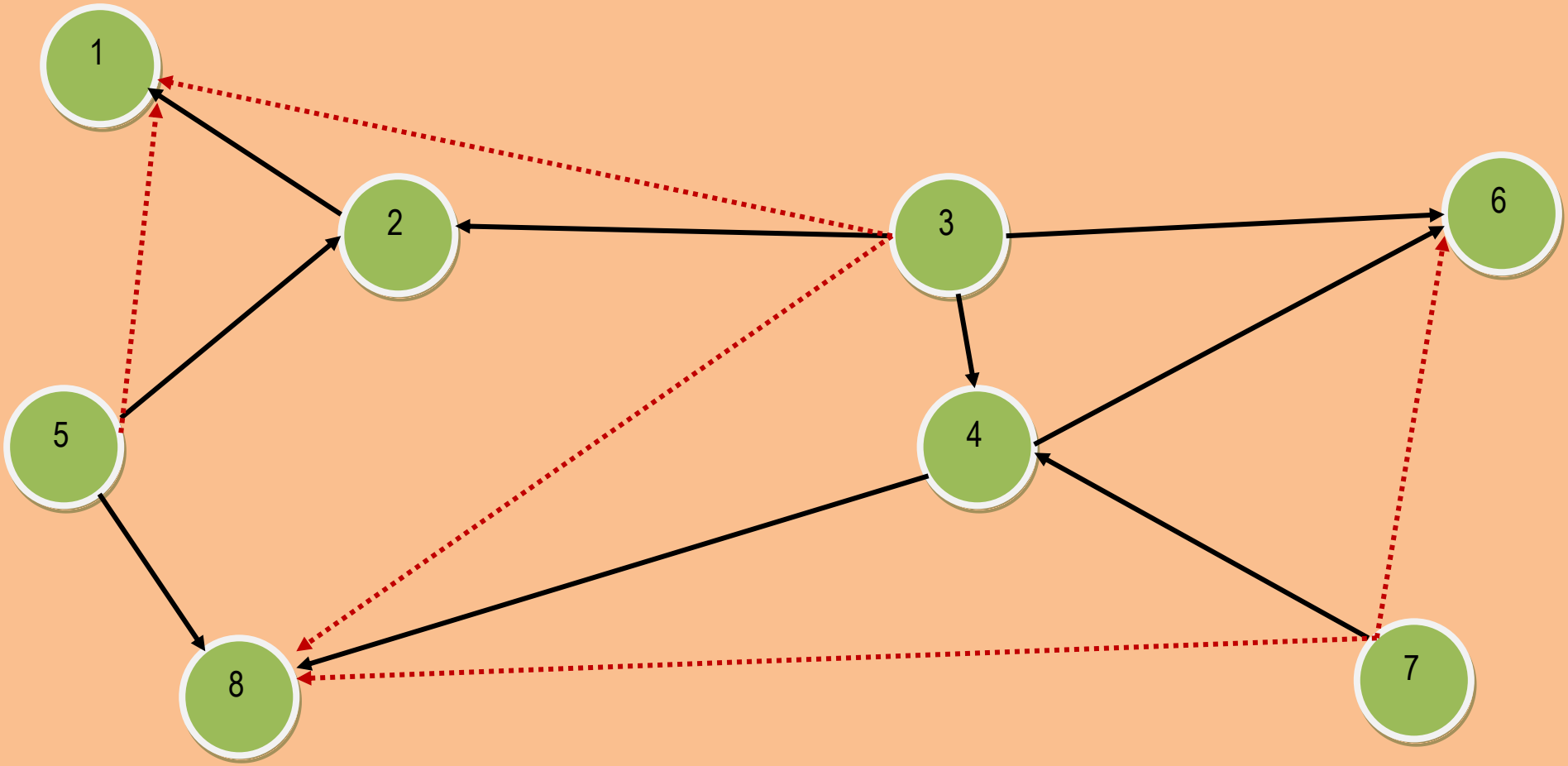
IV- FERMETURE TRANSITIVE D'UN GRAPHE

La **fermeture transitive** d'un graphe G = (S, A), est un graphe noté :

$$G^*=(S, A^*)$$

tel que :

$\mathbf{i} \rightarrow \mathbf{j} \in A^* \Leftrightarrow$ il existe dans G un chemin de **i** vers **j**.


1
2
3
6
5
4
8
7

Le calcul de la fermeture transitive d'un graphe G se fait :

- en répétant le produit de matrices M^p , p=1,2,3
- jusqu'à obtenir une matrice dont la valeur ne change plus.

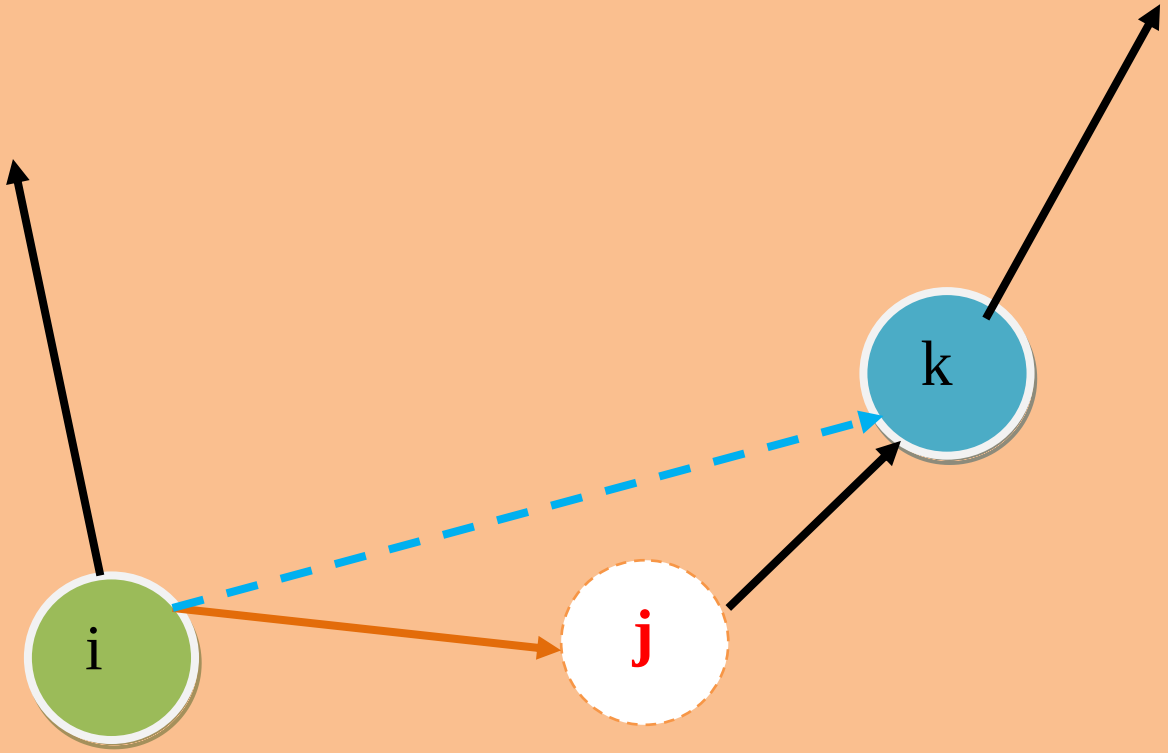
On remplace dans ce calcul :

- **'+'** par un '$\vee$'(**OU** logique)
- **'x'** par un '$\wedge$' (**ET** logique)

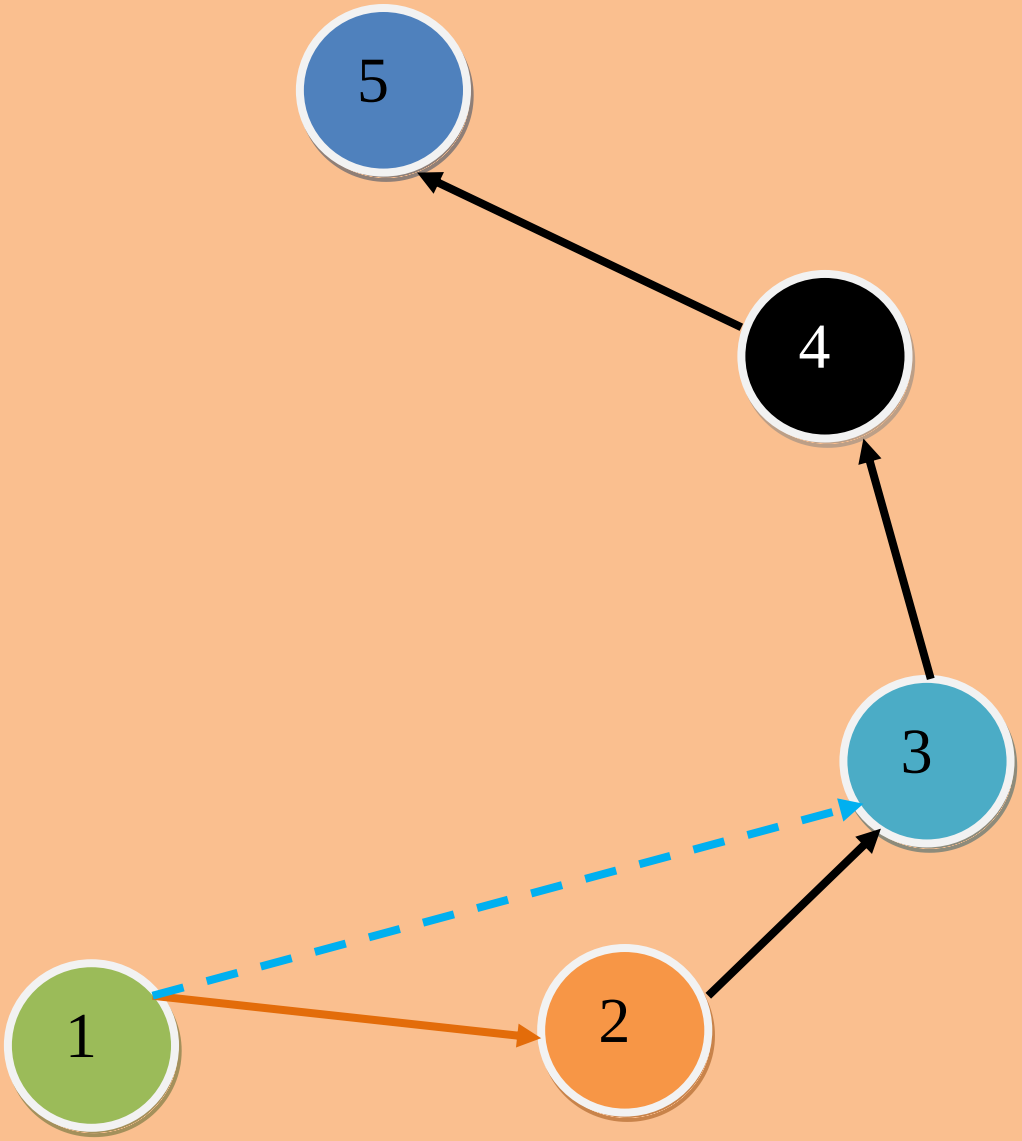
Algoritme Roy et Warshall

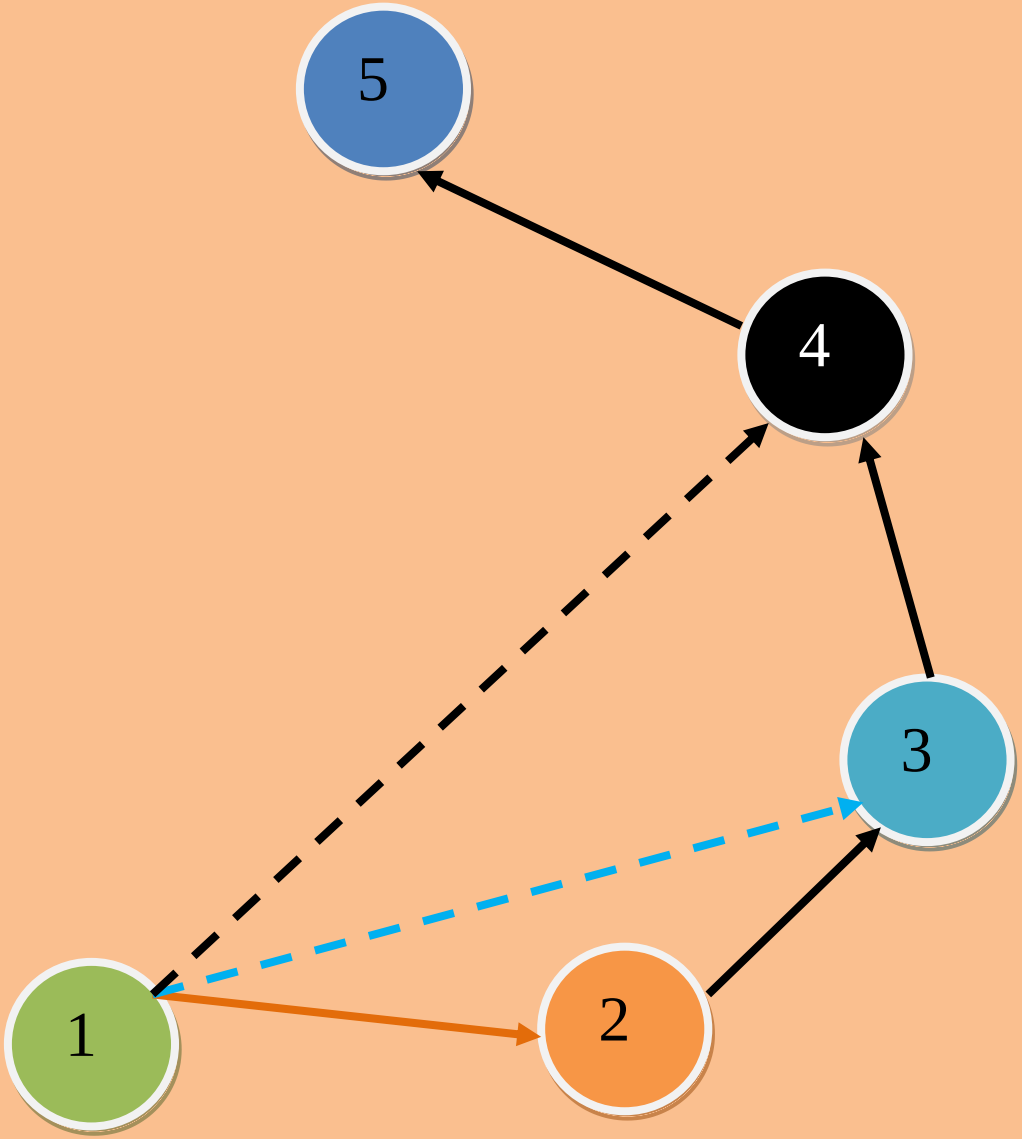
```
Phi (M: GrapheMat; j: integer; n: integer)
    var
        i, k: Integer;
    begin
    for i := 1 to n do
        if (M[i, j] = 1) then
                        for k := 1 to n do
                        if (M[j, k] = 1) then
                                        M[i, k] := 1;
    end;
```

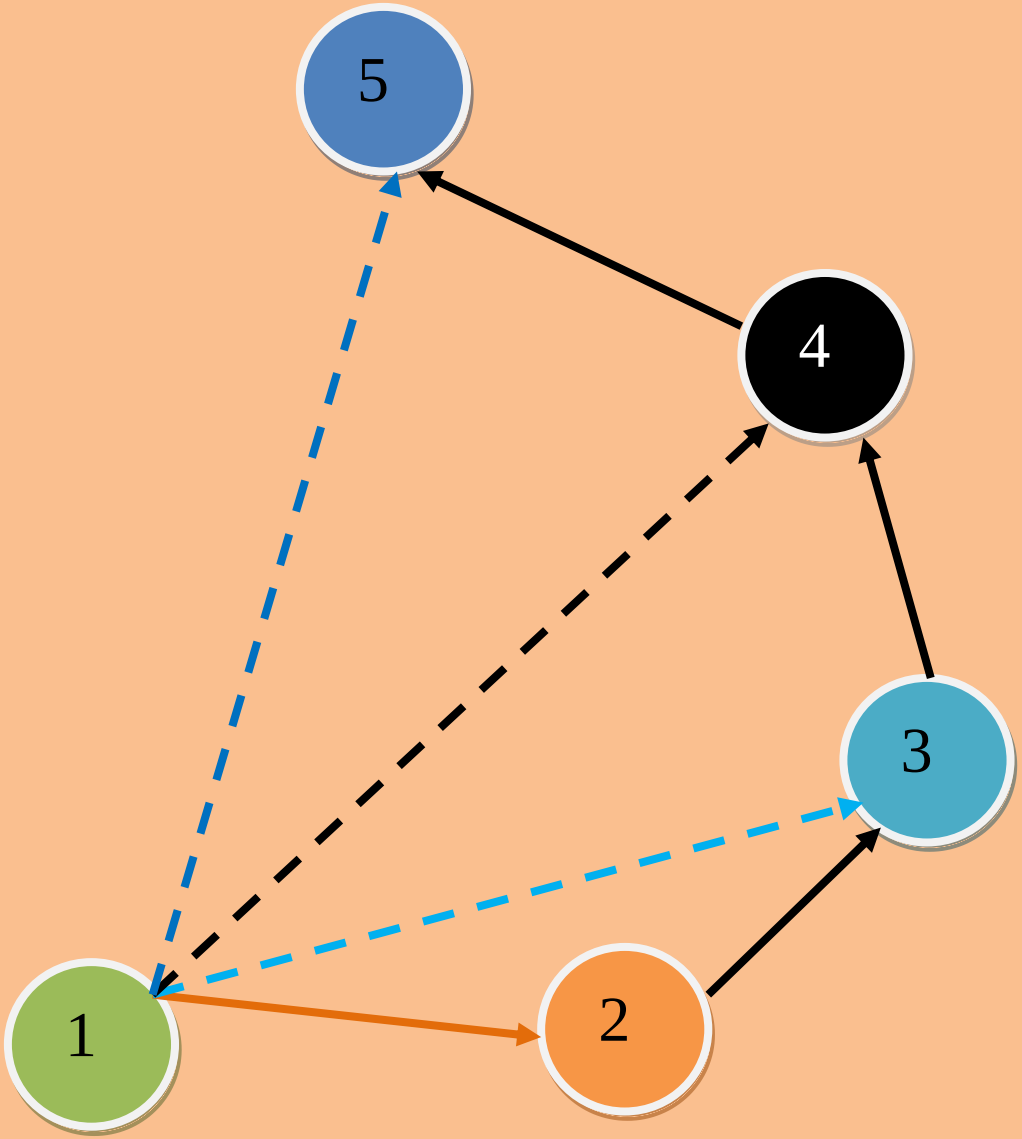
Illustration du principe de RoyWarshall:



```
RoyWarshall (M: GrapheMat; n: integer)
    var
        i: Integer;
    begin
    for i := 1 to n do
        Phi(M, i, n);
    end;
```


5
4
3
1
2


5
4
3
1
2


5
4
3
1
2

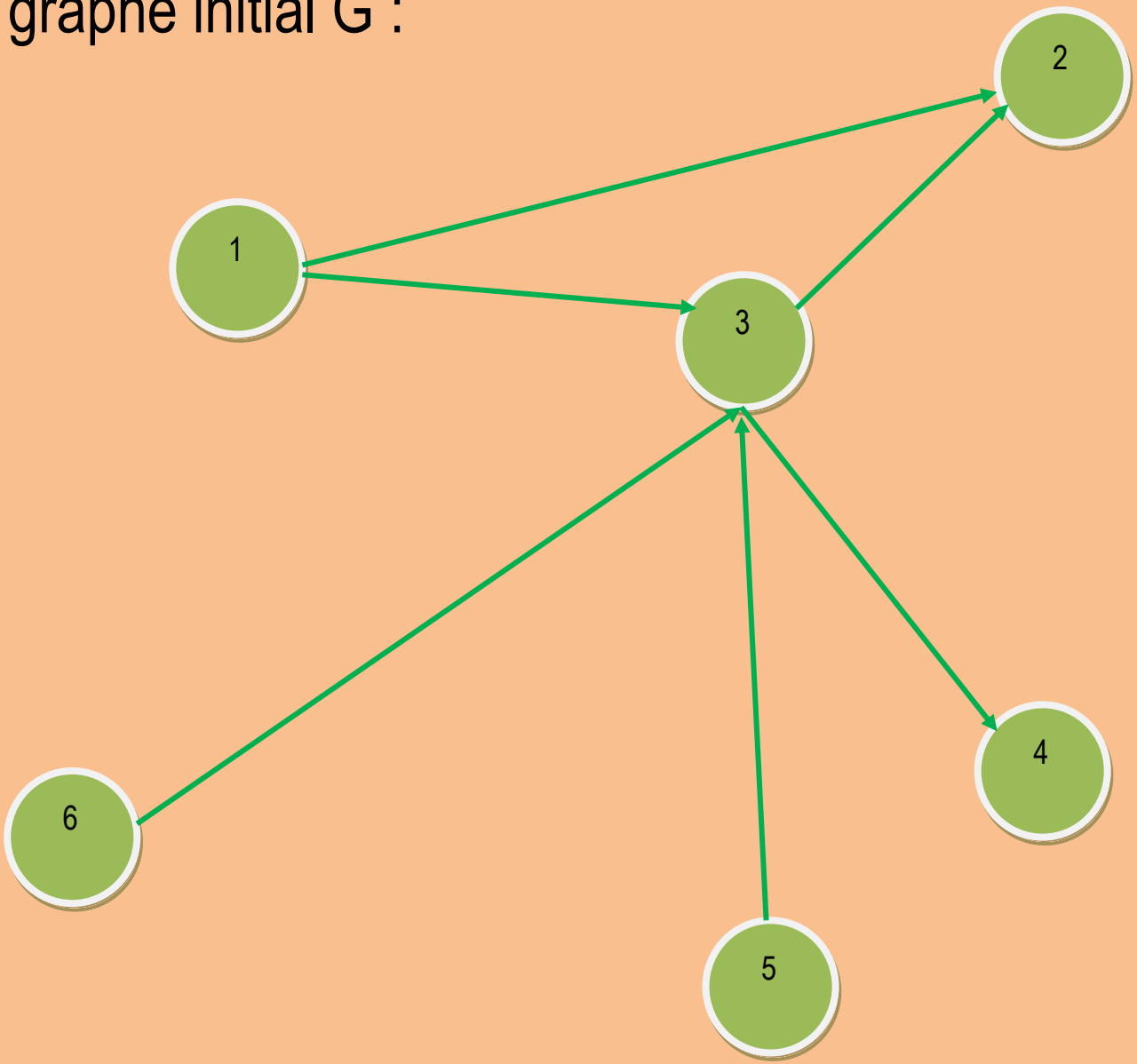
Remarque

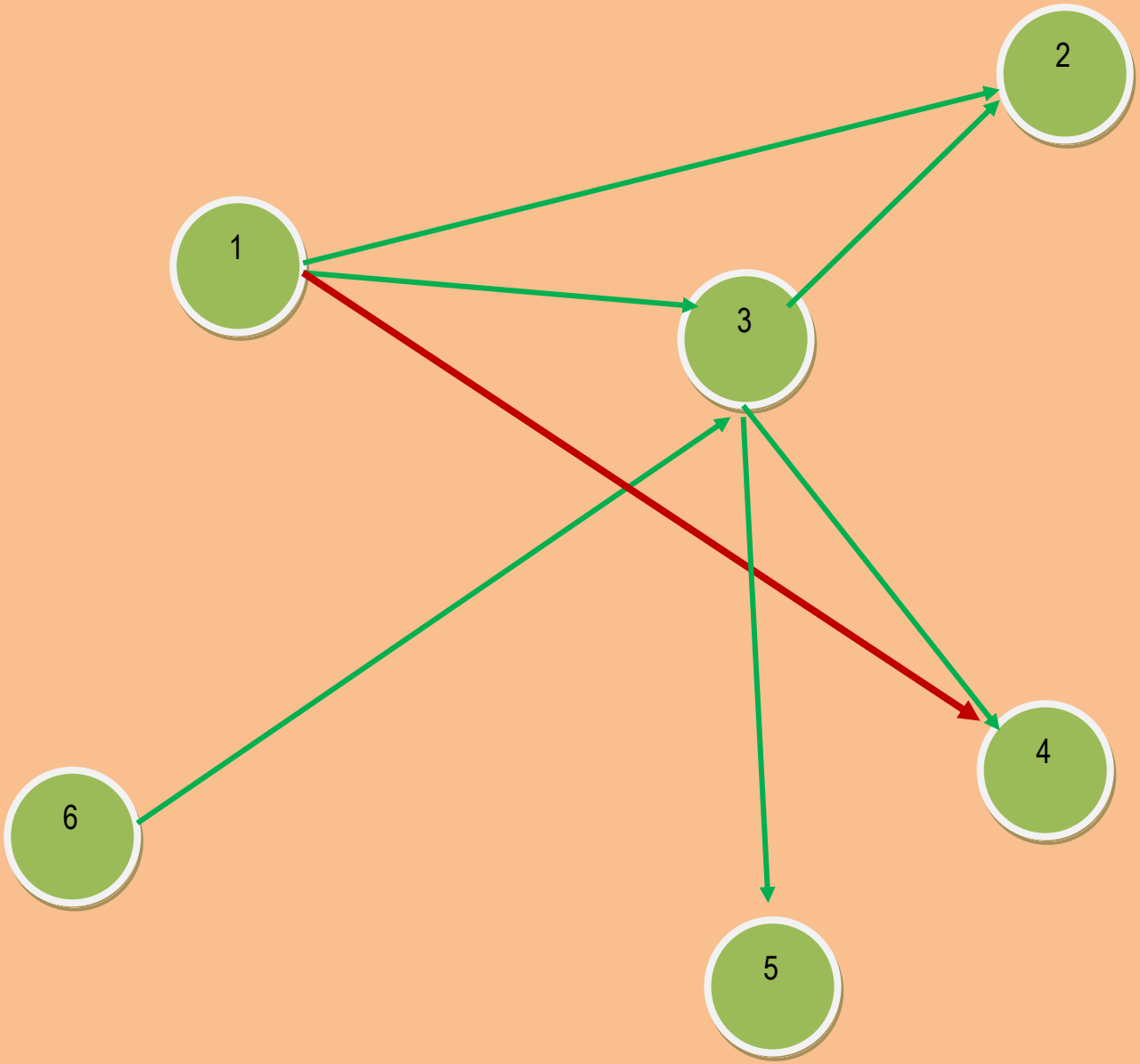
Chaque exécution de la procédure PHI pouvant nécessiter n^2 opérations.

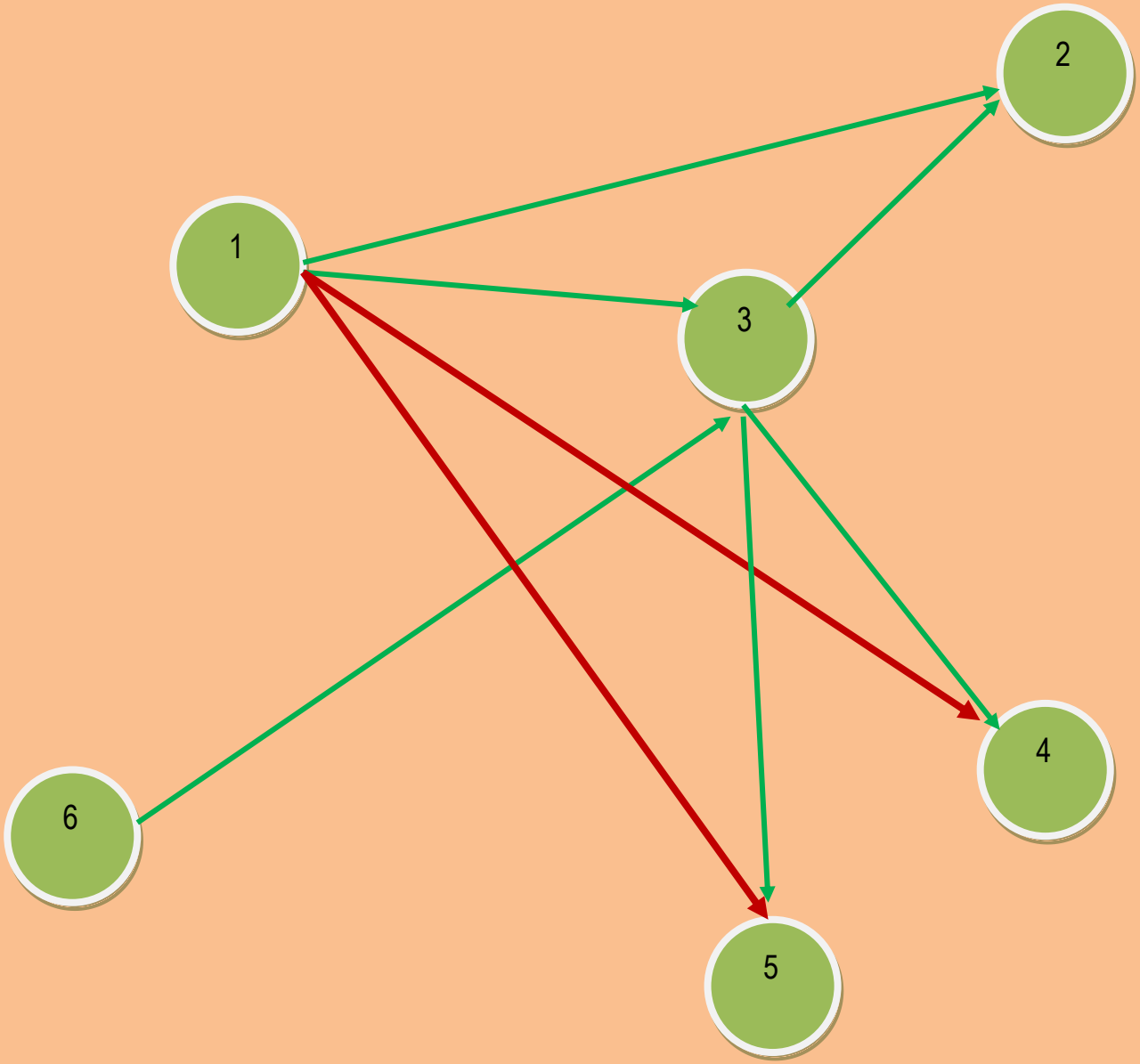
L'algorithme ci-dessus effectue un nombre d'opérations que l'on peut majorer par n^3.

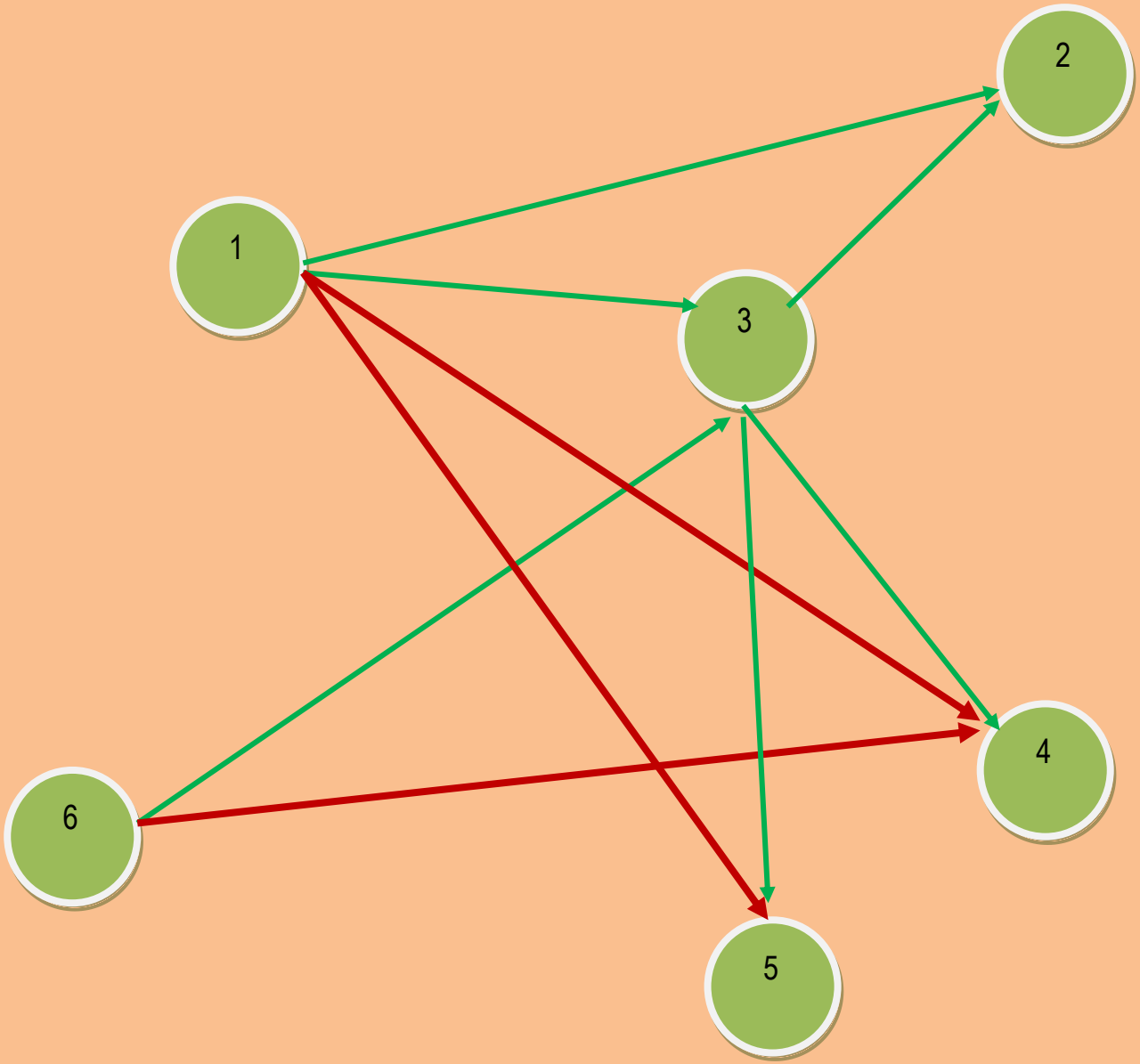
Cet algorithme est donc meilleur que le calcul des puissances successives de la matrice d'adjacence.

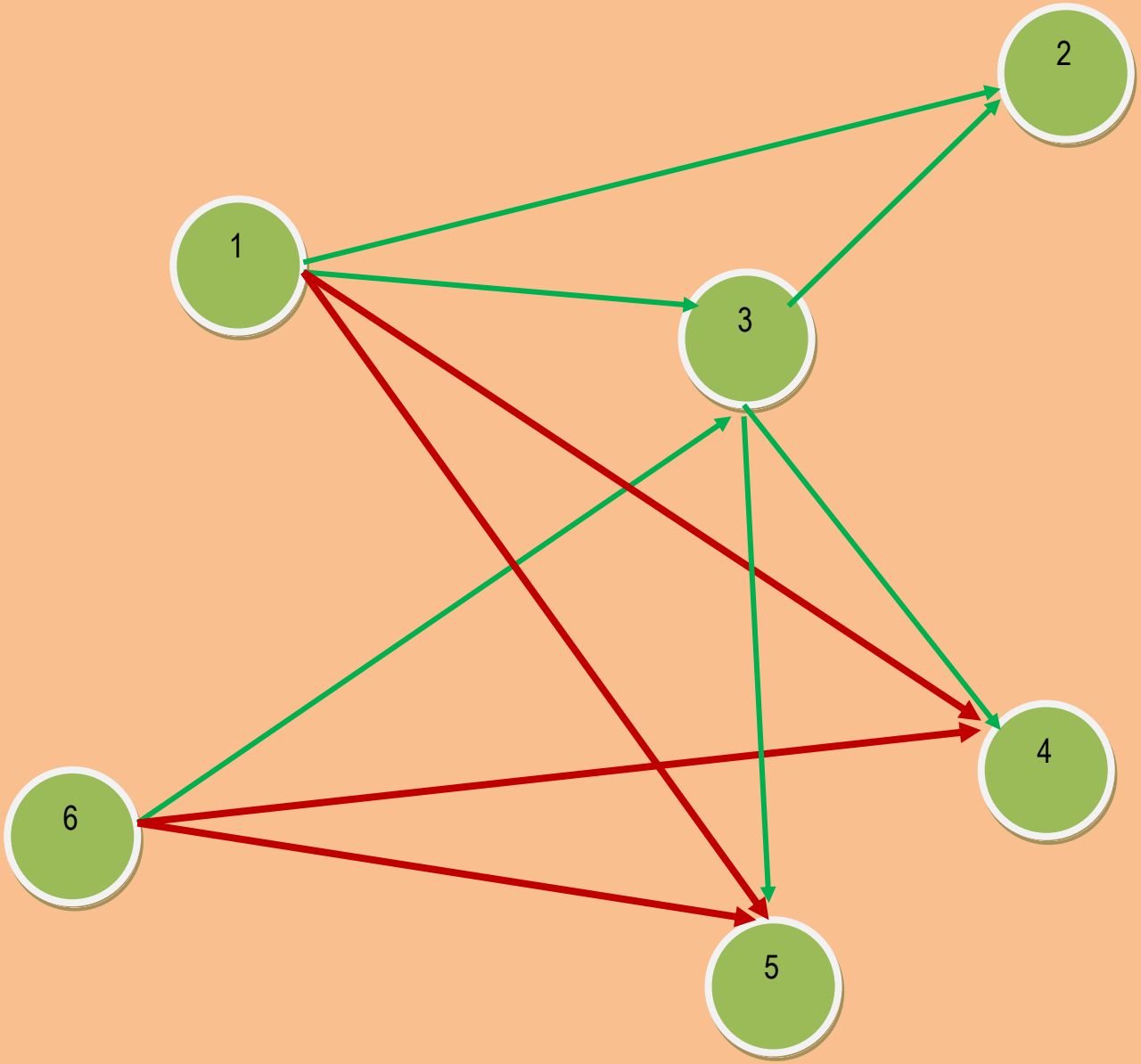
Soit le graphe initial G :




1
2
3
4
5
6


1
2
3
4
5
6


2
1
3
4
6
5


2
1
3
4
6
5

La fermeture transitive du graphe G est :