Graphes et Programmation

Ahcène Bounceur

Présentation

- Ahcène Bounceur
- Maître de Conférences
- Mail : Ahcene.Bounceur@univ-brest.fr
- Net : http://www.bounceur.com
- Téléphone : 62 17
- Adresse :
  - Département Informatique
  - Lc208 (2ème étage)

Partie 1

RAPPEL SUR LES GRAPHES

Définitions

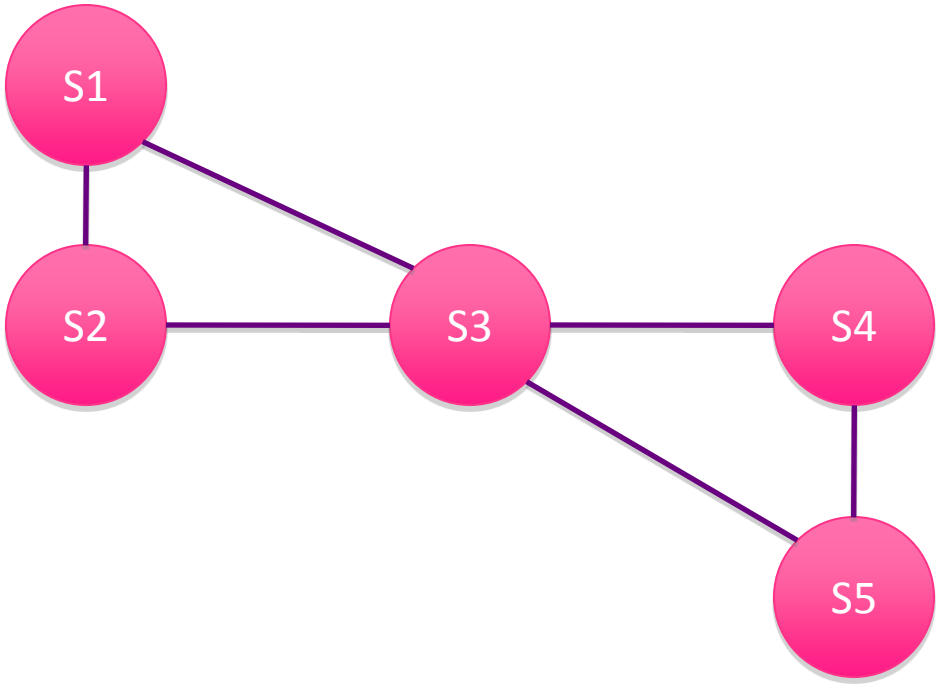
- Un graphe est une **structure de données** composée d'un ensemble de **sommets**, et d'un ensemble de **relations** entre ces sommets.

Définitions

- Un graphe est dit **non orienté** ou **symétrique** si les relations sont non orientées.
- Un graphe est dit **orienté** si les relations sont orientées.
- Une **relation** est aussi appelée un **arc** (pour les graphes orientées) et **arête** (pour les graphes non orientés).
- Les **sommets** sont aussi appelés **nœuds** ou **points**.

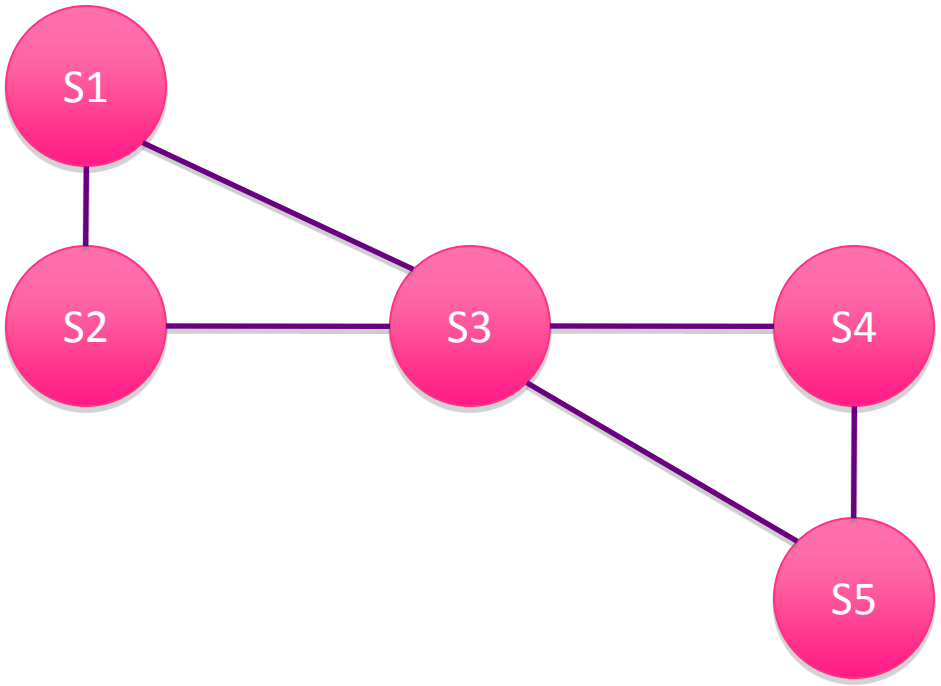
Graphe non orienté (symétrique)

- Ce graphe peut se noter comme suit
  - Les sommets : S = {S1, S2, S3, S4, S5}
  - Les arêtes : A = {S1S2, S1S3, S2S3, S3S4, S3S5, S4S5}



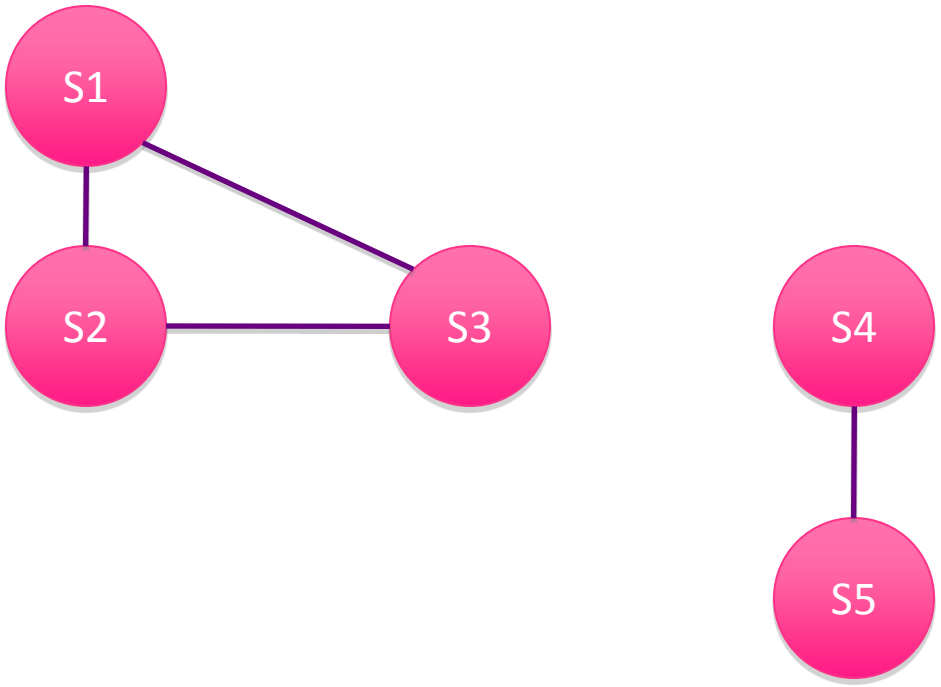
Graphe connexe

- un graphe non orienté est dit connexe si on peut aller de tout sommet vers tous les autres sommets. Ce graphe est connexe :



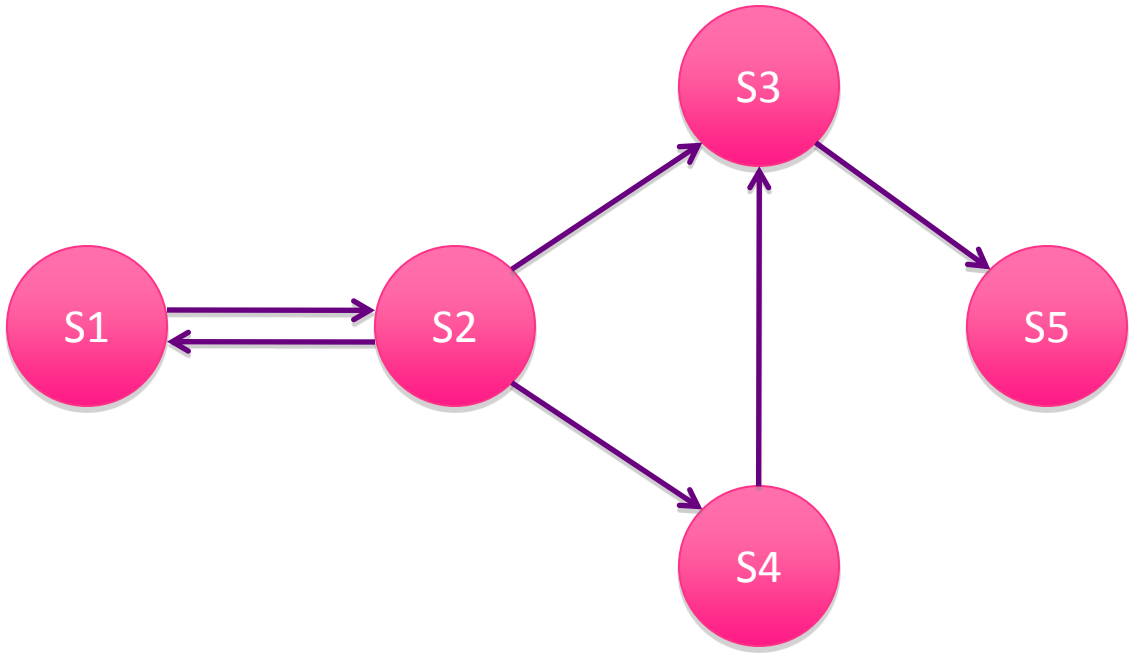
Graphe connexe

- Ce graphe n’est pas connexe :



Graphe orienté

- Si les relations sont orientées, le graphe est dit **orienté.**

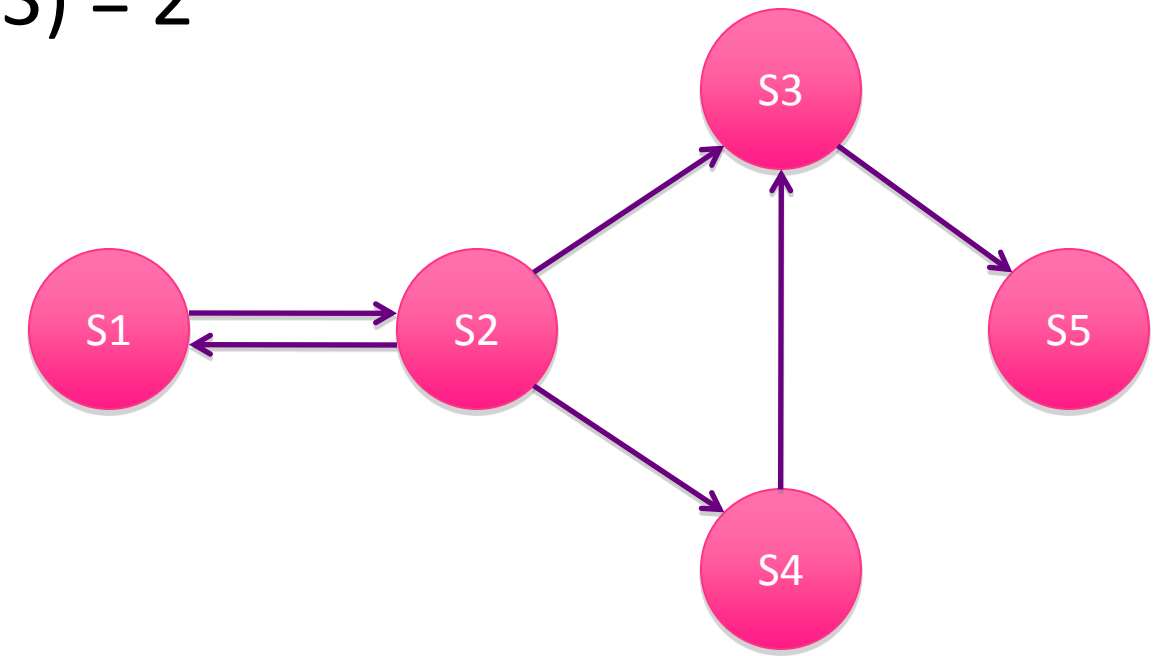


Degré d'un graphe orienté

- Le **degré** d'un sommet S est égal au nombre d'arcs entrants ou sortants de S
  - Représentation : d(S)
- Le **demi-degré extérieur** (ou degré d'émission) de S est égal au nombre d'arcs sortants de S
  - Représentation : d+(S)
- Le **demi-degré intérieur** (ou degré de réception) de S est égal au nombre d'arcs entrants à S
  - Représentation : d-(S)

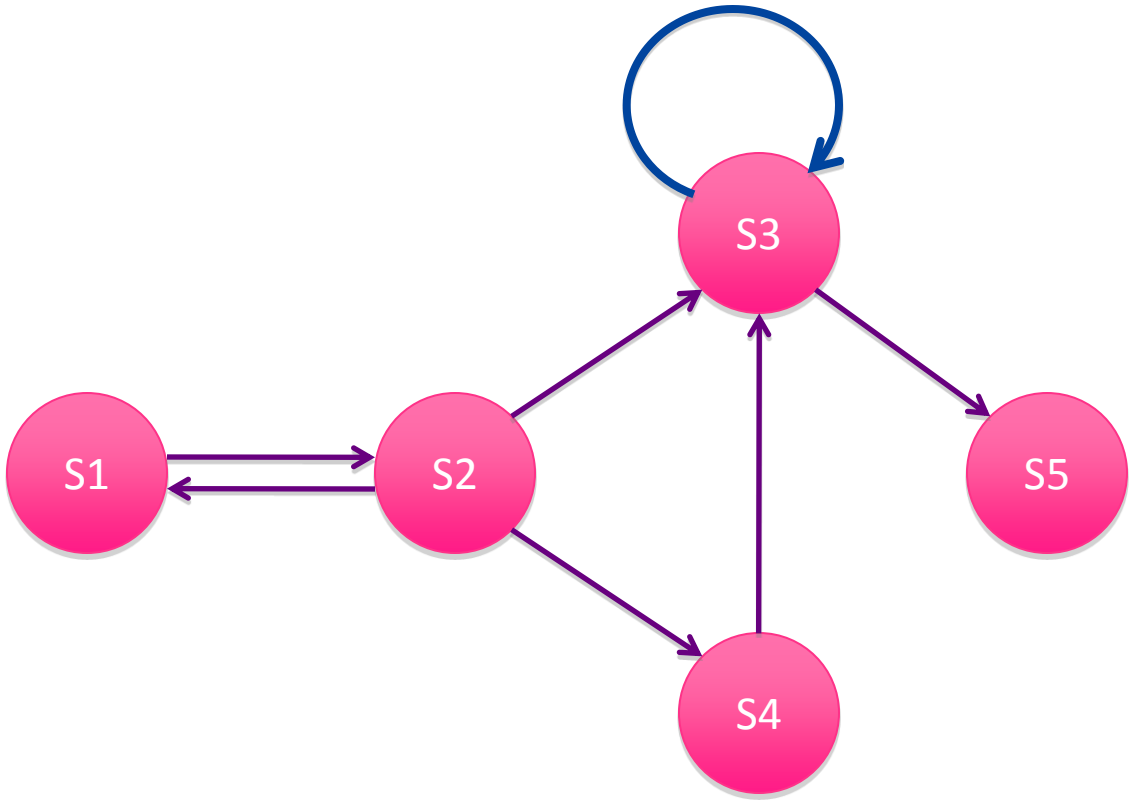
Degré d’un graphe orienté

- d(S3) = 5
- d+(S3) = 1
- d-(S3) = 2



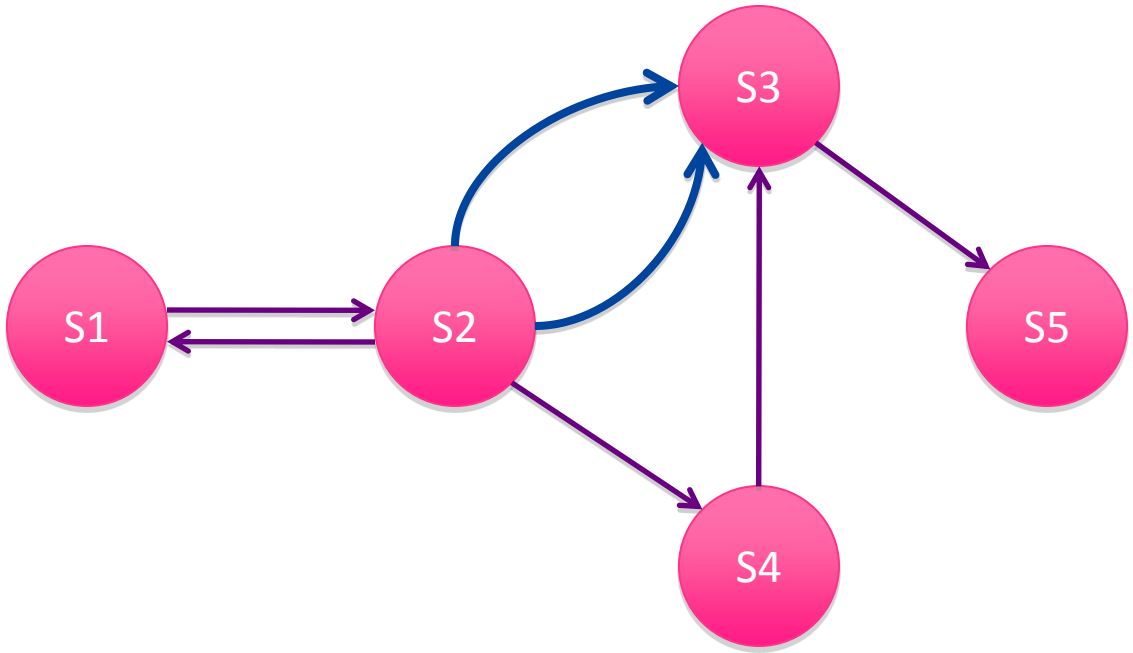
Boucle

- S3S3 est une **boucle** :



Graphe multiple (p-graphe)

- Ce graphe est un graphe **multiple** (ou 2-graphe) car il existe 2 arcs allants de S2 à S3 :

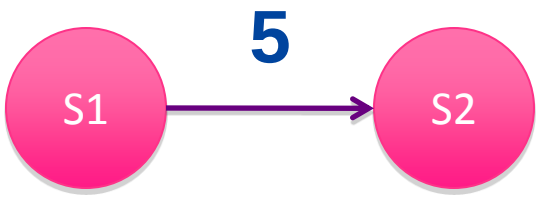


Graphe simple

- Un graphe **simple** est un graphe sans boucle et sans arcs multiples

Graphe valué

- Un graphe est dit **valué** (**pondéré**) si à chaque arc on associe une valeur représentant le coût de la transition de cet arc.



Définitions complémentaires

- Un **chemin** dans un graphe est une suite d'arcs consécutifs.
- La **longueur d'un chemin** est le nombre d'arcs constituant ce chemin.
- Un **chemin simple** est un chemin où aucun arc n'est utilisé 2 fois.
- Un **circuit simple** est un chemin simple tel que le premier et le dernier sommet sont les mêmes.

Définitions complémentaires

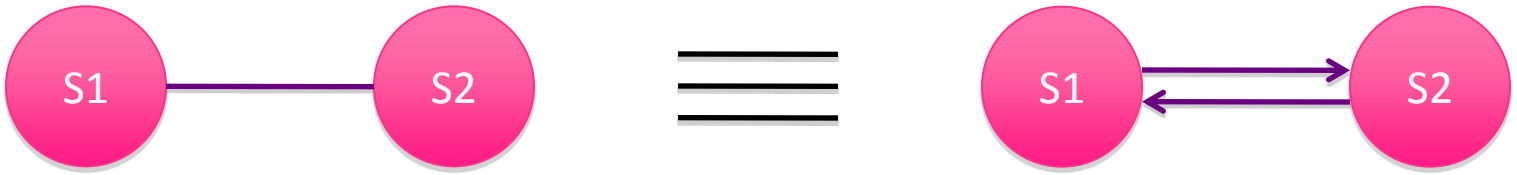
- Un **circuit eulérien** est un circuit qui passe une et une seule fois par tous les arcs.
- Un **circuit hamiltonien** est un circuit qui passe une et une seule fois par tous les sommets.
- Un graphe est **planaire** si aucun de ses arcs ne se coupent.

Remarques

- Dans les graphes non orientés, on parle quelquefois de **chaîne** au lieu de chemin et de **chaîne simple** pour un chemin simple. Le terme **arête** est aussi utilisé à la place d'arc. Le terme de **cycle** indique un circuit simple orienté.

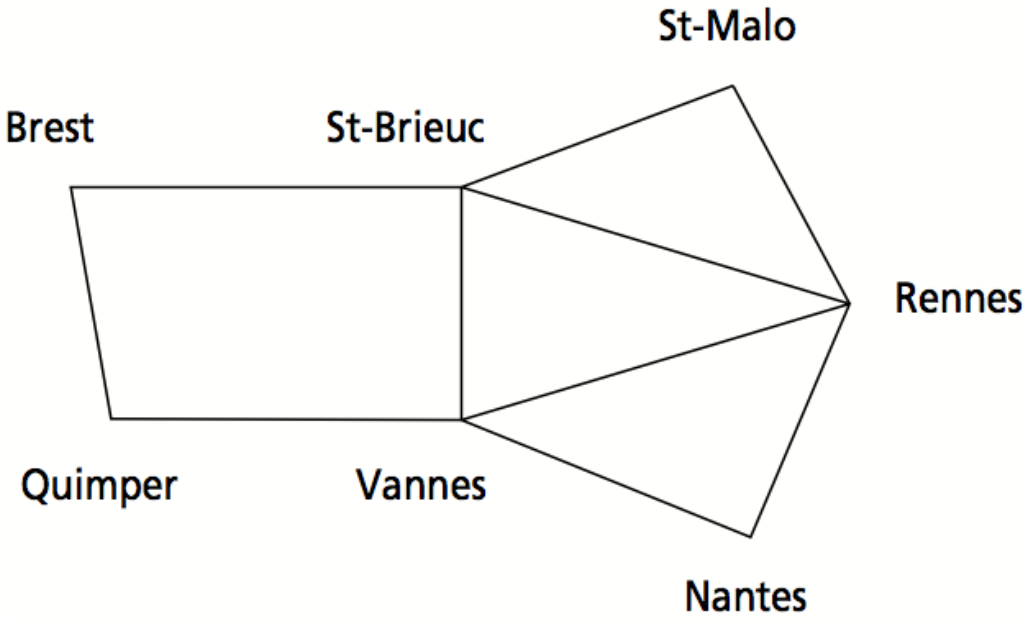
Remarques

- Un graphe non orienté peut toujours être considéré comme un graphe orienté où les relations symétriques sont explicitement mentionnées.



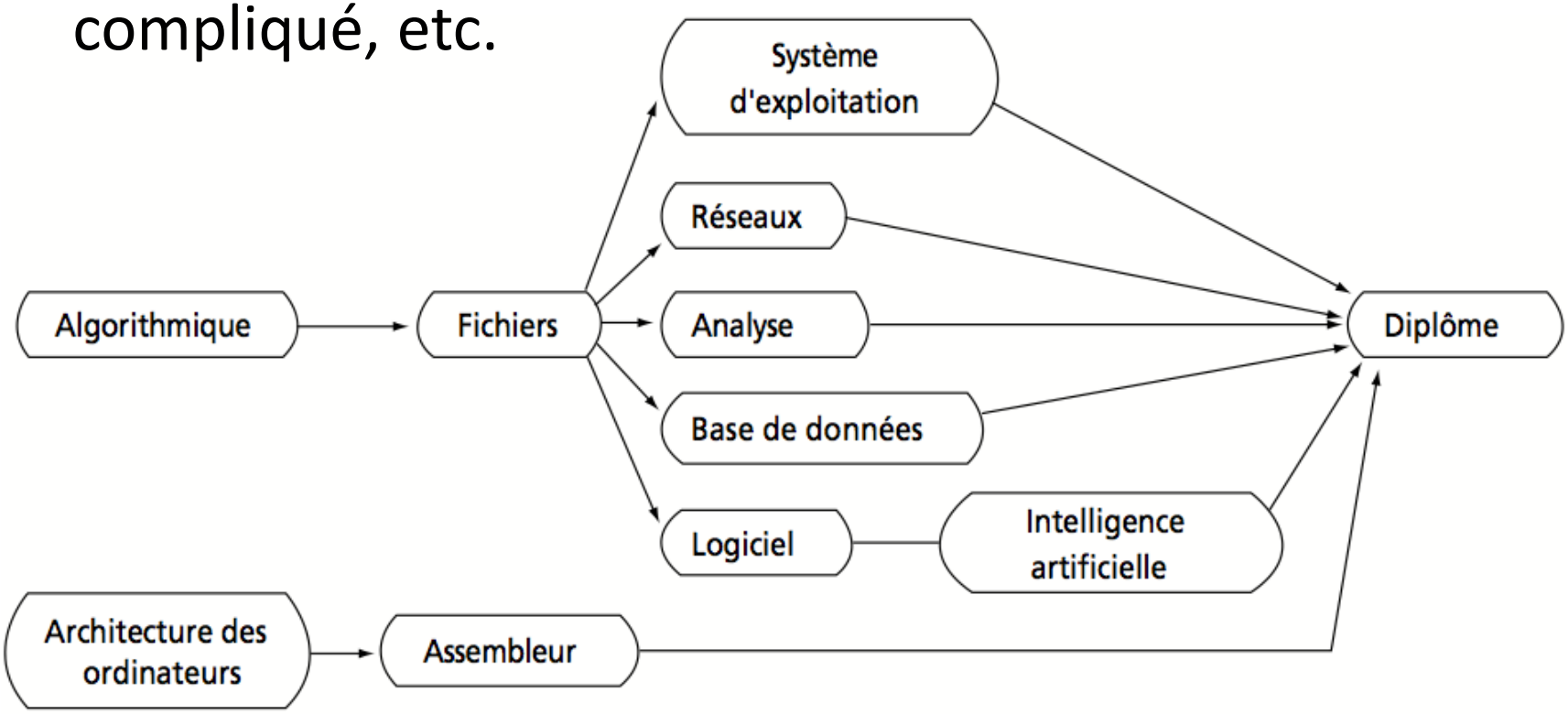
Exemples de graphes (1/4)

- Un réseau de communication (routier, aérien, électrique, d'alimentation en eau, etc.) entre différents lieux peut être schématisé sous forme d'un graphe (non orienté):



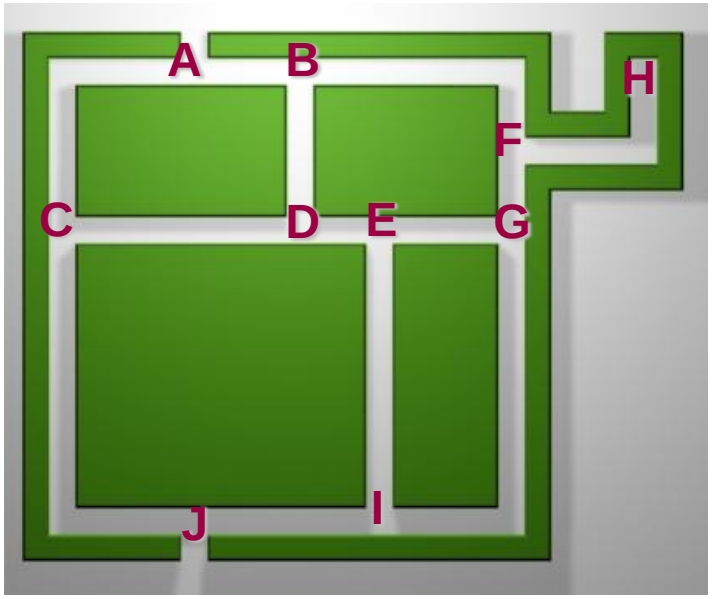
Exemples de graphes (2/4)

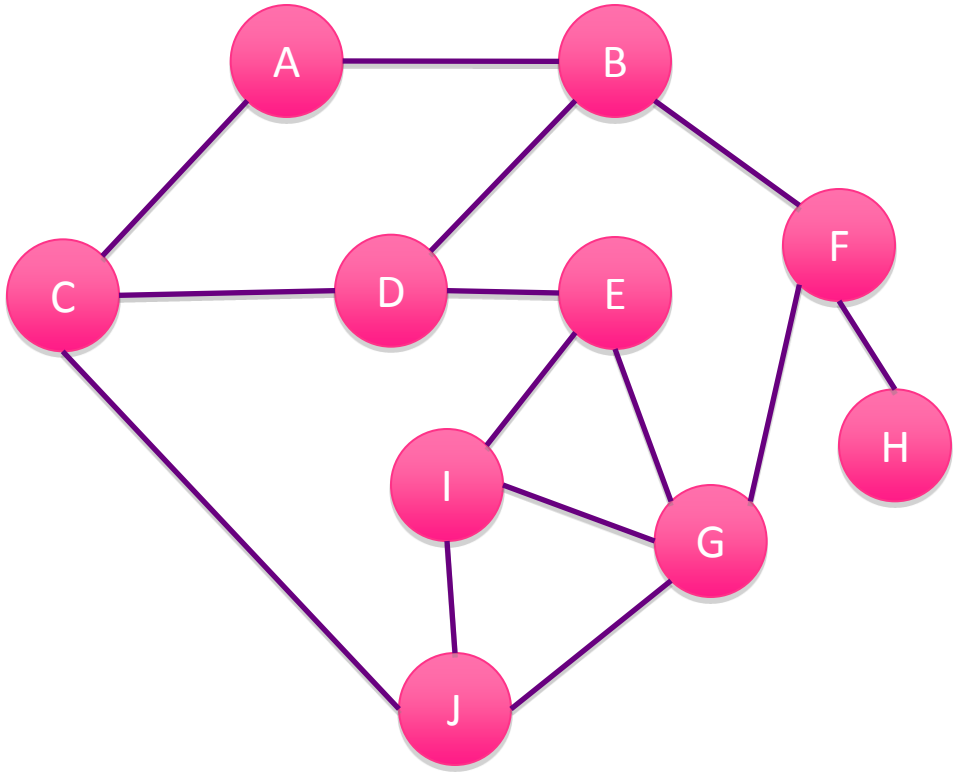
- ordonnancement de tâches (graphe orienté sans cycle) : la construction d'un complexe immobilier, ou plus simplement d'une maison, d'un appareil compliqué, etc.



Exemples de graphes (3/4)

- Un labyrinthe : l'entrée se fait en A, la sortie en H. Chaque carrefour présentant un choix de chemins est un sommet.

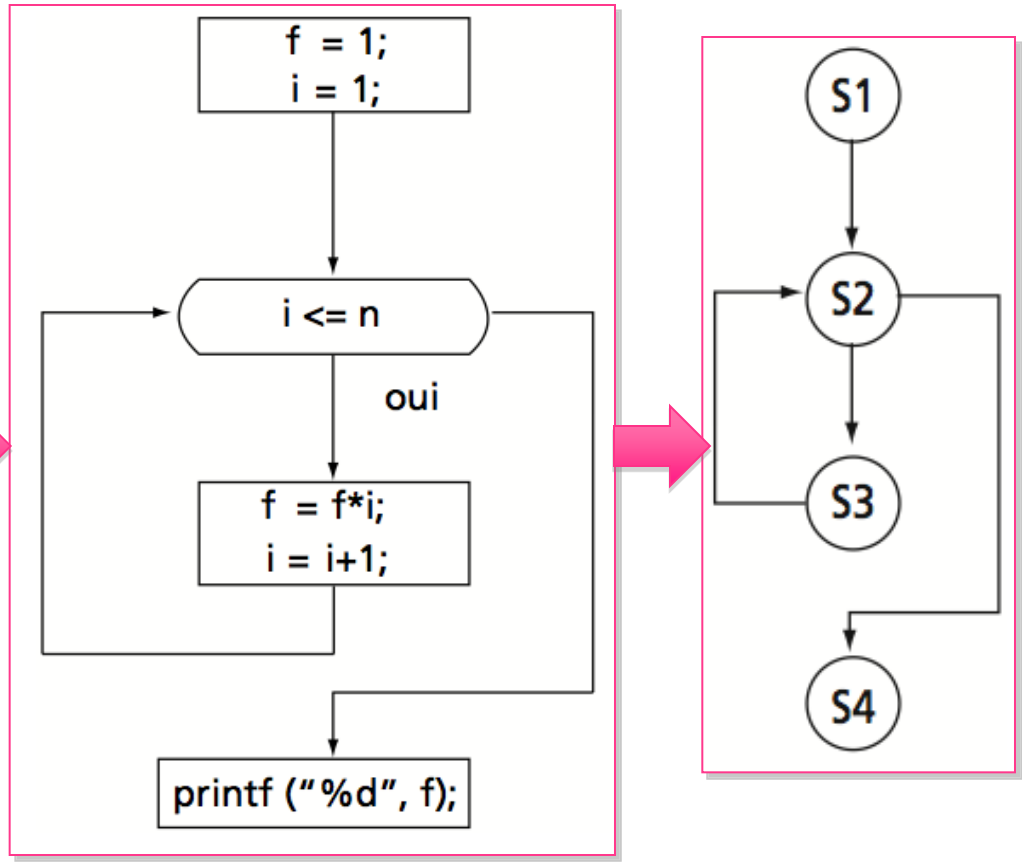




Exemples de graphes (4/4)

- Un programme peut être considéré comme un graphe orienté. Les sommets représentent les actions ; les arcs représentent l'enchaînement des actions.

```
int f, i;
f = 1;
for (i=1; i<=n; i++) {
        f = f * i;
}
printf ("%d", f);
```



Partie 2

MÉMORISATION DES GRAPHES

Introduction

- Suivant le rapport entre le nombre de sommets et le nombre d'arcs, on choisit soit une mémorisation sous forme de matrice (rapport important), soit une mémorisation sous forme de listes d'adjacence. Dans ce dernier cas, la matrice serait dite creuse avec beaucoup d'éléments mémorisés inutilement.

Matrice d’adjacence

- Chaque arc (i, j) est représenté par un 1 dans la matrice. Une valeur 0 indique une absence de relation entre le sommet i et le sommet j.

0	1	1	0
1	0	1	1
1	1	0	0
0	1	0	0

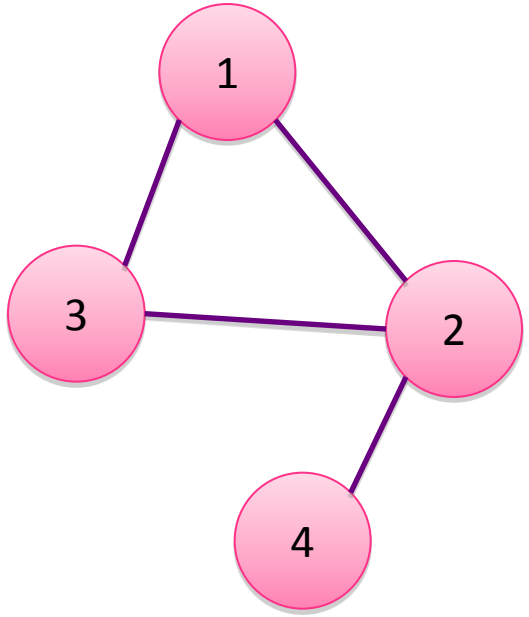
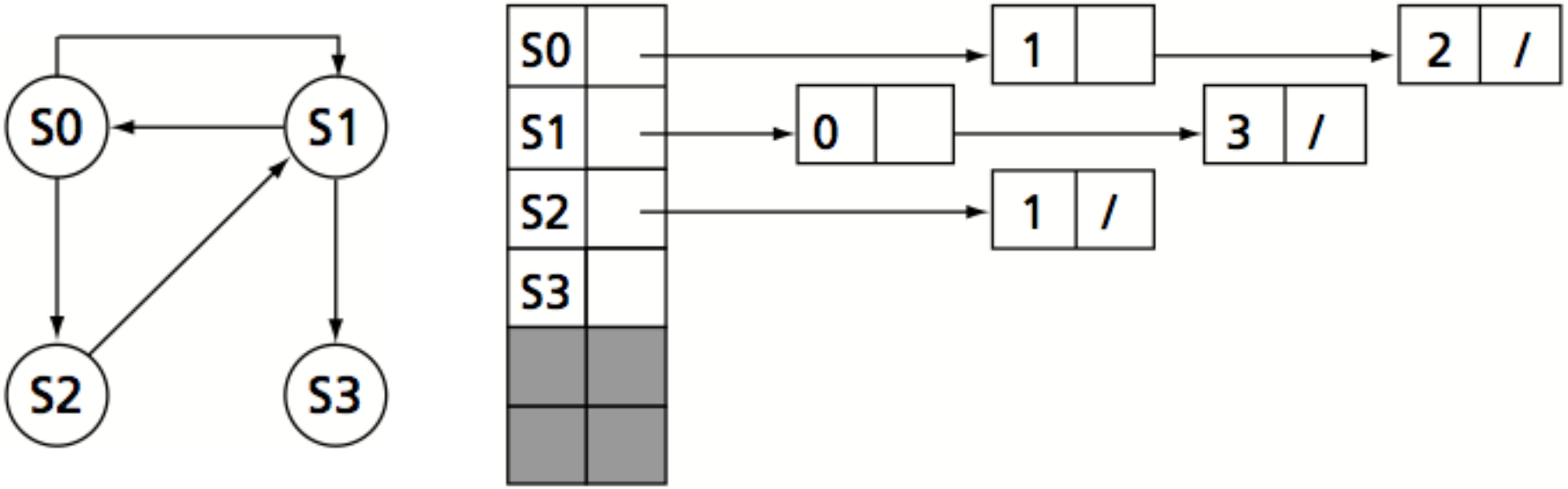


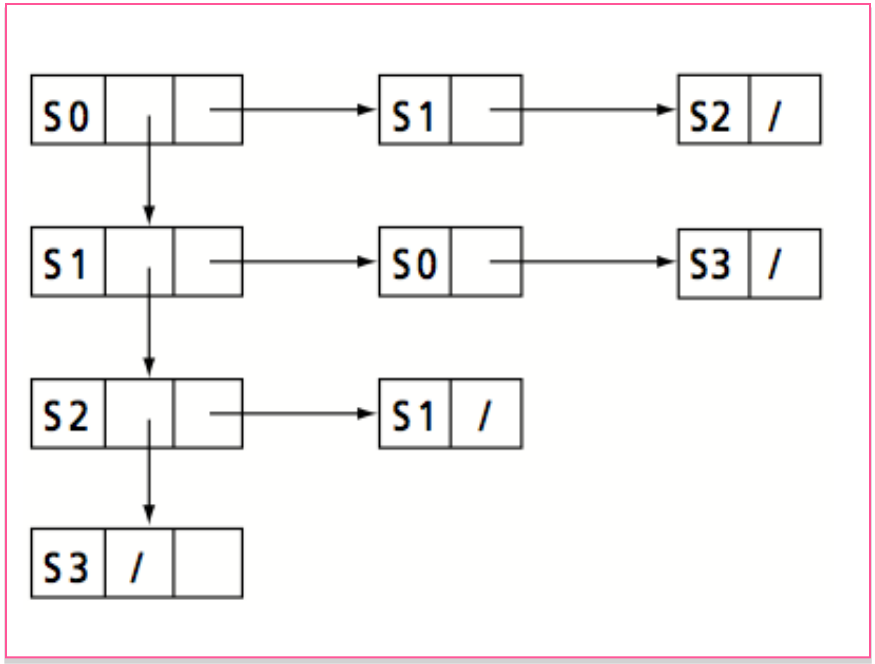
Table de listes d'adjacence

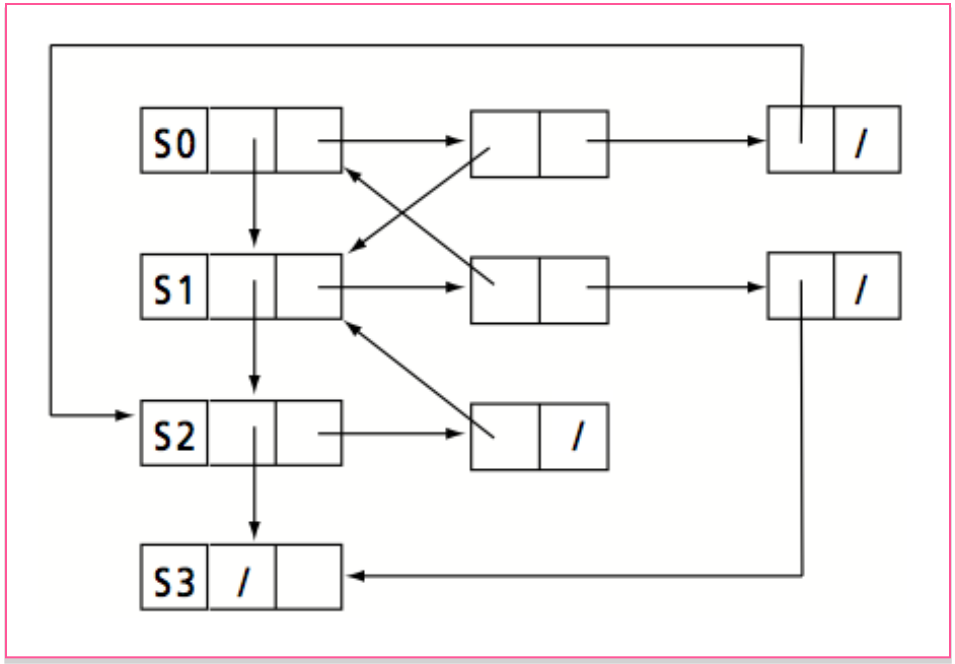
- La partie concernant les caractéristiques des sommets est mémorisée dans une table contenant, pour chaque entrée, une liste des sommets que l'on peut atteindre directement en partant du sommet correspondant à cette entrée.



Listes d'adjacence (dynamique)

- On peut tout allouer dynamiquement : la liste des sommets, et pour chaque sommet, la liste des sommets successeurs. L'allocation est entièrement dynamique.



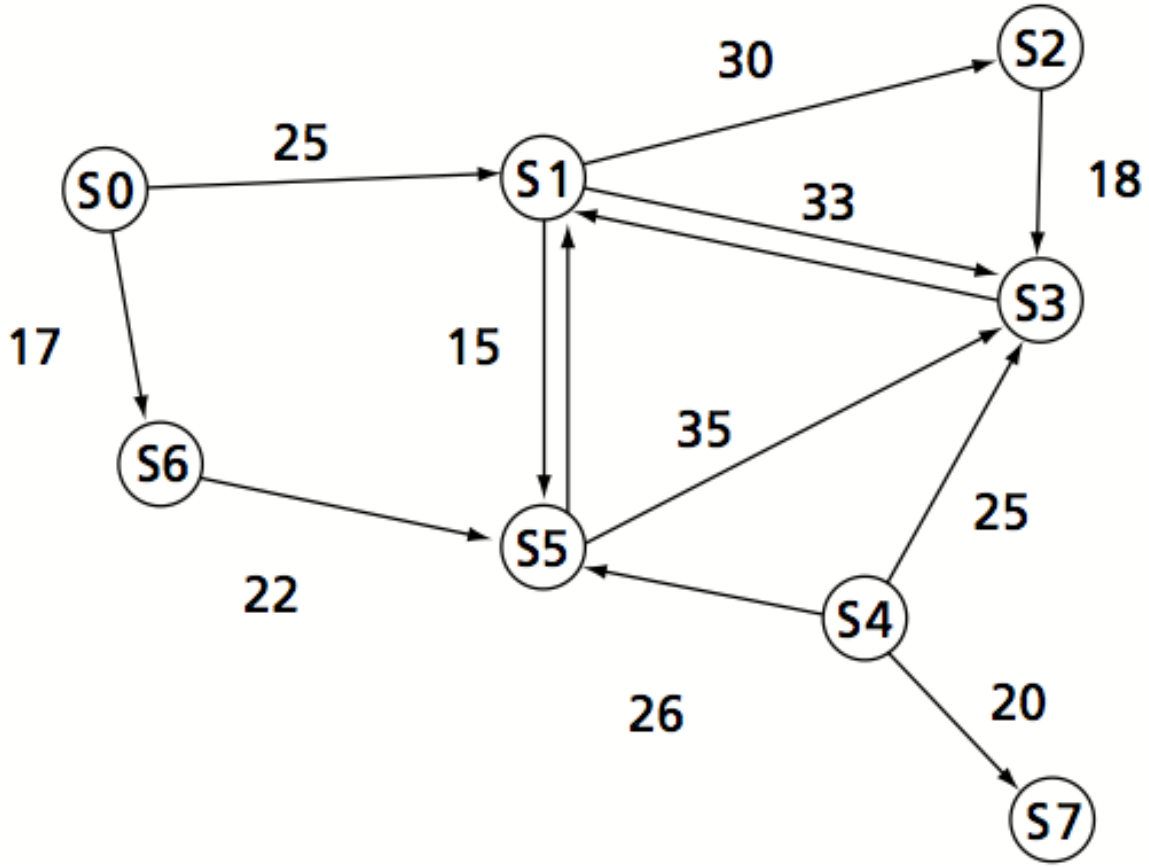


Parcours d’un graphe

- Il s’agit d’écrire un algorithme qui permet d’examiner les sommets une et une seule fois. La présence de circuits doit être prise en considération de façon à ne pas visiter plusieurs fois le même sommet. Il faut donc **marquer** les sommets déjà visités. On distingue deux types de parcours : **le parcours en profondeu**r et **le parcours en largeur**.

Parcours d'un graphe

- Soit le graphe suivant. C'est un graphe valué de distances entre lieux.



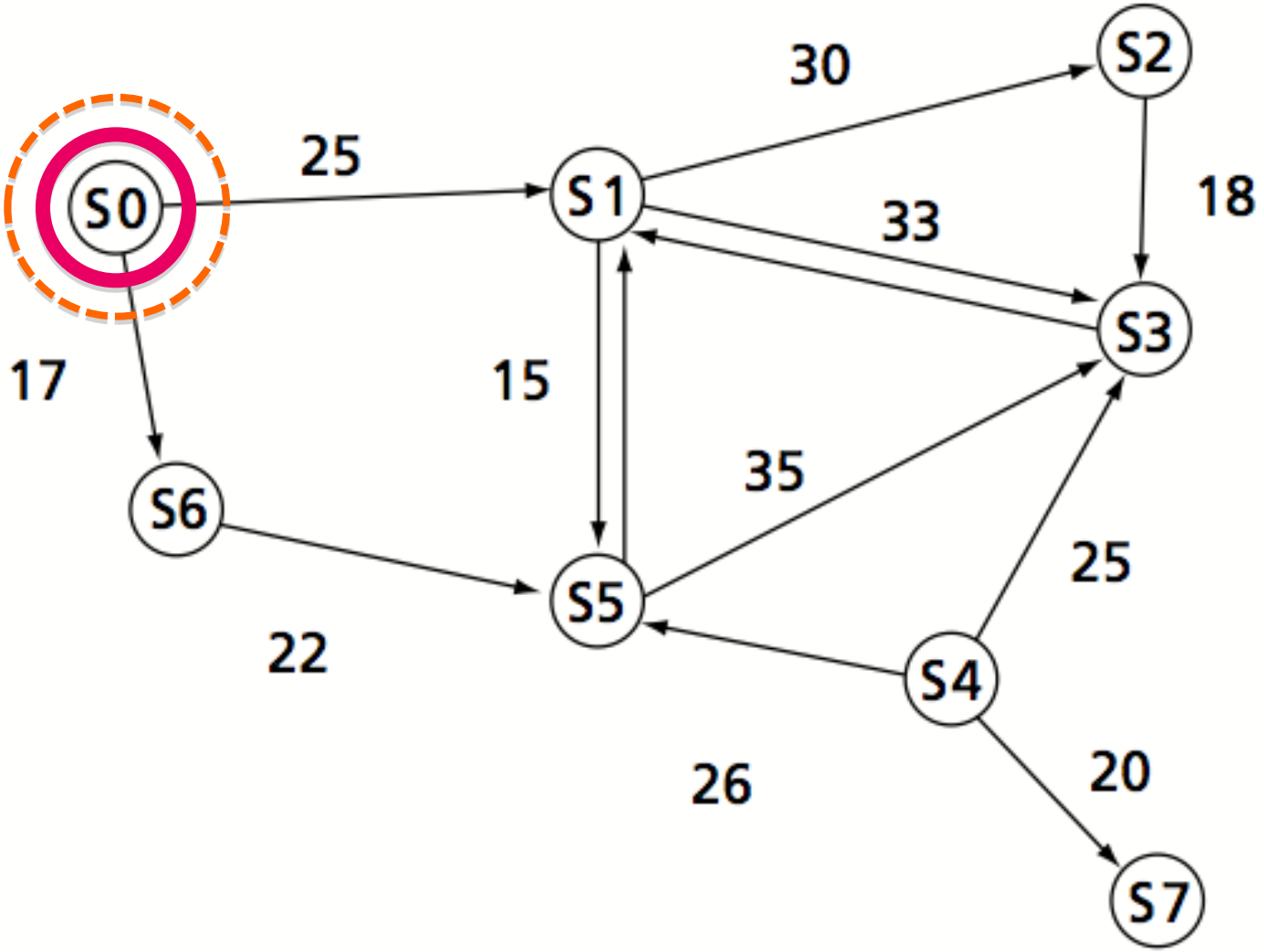
Parcours d'un graphe

- Ce graphe peut être décrit comme suite :
  - S0 S1 S2 S3 S4 S5 S6 S7 ; → Les sommets
  - S0: S1 (25) S6 (17) ;
  - S1: S2 (30) S3 (33) S5 (15) ;
  - S2: S3 (18) ;
  - S3: S1 (33) ;
  - S4: S3 (25) S5 (26) S7 (20) ;
  - S5: S1 (15) S3 (35) ;
  - S6: S5 (22) ;
  - S7:

Principe du parcours en profondeur

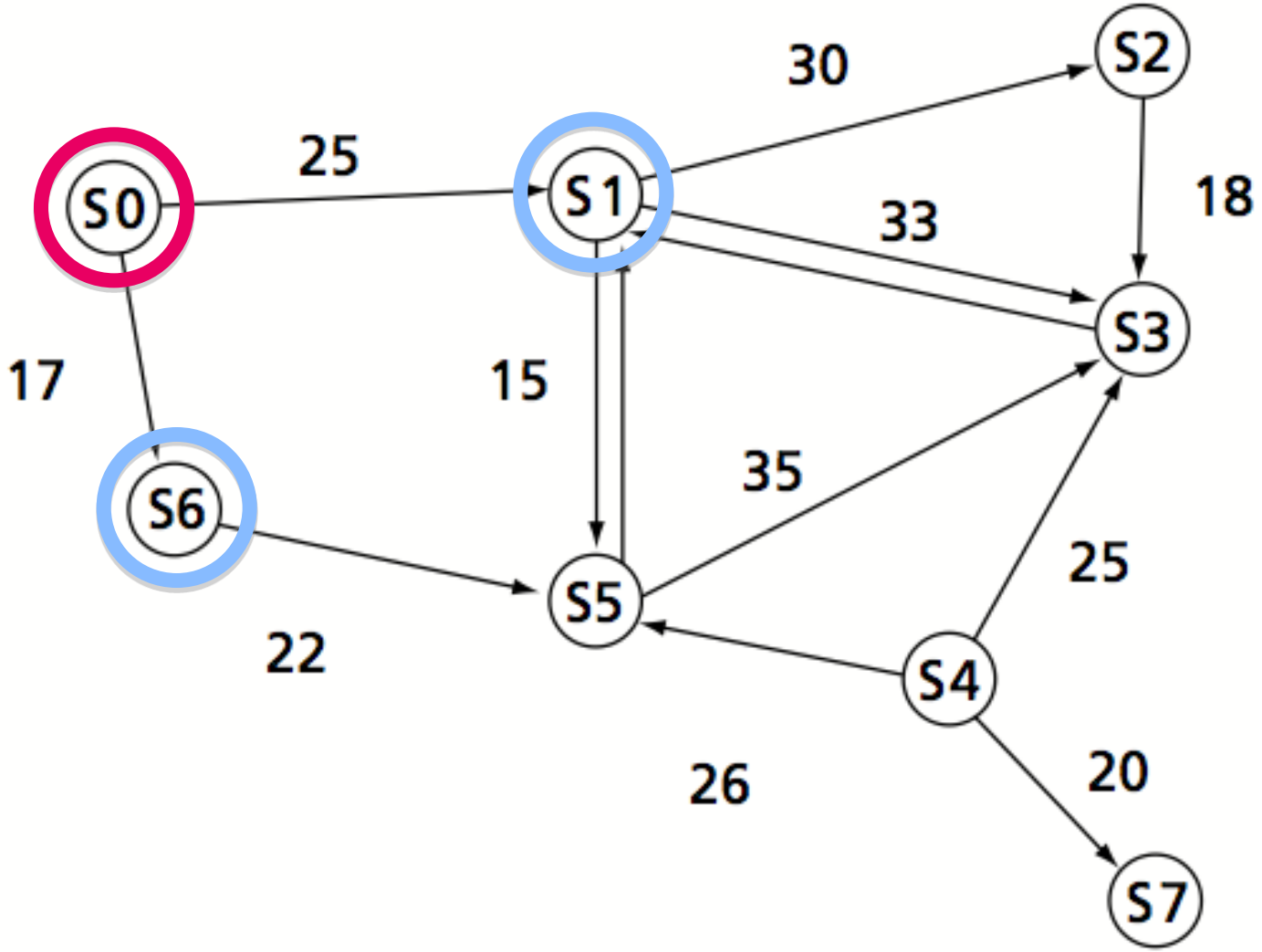
- On part d'un sommet donné. On énumère le premier fils de ce sommet (par ordre alphabétique par exemple), puis on repart de ce dernier sommet pour atteindre le premier petit-fils, etc. Il s'agit pour chaque sommet visité, de choisir un des sommets successeurs du sommet en cours, jusqu'à arriver sur une impasse ou un sommet déjà visité. Dans ce cas, on revient en arrière pour repartir avec un des successeurs non visité du sommet courant.

Principe du parcours en profondeur



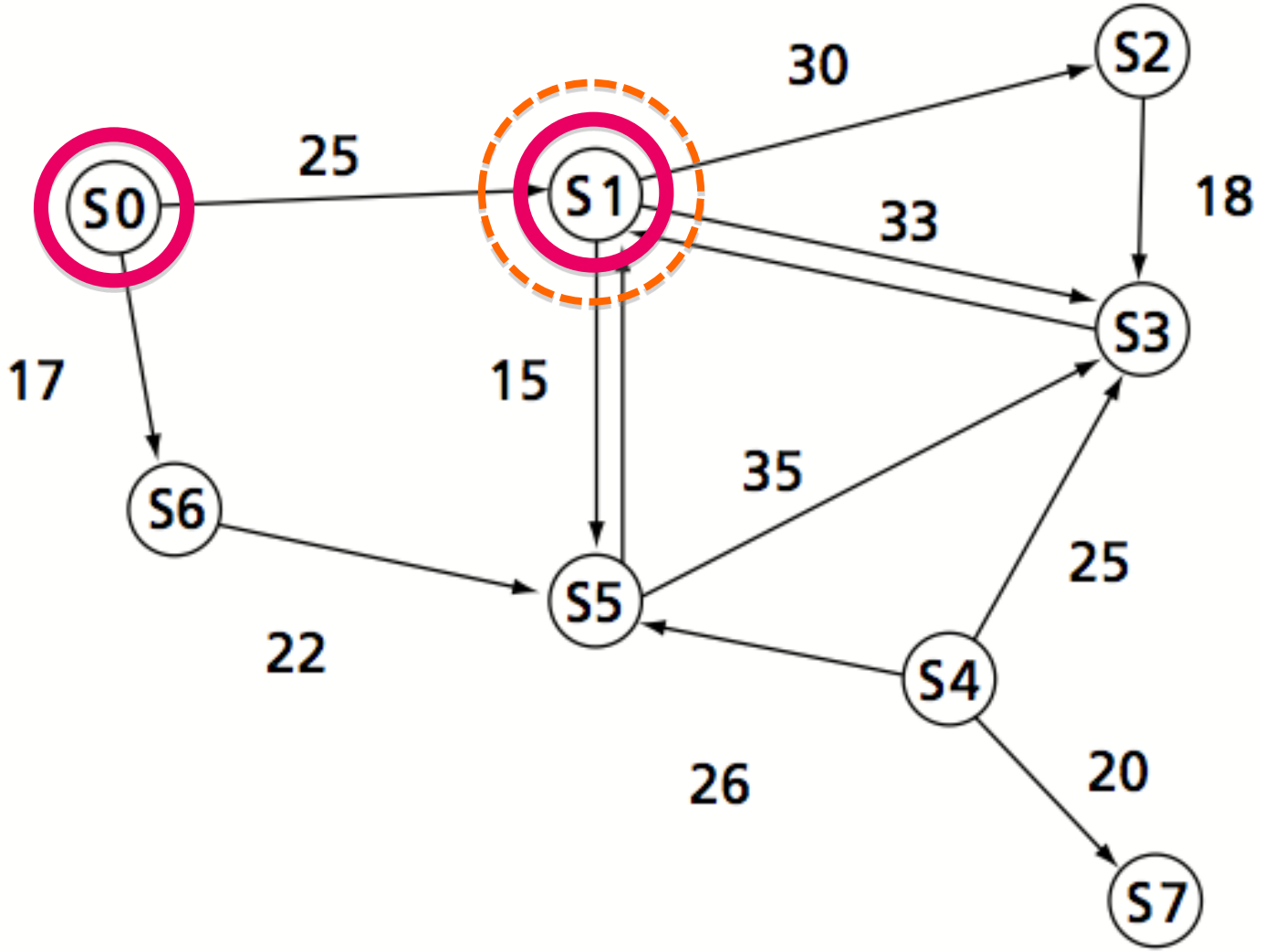
S0

Principe du parcours en profondeur



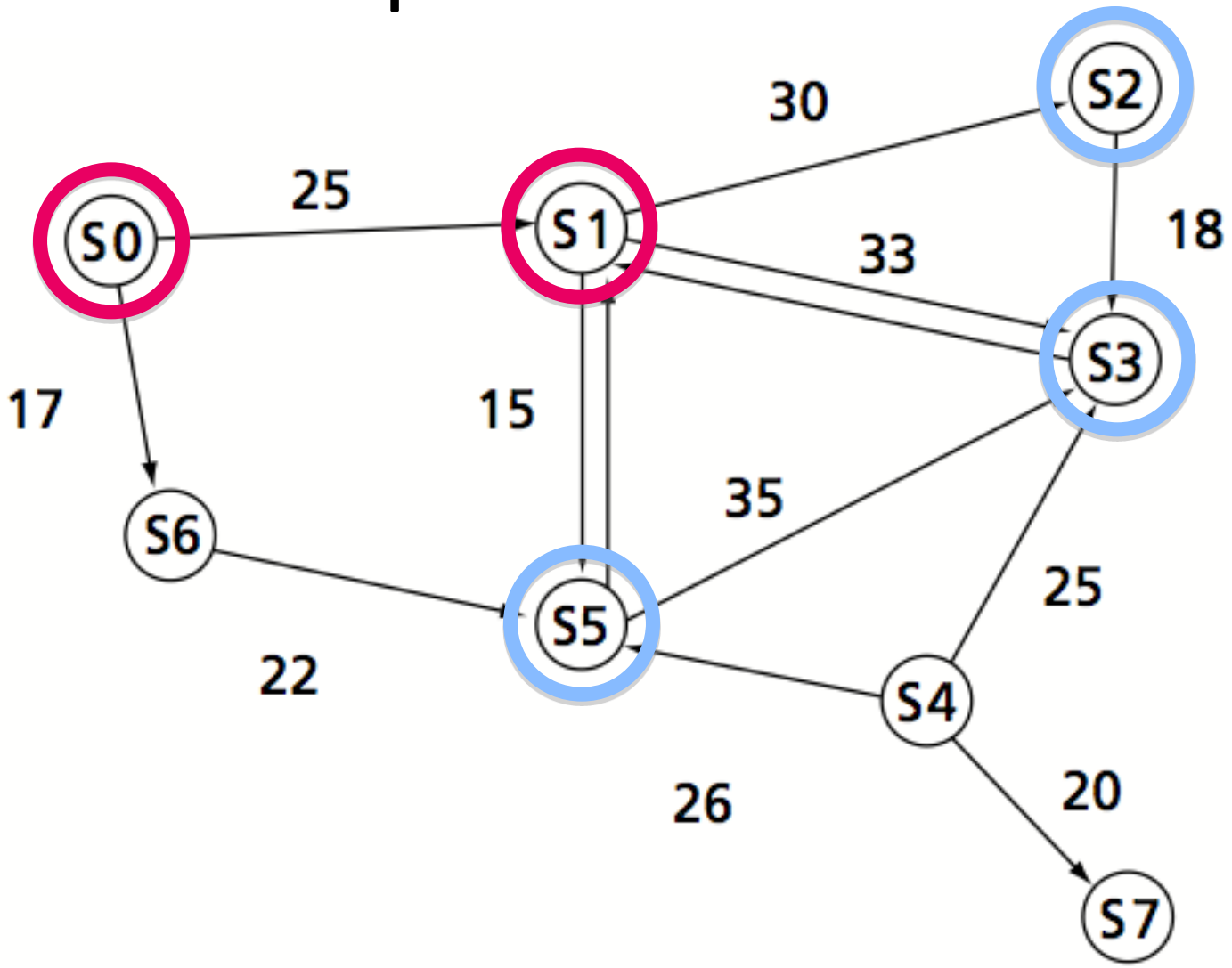
S0

Principe du parcours en profondeur



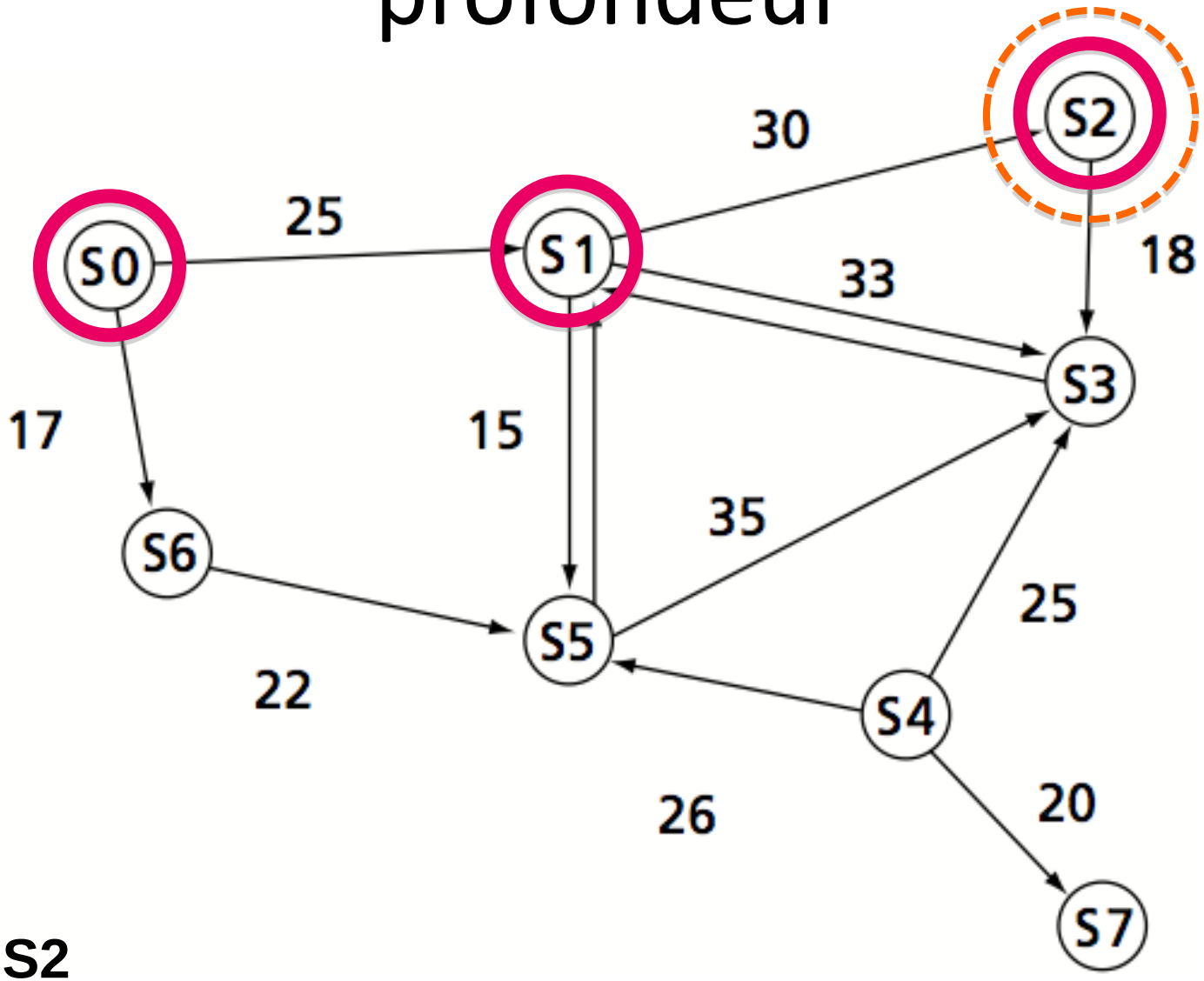
S0-S1

Principe du parcours en profondeur



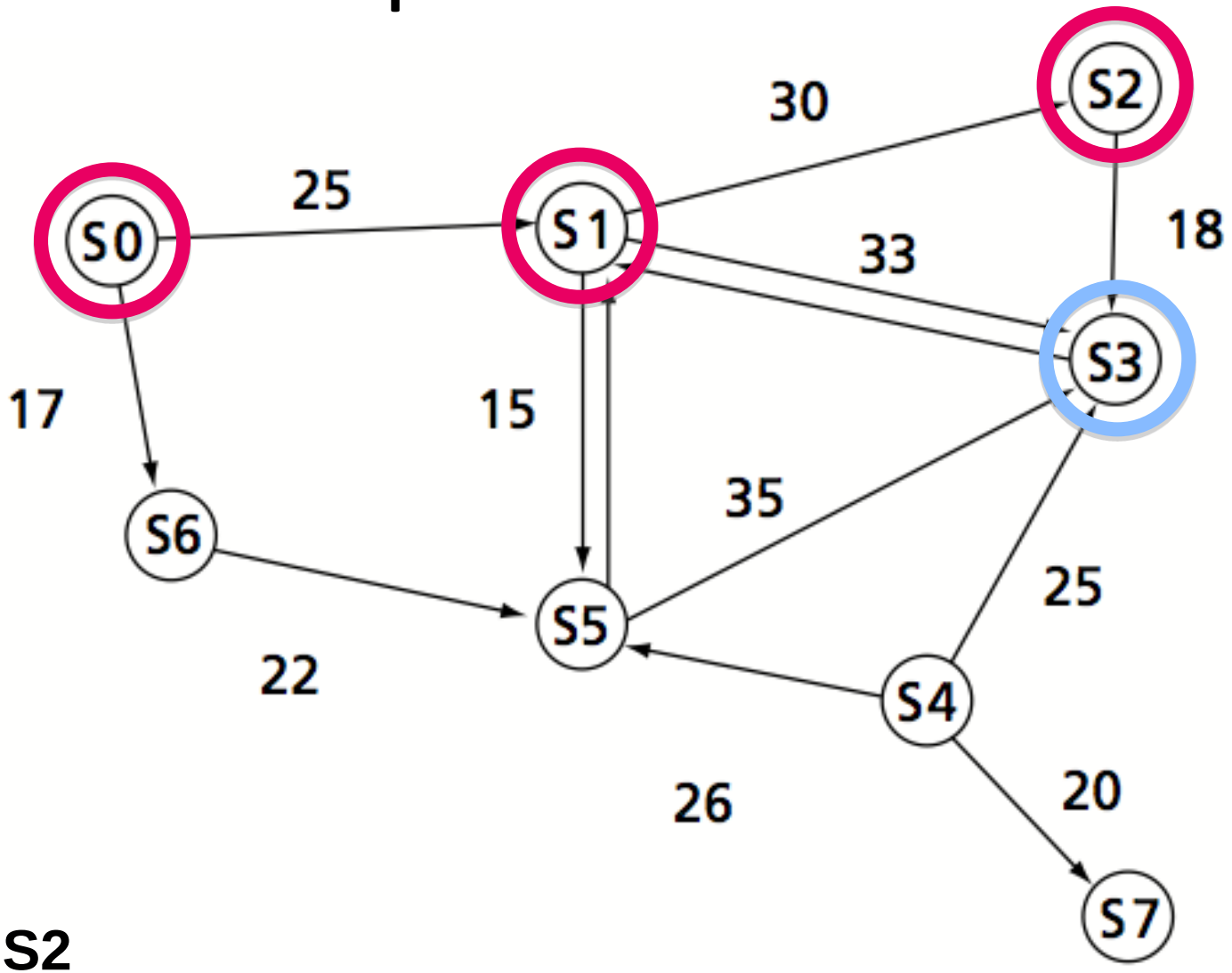
S0-S1

Principe du parcours en profondeur



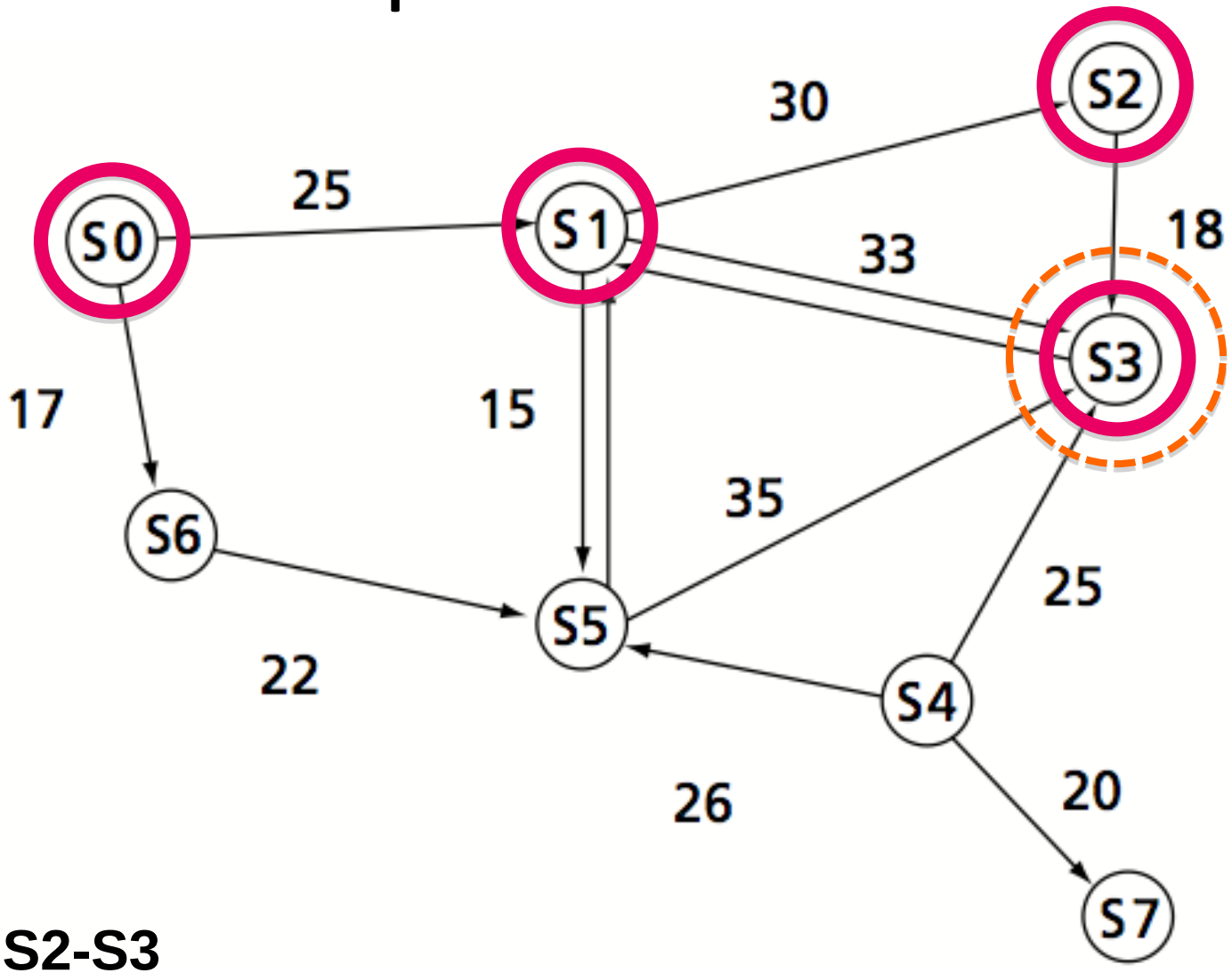
S0-S1-S2

Principe du parcours en profondeur



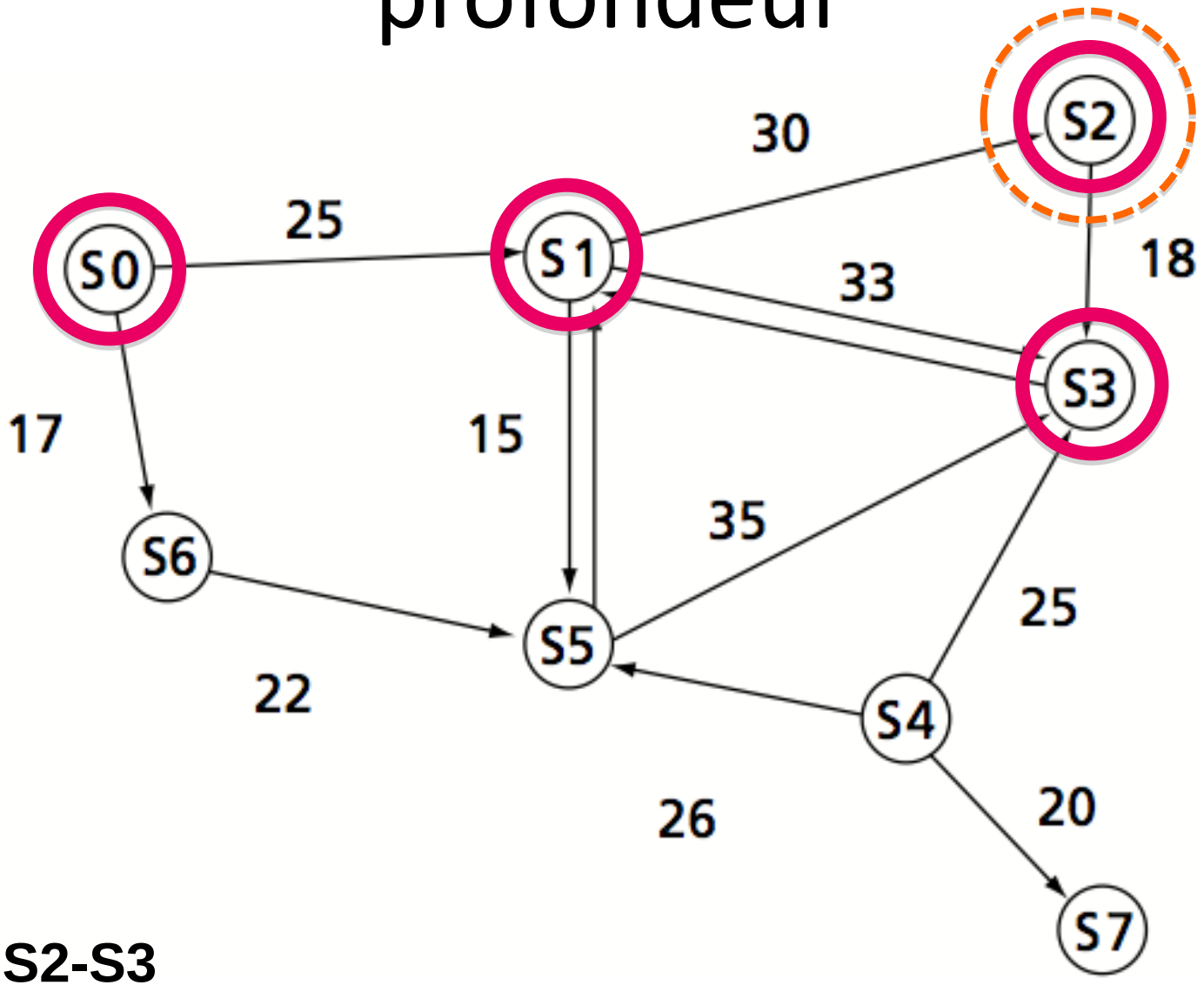
S0-S1-S2

Principe du parcours en profondeur



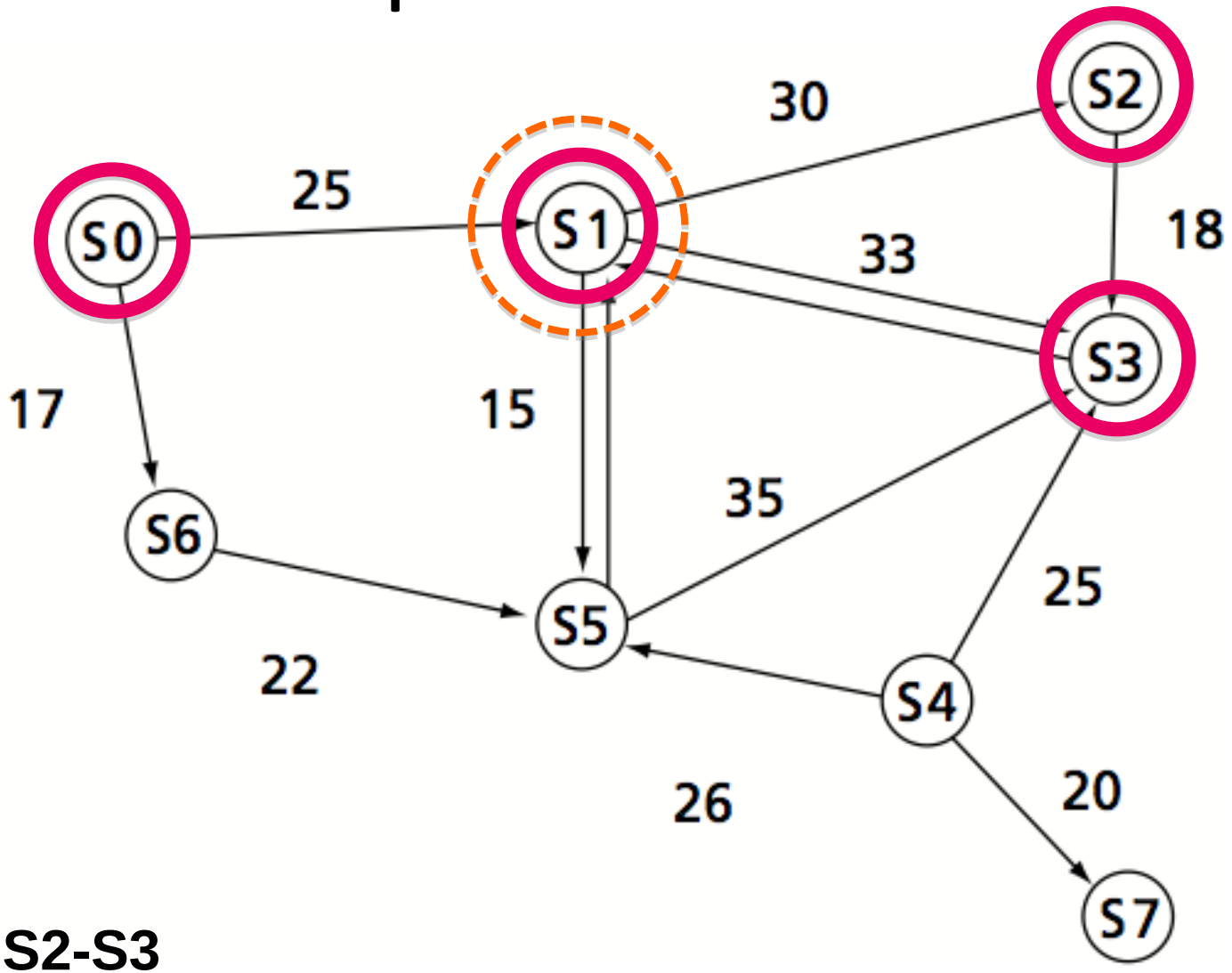
S0-S1-S2-S3

Principe du parcours en profondeur



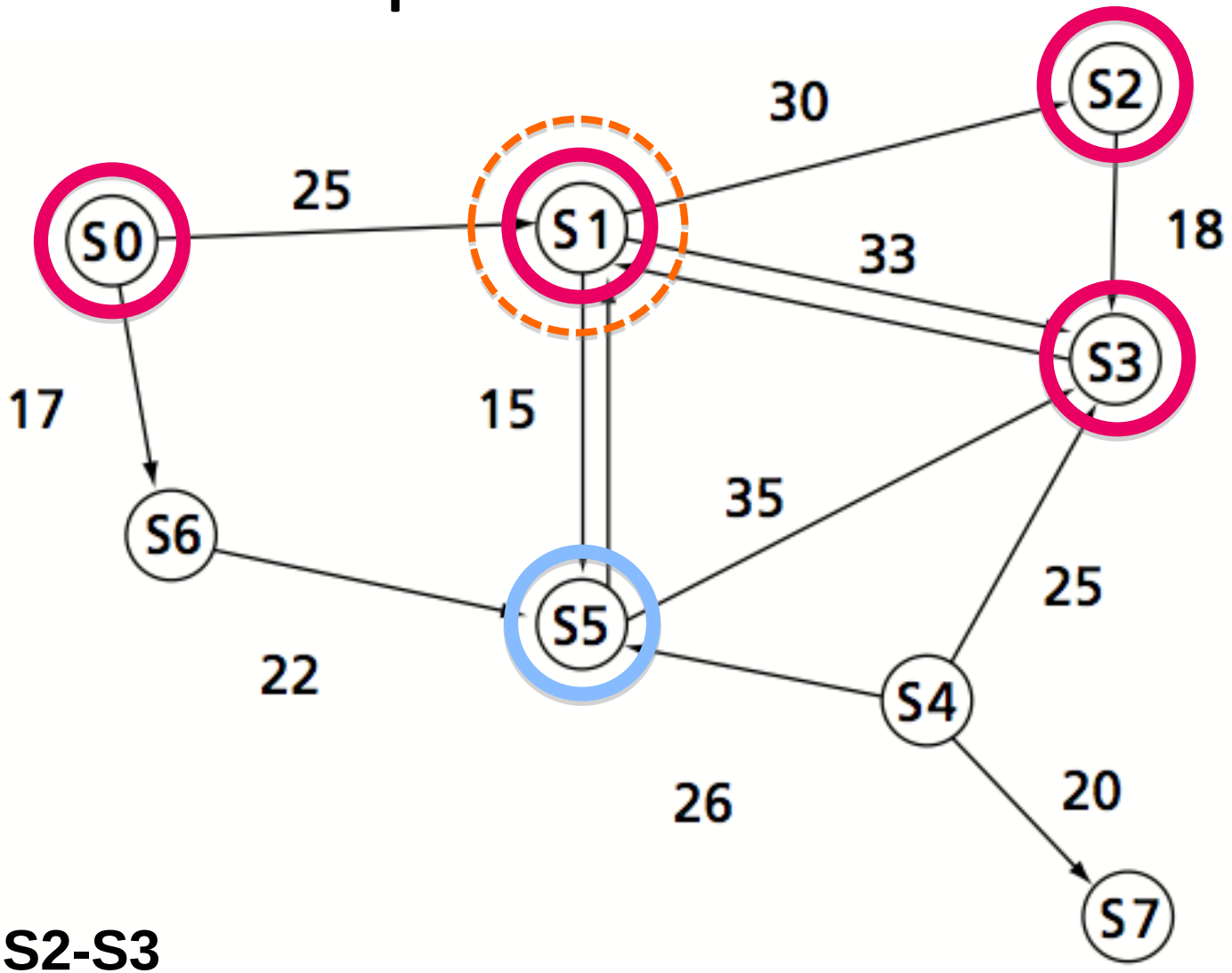
S0-S1-S2-S3

Principe du parcours en profondeur



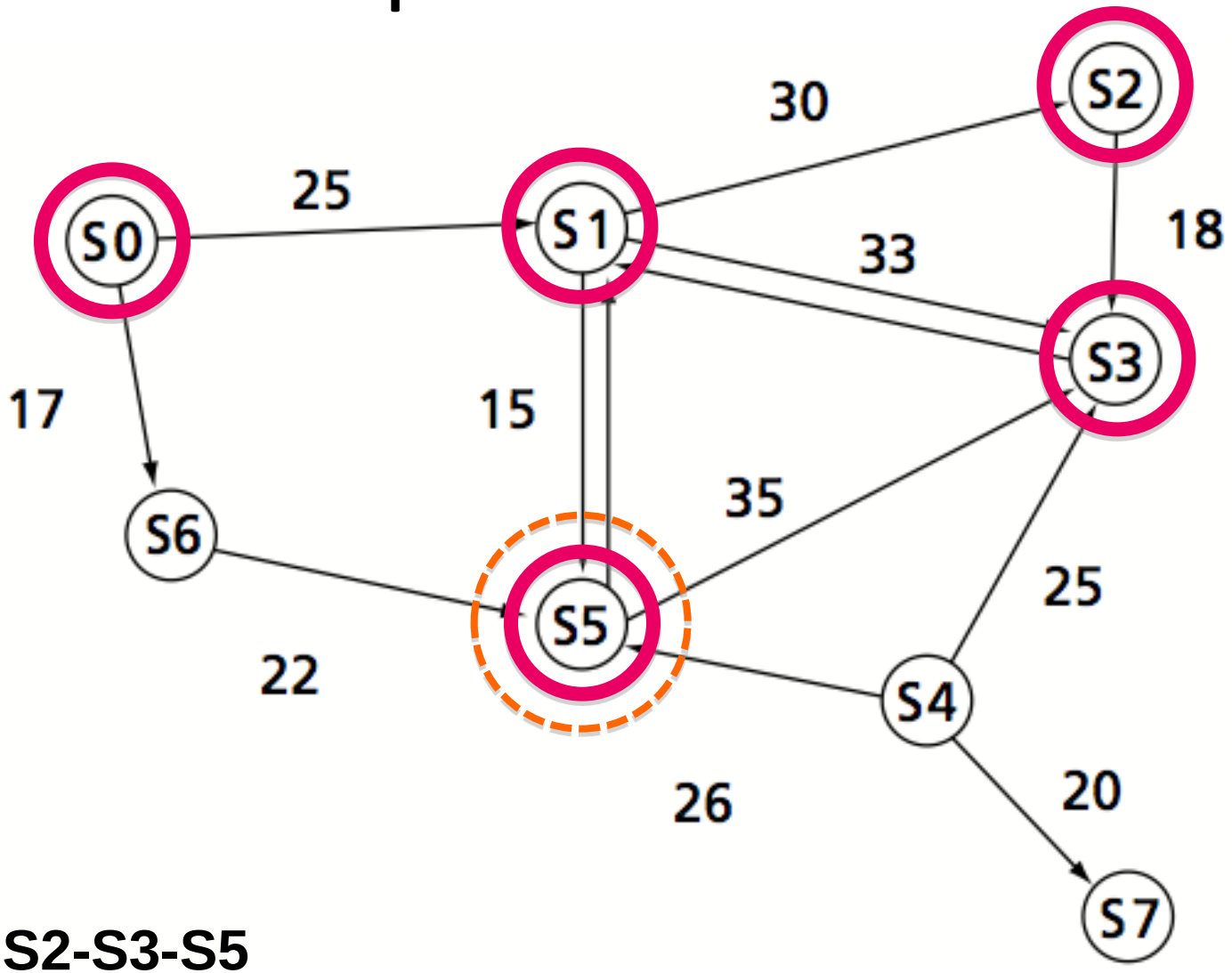
S0-S1-S2-S3

Principe du parcours en profondeur



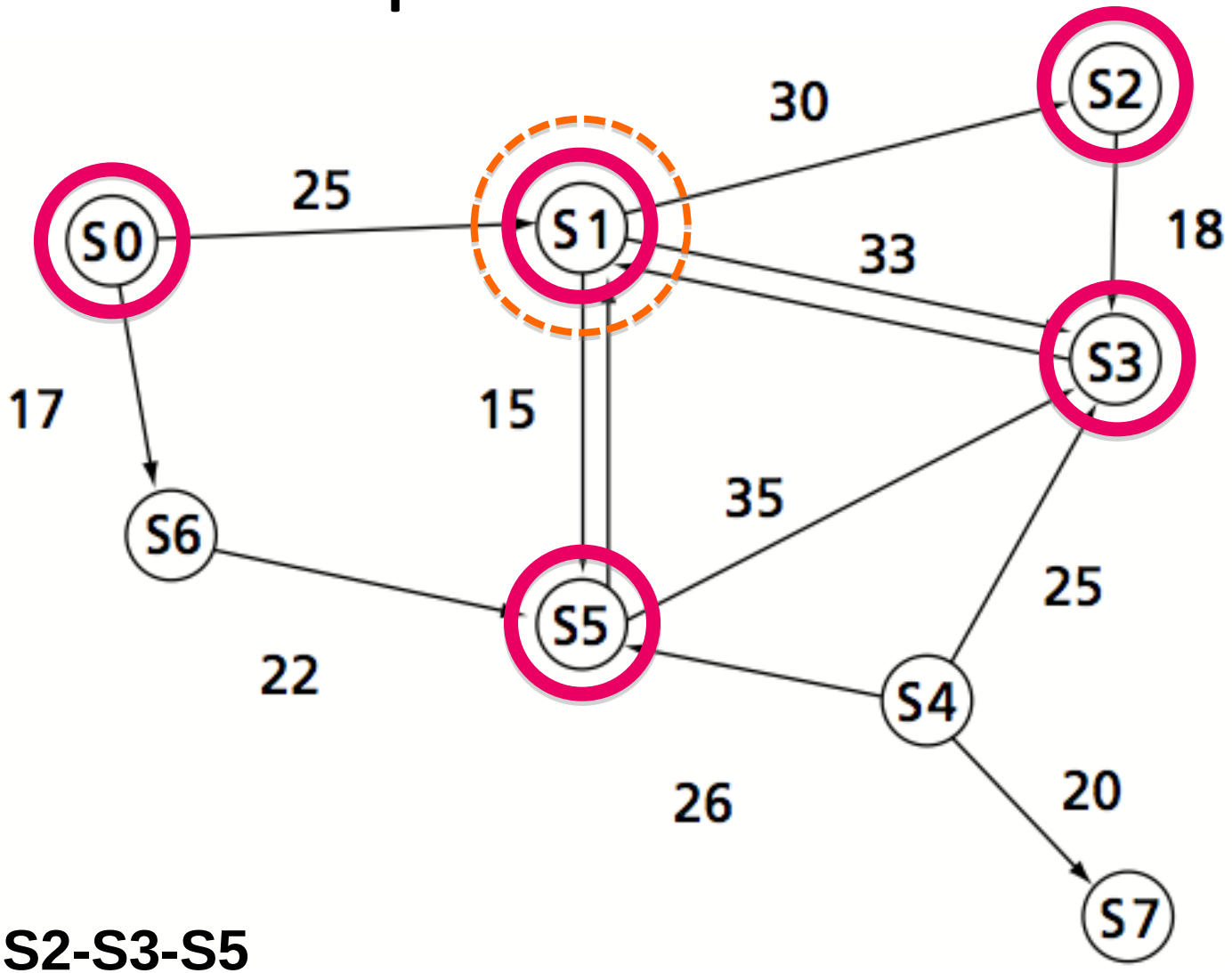
S0-S1-S2-S3

Principe du parcours en profondeur



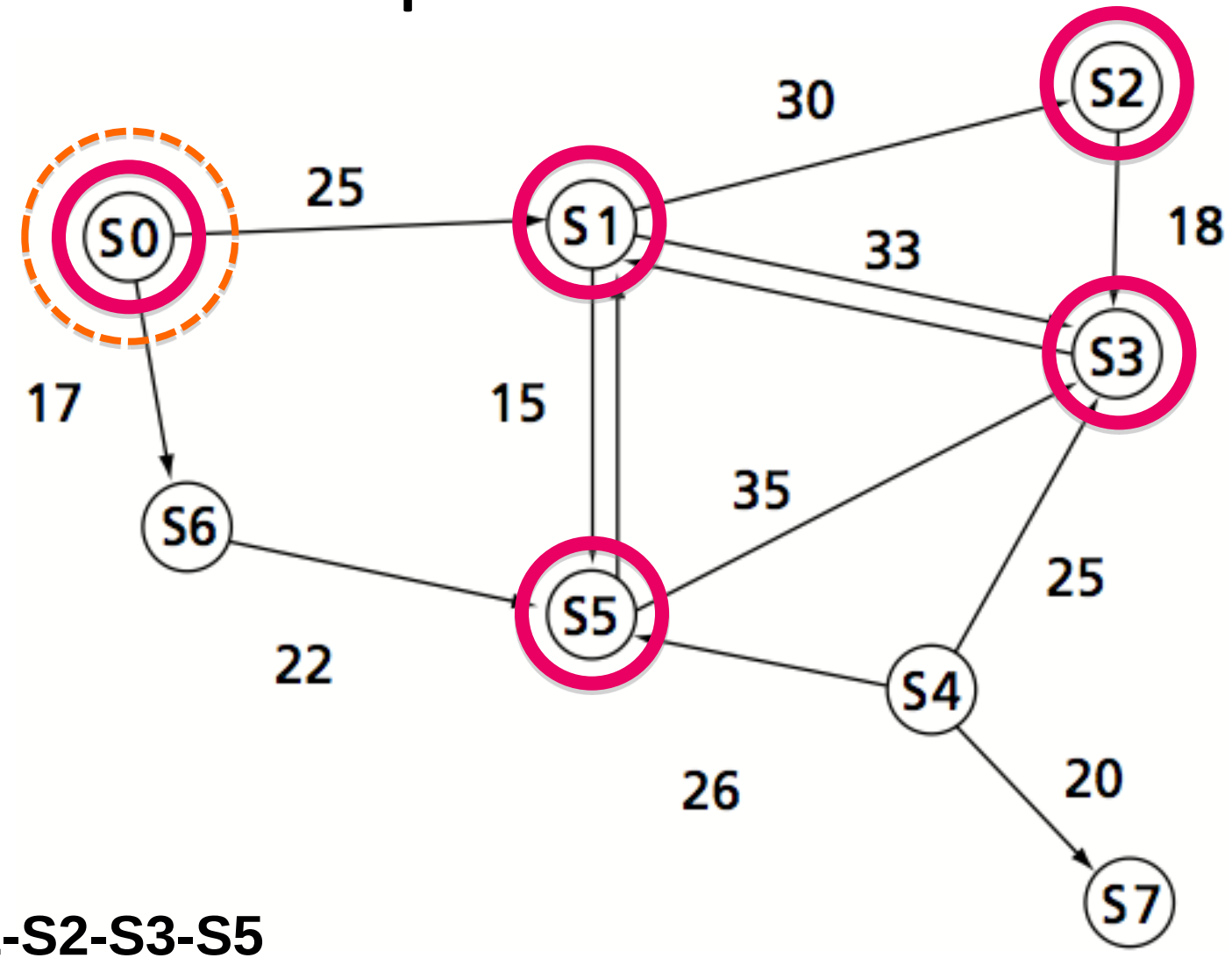
S0-S1-S2-S3-S5

Principe du parcours en profondeur



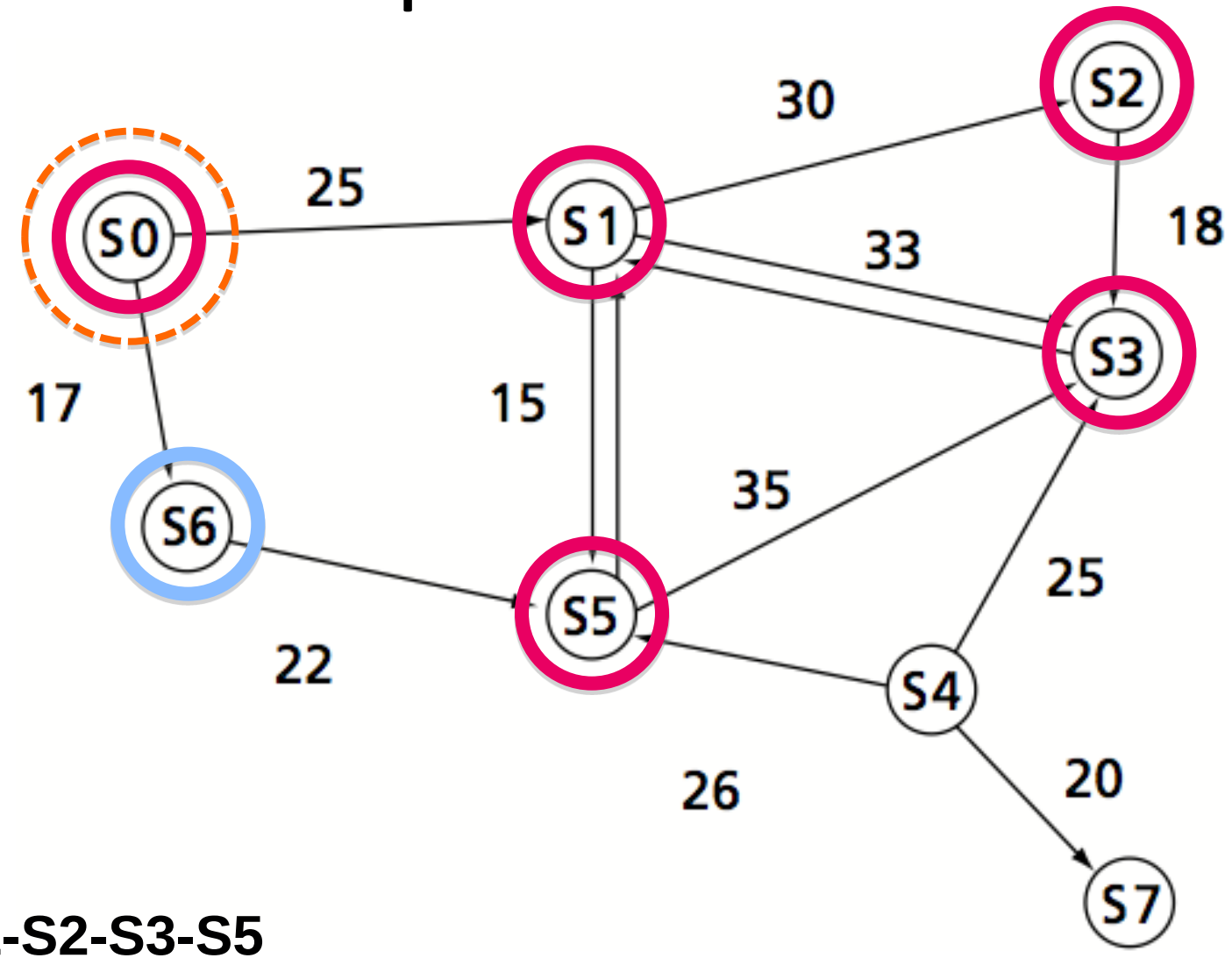
S0-S1-S2-S3-S5

Principe du parcours en profondeur



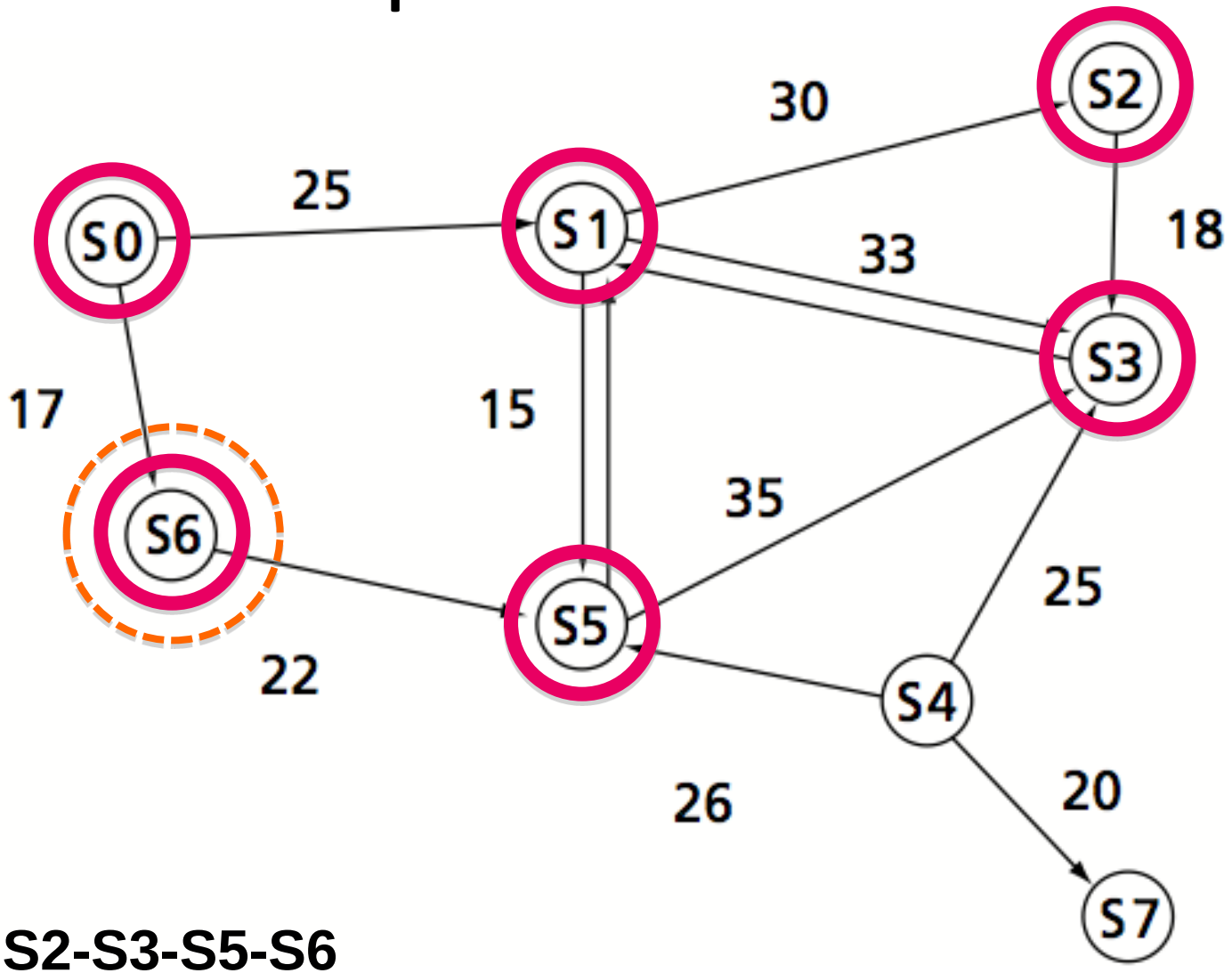
S0-S1-S2-S3-S5

Principe du parcours en profondeur



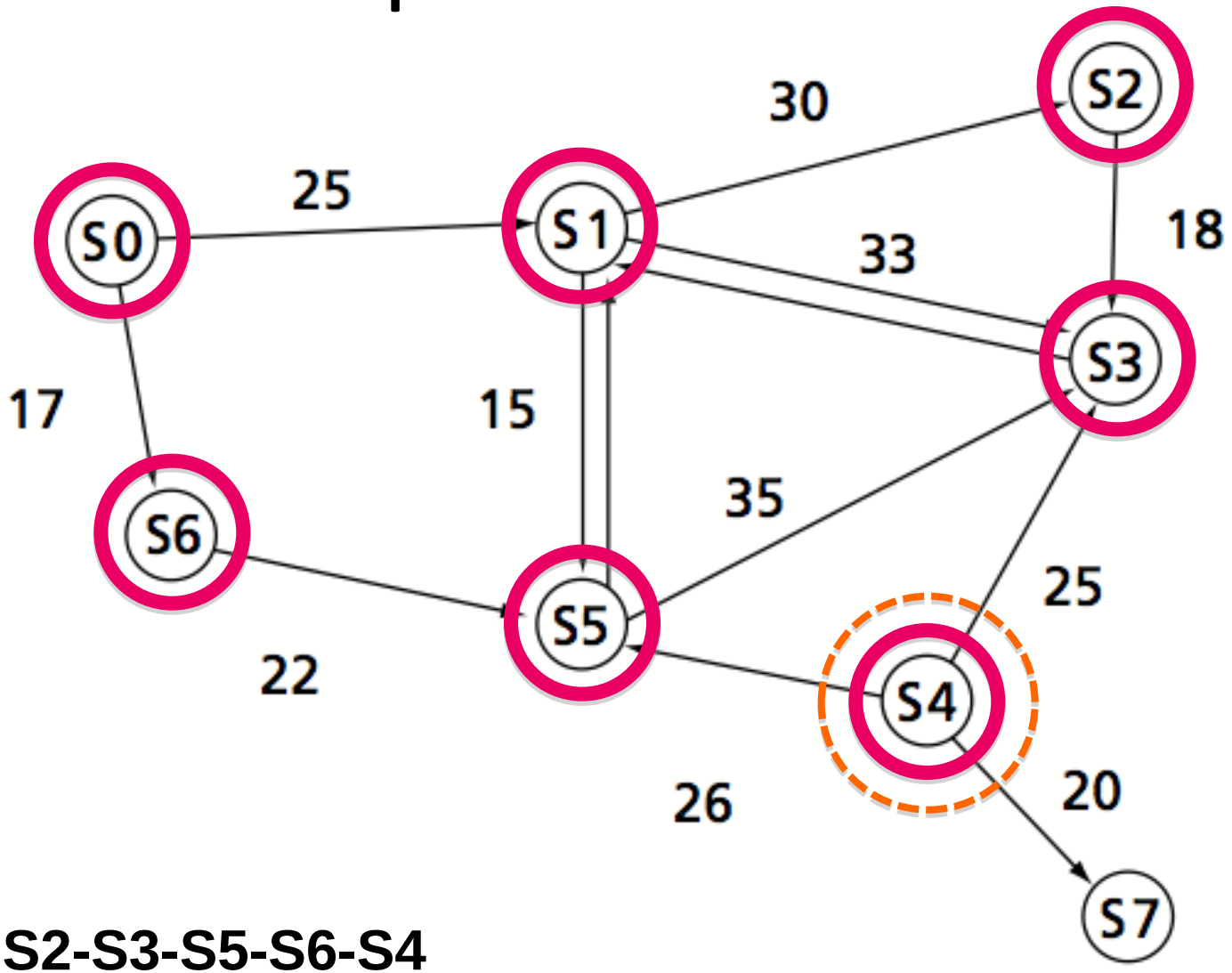
S0-S1-S2-S3-S5

Principe du parcours en profondeur



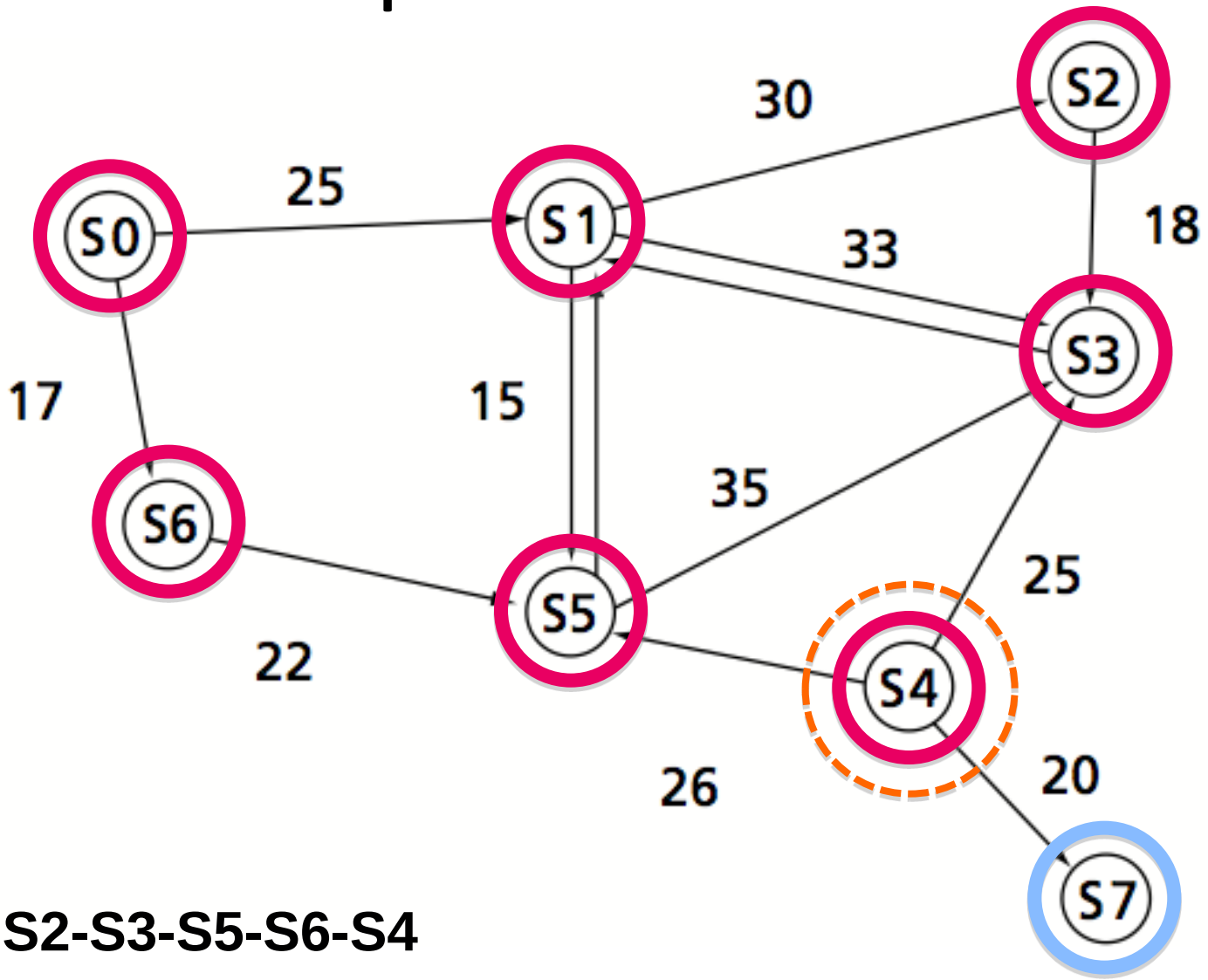
S0-S1-S2-S3-S5-S6

Principe du parcours en profondeur



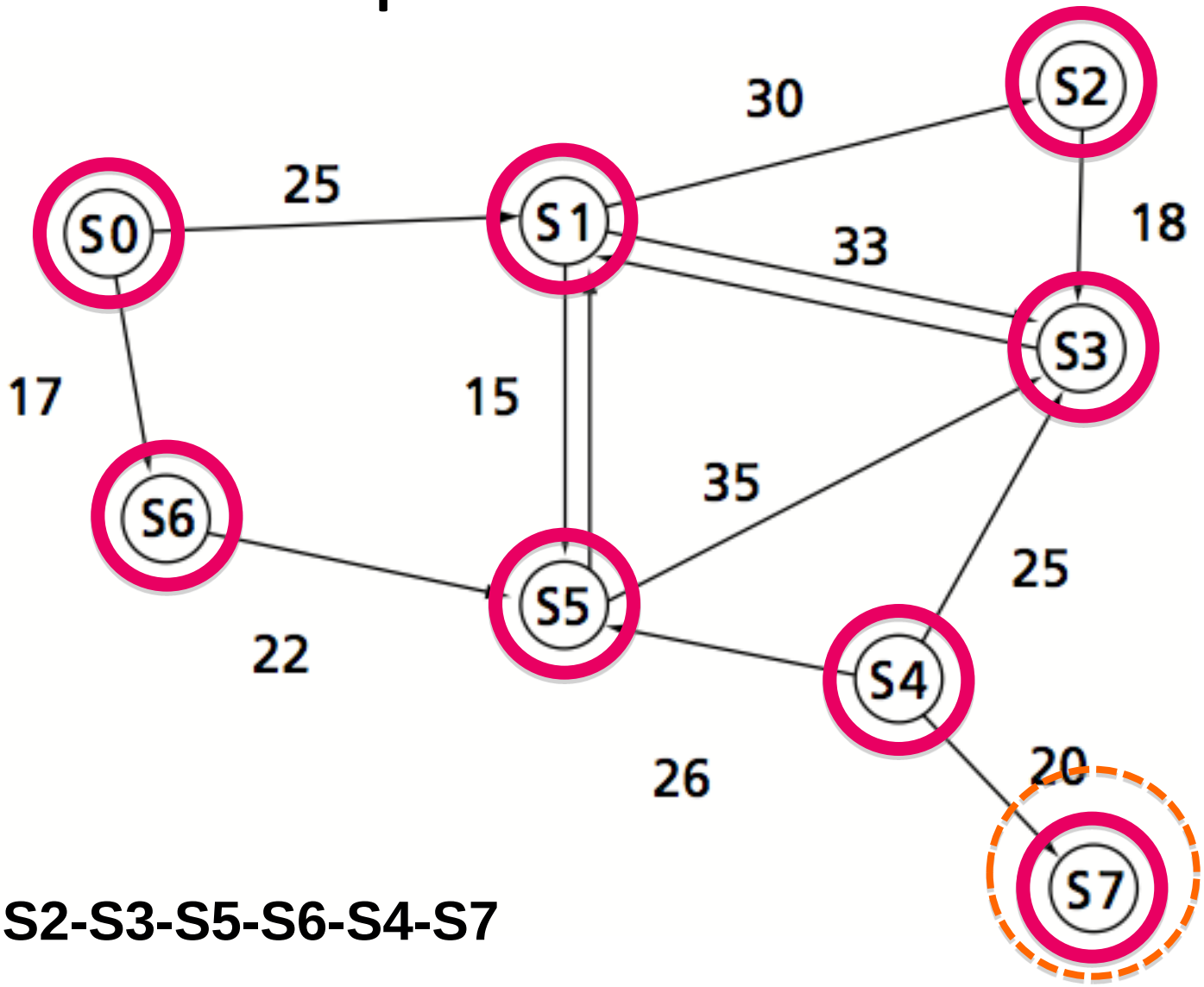
S0-S1-S2-S3-S5-S6-S4

Principe du parcours en profondeur



S0-S1-S2-S3-S5-S6-S4

Principe du parcours en profondeur

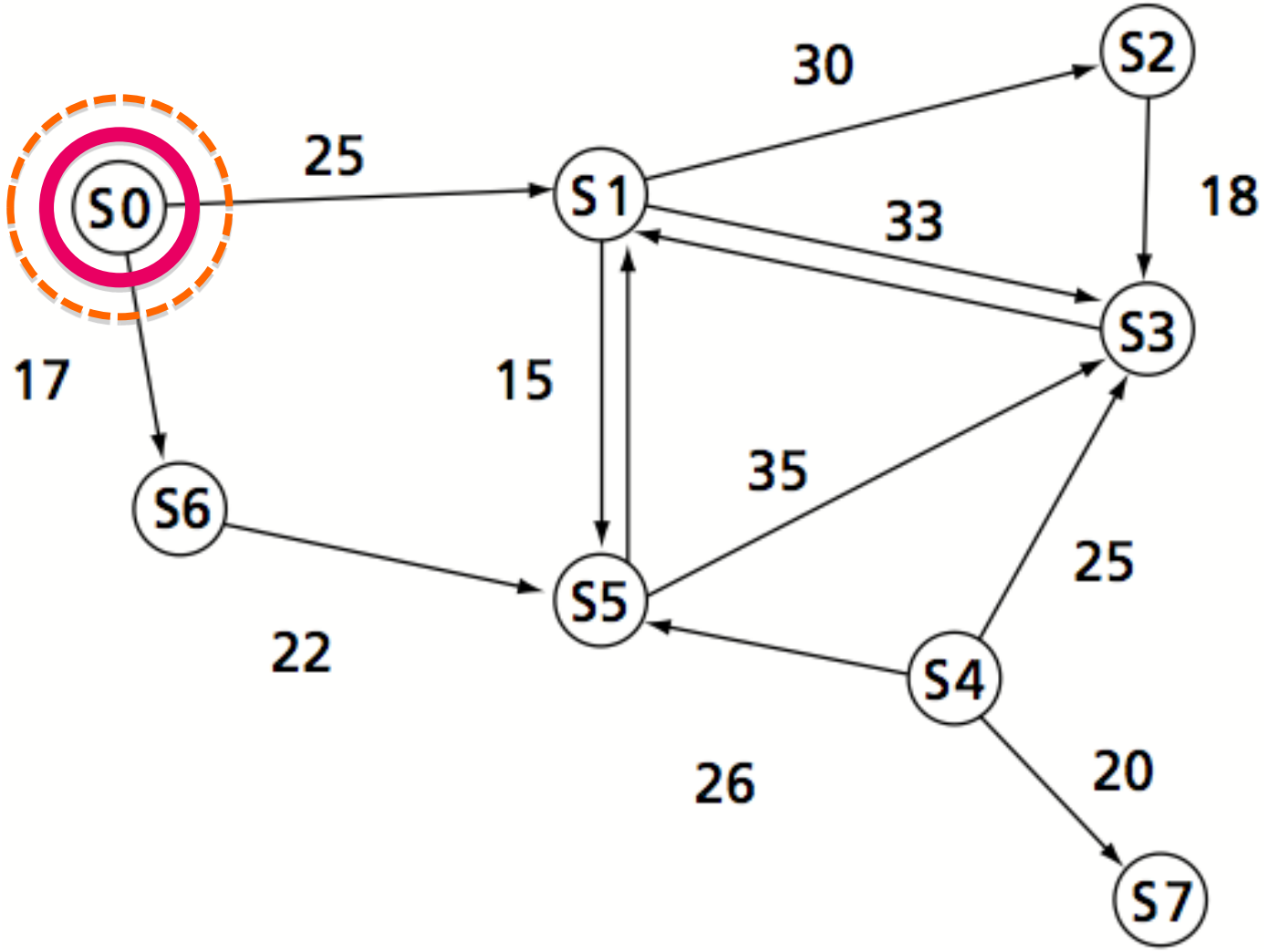


S0-S1-S2-S3-S5-S6-S4-S7

Principe du parcours en largeur

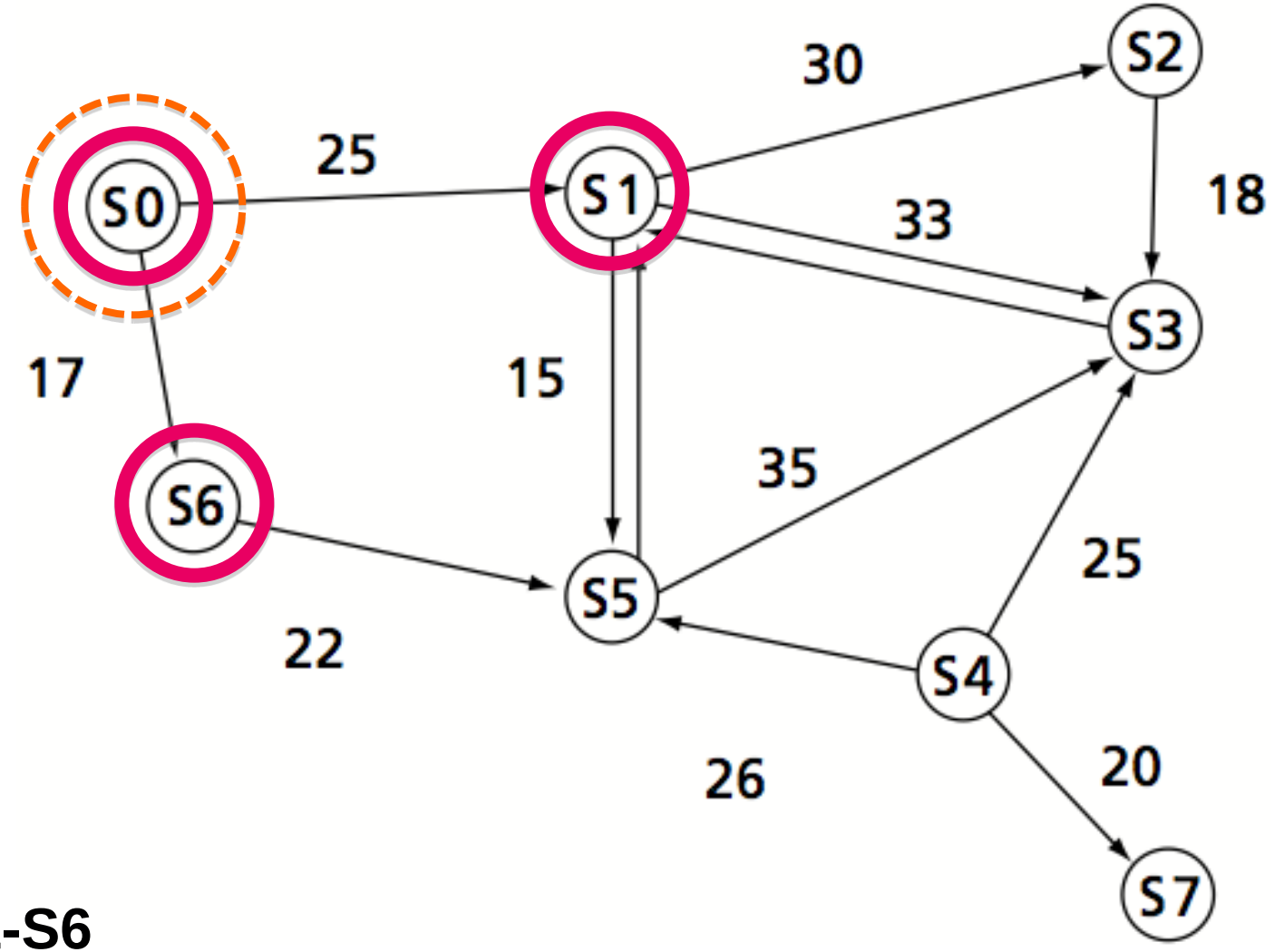
- On part d'un sommet donné. On énumère tous les fils (les suivants) de ce sommet, puis tous les petits-fils non encore énumérés, etc. C'est une énumération par génération : les successeurs directs, puis les successeurs au 2e degré, etc.

Principe du parcours en largeur



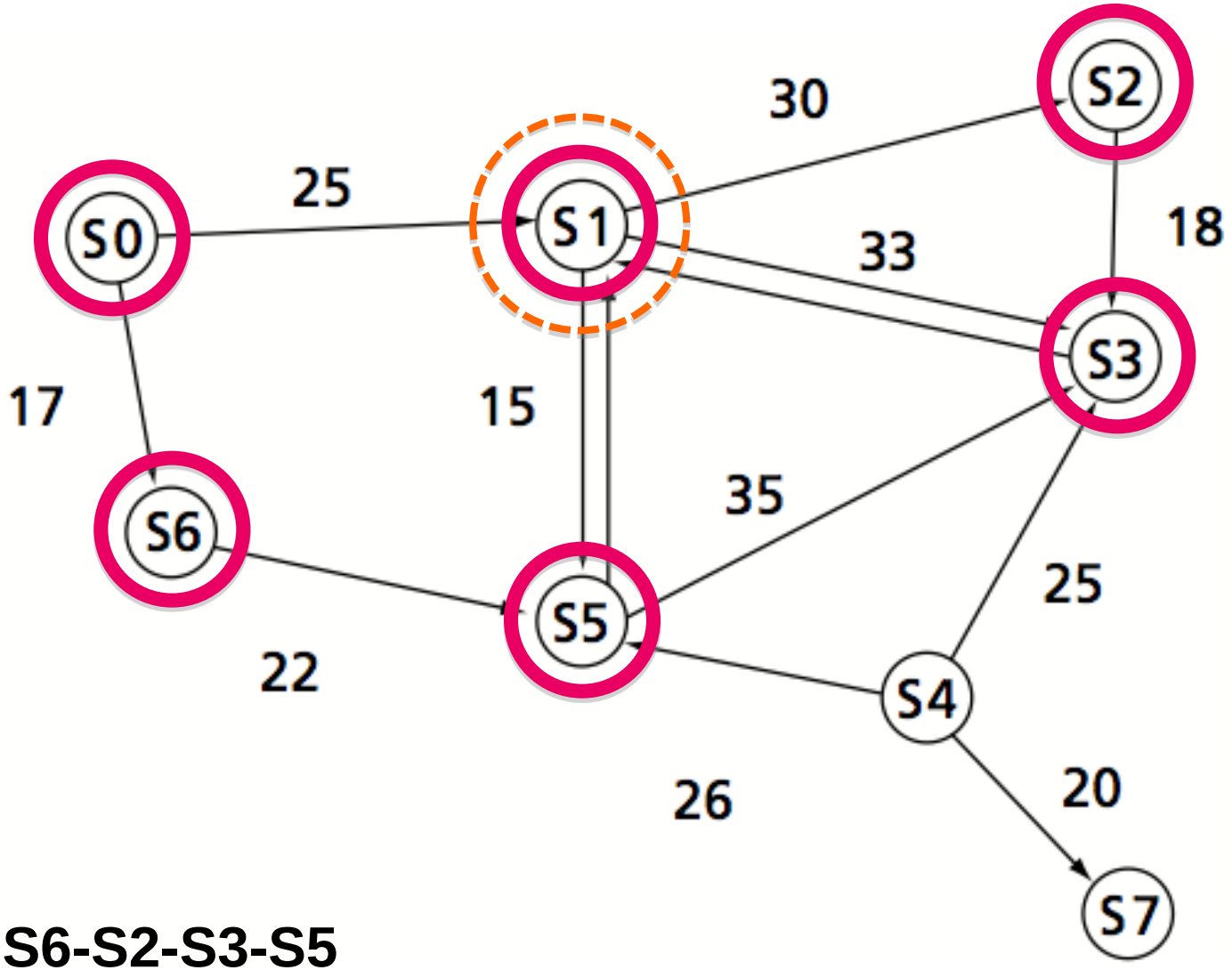
S0

Principe du parcours en largeur



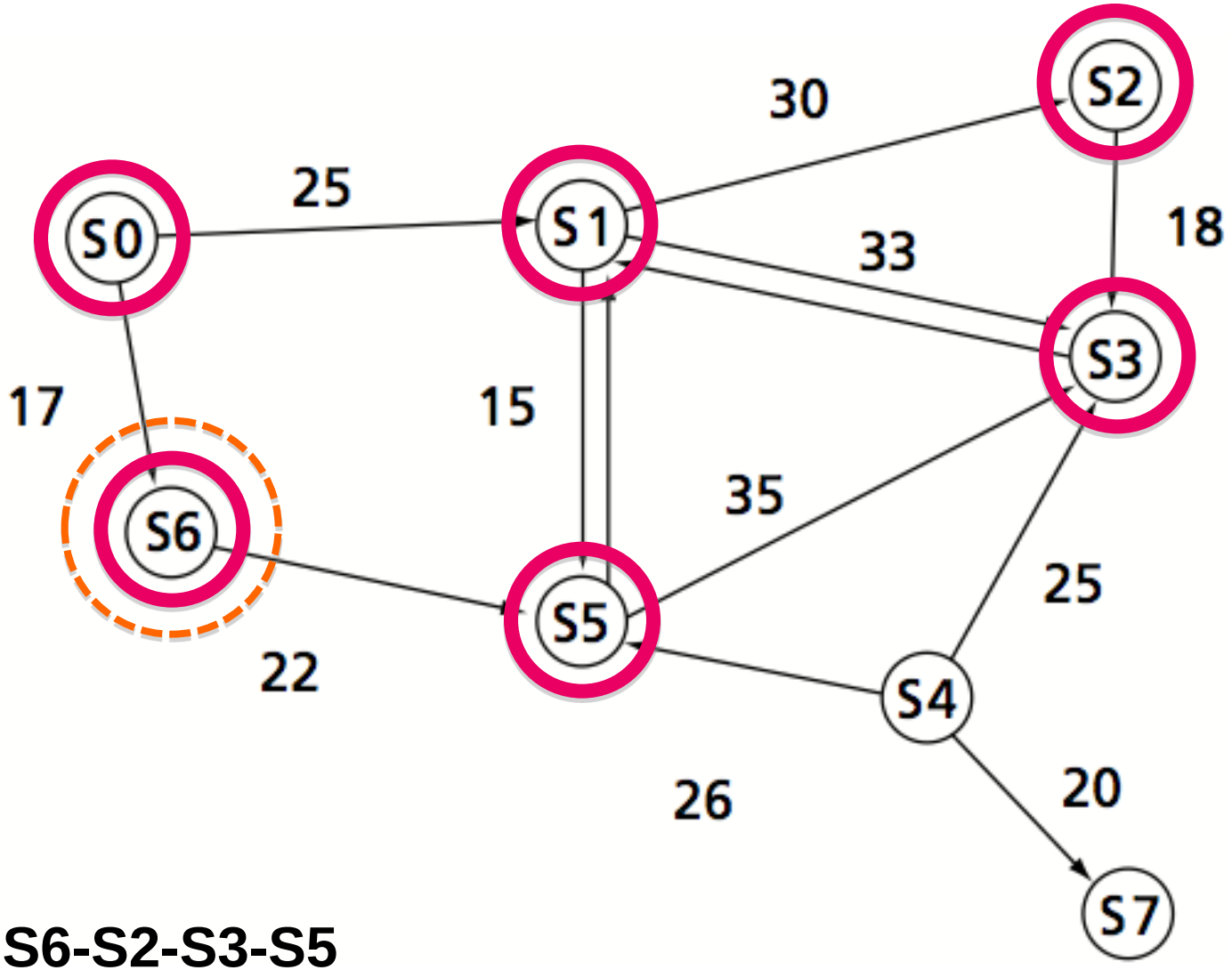
S0-S1-S6

Principe du parcours en largeur



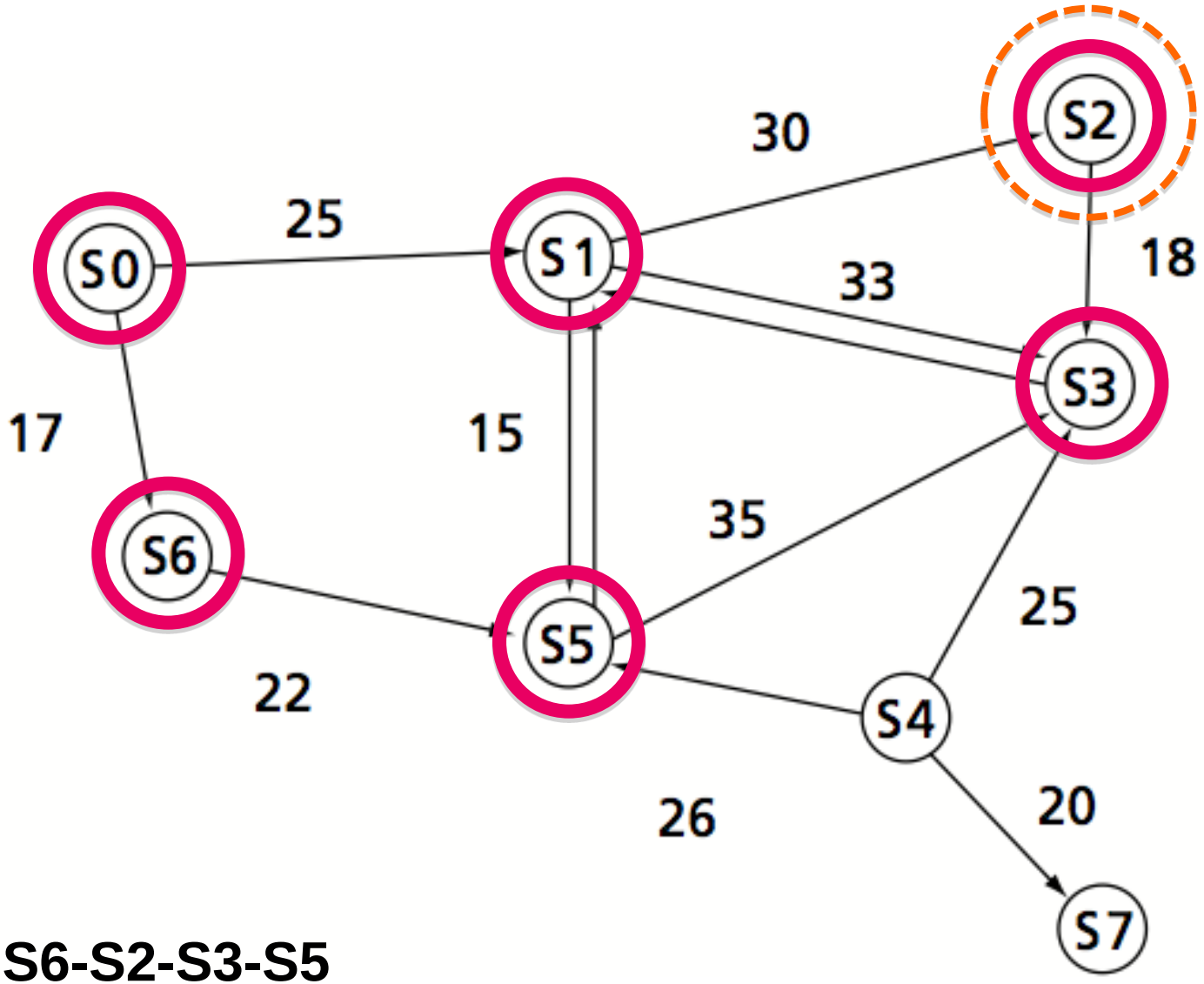
S0-S1-S6-S2-S3-S5

Principe du parcours en largeur



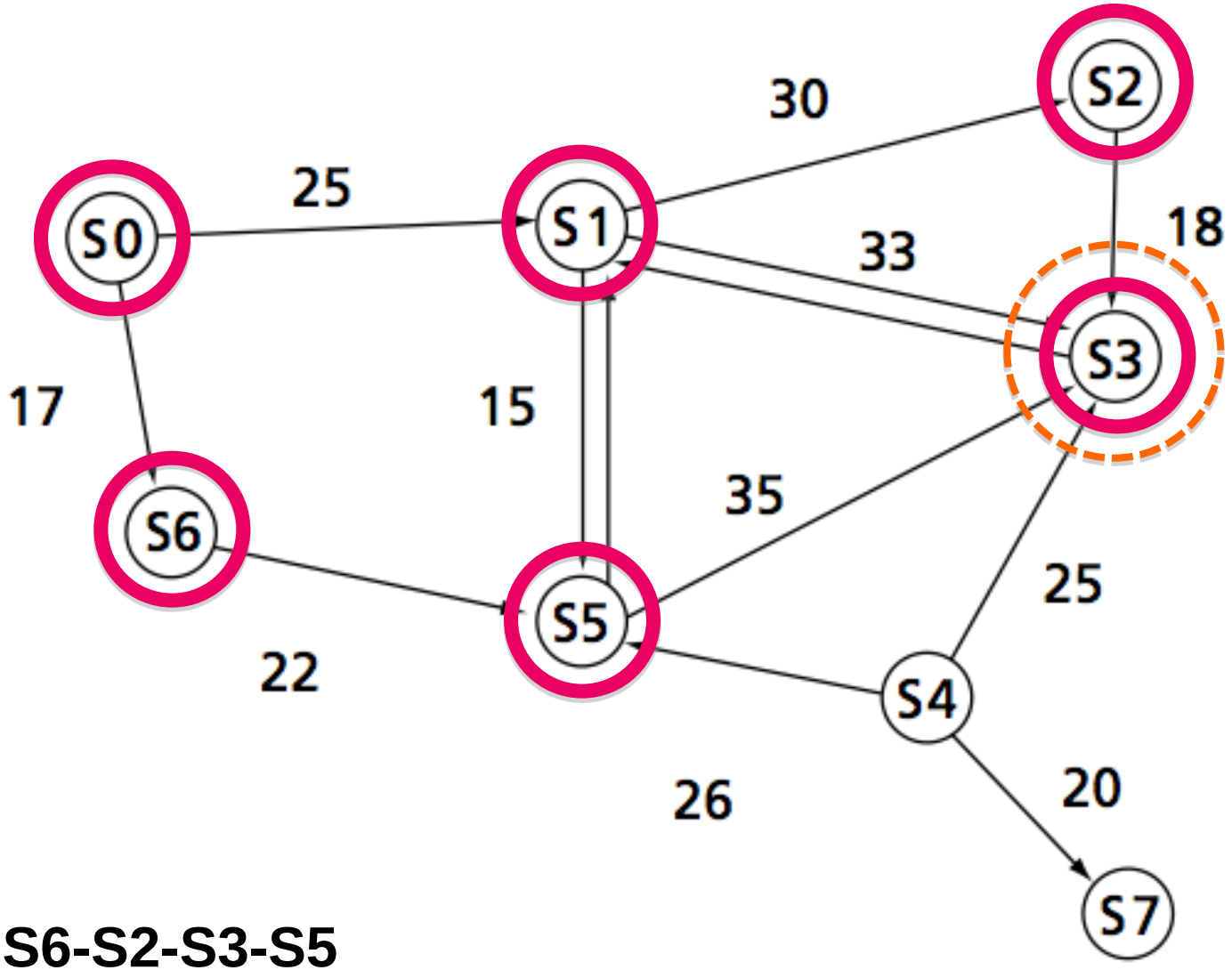
S0-S1-S6-S2-S3-S5

Principe du parcours en largeur



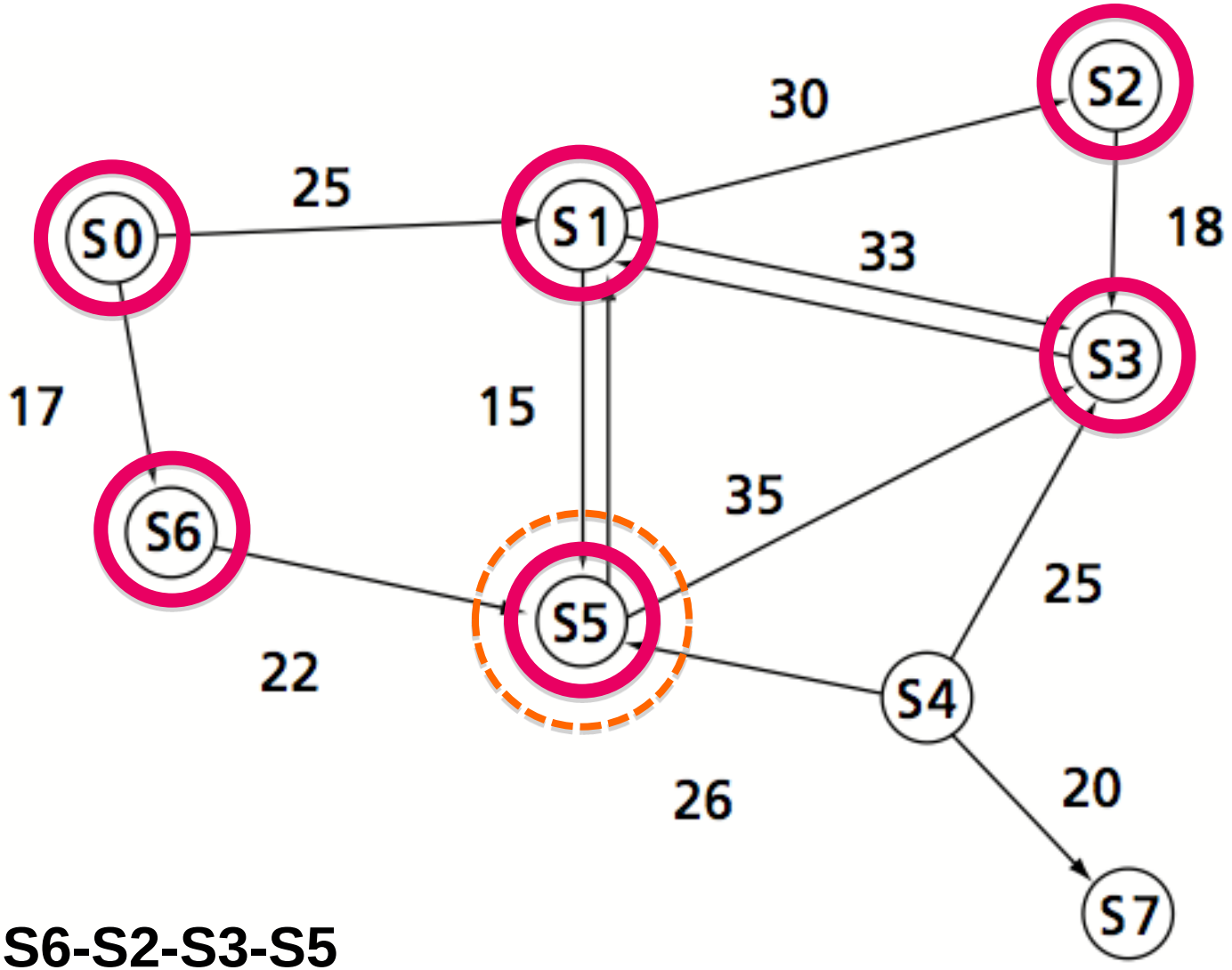
S0-S1-S6-S2-S3-S5

Principe du parcours en largeur



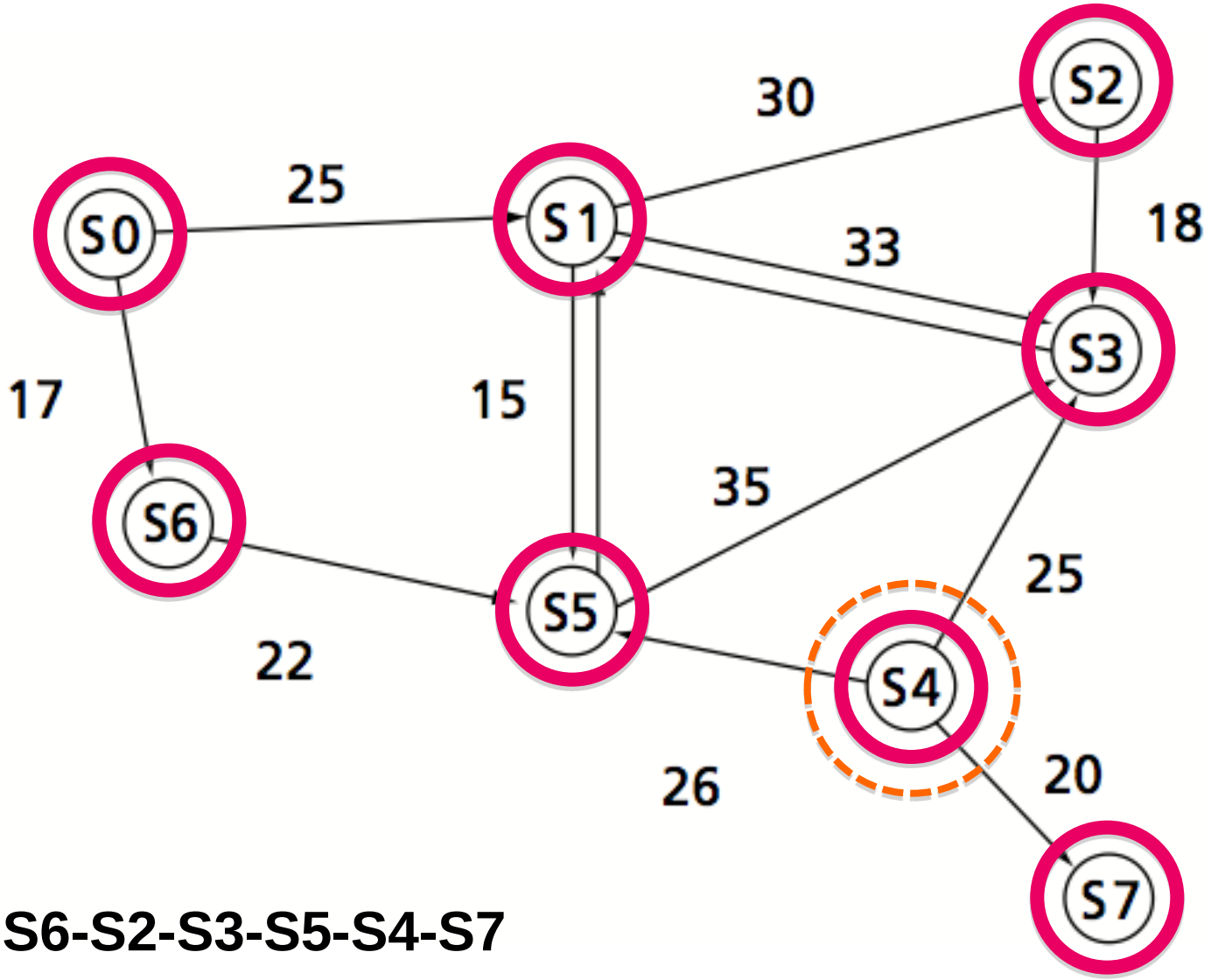
S0-S1-S6-S2-S3-S5

Principe du parcours en largeur



S0-S1-S6-S2-S3-S5

Principe du parcours en largeur



S0-S1-S6-S2-S3-S5-S4-S7

Algorithme de la fermeture transitive

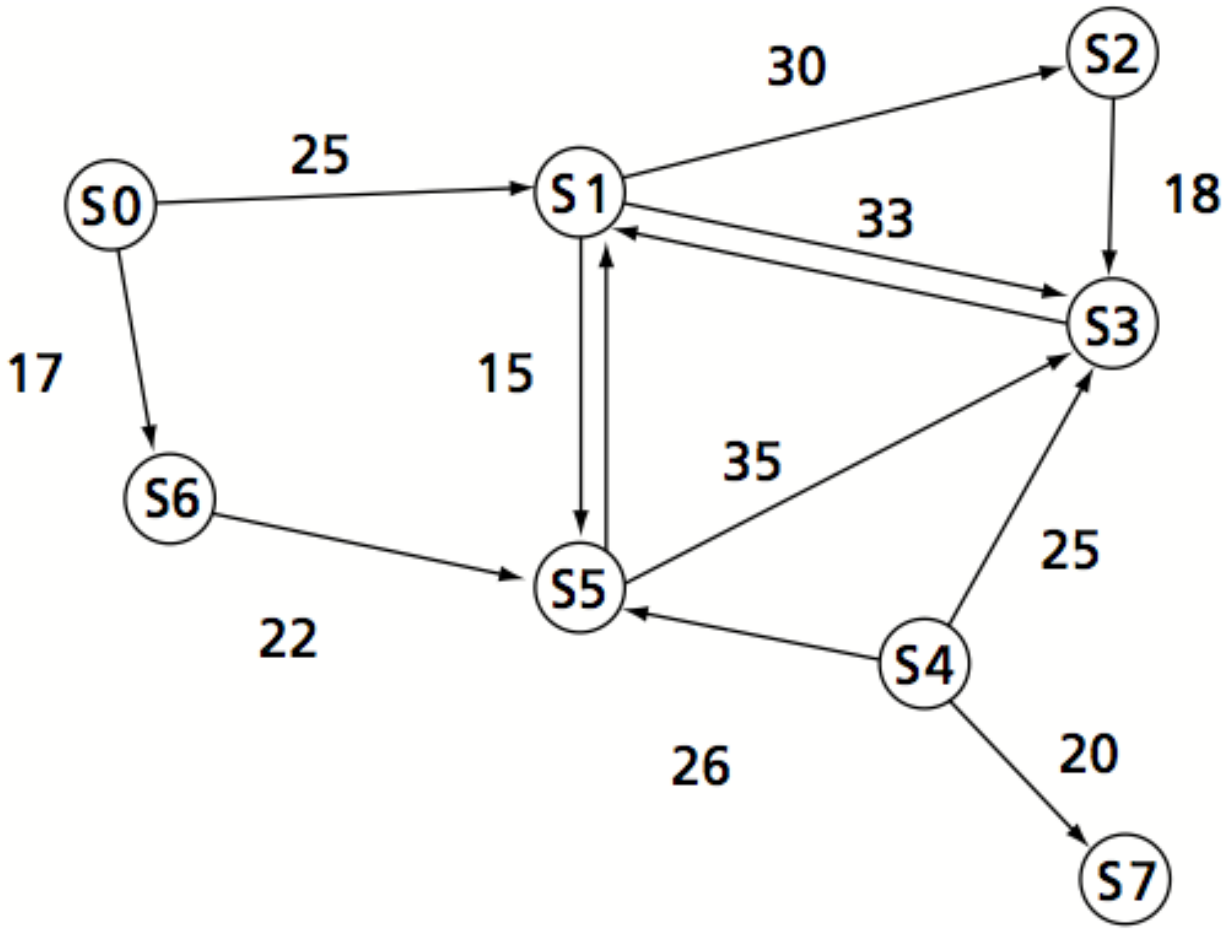
- La fermeture transitive permet de connaître l'existence d'un chemin de longueur quelconque entre 2 sommets i et j.
- Si **M est la matrice représentant l'existence d'un chemin élémentaire entre i et j**, le produit :
  - M2 = M*M représente l'existence d'un chemin de longueur 2 entre i et j ;
  - M3 = M*M*M, l'existence d'un chemin de longueur 3.
  - S'il y a N sommets, on peut calculer jusqu'à MN (M à la puissance N).

Algorithme de la fermeture transitive

- La somme SM des matrices M + M2 + ... + MN représente l'existence d'un chemin de longueur 1, 2, ... ou N, entre i et j.
- C'est la **fermeture transitive** ainsi appelée car il y a transitivité dans l'existence de chemins

Algorithme de la fermeture transitive

0	1	1	1	0	1	1	0
0	1	1	1	0	1	0	0
0	1	1	1	0	1	0	0
0	1	1	1	0	1	0	0
0	1	1	1	0	1	0	1
0	1	1	1	0	1	0	0
0	1	1	1	0	1	0	0
0	0	0	0	0	0	0	0



Problème de coloration

- Soit un graphe G=(X,U) non orienté
- Trouver le nombre de coloration minimale des sommets de G de telle sorte que 2 sommets adjacents n'aient pas la même couleur.
- Il n'existe pas d'algorithme qui permet de le résoudre en un temps polynomial
- On peut utiliser la méthode Powel & Welsh (solution approchée)

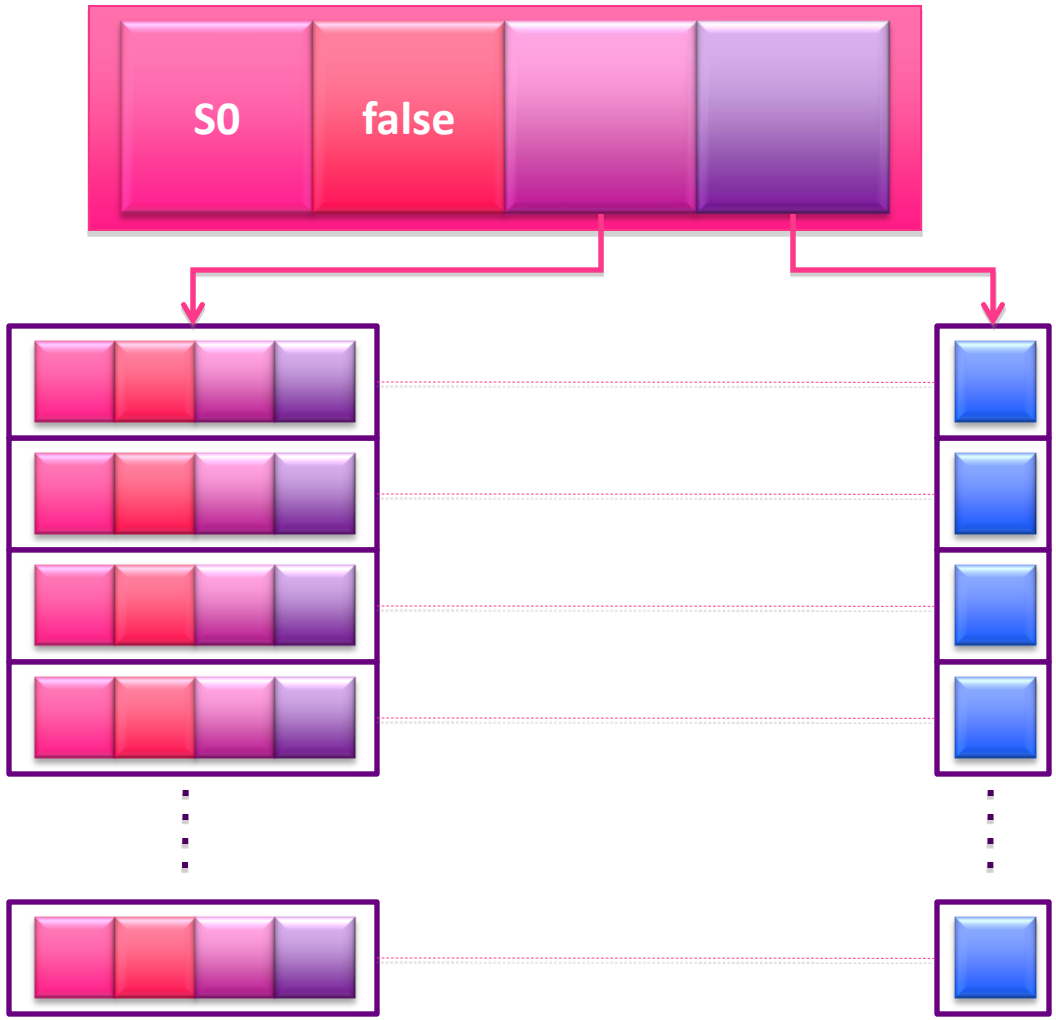
Algorithme de Powel & Welsh

1. Déterminer la matrice d'adjacence M, puis ranger les sommets selon leur ordre non croissant de leur degré. Soit M' la matrice obtenue. Poser k=1 et N=M'
2. Colorier par la couleur Ck la première ligne non encore coloriée dans N ainsi que la colonne correspondante
3. Soit N l'ensemble des lignes non encore coloriées ayant un 0 dans les colonnes de couleur Ck
4. Tester si N est vide :
    1. Si OUI : aller en 5
    2. SI NON : aller en 2
5. Tester si toutes les lignes sont coloriées :
    1. Si OUI : FIN
    2. Si NON : poser k=k+1, N=M', aller en 2

Mémorisation d'un graphe

Nom	Marque	Liste des voisins	Listes des coûts
"S0"	false	S1,S6	25, 17
"S1"	false	S2, S3, S5	30, 33, 15
"S2"	false	S3	18
"S3"	false	S1	33
"S4"	false	S3, S5, S7	25, 26, 20
"S5"	false	S1, S3	15, 35
"S6"	false	S5	22
"S7"	false		

Mémorisation d'un graphe



Graphe : programme Java

La classe Sommet :

```
public class Sommet {
    private String nom ;
    private boolean marque = false ;
    private LinkedList<Sommet> voisins ;
    private LinkedList<Integer> couts ;
...
}
```

Graphe : programme Java

Le constructeur :

```
public Sommet(String nom) {
    voisins = new LinkedList<Sommet>() ;
    couts = new LinkedList<Integer>() ;
    this.nom = nom ;
}
```

Graphe : programme Java

Ajoute d’un sommet voisin :

```
public void ajouterVoisin(Sommet s, int c) {
    voisins.add(s) ;
    couts.add(c) ;
}
```

Graphe : programme Java

Récupération du nom du sommet :

```
public String getNom() {
    return nom ;
}
```

Graphe : programme Java

Marquer un sommet :

```
public void marquer() {
    marque = true ;
}
```

Graphe : programme Java

Retourner l'état d'un sommet (marqué ou non) :

```
public boolean etat() {
    return marque ;
}
```

Graphe : programme Java

Retourner la distance entre le sommet et l'un de ses voisins :

```
public int getCout(Sommet s) {
    for(int i=0; i <voisins.size(); i++) {
      if(s == voisins.get(i)) {
       return couts.get(i);
      }
    }
    return -1 ;
}
```

Graphe : programme Java

Retourner la liste des successeurs d'un sommet :

```
public LinkedList<Sommet> getVoisins() {
    return voisins ;
}
```

Graphe : programme Java

Afficher un sommet et ses voisins :

```
public void afficher() {
    System.out.print(getNom()+" : ");
    for(Sommet s : voisins) {
     System.out.print(s.getNom()+" ") ;
    }
}
```

Graphe : programme Java

La classe Graphe :

```
public class Graphe {
    private LinkedList<Sommet> liste ;
    ...
}
```

Graphe : programme Java

Le constructeur :

```
public Graphe() {
    liste = new LinkedList<Sommet>() ;
}
```

Graphe : programme Java

Ajouter un sommet au graphe :

```
public void ajouter(Sommet s) {
    liste.add(s);
}
```

Graphe : programme Java

Récupérer un sommet par son indexe :

```
public Sommet get(int index) {
    return liste.get(index) ;
}
```

Graphe : programme Java

Récupérer l'index d'un sommet :

```
public int getIndex(Sommet s) {
    for(int i=0; i<liste.size(); i++) {
     if(liste.get(i).getNom() == s.getNom())
           return i ;
    }
    return -1;
}
```

Graphe : programme Java

Récupérer le nombre de sommets du graphe :

```
public int size() {
    return liste.size() ;
}
```

Graphe : programme Java

Récupérer un sommet par son nom :

```
public Sommet getSommetParNom(String nom) {
    for(Sommet s : liste) {
      if(s.getNom() == nom)
      return s ;
    }
    return null;
}
```

Graphe : programme Java

Récupérer le coût entre deux sommets :

```
public int getCout(Sommet s1, Sommet s2) {
    return (s1.getCout(s2));
}
```

Graphe : programme Java

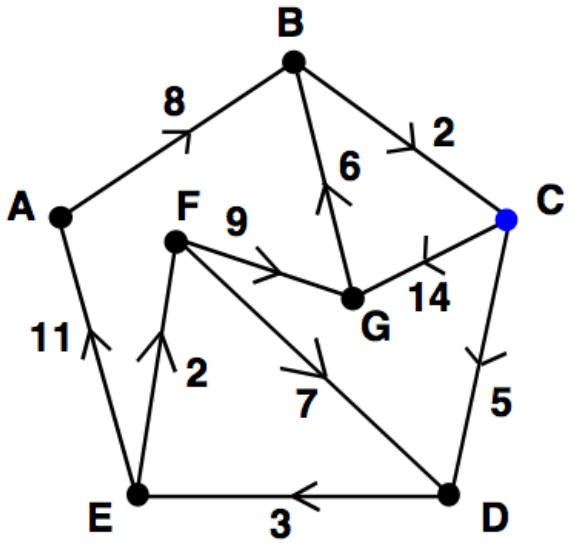
Afficher le graphe :

```
public void afficher() {
    for(Sommet s : liste) {
      s.afficher() ;
      System.out.println() ;
    }
}
```

Graphe : programme Java

Exécution : la classe Test

```
public static void main(String [] args) {
      Graphe gr = new Graphe() ;
      Sommet a = new Sommet("A1");
      Sommet b = new Sommet("B2");
      Sommet c = new Sommet("C3");
      Sommet d = new Sommet("D4");
      Sommet e = new Sommet("E5");
      Sommet f = new Sommet("F6");
      Sommet g = new Sommet("G7");
...
}
```



Graphe : programme Java

Exécution : la classe Test

```
a.ajouterVoisin(b,8);
b.ajouterVoisin(c,2);
c.ajouterVoisin(d,5);    c.ajouterVoisin(g,14);
d.ajouterVoisin(e,3);
e.ajouterVoisin(a,11);   e.ajouterVoisin(f,2);
f.ajouterVoisin(d,7);    f.ajouterVoisin(g,9);
g.ajouterVoisin(b,6);
```

Graphe : programme Java

Exécution : la classe Test

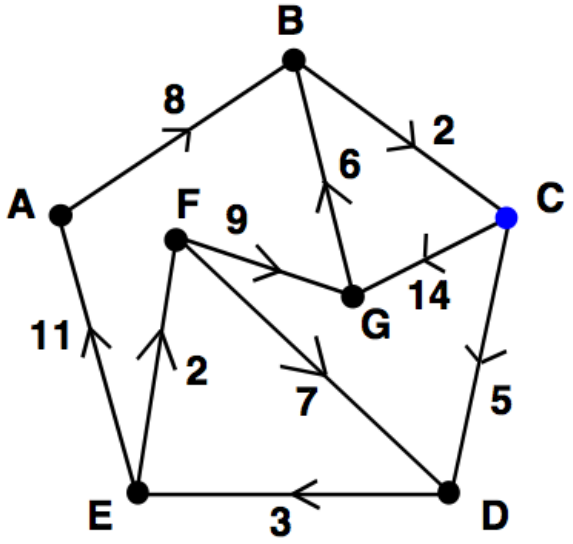
```
gr.ajouter(a);
gr.ajouter(b);
gr.ajouter(c);
gr.ajouter(d);
gr.ajouter(e);
gr.ajouter(f);
gr.ajouter(g);
```

Graphe : programme Java

Exécution : la classe Test

gr.afficher() ;

```
A : B
B : C
C : D G
D : E
E : A F
F : D G
G : B
```



Question ?