Algorithmique des graphes

David Pichardie

13 Avril 2018

Bilan du CM6

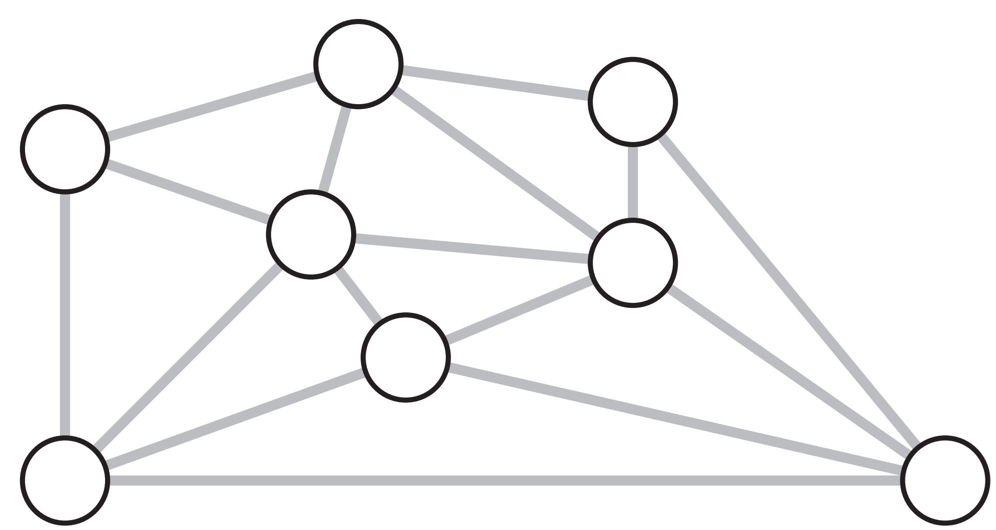
- Graphes pondérés
- Arbres couvrants
- Arbres couvrants minimaux (ACM)
  - Unicité de l'ACM
  - Propriété de la coupure
  - Algorithme glouton abstrait
  - Algorithm de Prim

Propriété de la coupure

Soit une coupure (A,B) de G. Soit *e* l'arête traversante de poids minimum vis à vis de cette coupure. Alors *e* appartient forcement à ACM.

Algorithme (abstrait) glouton

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.
- On choisie une coupure avec aucune arête traversante noire.
- On prend l'arête traversante de poids minimum et on la colorie en noir.
- On répète jusqu'à avoir colorié S-1 arêtes en noir.



Algorithme de Prim

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM. A chaque étape, le sous-graphe noir est un arbre. On colorie en noir les sommets reliés par des arêtes noirs.
- On choisie un sommet de départ.
- On considère la coupure partitionnant les sommets noirs des autres. On choisi un arc traversant minimal pour cette coupure et on le colorie en noir. On colorie en noir le sommet associé qui ne l'était pas encore.
- On répète jusqu'à avoir colorié tous les sommets en noir.

Algorithme de Prim

Correction

Algorithme de Prim

Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe

Algorithme de Prim
Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe
    - c’est vrai au départ (graphe vide)

Algorithme de Prim
Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe
    - c'est vrai au départ (graphe vide)
    - c'est encore vrai après avoir ajouté une arête et un sommet noir

Algorithme de Prim
Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe
  - c'est vrai au départ (graphe vide)
  - c'est encore vrai après avoir ajouté une arête et un sommet noir
- A chaque étape, l'arête choisie vérifie les hypothèses de l'algorithme glouton générique

Algorithme de Prim
Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe
  - c’est vrai au départ (graphe vide)
  - c’est encore vrai après avoir ajouté une arête et un sommet noir
- A chaque étape, l’arête choisie vérifie les hypothèses de l’algorithme glouton générique
  - c’est l’arête minimum pour la coupure (noir / pas-noir) courante

Algorithme de Prim

Correction

- A chaque étape, le sous graphe noir (sommets+arêtes) est connexe
  - c'est vrai au départ (graphe vide)
  - c'est encore vrai après avoir ajouté une arête et un sommet noir
- A chaque étape, l'arête choisie vérifie les hypothèses de l'algorithme glouton générique
  - c'est l'arête minimum pour la coupure (noir / pas-noir) courante
- Chaque étape ajoute donc une arête appartenant à l'ACM

Algorithme de Kruskal

Algorithme de Kruskal

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.

Algorithme de Kruskal

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.
- On considère tous les arêtes par ordre croissant de poid. Et on décide à chaque étape si l'arête appartient à l'ACM ou pas.

Algorithme de Kruskal

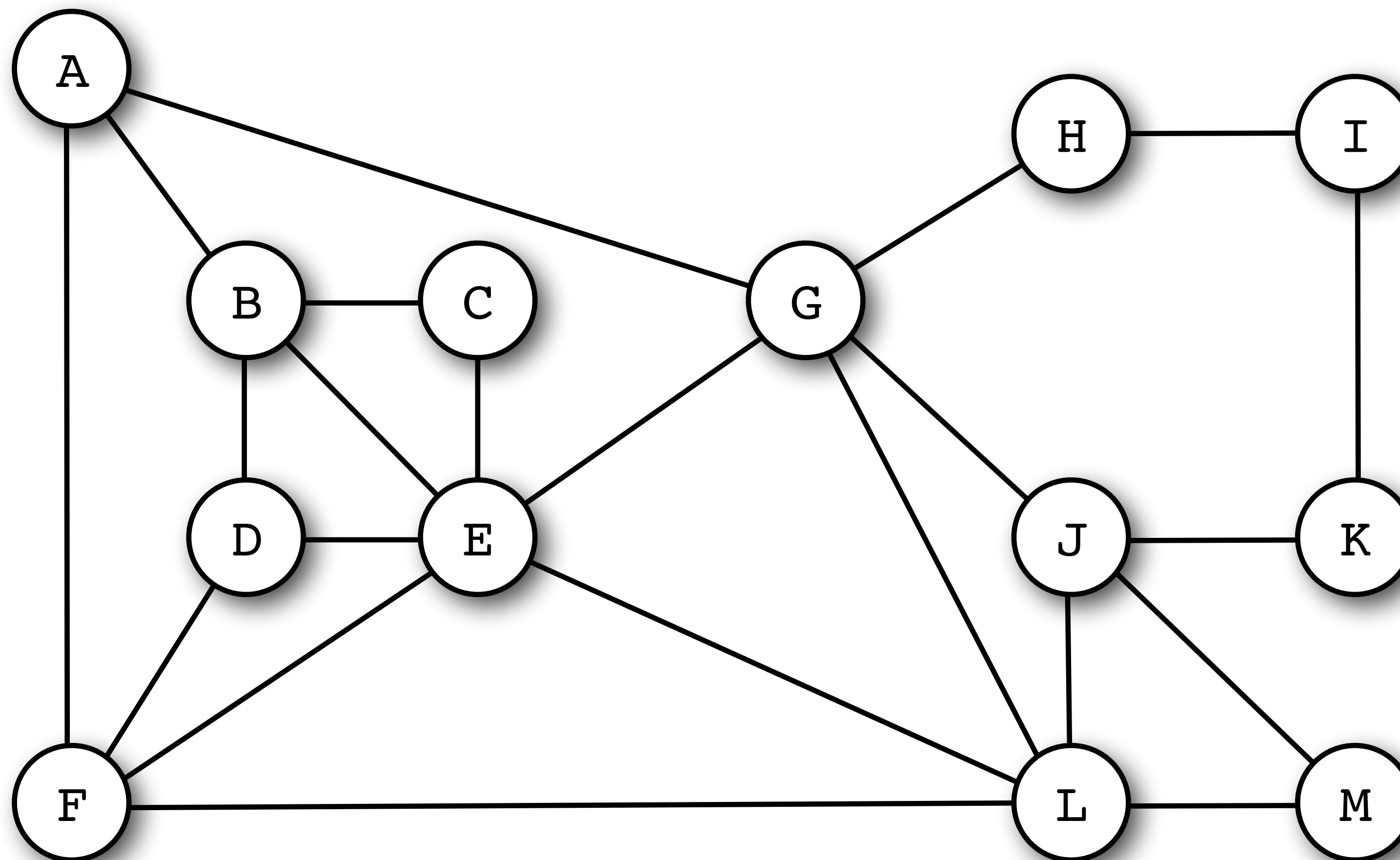
- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.
- On considère tous les arêtes par ordre croissant de poid. Et on décide à chaque étape si l'arête appartient à l'ACM ou pas.
  - Si ajouter l'arête au sous-graphe noir crée un cycle noir, on ne le colorie pas.

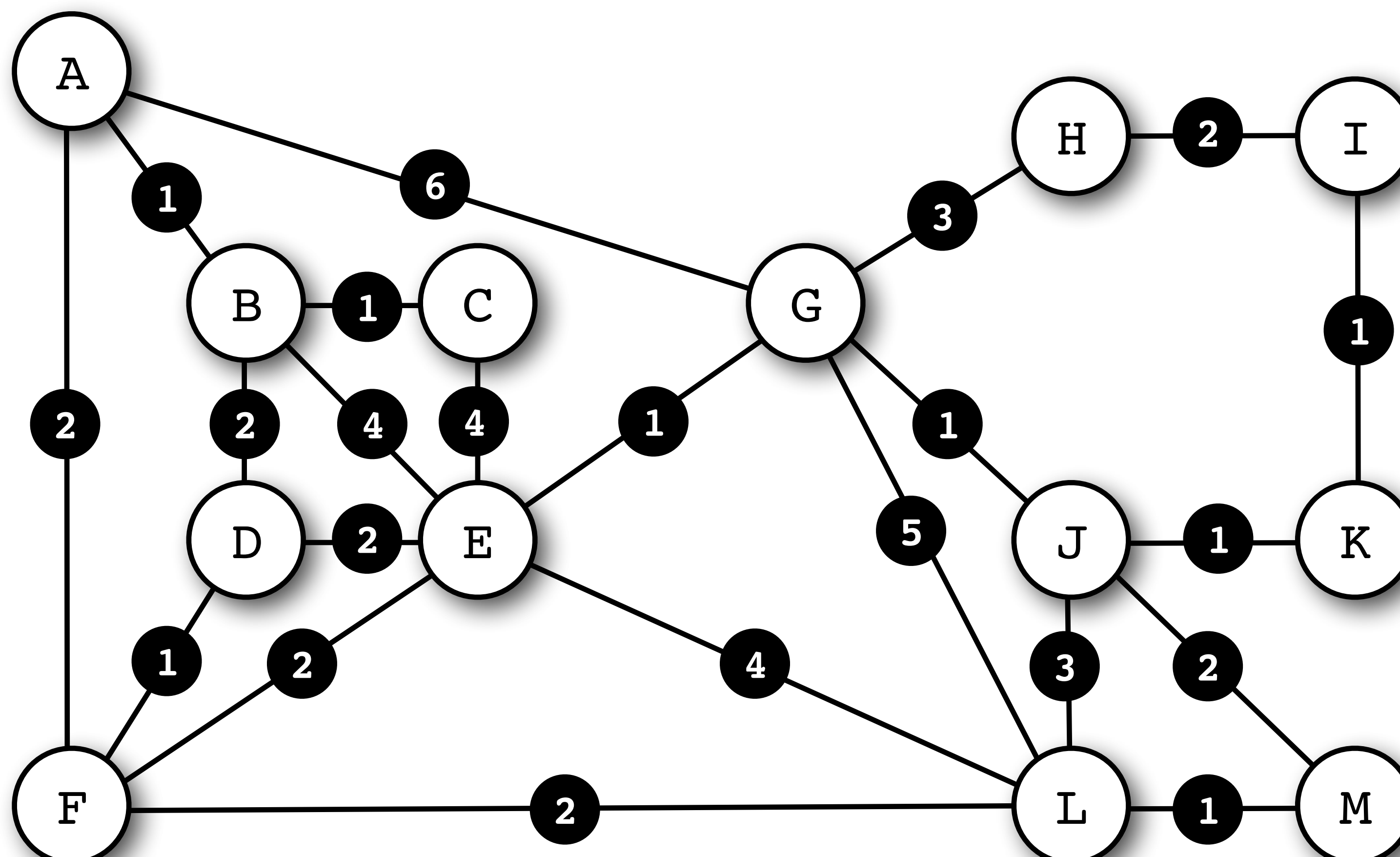
Algorithme de Kruskal

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.
- On considère tous les arêtes par ordre croissant de poid. Et on décide à chaque étape si l’arête appartient à l’ACM ou pas.
  - Si ajouter l’arête au sous-graphe noir crée un cycle noir, on ne le colorie pas.
  - Sinon on le colorie

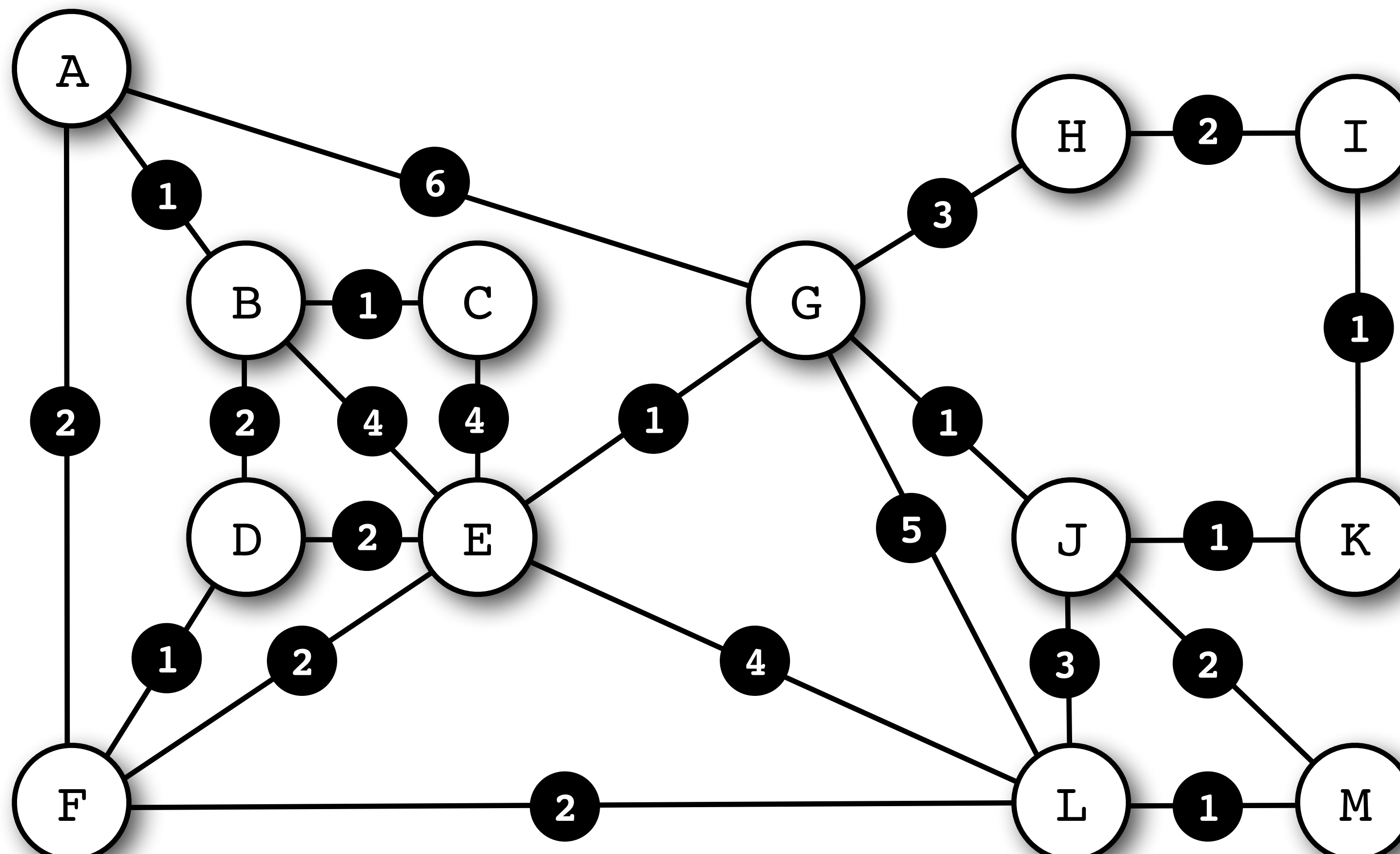
Algorithme de Kruskal

- On colorie toutes les arêtes de G en gris. Puis on va progressivement colorier en noir les arêtes de ACM.
- On considère tous les arêtes par ordre croissant de poid. Et on décide à chaque étape si l'arête appartient à l'ACM ou pas.
    - Si ajouter l'arête au sous-graphe noir crée un cycle noir, on ne le colorie pas.
    - Sinon on le colorie
- On répète jusqu'à avoir colorié S-1 arêtes en noir.

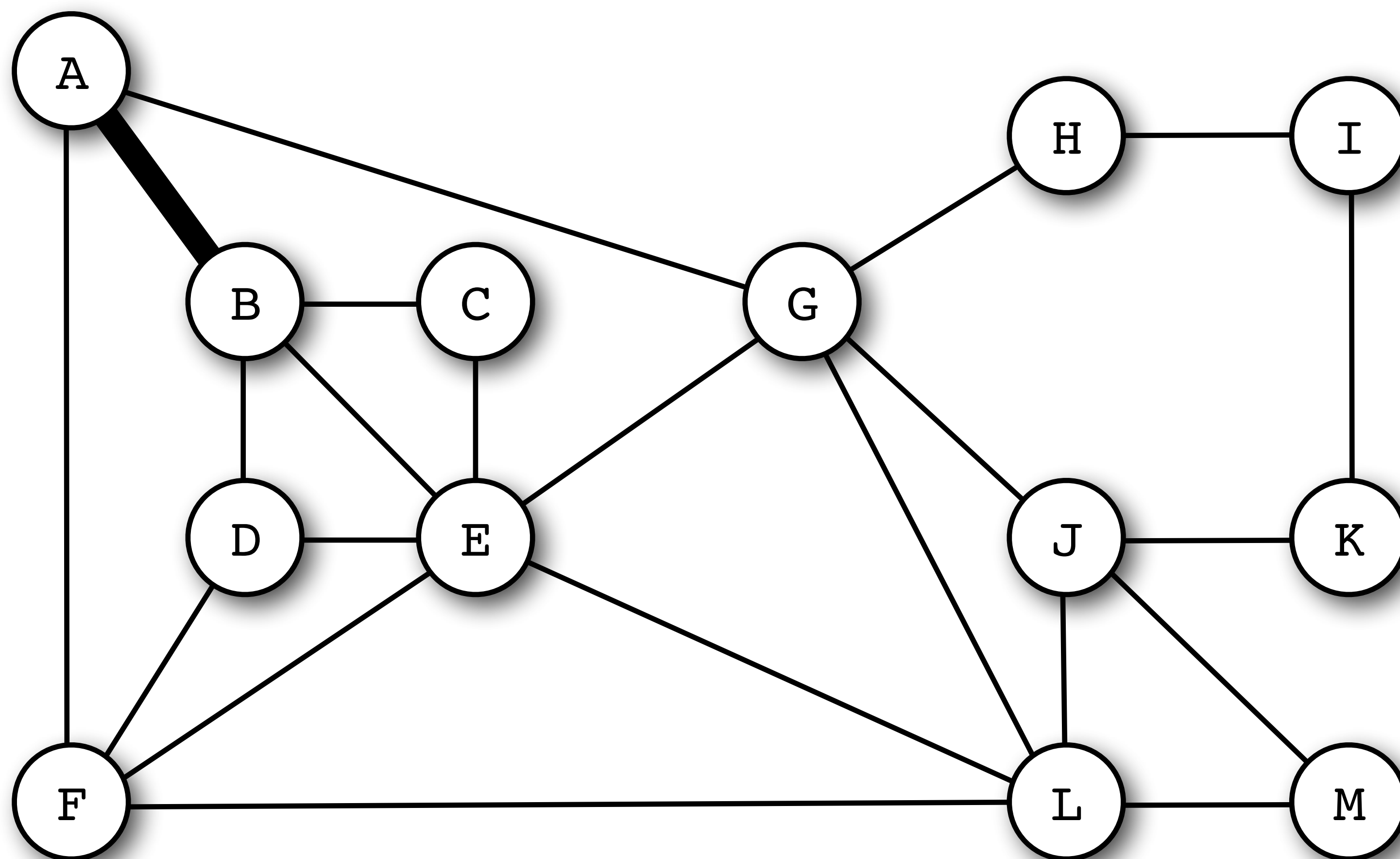

A
B
C
D
E
F
G
H
I
J
K
L
M


A
B
C
D
E
F
G
H
I
J
K
L
M
1
6
2
1
2
4
4
1
3
2
1
2
1
1
5
1
1
2
4
3
2
2
1

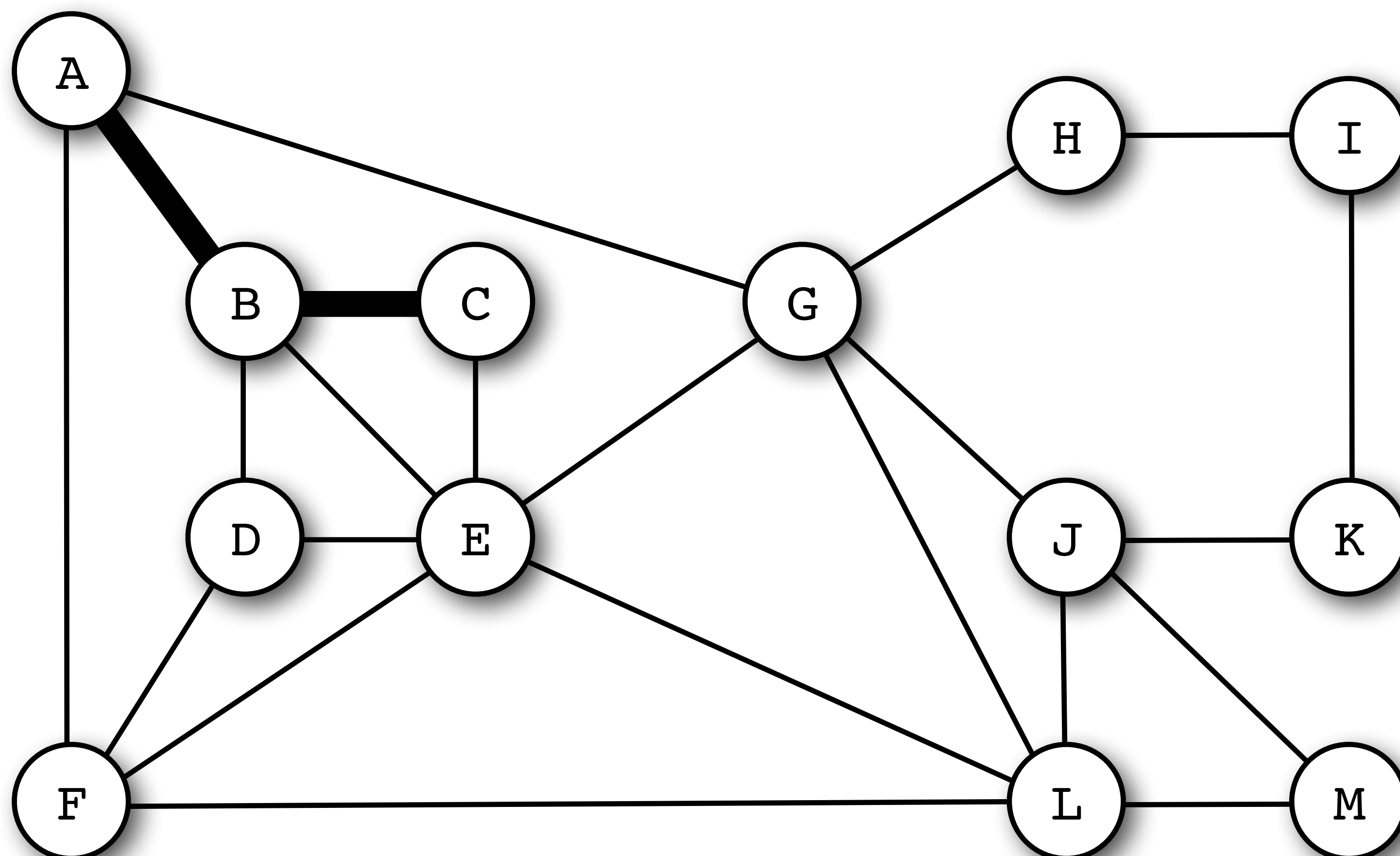
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



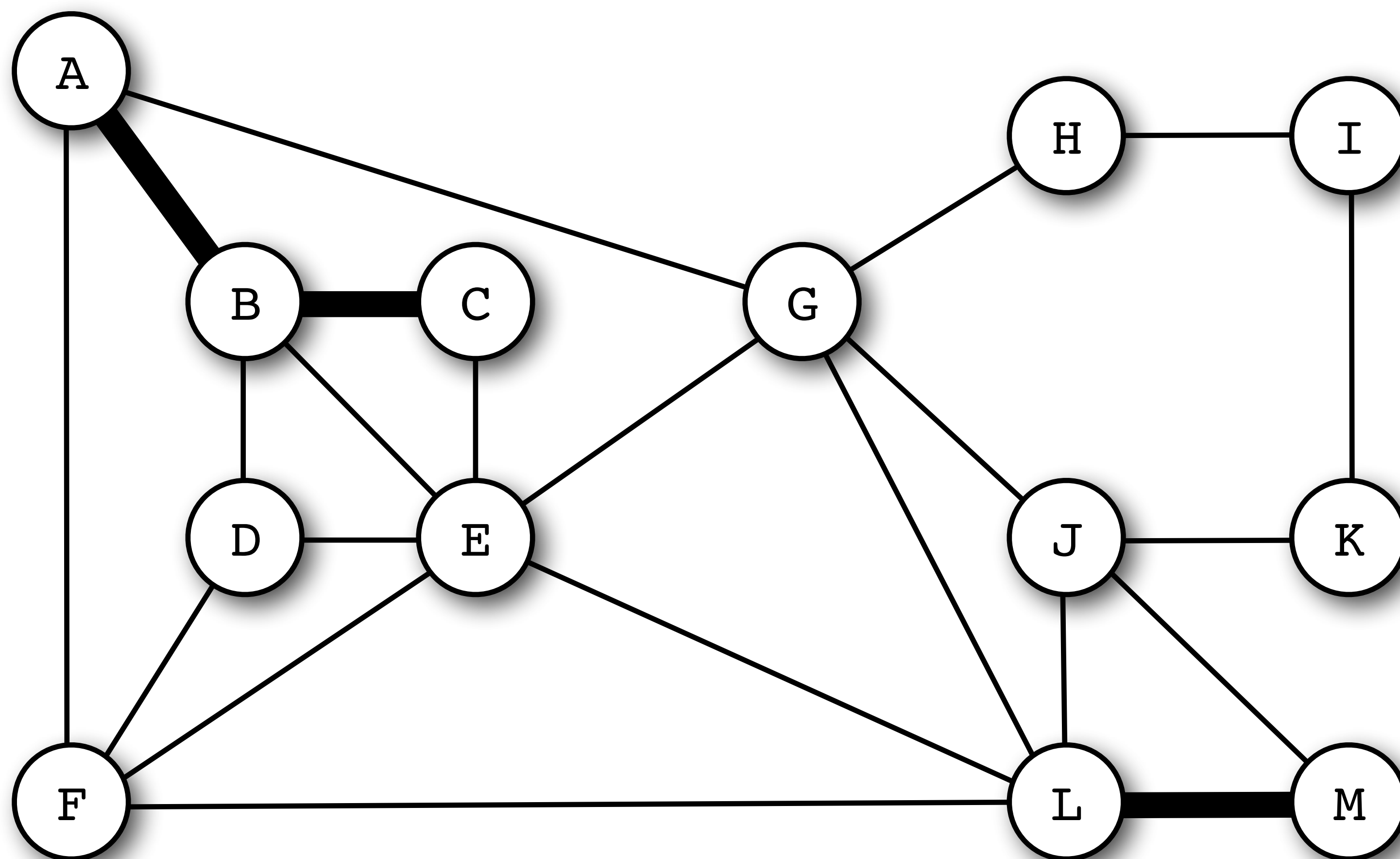
A-B 1
B-C 1
L-M 1
J-K 1
F-D 1
E-G 1
G-J 1
I-K 1
J-M 2
D-E 2
F-E 2
B-D 2
A-F 2
H-I 2
F-L 2
G-H 3
J-L 3
E-L 4
B-E 4
C-E 4
G-L 5
A-G 6



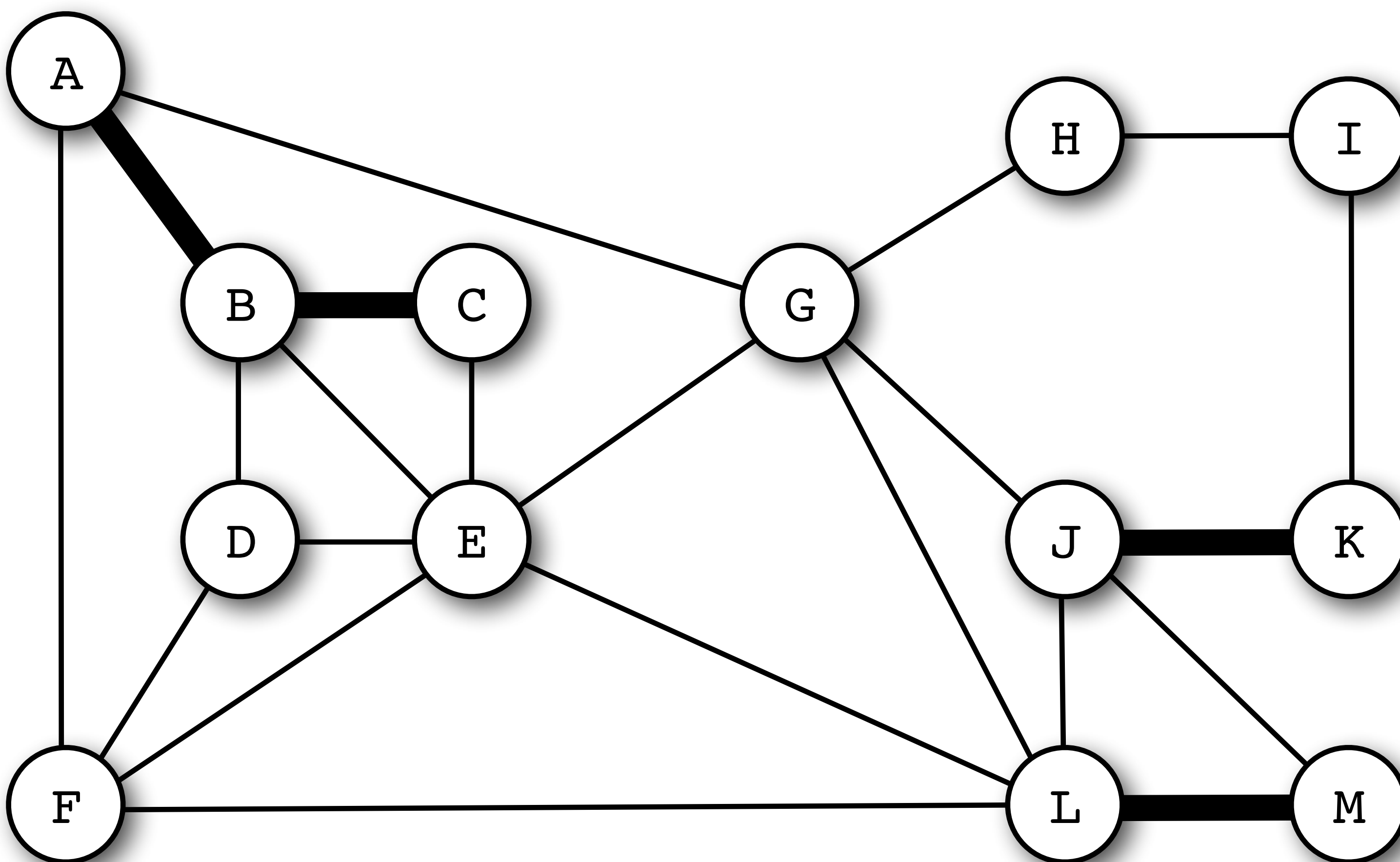
A-B 1
B-C 1
L-M 1
J-K 1
F-D 1
E-G 1
G-J 1
I-K 1
J-M 2
D-E 2
F-E 2
B-D 2
A-F 2
H-I 2
F-L 2
G-H 3
J-L 3
E-L 4
B-E 4
C-E 4
G-L 5
A-G 6



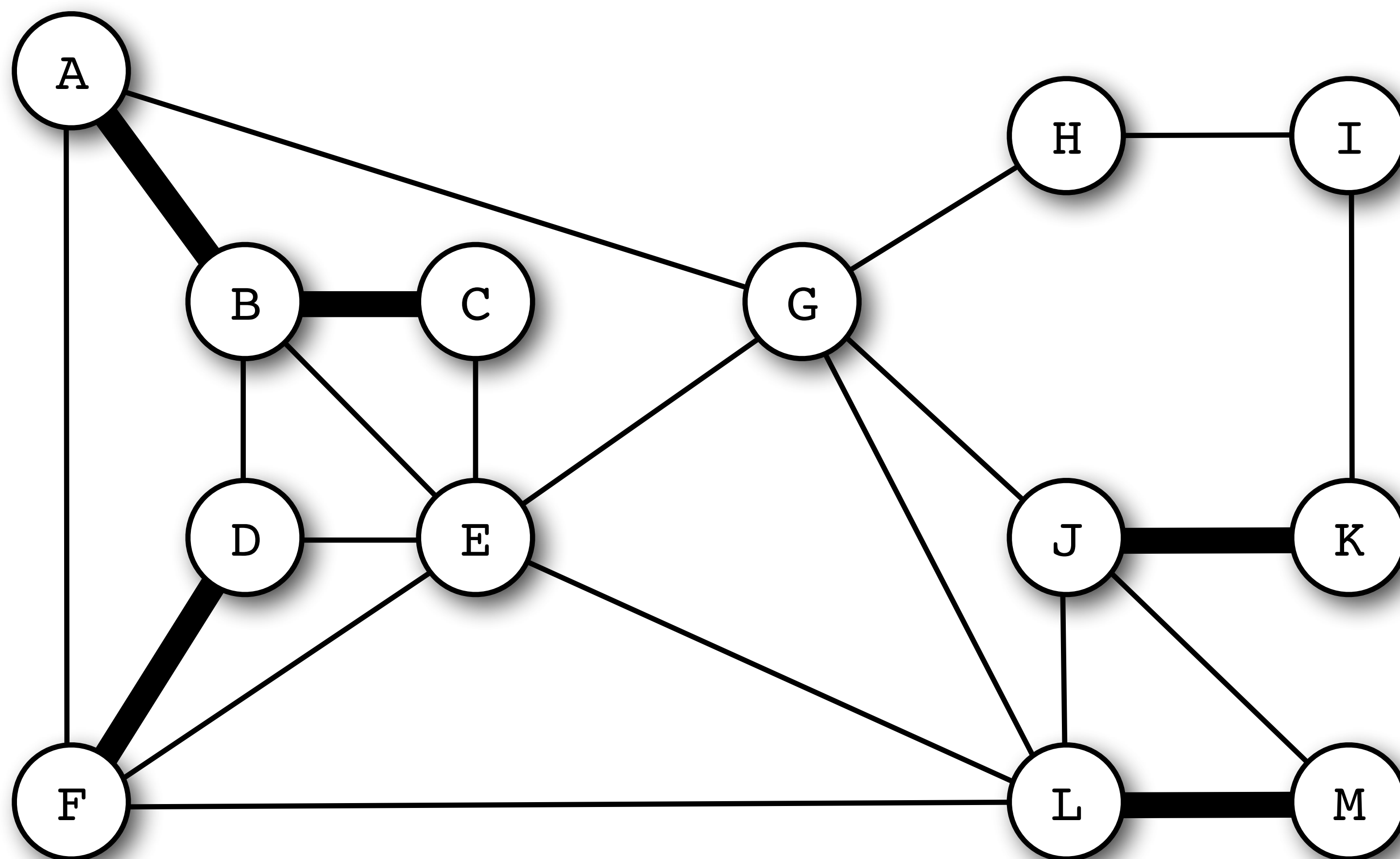
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



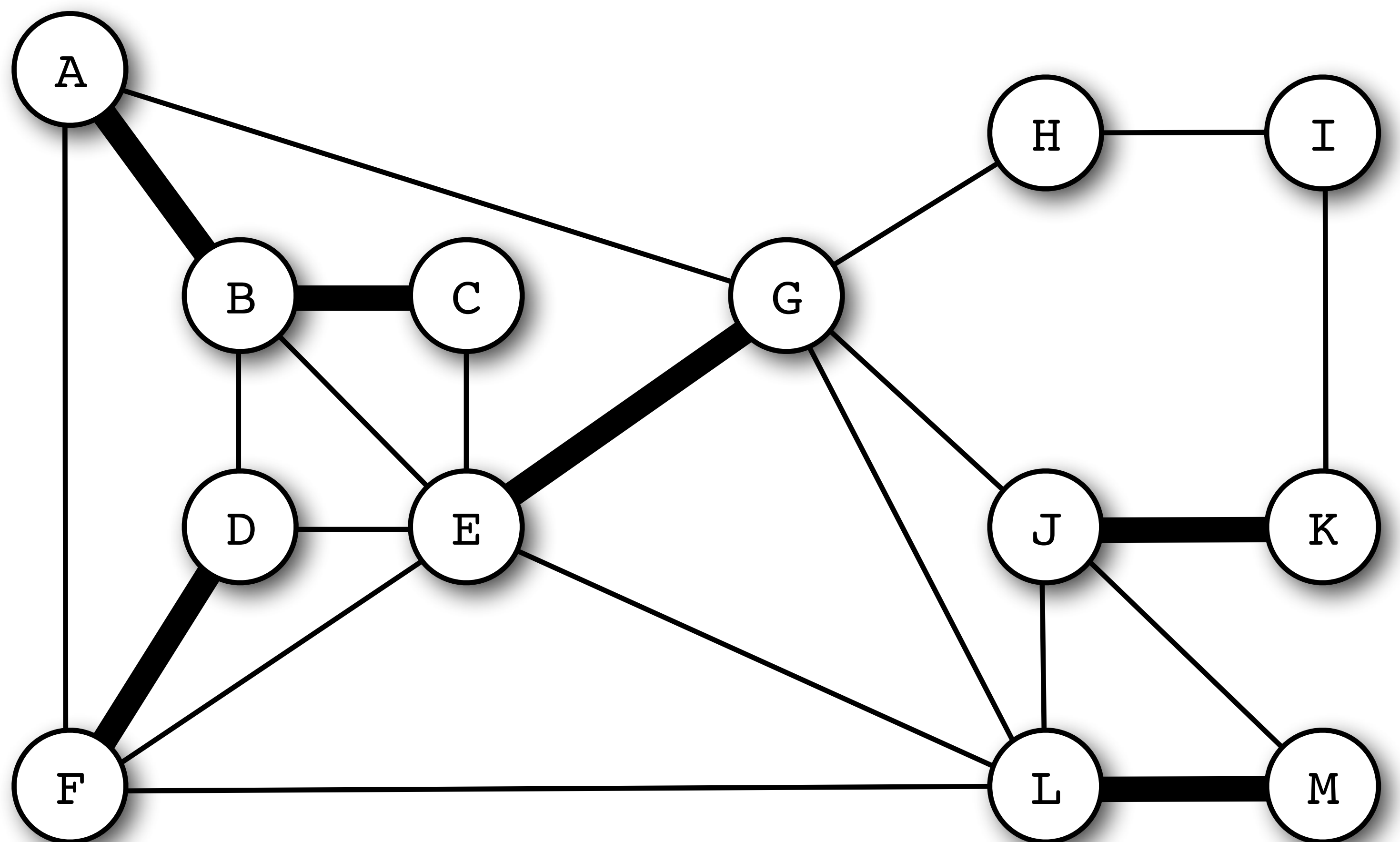
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



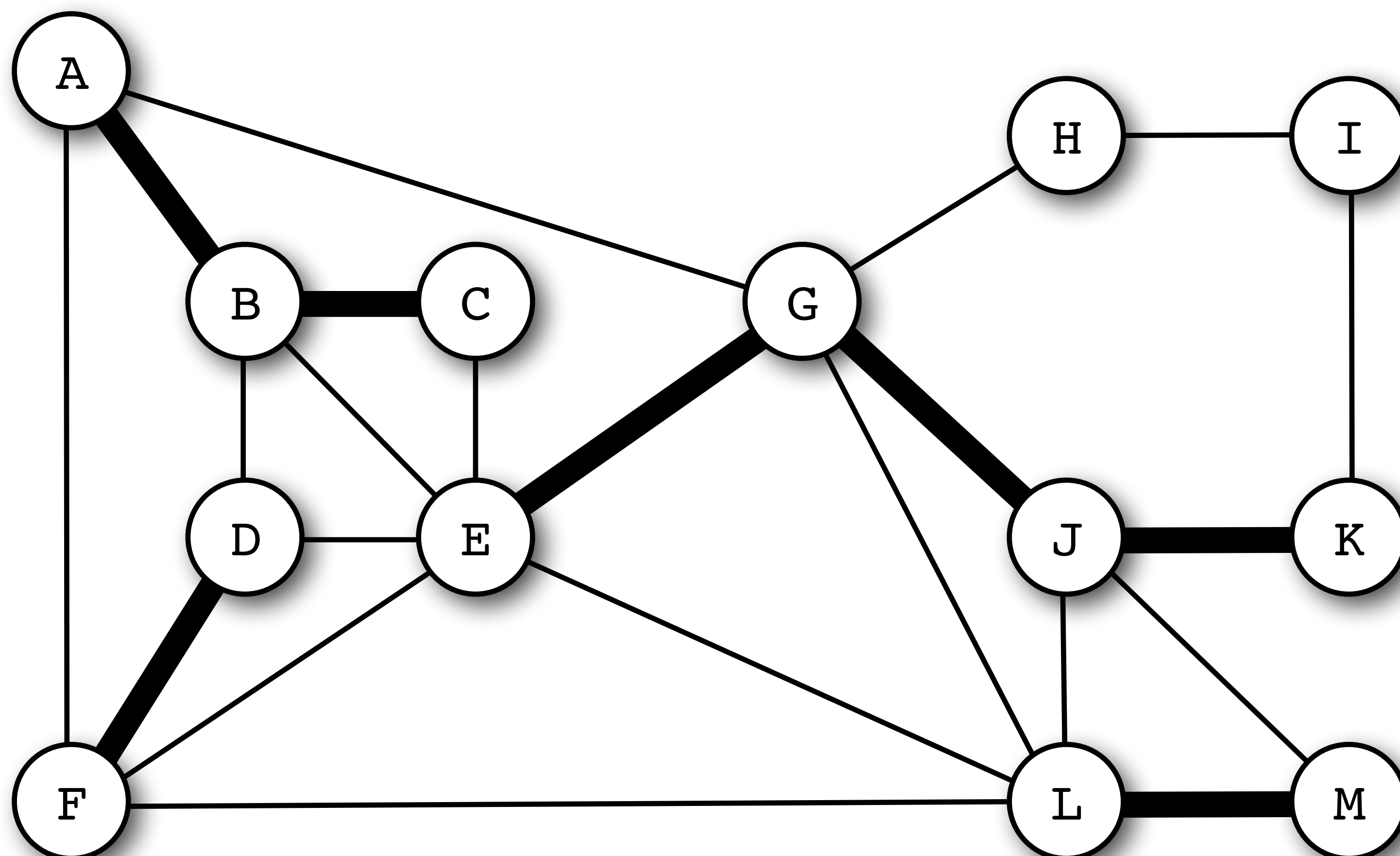
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



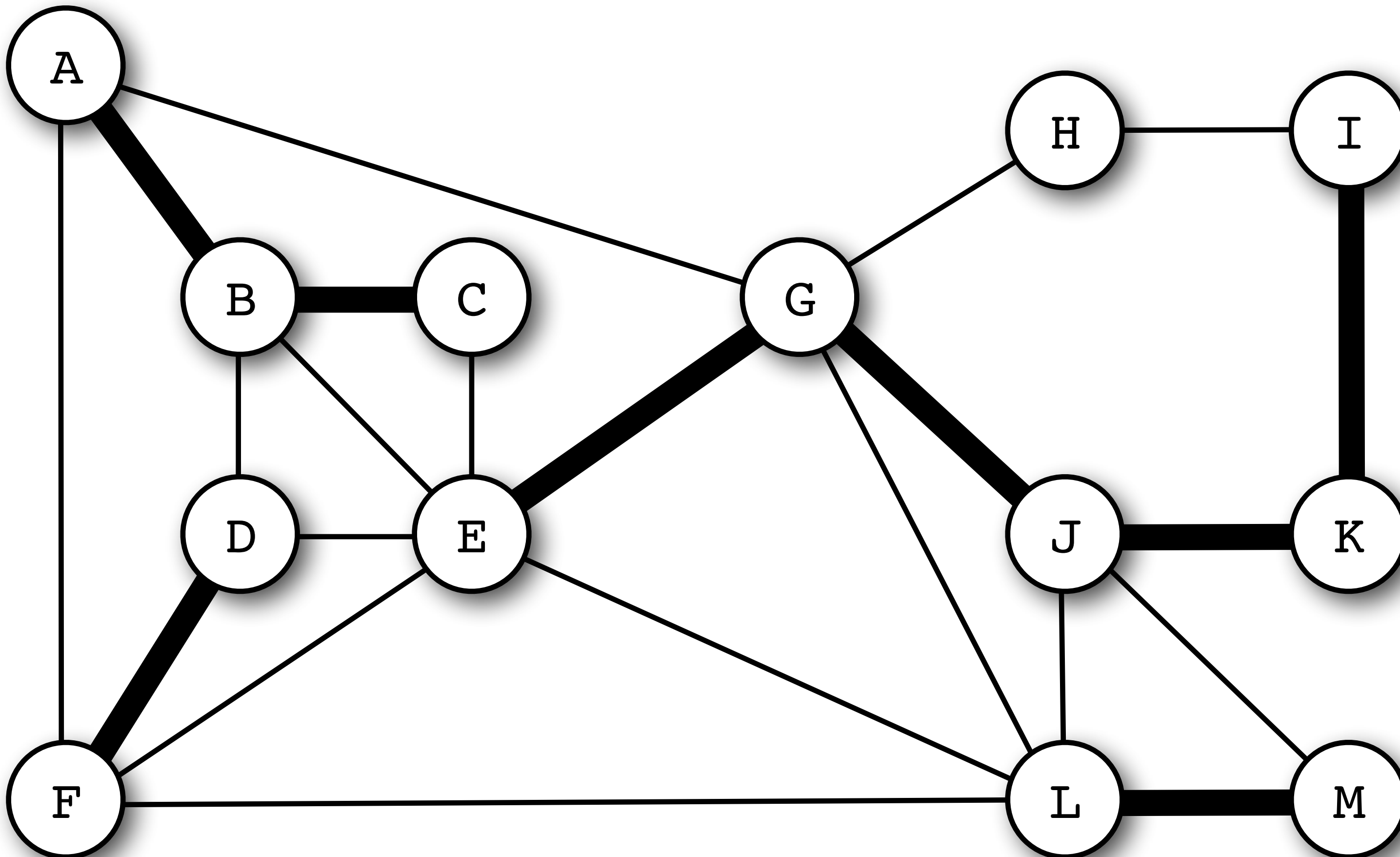
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



A-B 1
B-C 1
L-M 1
J-K 1
F-D 1
E-G 1
G-J 1
I-K 1
J-M 2
D-E 2
F-E 2
B-D 2
A-F 2
H-I 2
F-L 2
G-H 3
J-L 3
E-L 4
B-E 4
C-E 4
G-L 5
A-G 6



A–B 1

B–C 1

L–M 1

J–K 1

F–D 1

E–G 1

G–J 1

I–K 1

J–M 2

D–E 2

F–E 2

B–D 2

A–F 2

H–I 2

F–L 2

G–H 3

J–L 3

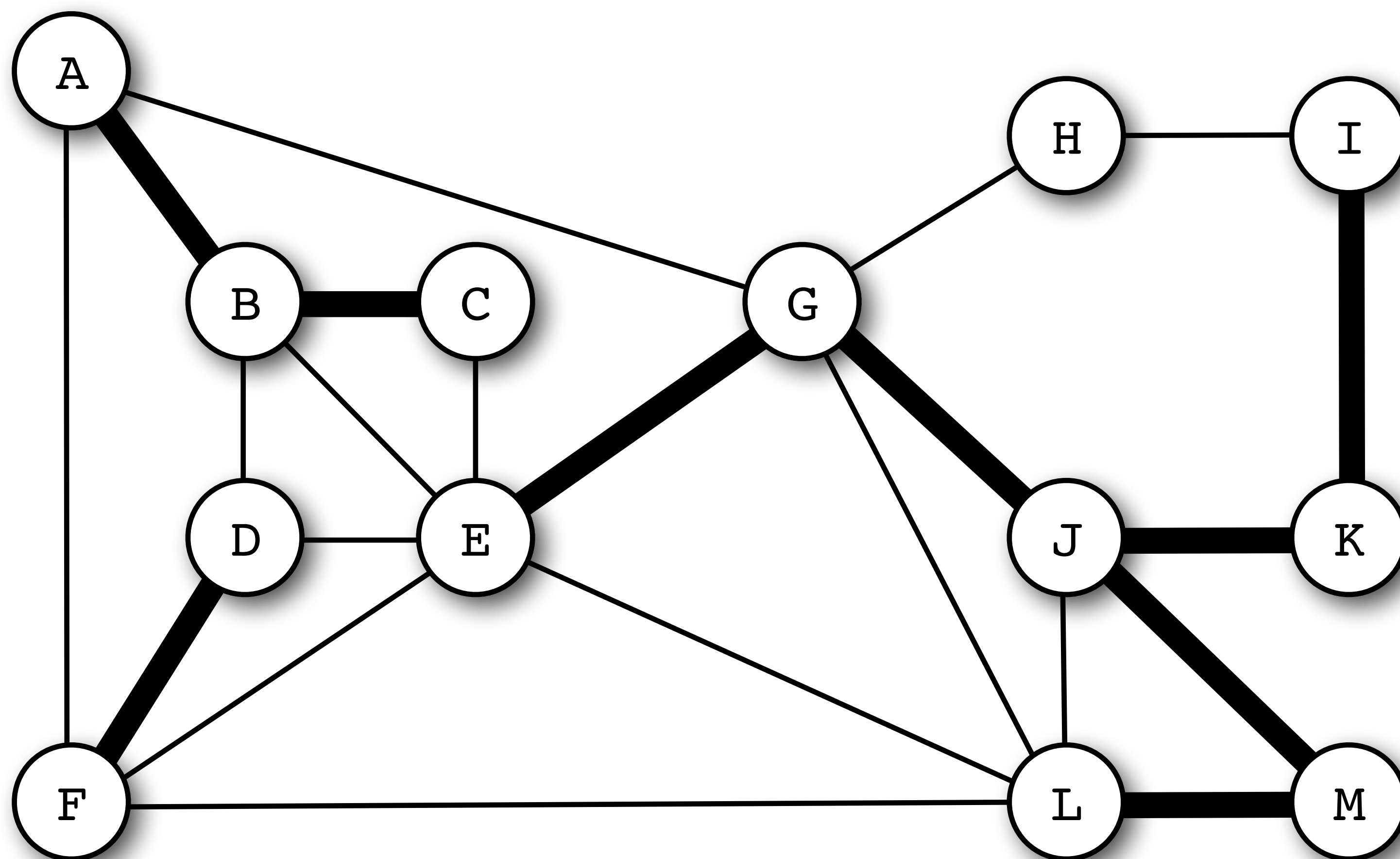
E–L 4

B–E 4

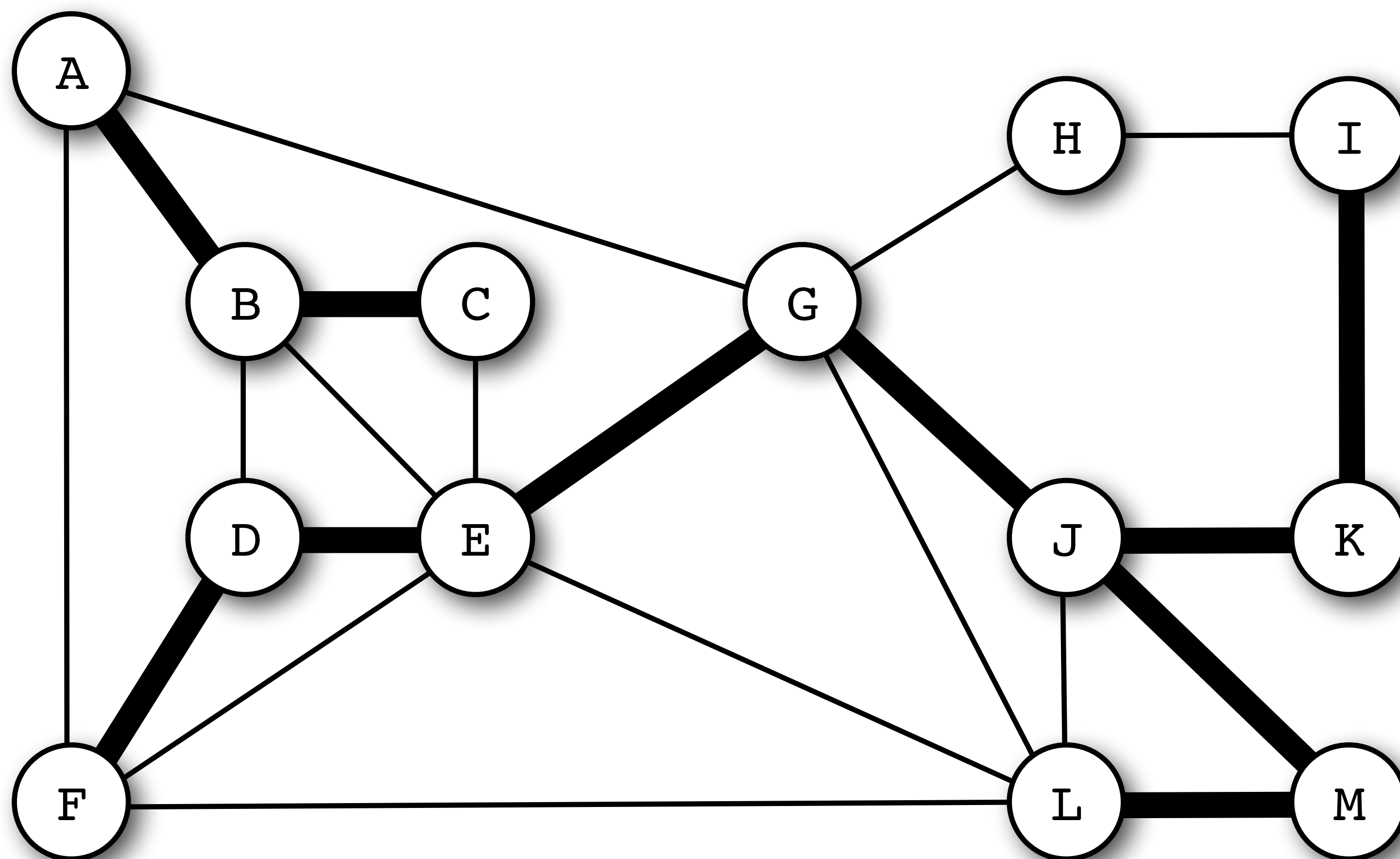
C–E 4

G–L 5

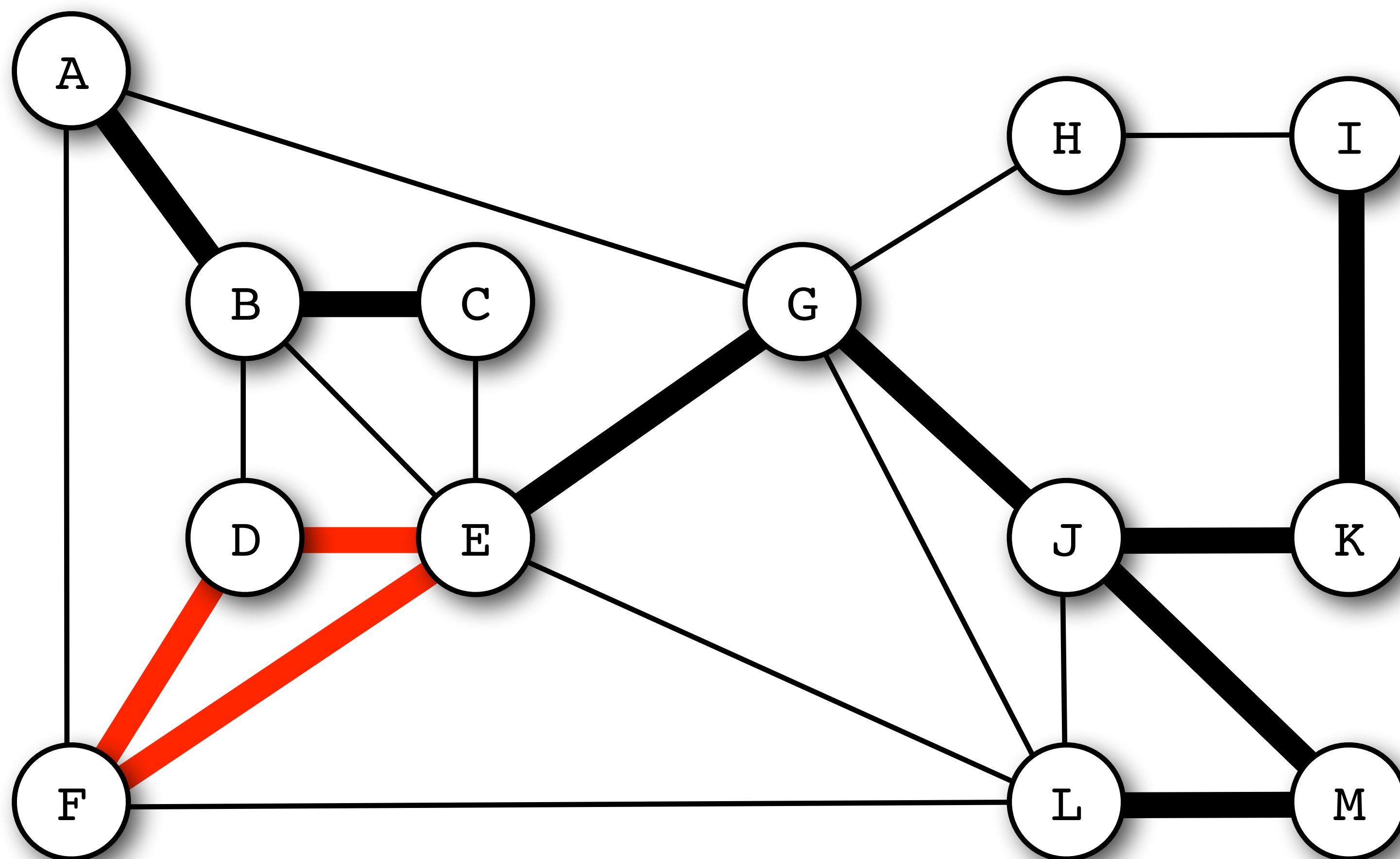
A–G 6



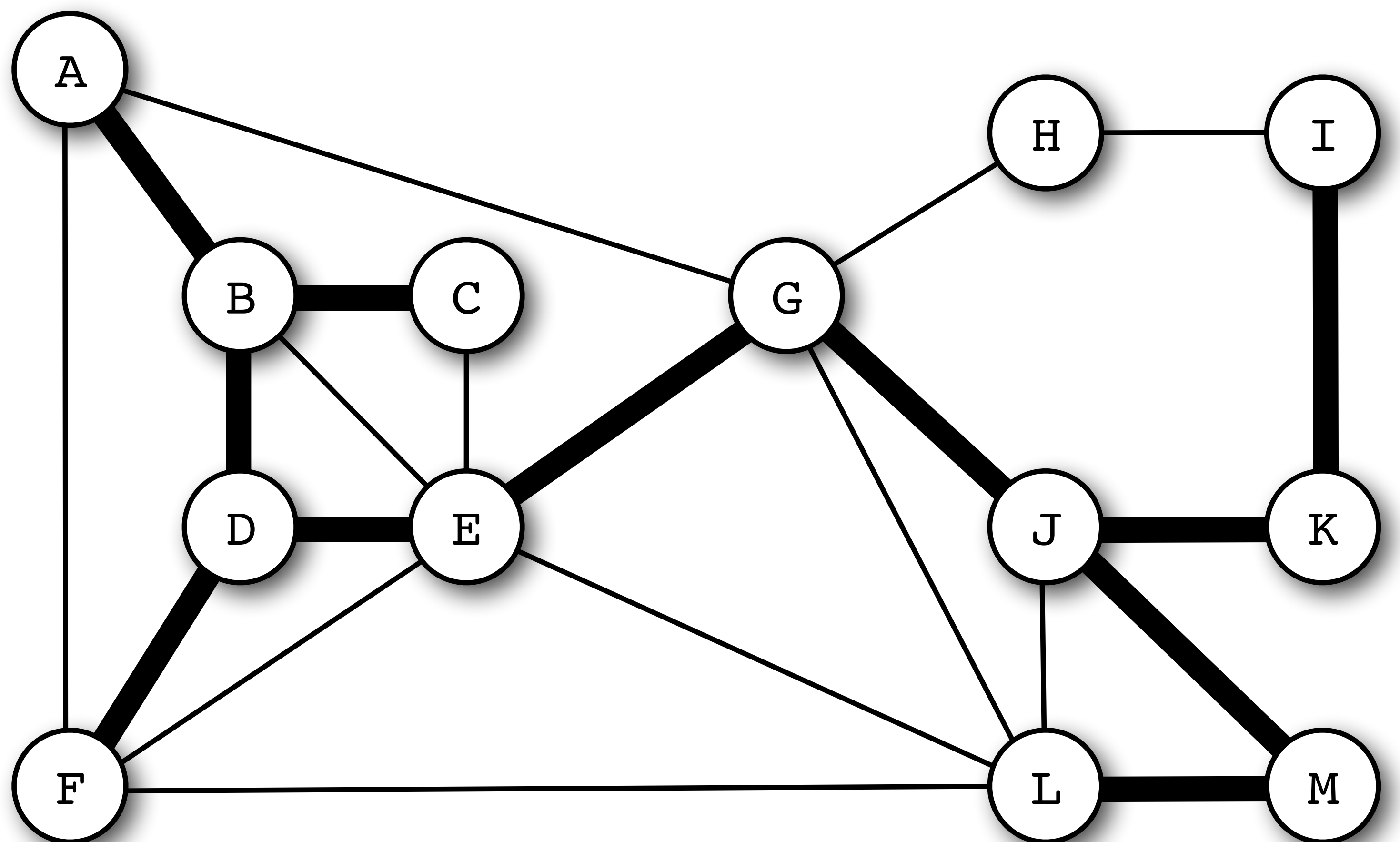
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



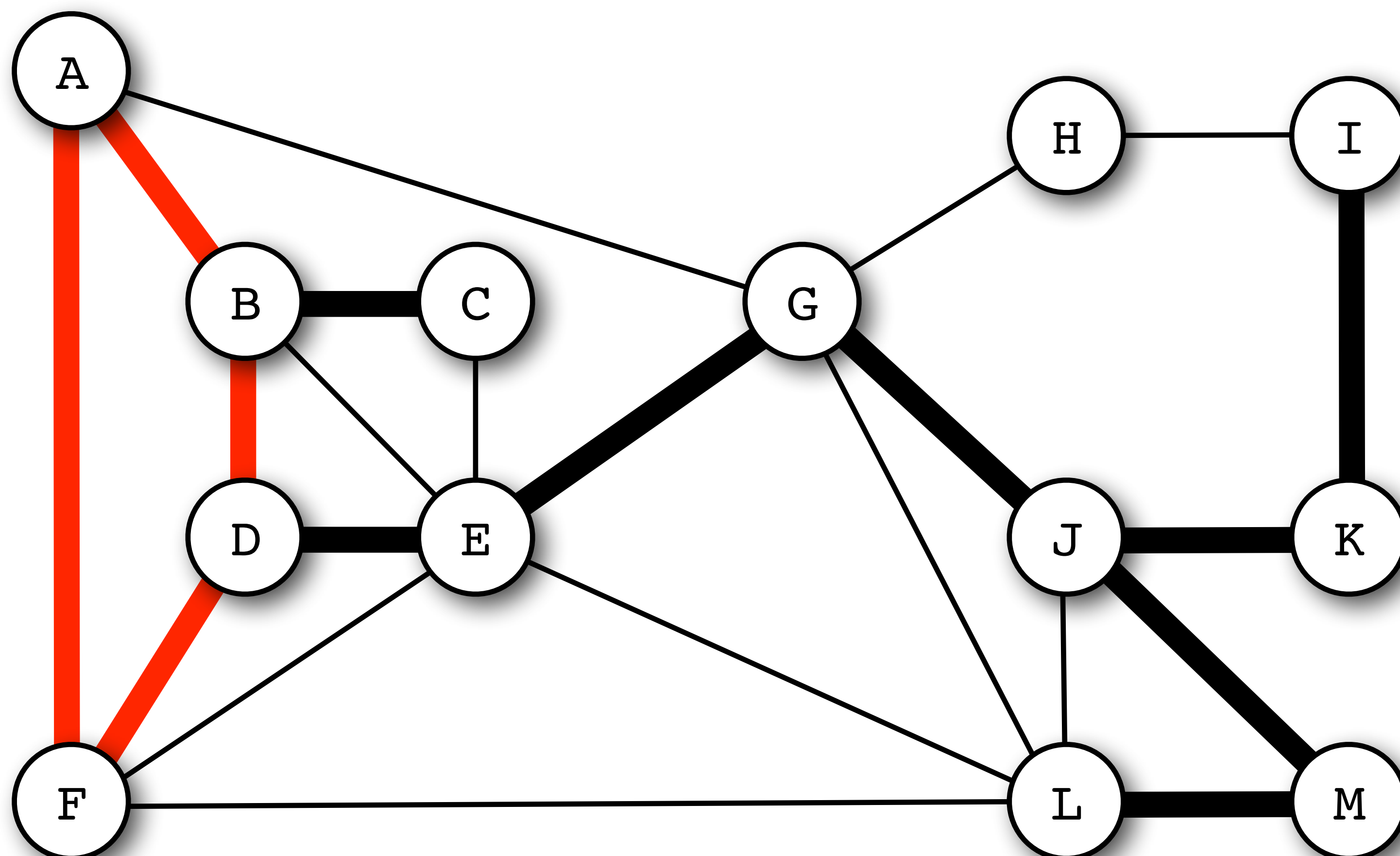
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



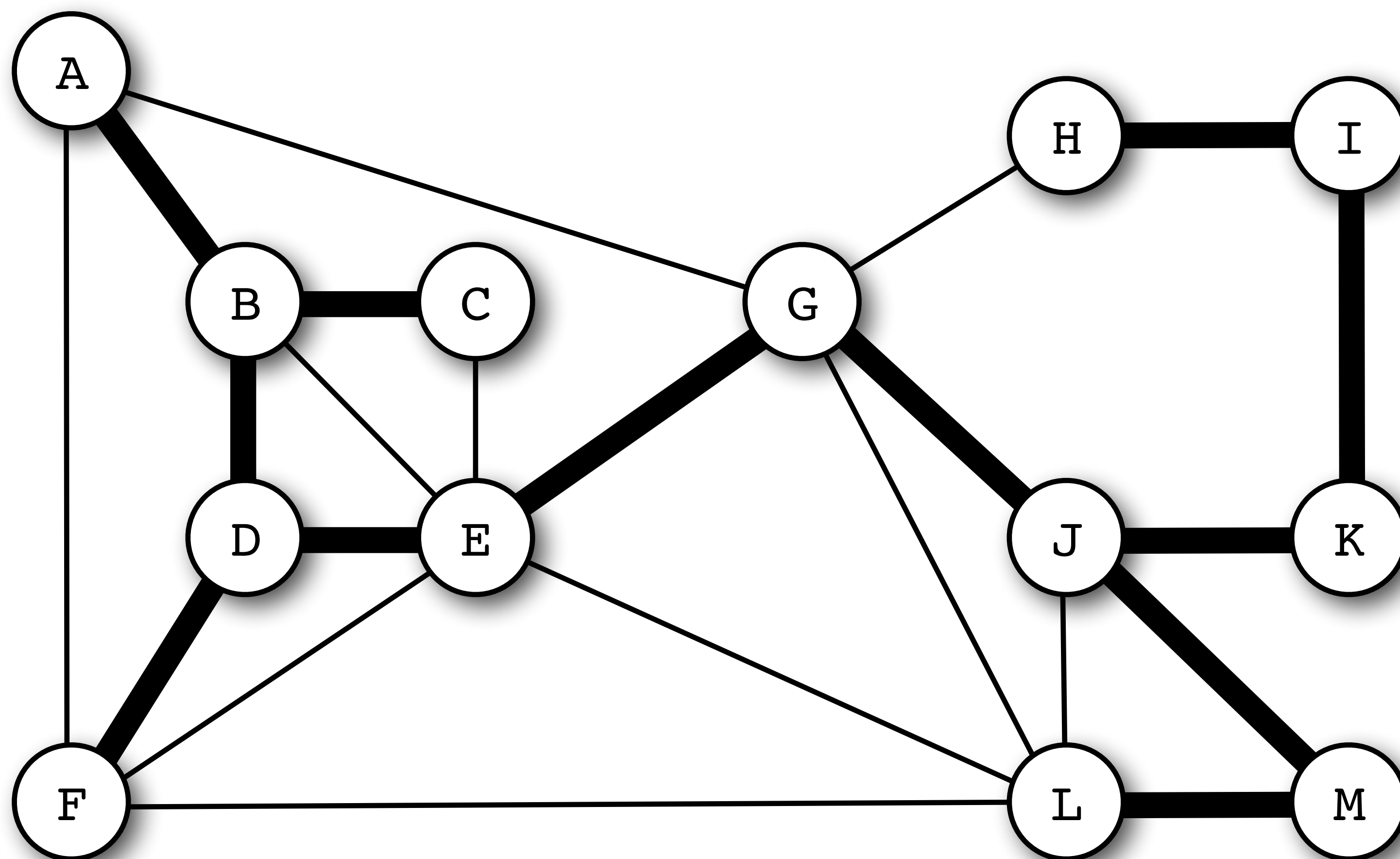
A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



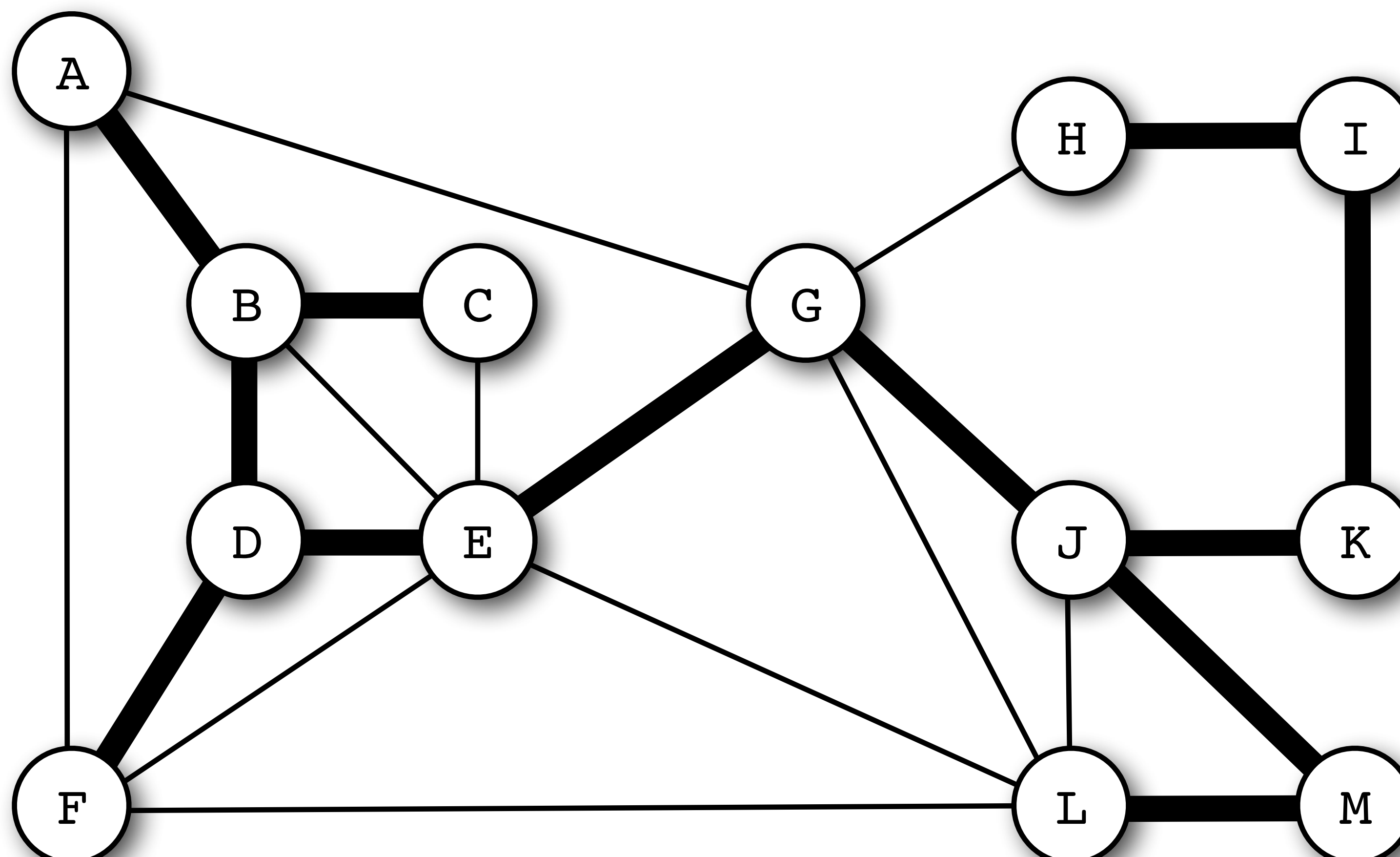
A-B 1
B-C 1
L-M 1
J-K 1
F-D 1
E-G 1
G-J 1
I-K 1
J-M 2
D-E 2
F-E 2
B-D 2
A-F 2
H-I 2
F-L 2
G-H 3
J-L 3
E-L 4
B-E 4
C-E 4
G-L 5
A-G 6



A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



A–B 1
B–C 1
L–M 1
J–K 1
F–D 1
E–G 1
G–J 1
I–K 1
J–M 2
D–E 2
F–E 2
B–D 2
A–F 2
H–I 2
F–L 2
G–H 3
J–L 3
E–L 4
B–E 4
C–E 4
G–L 5
A–G 6



Algorithme de Kruskal

Correction

Algorithme de Kruskal

Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM

Algorithme de Kruskal
Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)

Algorithme de Kruskal

Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)
  - la propriété est maintenue à chaque ajout d'arête

Algorithme de Kruskal

Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)
  - la propriété est maintenue à chaque ajout d'arête
    - appelons (a,b) cette arête

Algorithme de Kruskal
Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
    - la propriété est vraie au départ (sous-graphe vide)
    - la propriété est maintenue à chaque ajout d'arête
        - appelons (a,b) cette arête
        - considérons la coupure formée par la composante connexe de a (pour le sous-graphe noir) et son complémentaire

Algorithme de Kruskal
Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)
  - la propriété est maintenue à chaque ajout d'arête
    - appelons (a,b) cette arête
    - considérons la coupure formée par la composante connexe de a (pour le sous-graphe noir) et son complémentaire
    - il n'existe pas d'arête traversante noire (sinon elle connecterait un sommet de la composante connexe de a avec un sommet qui n'est pas censé être connecté à a).

Algorithme de Kruskal
Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)
  - la propriété est maintenue à chaque ajout d'arête
    - appelons (a,b) cette arête
    - considérons la coupure formée par la composante connexe de a (pour le sous-graphe noir) et son complémentaire
    - il n'existe pas d'arête traversante noire (sinon elle connecterait un sommet de la composante connexe de a avec un sommet qui n'est pas censé être connecté à a).
    - c'est l'arête traversante minimale car les arêtes plus petites sont noires, ou ont été écartées parce qu'elle n'appartiennent pas à l'ACM

Algorithme de Kruskal

Correction

- A chaque étape, le sous-graphe noir est un sous-graphe de l'ACM
  - la propriété est vraie au départ (sous-graphe vide)
  - la propriété est maintenue à chaque ajout d'arête
    - appelons (a,b) cette arête
    - considérons la coupure formée par la composante connexe de a (pour le sous-graphe noir) et son complémentaire
    - il n'existe pas d'arête traversante noire (sinon elle connecterait un sommet de la composante connexe de a avec un sommet qui n'est pas censé être connecté à a).
    - c'est l'arête traversante minimale car les arêtes plus petites sont noires, ou ont été écartées parce qu'elle n'appartiennent pas à l'ACM
- Les arêtes écartées ne risquent pas de manquer à l'ACM final car celui ci est acyclique

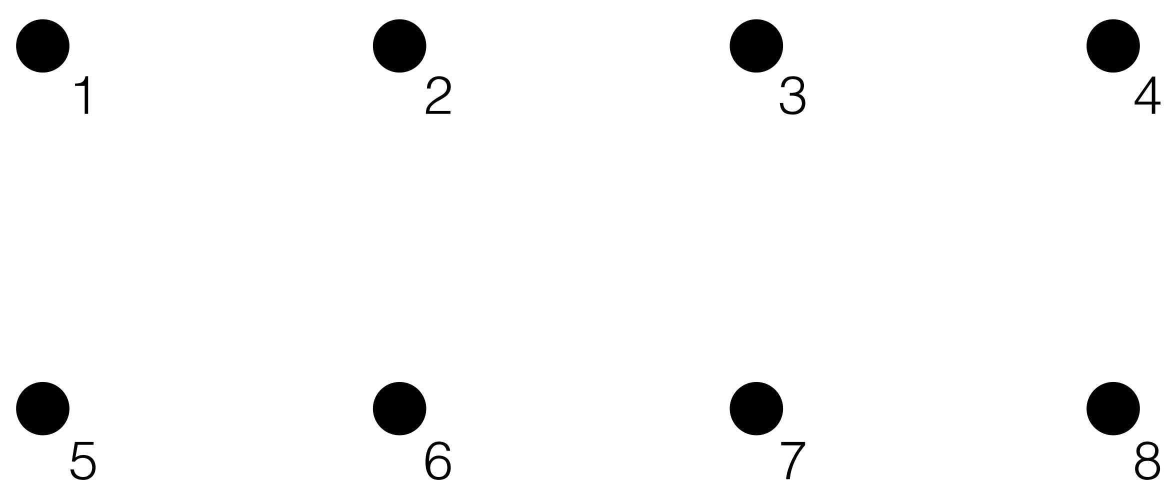
Algorithme de Kruskal

Implémentation

```
KRUSKAL(S,A) =
   E <- creer_classes_equiv()
   ACM <- ∅
   pour tout (a,b) dans A (par poids croissant)
      si a et b non déjà équivalents dans E
      alors
         E.fusionne(a,b)
         ACM <- ACM u {(a,b)}
   retourne ACM
```

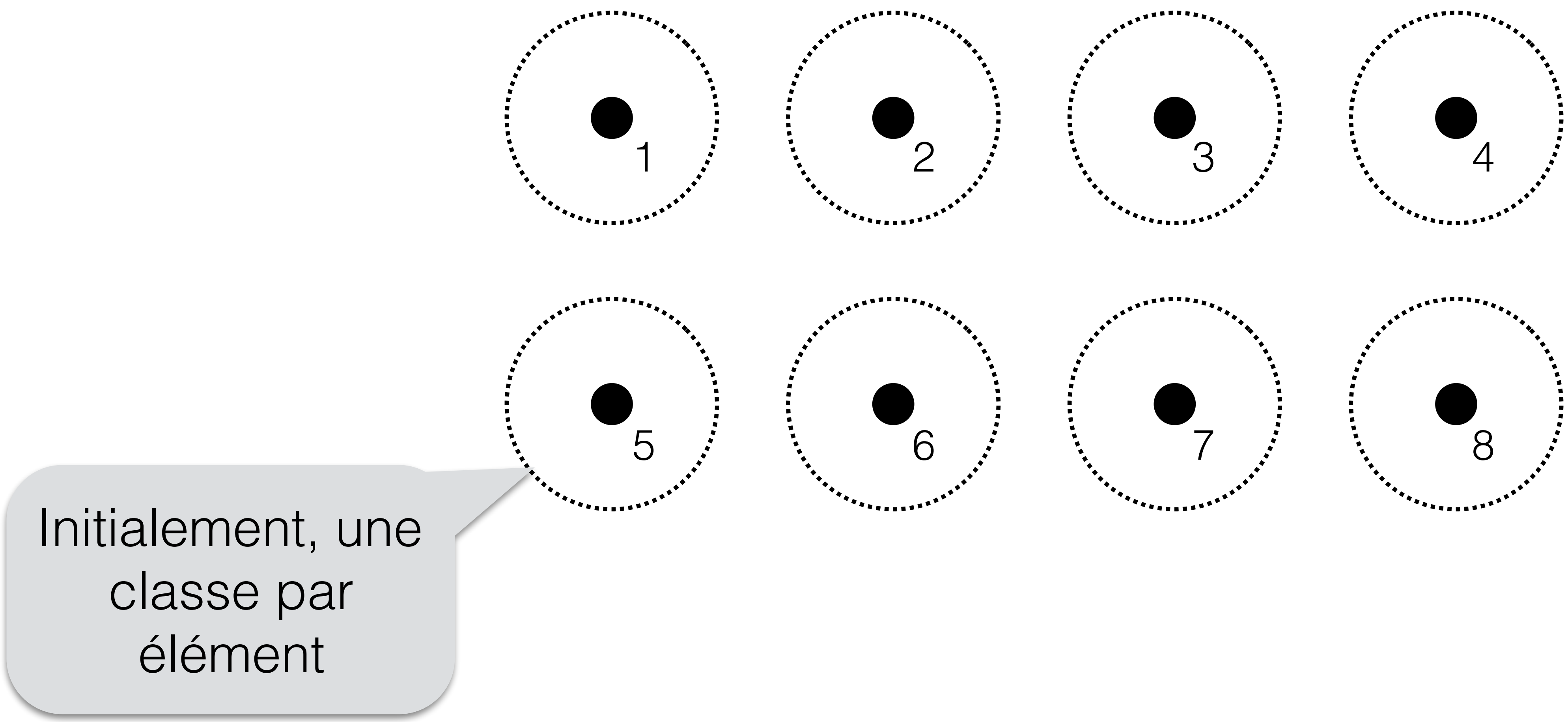
Structure de donnée

- Comment implémenter une structure de classe d'équivalence avec les opérations suivantes ?
  - initialisation
  - fusionner deux classes
  - tester si deux éléments sont dans la même classe

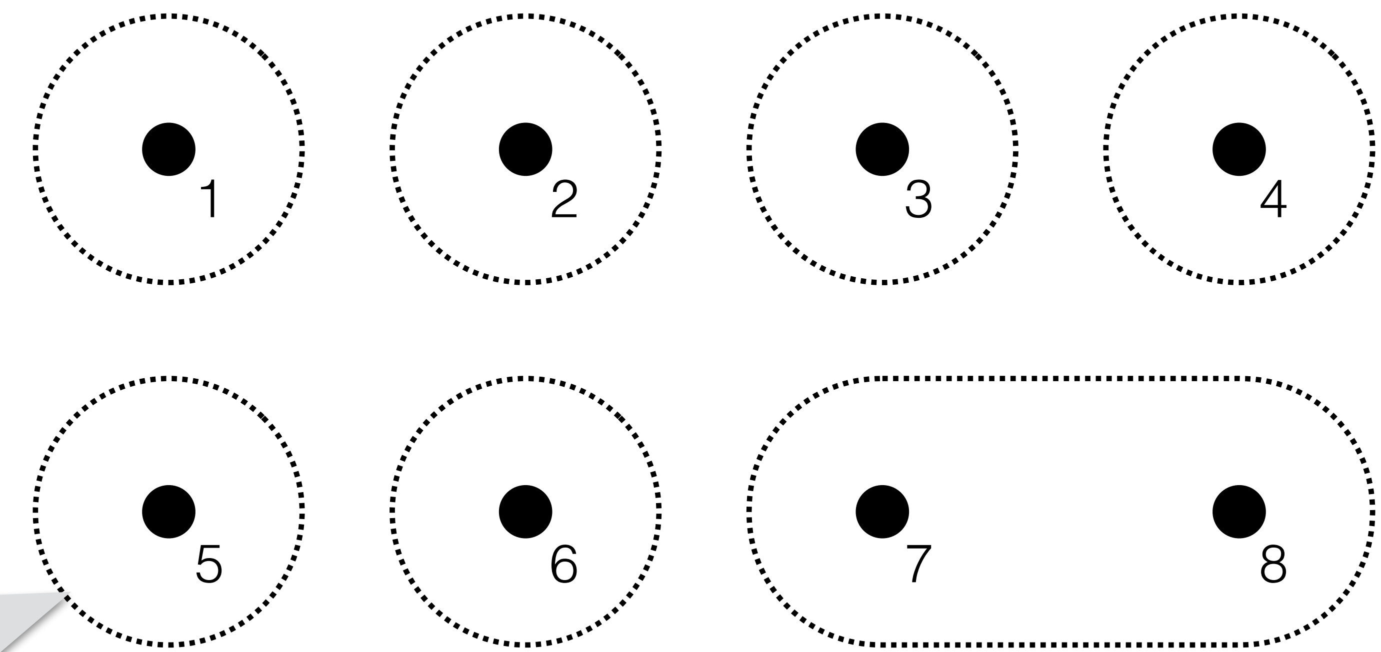


N éléments à manipuler

Initialisation

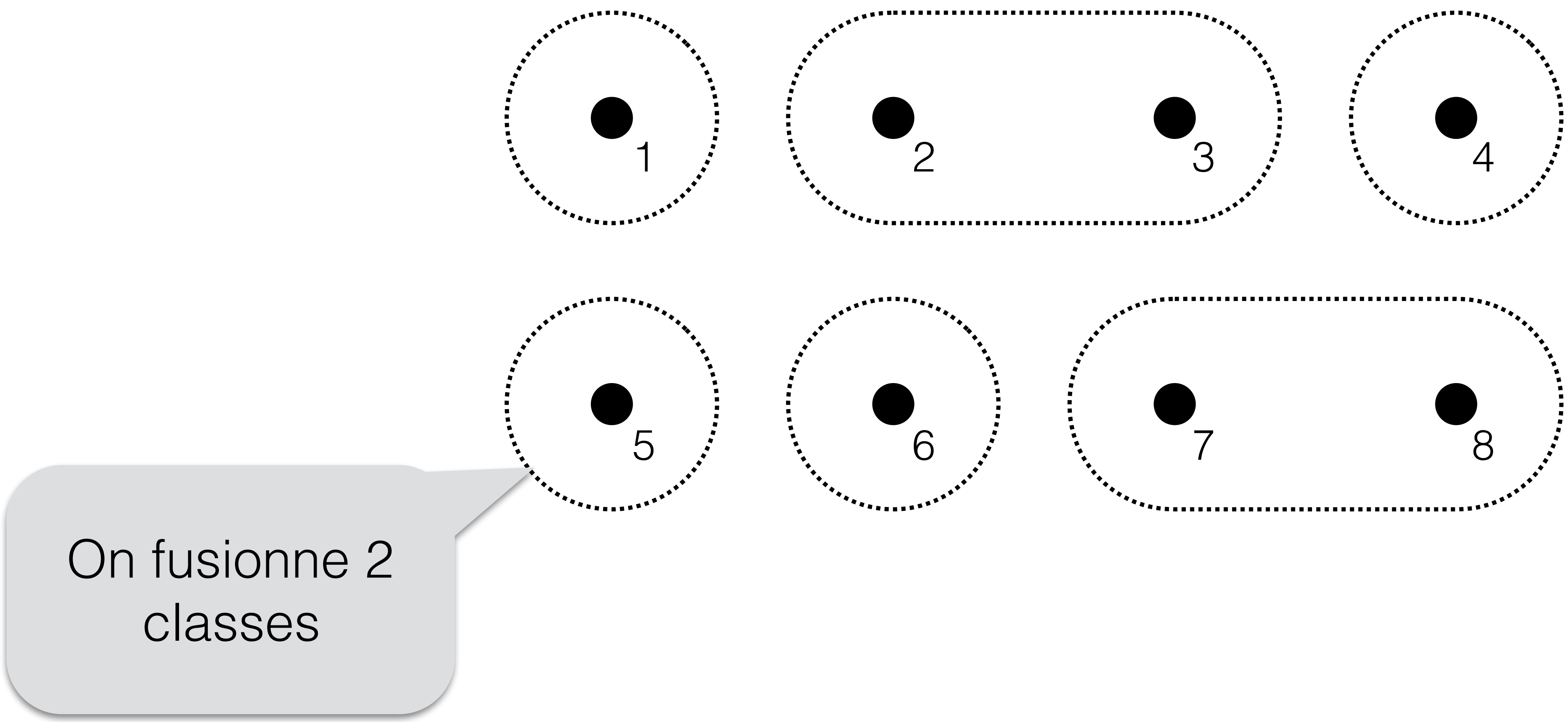


Fusion(7,8)

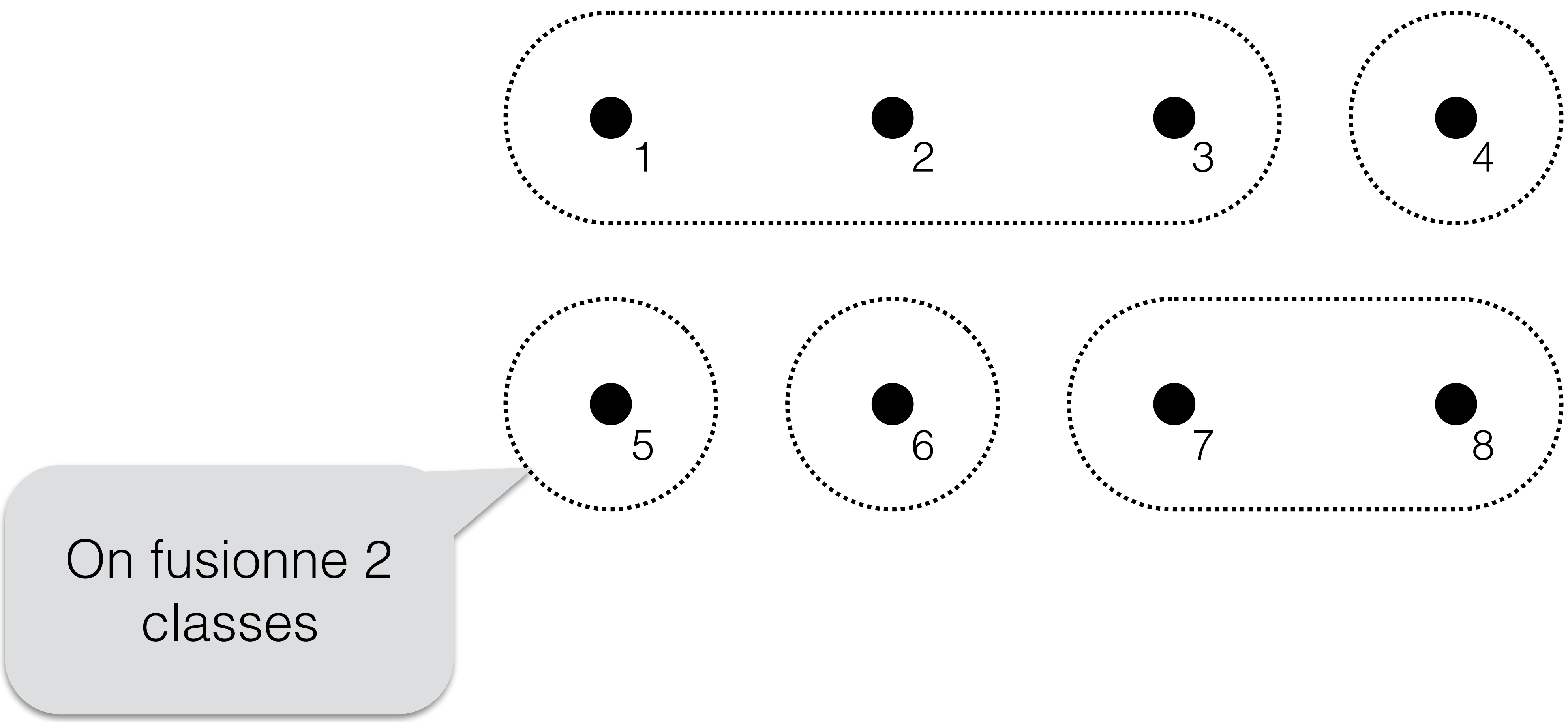


On fusionne 2 classes

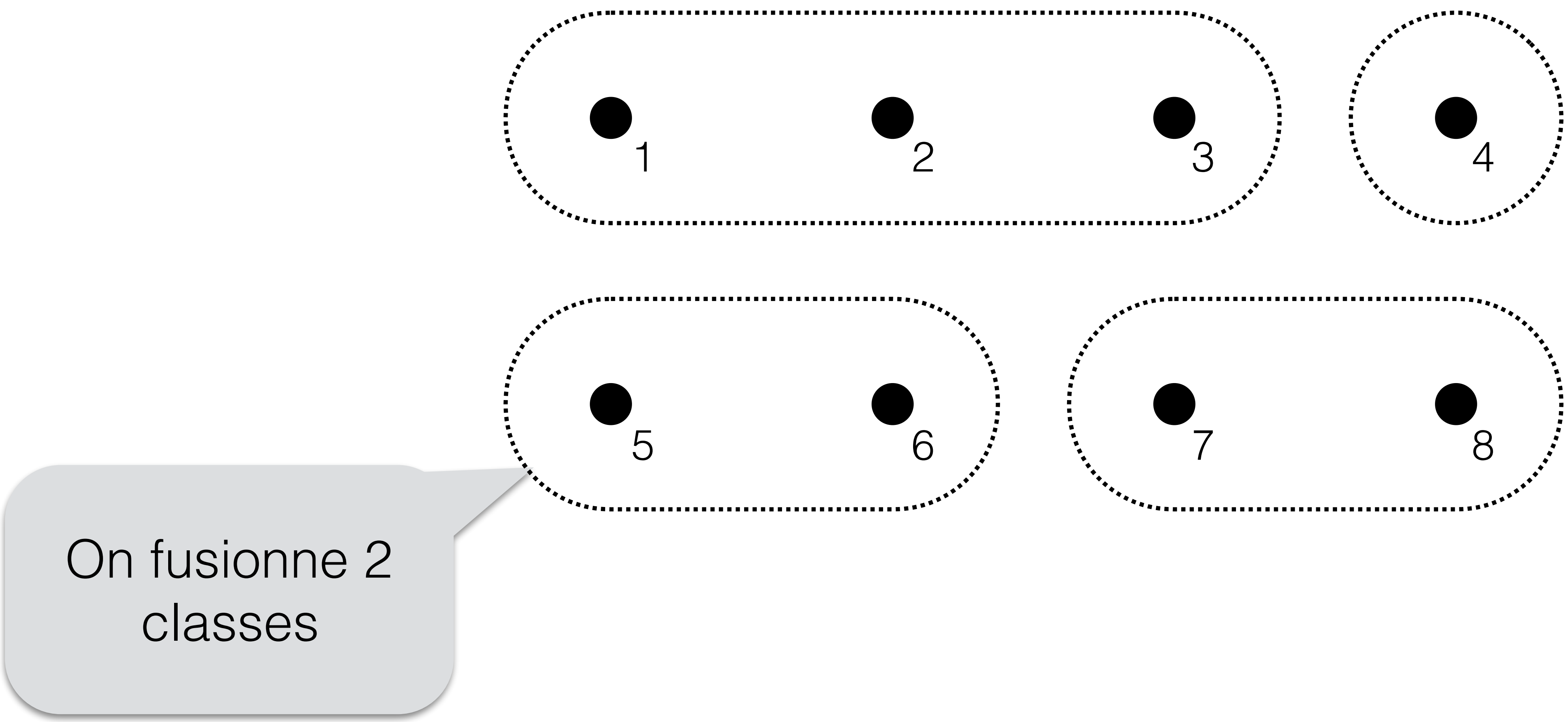
Fusion(2,3)



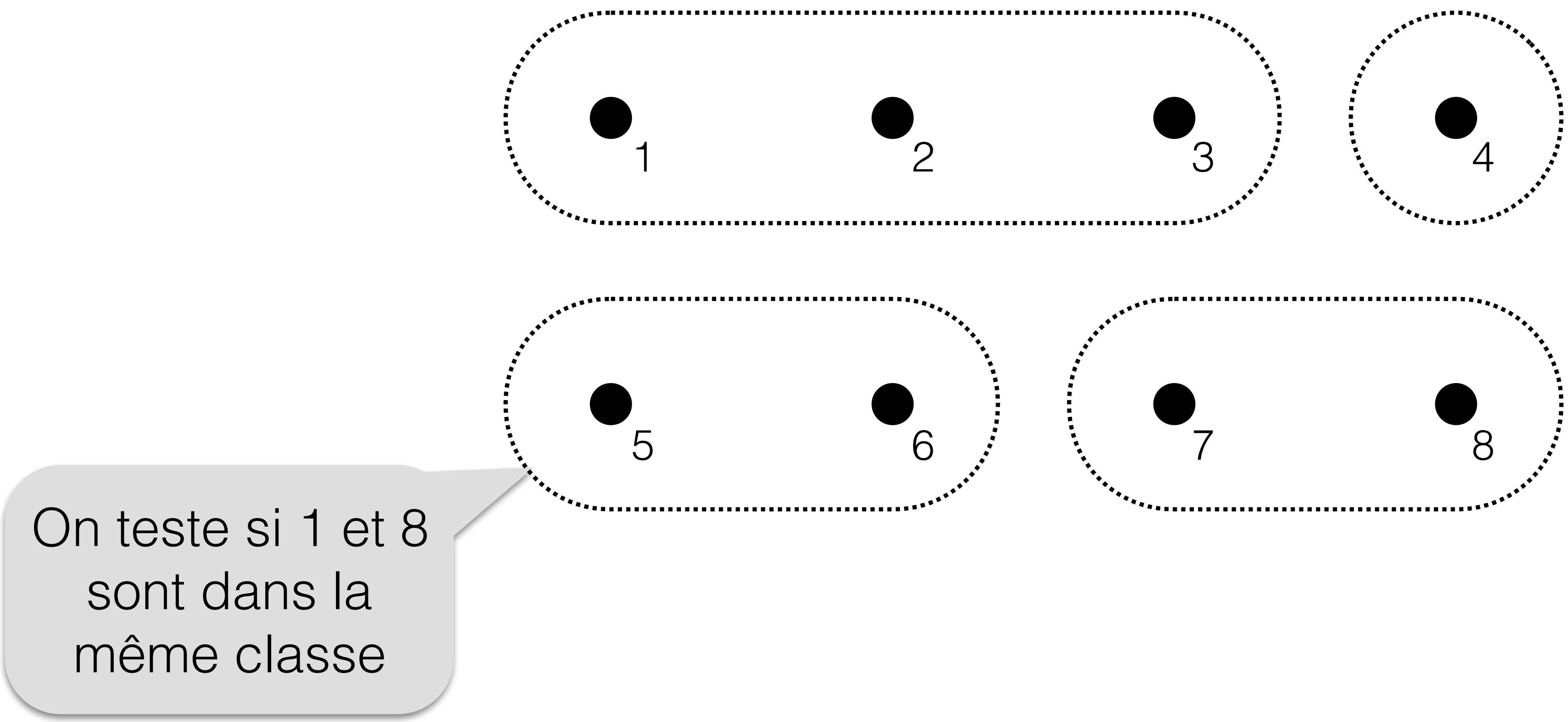
Fusion(1,3)



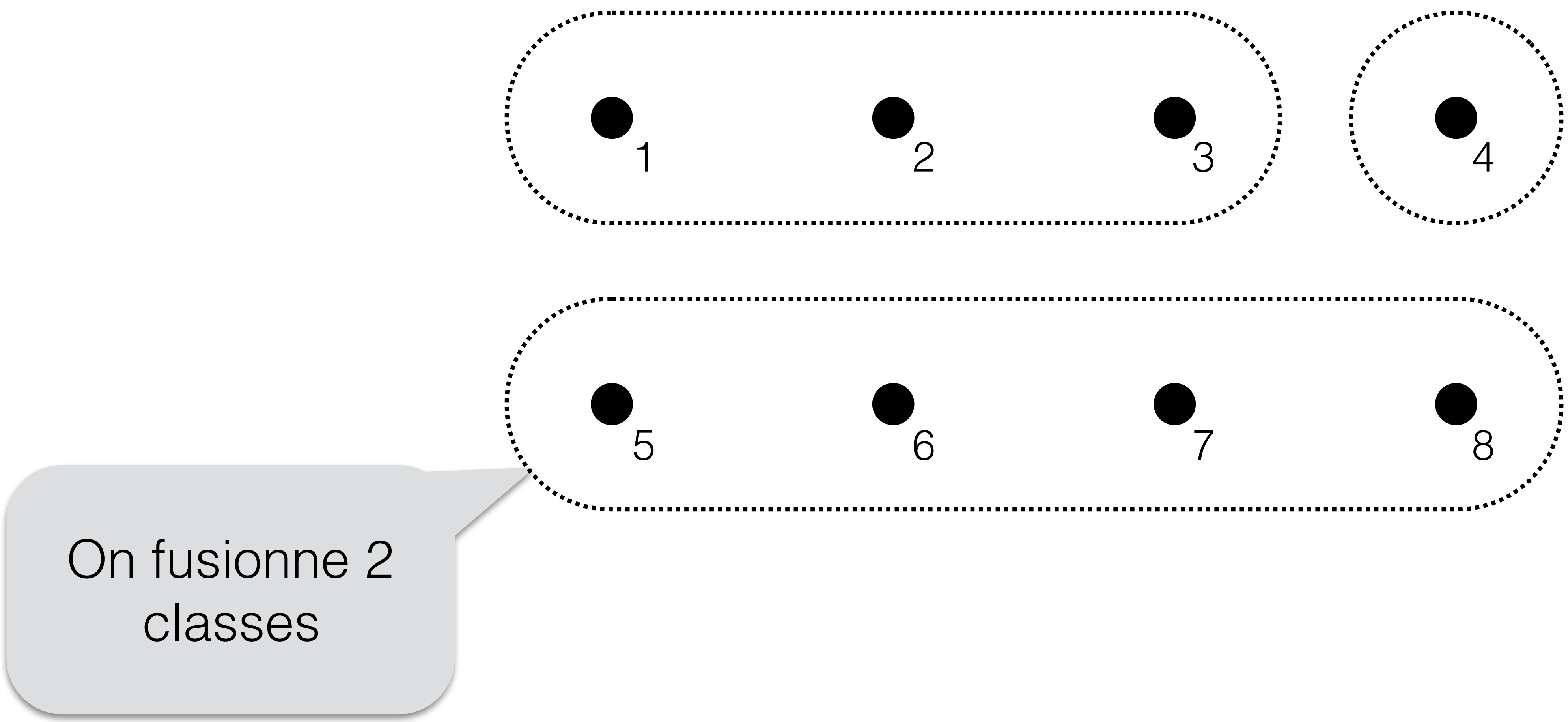
Fusion(5,6)



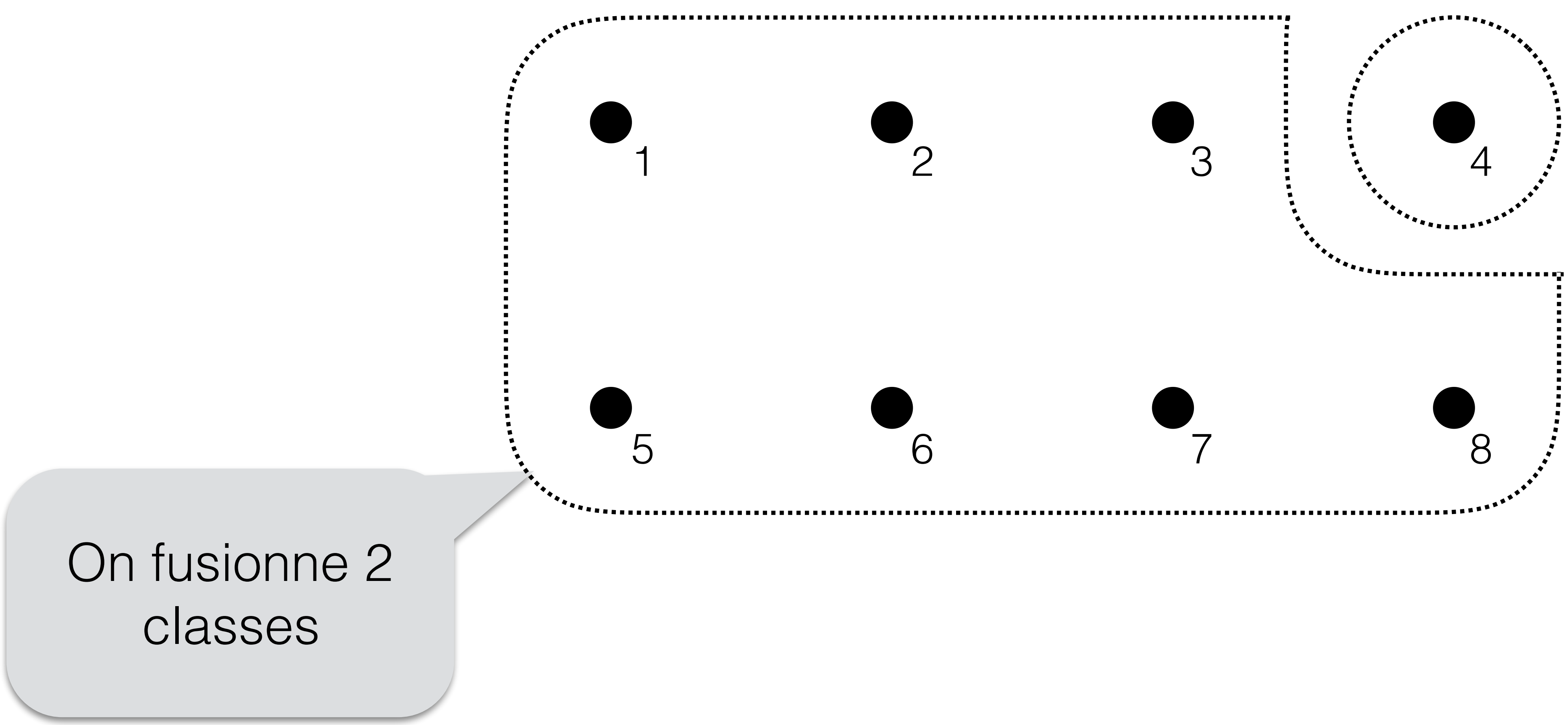
Equiv?(1,8)



Fusion(5,8)



Fusion(1,7)



Deux opérations clés

- Union(a,b)
  - fusionne les classes de a et b
- Find(a)
  - renvoie le numéro de la classe de a
  - Equiv?(a,b) est équivalent à Find(a)==Find(b)

Union-Find

- Il existe une implémentation astucieuse permettant d'obtenir des coûts *amortis* constant

- Coût *amorti constant* : M opérations successives coûtent **O**(log*(N)·M)

nombre de fois qu'il faut appliquer log pour être inférieur à 1

n	1	2	4	16	65536	2^{65536}
log*(n)	0	1	2	3	4	5

1ère idée

- On utilise un tableau repr de taille N tel que repr[i] contient le représentant de la classe de i

```
FIND(i) =
   retourne repr[i]
UNION(i,j) =
   si repr[i]!=repr[j]
      pour k=0 à N-1 faire
         si repr[k]==repr[j]
         alors repr[k] <- repr[i]
```

1ère idée

- On utilise un tableau repr de taille N tel que repr[i] contient le représentant de la classe de i

```
FIND(i) =
    retourne repr[i]
UNION(i,j) =
    si repr[i]!=repr[j]
        pour k=0 à N-1 faire
            si repr[k]==repr[j]
            alors repr[k] <- repr[i]
```

Incorrect !
Pourquoi ?

1ère idée

- On utilise un tableau repr de taille N tel que repr[i] contient le représentant de la classe de i

```
FIND(i) =
   retourne repr[i]
UNION(i,j) =
   si repr[i]!=repr[j]
      repr_j = repr[j]
      pour k=0 à N-1 faire
         si repr[k]==repr_j
         alors repr[k] <- repr[i]
```

À partir de k=j, on oublierait de faire des mises à jour !

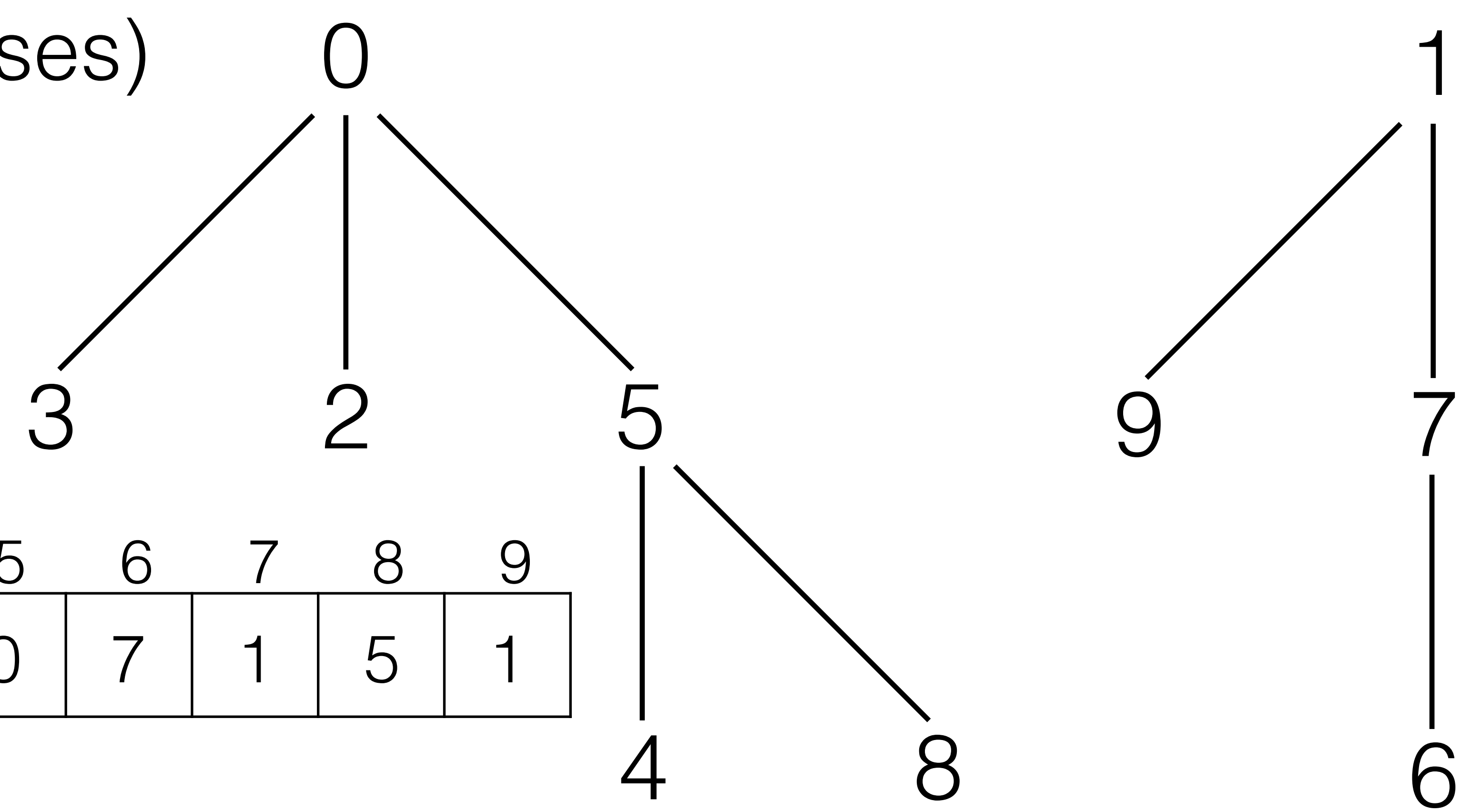
Coût

	1ère idée (*quick-find*)	2ème idée (*quick-union*)	Union-Find
union	O(N)		
find	O(1)		

2ème idée

- On utilise un tableau repr de taille N tel que repr[i] contient le *père* de i dans une forêt F.
  Cette forêt est telle que chaque classe d'équivalence est exactement représentée par un arbre de la forêt

- Exemple (2 classes)



0	1	2	3	4	5	6	7	8	9
0	1	0	0	5	0	7	1	5	1

2ème idée

```
FIND(i) =
    p = repr[i]
    si i!=p alors retourne FIND(p)
    sinon retourne i
```

on remonte dans l'arbre depuis i jusqu'à sa racine

```
UNION(i,j) =
    p = repr[i]
    q = repr[j]
    si p!=q alors
        repr[q] = p
```

l'arbre de j vient s'enraciner dans l'arbre de i

2ème idée

- La complexité de FIND est proportionnelle à la hauteur des arbres. On peut assurer que les arbres sont équilibrés en augmentant le moins souvent possible leurs hauteurs.

```
UNION(i,j) =
   p = repr[i]
   q = repr[j]
   si p!=q alors
      si hauteur(p) < hauteur(q)
      alors repr[p] = q
      sinon repr[q] = p
```

hauteur de l'arbre enraciné en p (calculable en temps constant avec un tableau auxiliaire)

la hauteur de q n'augmente pas

la hauteur de p n'augmente que si p et q avait la même hauteur

Coût

	1ère idée (*quick-find*)	2ème idée (*quick-union*)	Union-Find
union	O(N)	O(1)	
find	O(1)	O(log(N))	

la hauteur d'un arbre ne dépassera pas log(N) (admis)

3ème idée

- Lors du calcul de FIND[i], on peut *compresser les chemins* : pour chaque ancêtre a de i rencontré, on fait la mise à jour `repr[a] = FIND[i]`

- Les arbres vont êtres *aplatis* par chaque appel de FIND.