Arbres couvrants : gloutonnons

G. Aldon - J. Germoni - J.-M. Mény

mars 2012

Arbre couvrant d'un graphe

Un arbre est un graphe connexe sans cycle.

Un arbre couvrant d'un graphe G est un sous-graphe de G qui est connexe, sans cycle et contient tous les sommets de G.

Algorithme générique de construction d'un arbre couvrant

- Input – Un graphe connexe G, un sommet "source" s.
- Output – T arbre couvrant de G.

Algorithme générique de construction d'un arbre couvrant

Traitement.
T sera à chaque étape un arbre sous-graphe de G. On initialise l'arbre T avec le seul sommet source s.
$\mathcal{F}$ désignera à chaque étape l'ensemble des "arêtes frontières", c'est à dire l'ensemble des arêtes reliant un sommet de l'arbre T à un sommet de $G - T$. On initialise l'ensemble $\mathcal{F}$ à l'ensemble des arêtes incidentes au sommet source s.

Tant que $\mathcal{F} \neq \varnothing$

- $e :=$ ProchaineArêteFrontière (le choix de cette prochaine arête sera défini de plusieurs façons possibles dans la suite).
- $w :=$ le sommet de $G - T$ incident à e.
- $T := T$ auquel on ajoute e et w
- On met à jour l'ensemble $\mathcal{F}$ des arêtes frontières.

Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

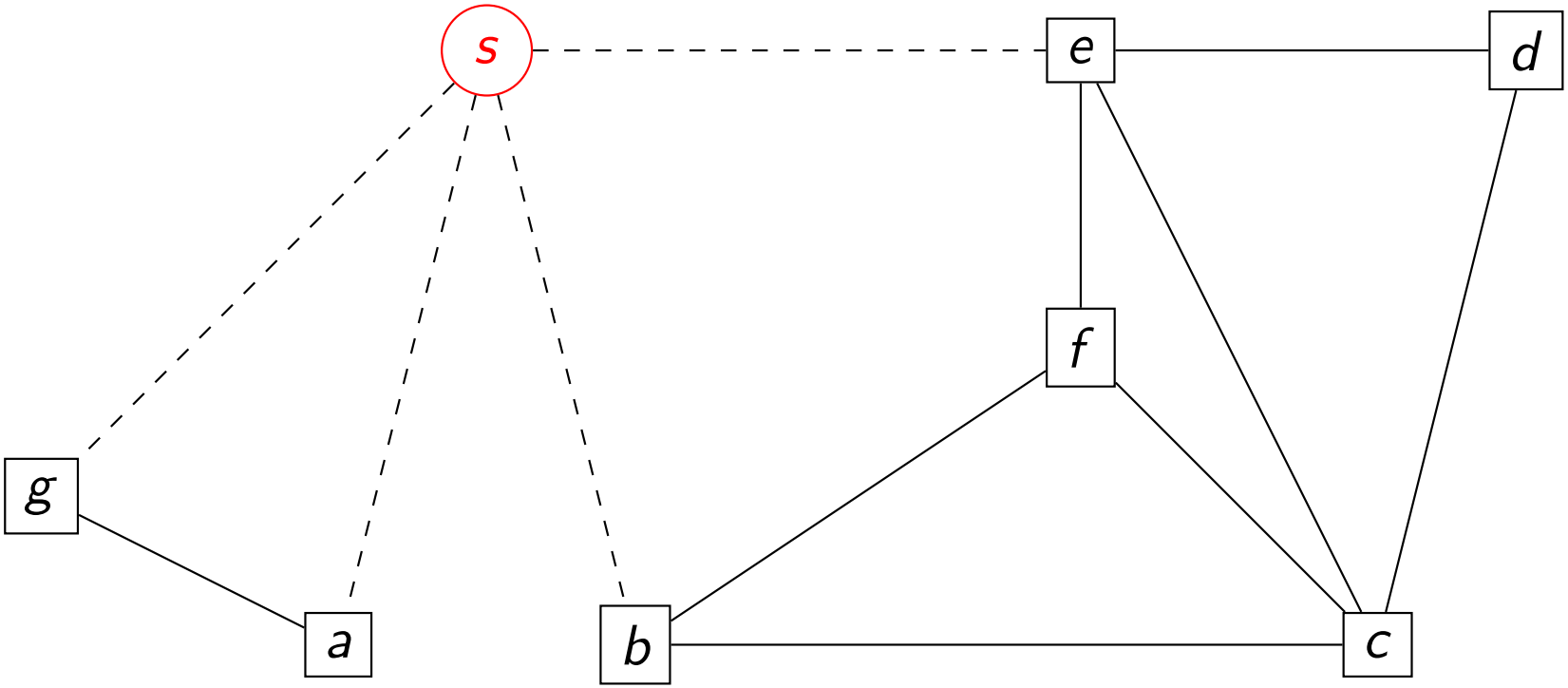


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

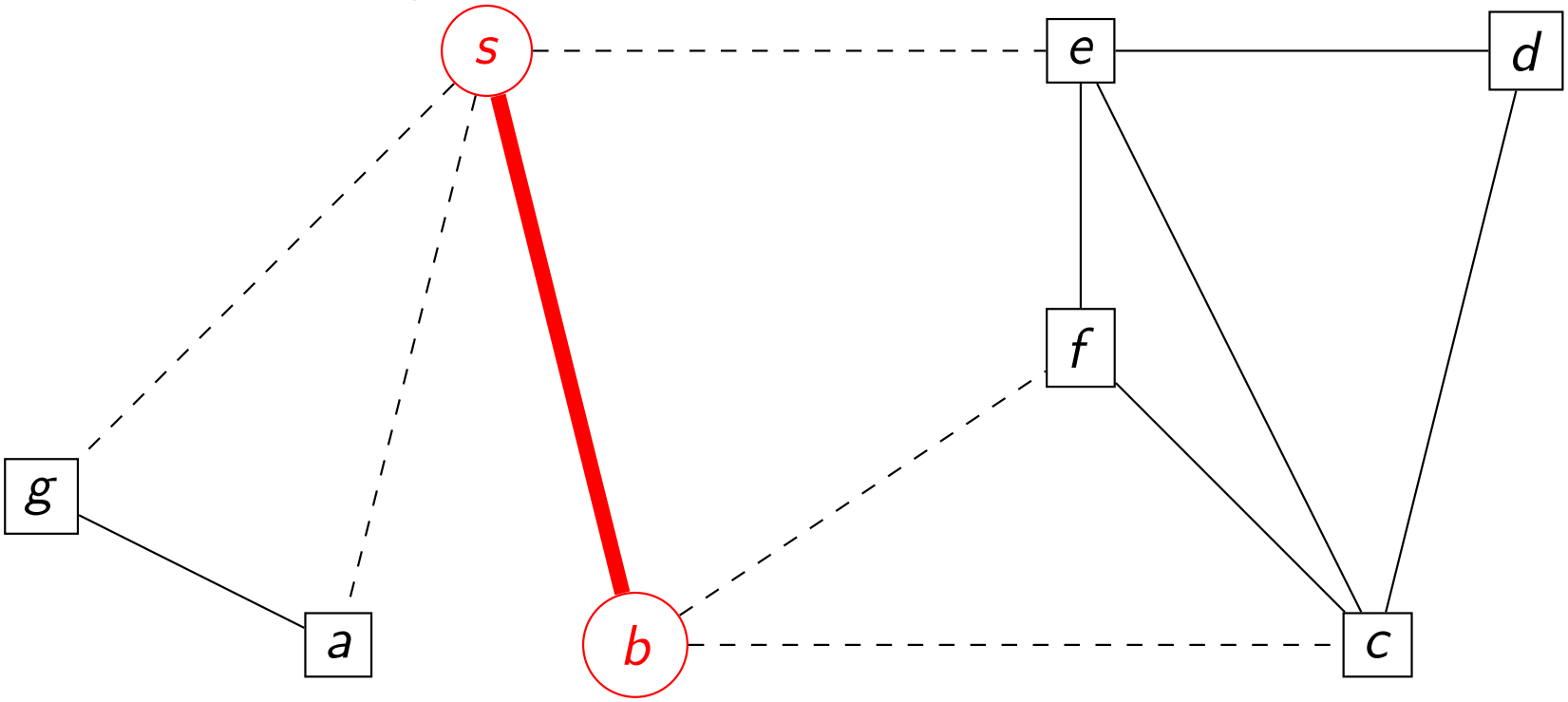


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

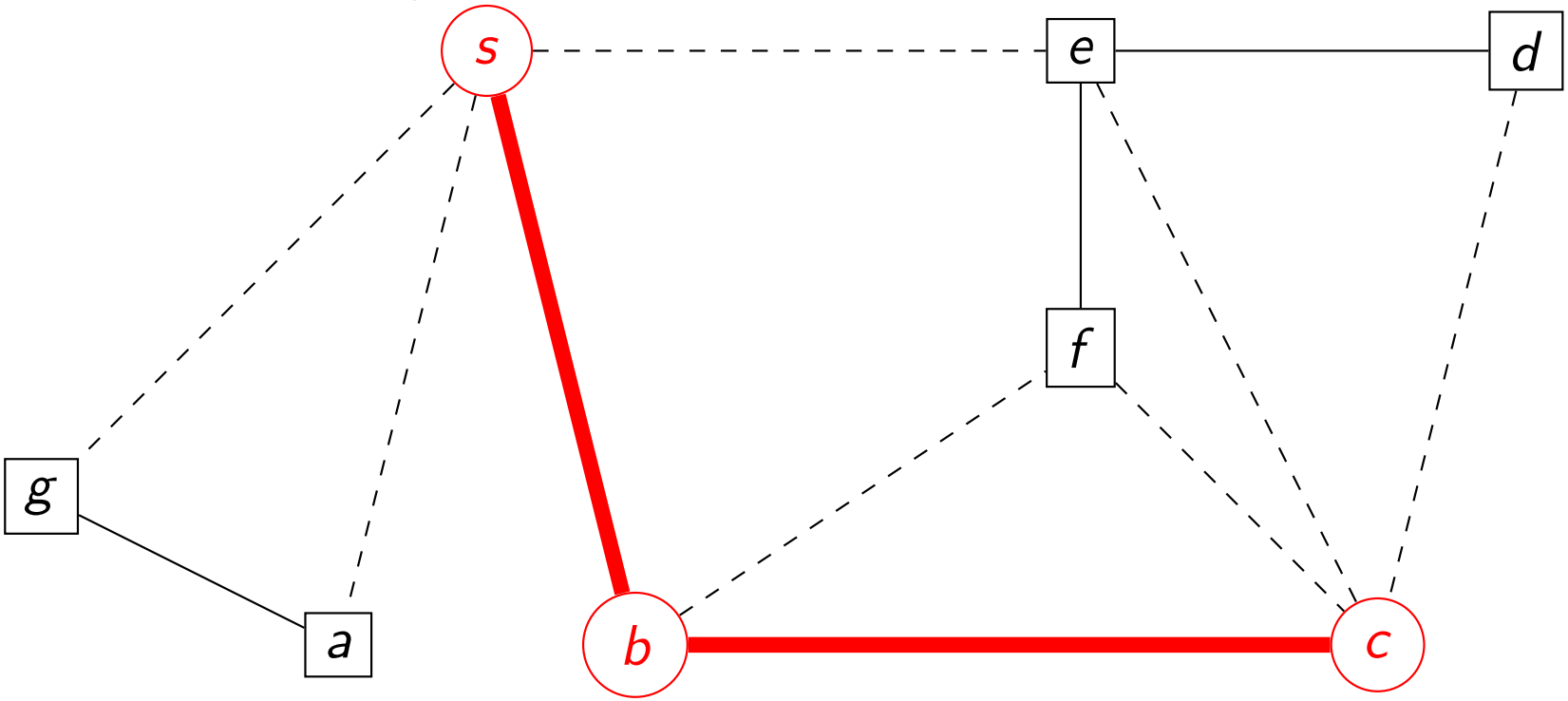


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

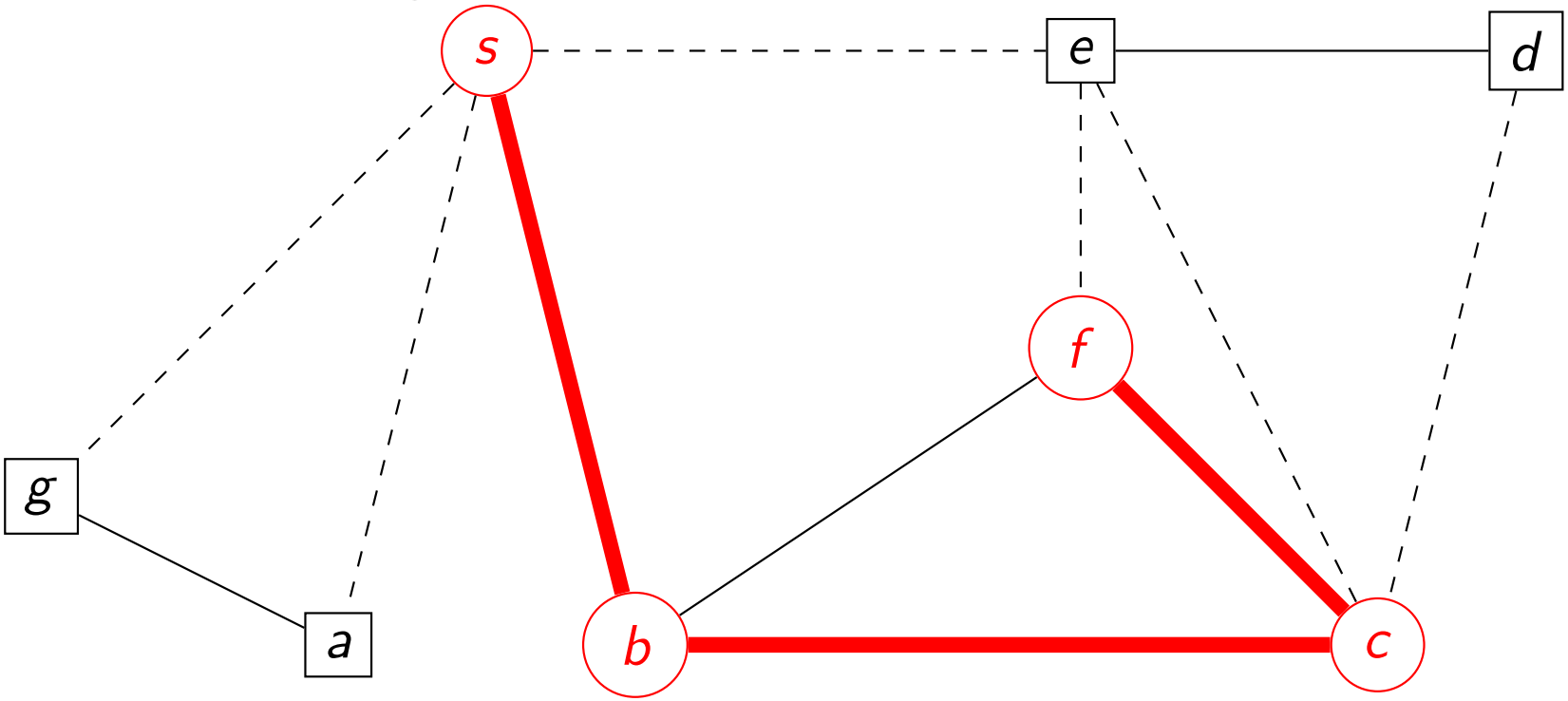


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

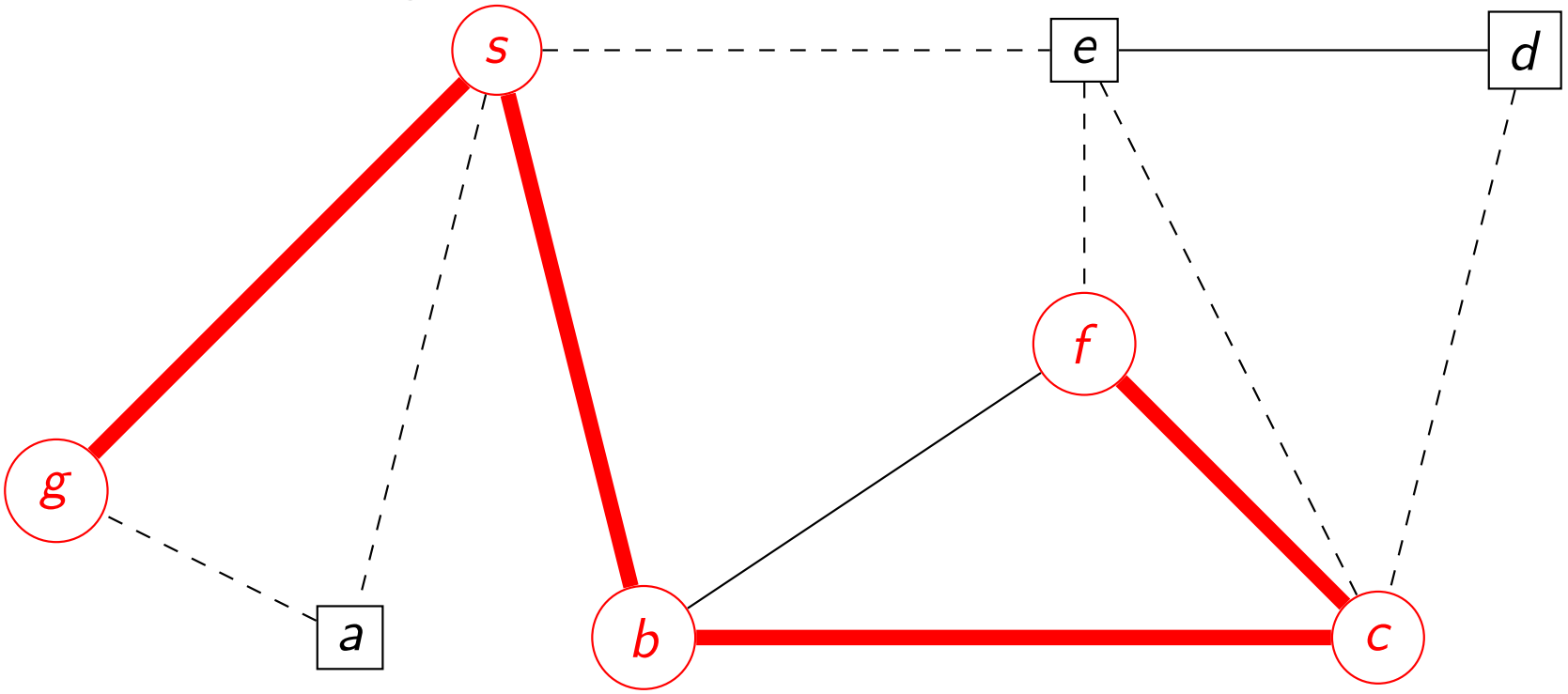


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

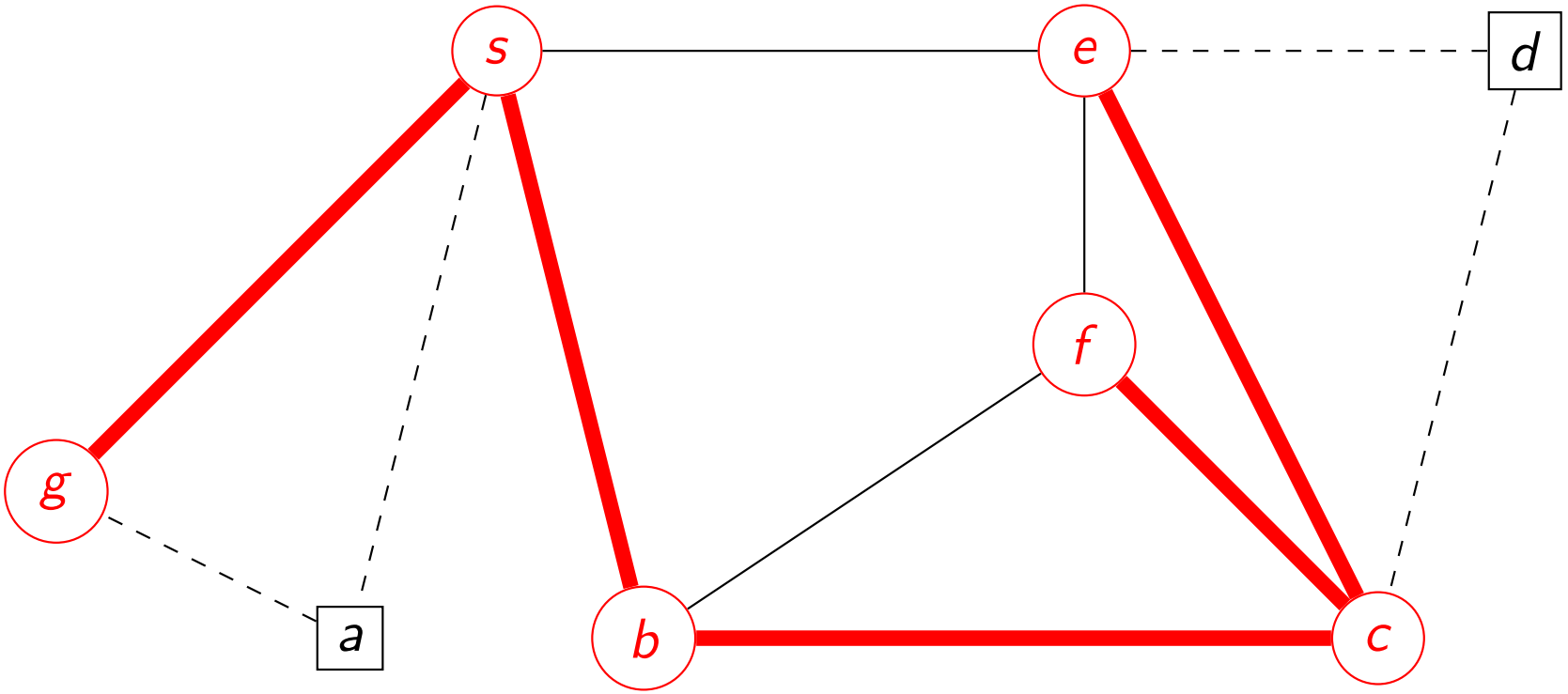


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.

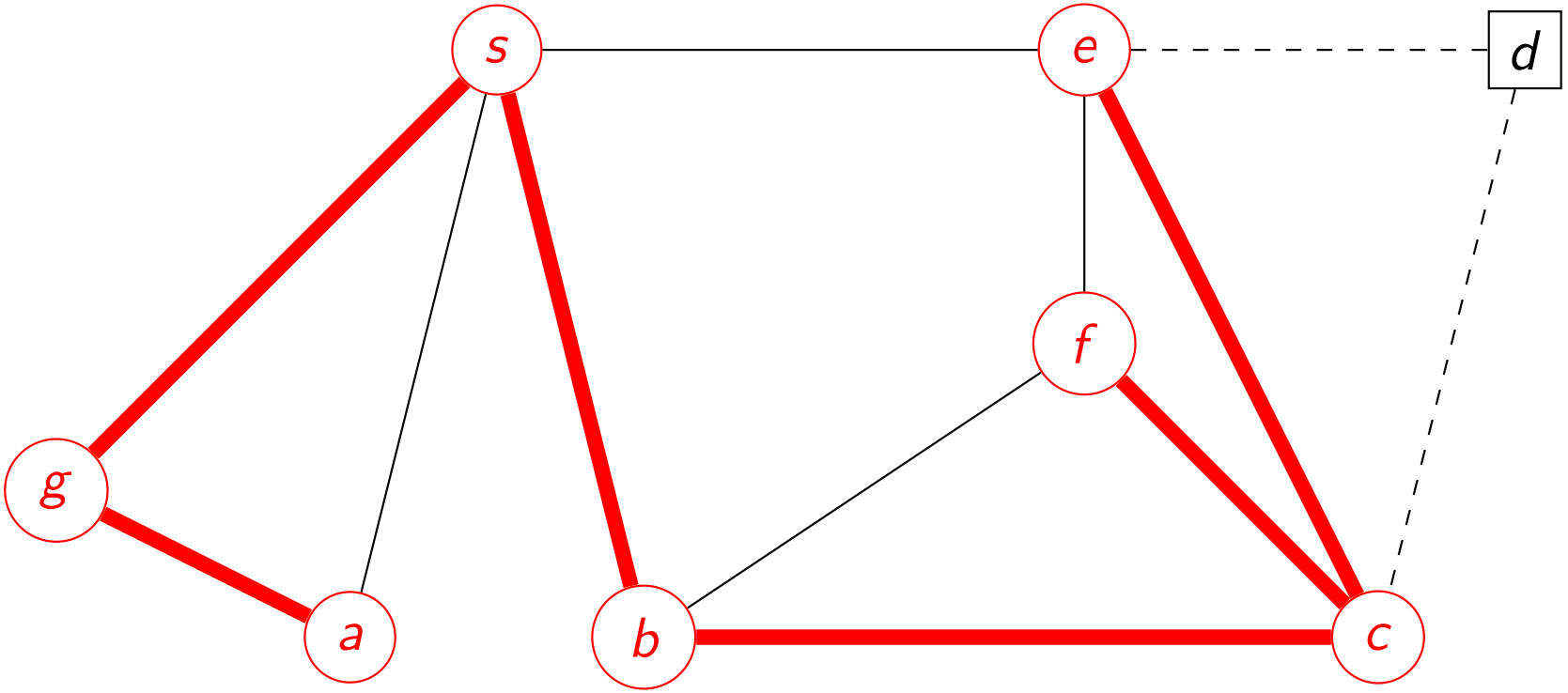
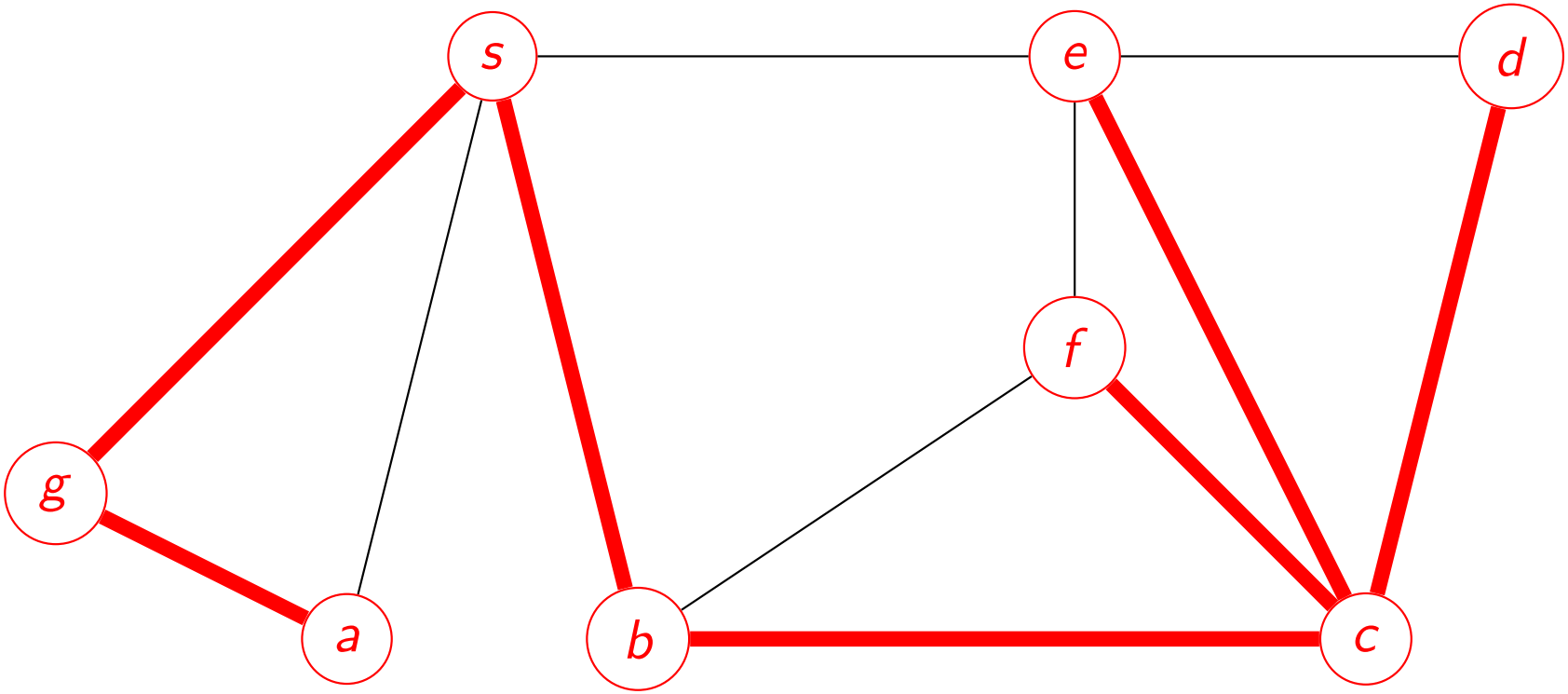


Illustration avec un choix au hasard de l'arête frontière à chaque étape

T est l'arbre rouge, $\mathcal{F}$ est constitué des arêtes pointillées.



Validité de l'algorithme

Exercice. Justifier que cet algorithme donne bien un arbre couvrant du graphe G.

Validité de l'algorithme

1. T est à chaque étape un arbre :
   1. L'initialisation en fait un arbre.
   2. A chaque étape l'ajout d'une arête frontière préserve :
      - la connexité puisque la nouvelle arête est incidente à un sommet déjà dans T.
      - l'absence de cycle puisque le second sommet de l'arête ajoutée n'est pas dans T (cette nouvelle arête ne saurait donc créer un cycle).
2. Il reste à vérifier que l'algorithme s'arrête lorsque tous les sommets de G appartiennent à T.
   Si un sommet x de G n'est pas dans T à une étape donnée, il existe une chaîne de v à x (connexité de G) : $(v_0 = v, v_1, v_2, \ldots, v_k = x)$.
   Soit i le plus petit indice tel que v_i est dans T : l'arête v_i–v_{i+1} est une arête frontière et l'algorithme n'est pas terminé.

Le problème de l'arbre couvrant de poids minimum

Chaque arête d'un graphe connexe G est muni d'un nombre positif ("poids de l'arête").

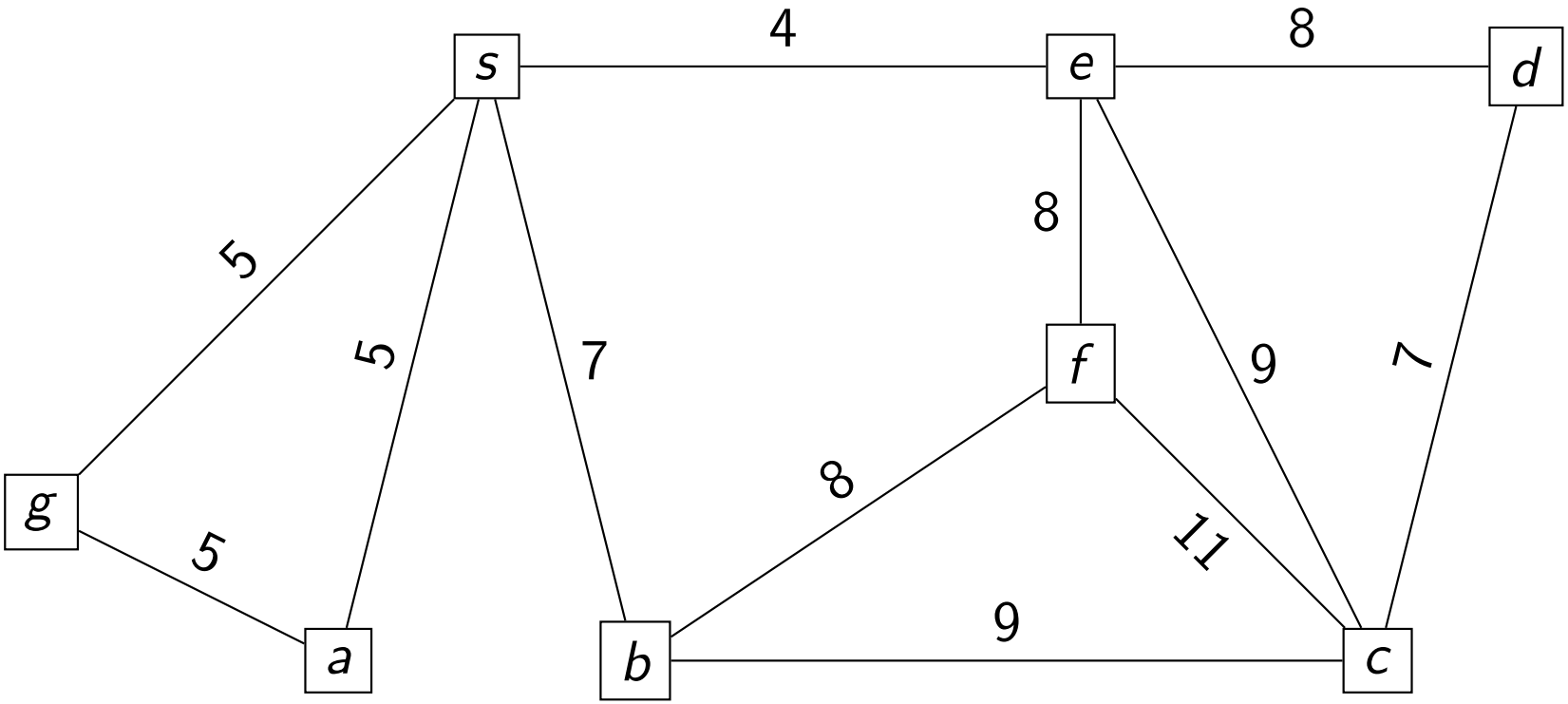
On aimerait obtenir un arbre recouvrant de G de poids minimum parmi les arbres recouvrants de G.

Un choix glouton : Prim

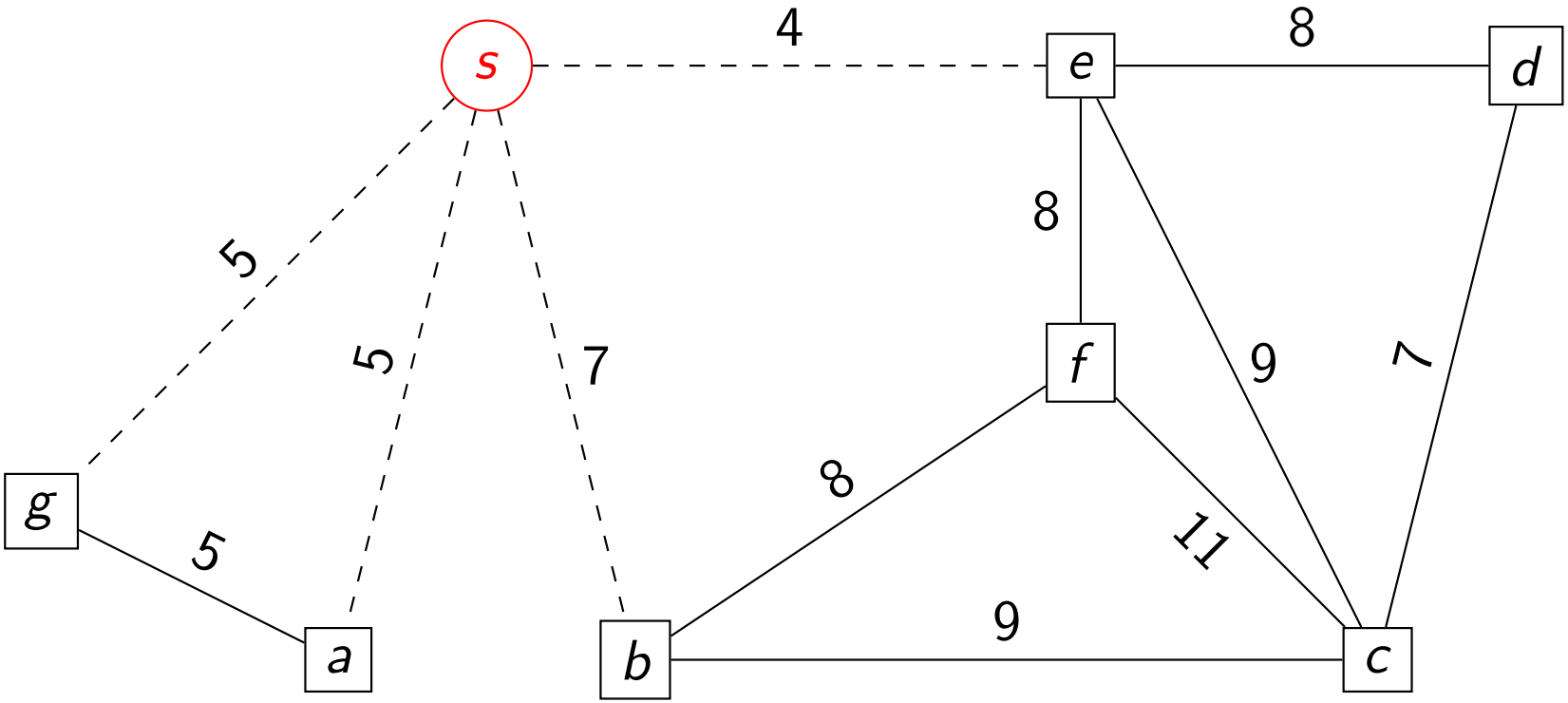
Dans l'algorithme précédent, on gloutonne : on choisit, à chaque étape, un minimum local en sélectionnant l'arête frontière de poids minimum parmi les arêtes frontières.

Un choix glouton : Prim

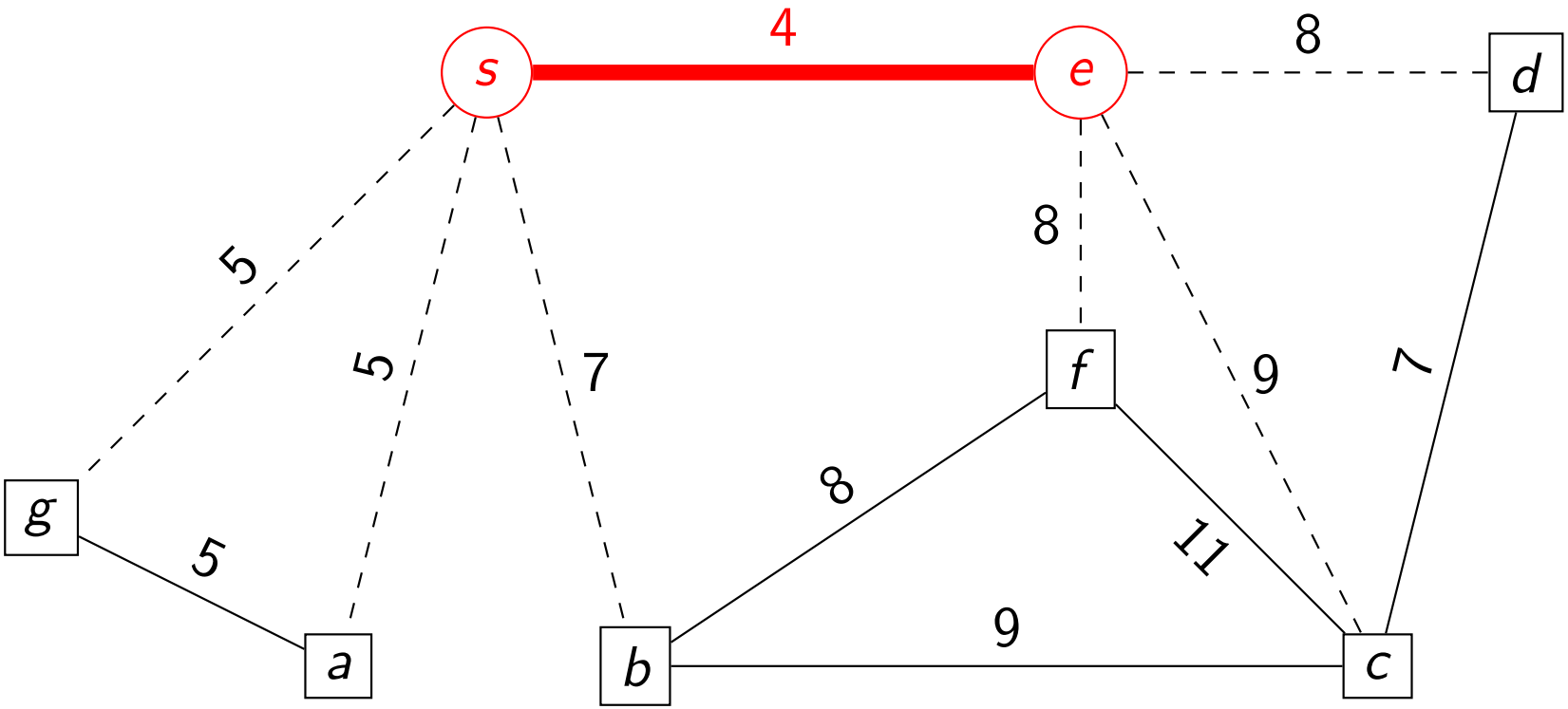
Exercice. Appliquer l'algorithme de Prim au graphe suivant :



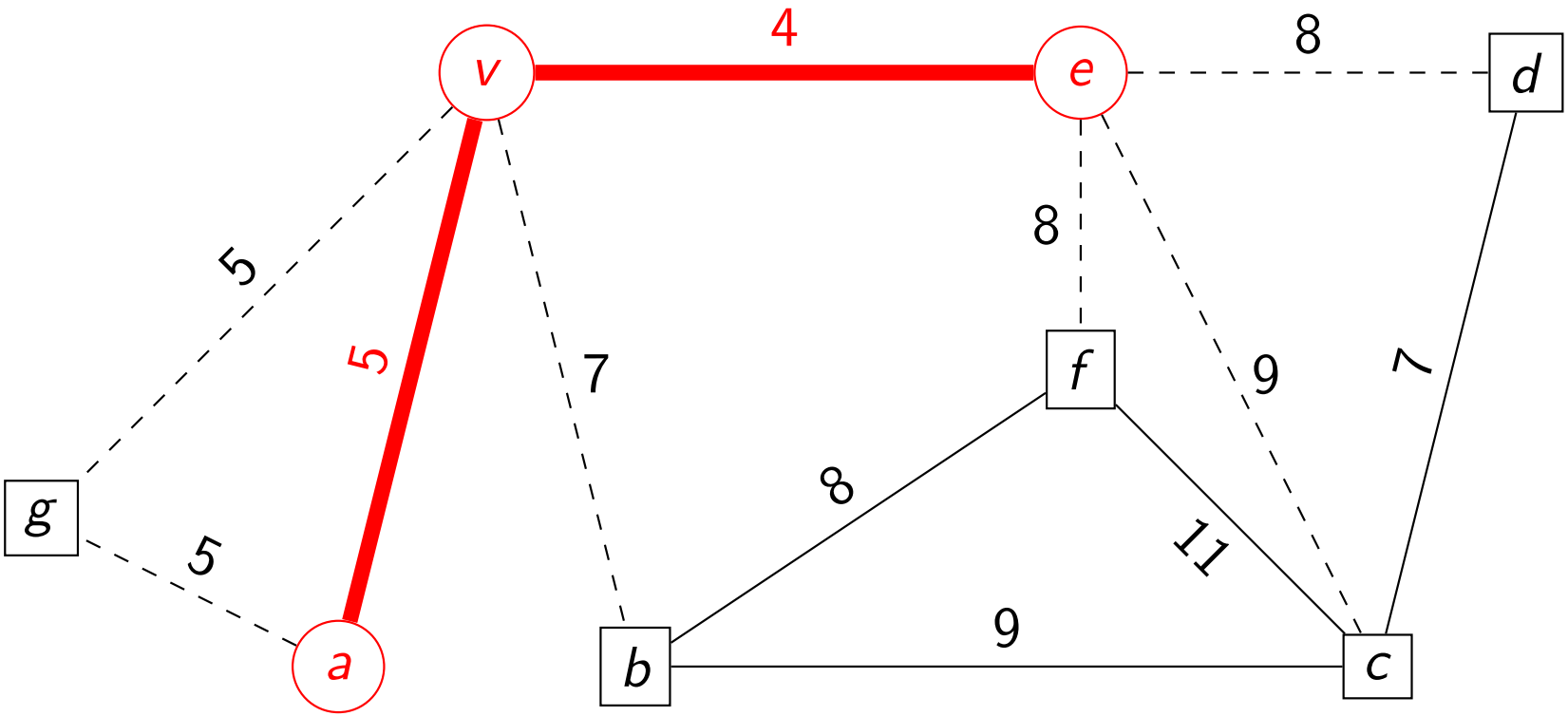
Mise en oeuvre de l'algorithme de Prim



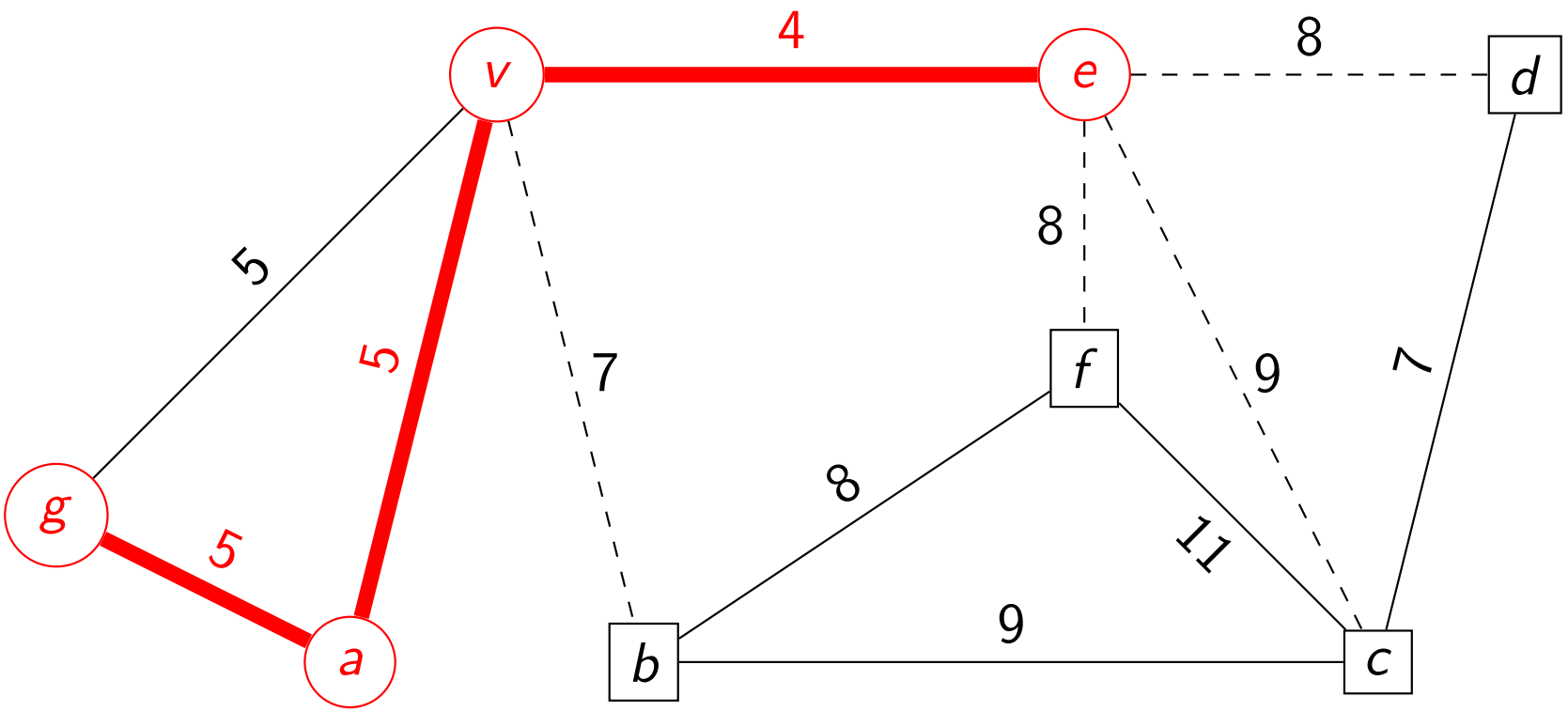
Mise en oeuvre de l'algorithme de Prim



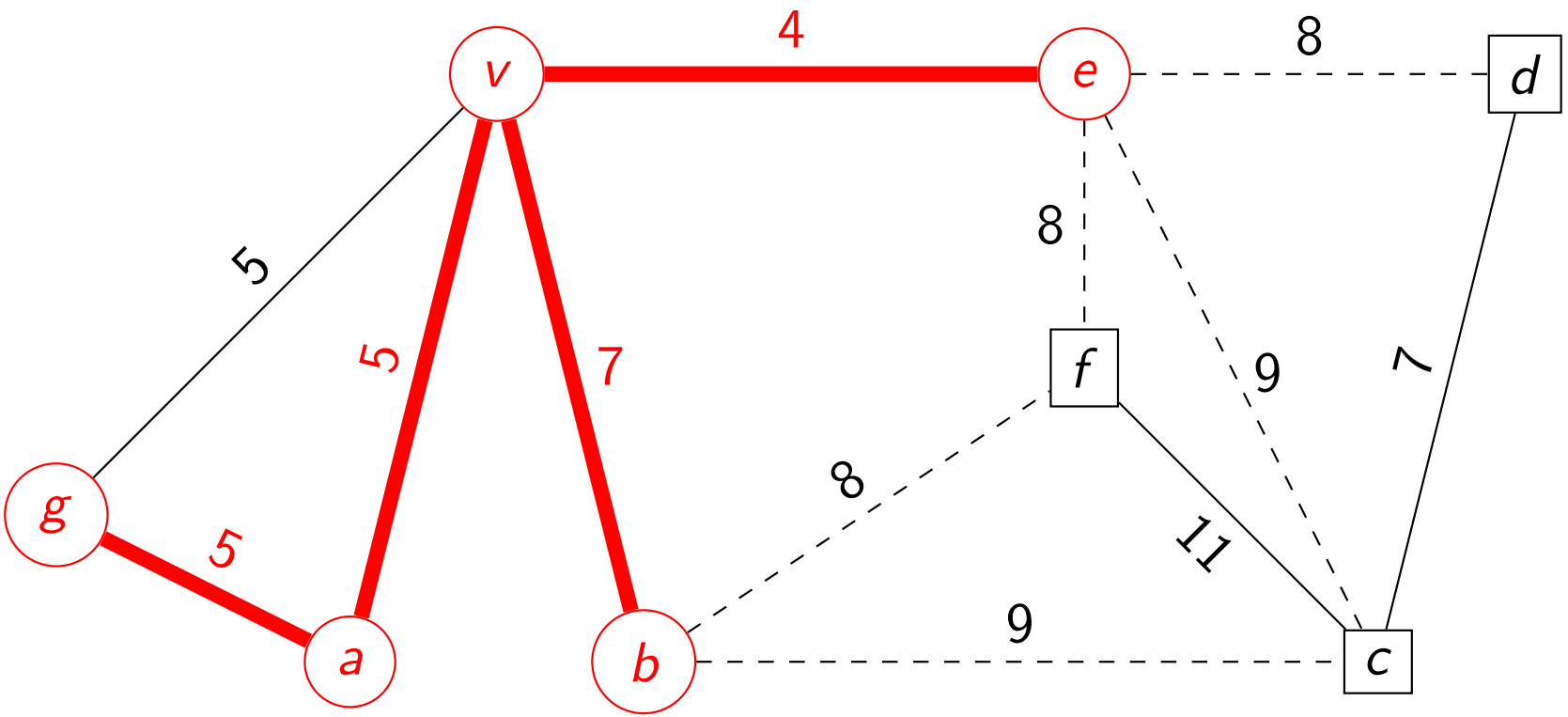
Mise en oeuvre de l'algorithme de Prim



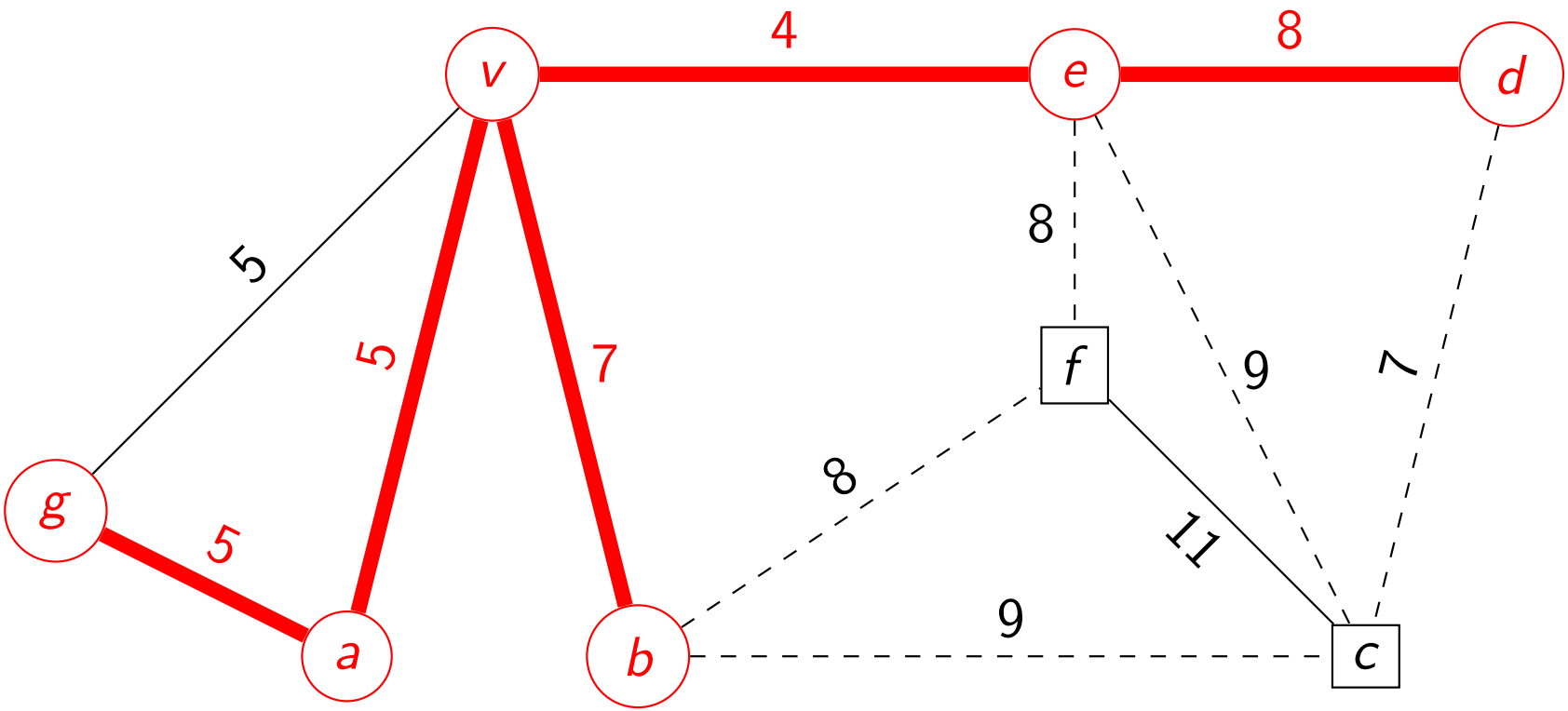
Mise en oeuvre de l'algorithme de Prim



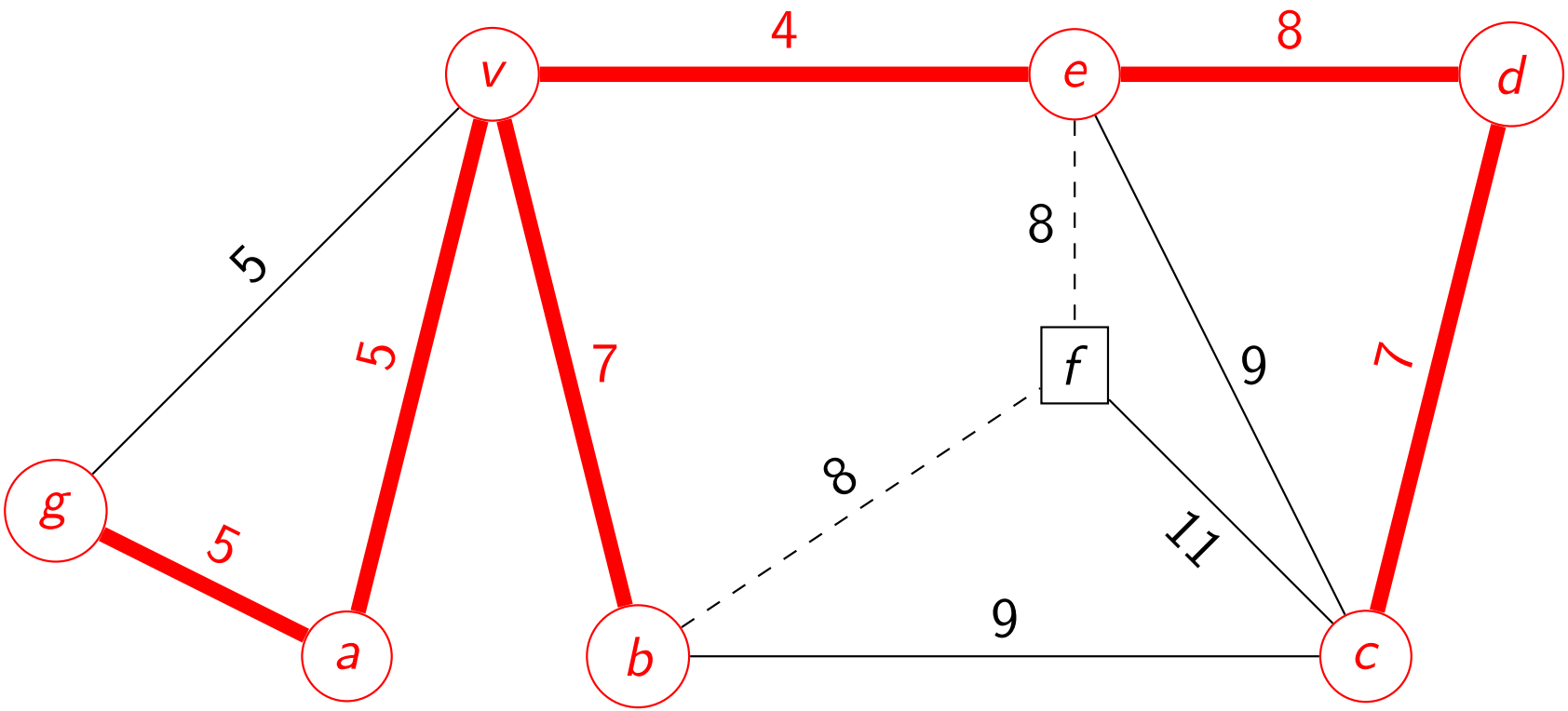
Mise en oeuvre de l'algorithme de Prim



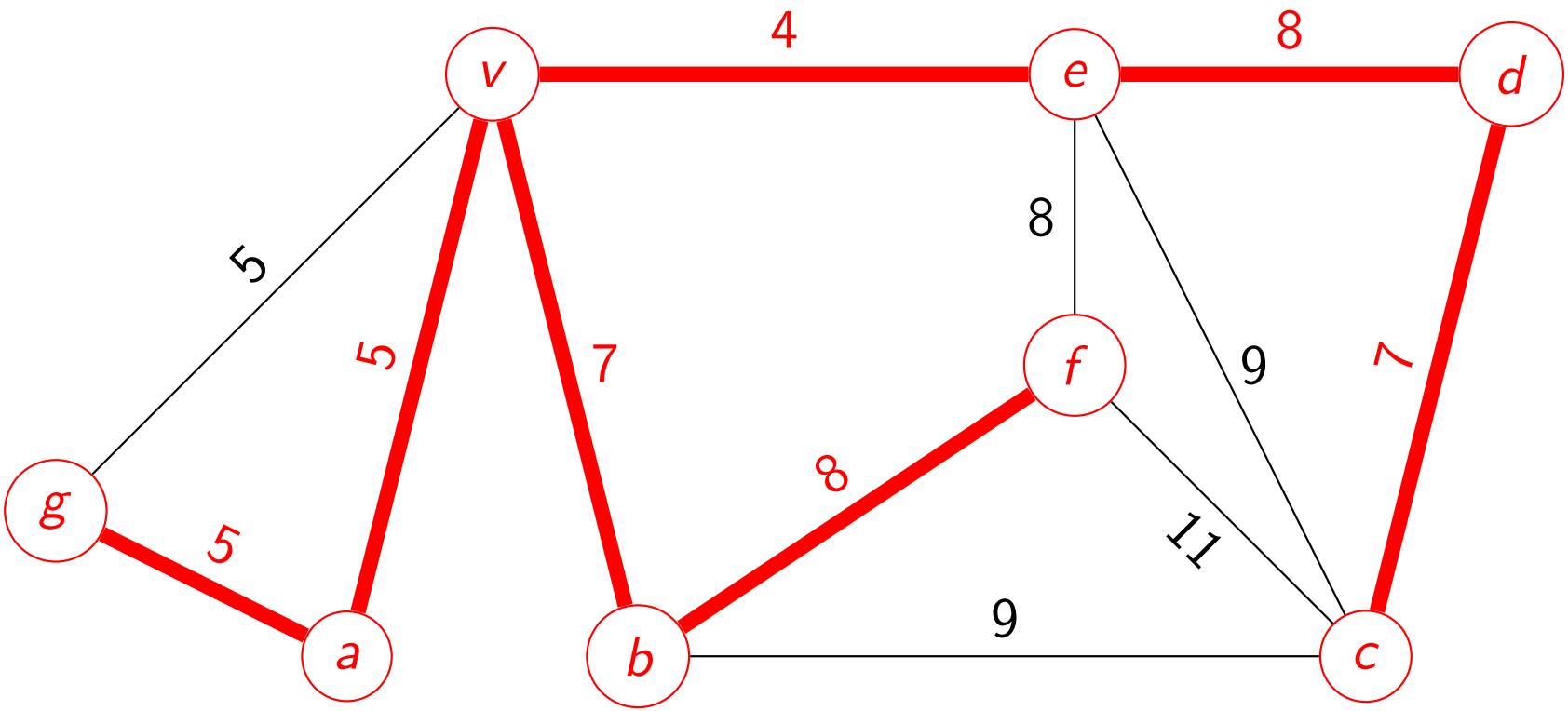
Mise en oeuvre de l'algorithme de Prim



Mise en oeuvre de l'algorithme de Prim



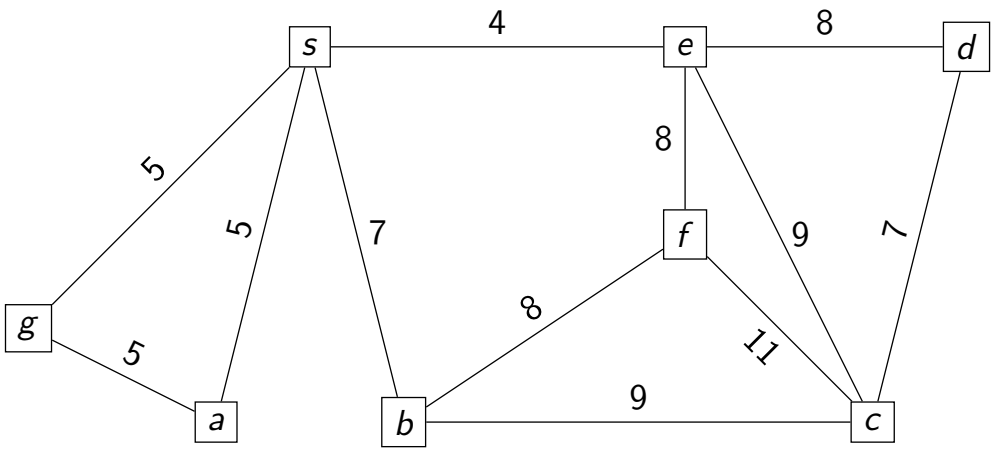
Mise en oeuvre de l'algorithme de Prim



Programmation

Pour programmer l'algorithme, on décide de représenter le graphe précédent par une matrice :

0	0	0	0	0	0	5	5	a
0	0	9	0	0	8	0	7	b
0	9	0	7	9	11	0	0	c
0	0	7	0	8	0	0	0	d
0	0	9	8	0	8	0	4	e
0	8	11	0	8	0	0	0	f
5	0	0	0	0	0	0	5	g
5	7	0	0	4	0	5	0	s
a	b	c	d	e	f	g	s	



Codage Xcas

Xcas

```
G := [[0,0,0,0,0,0,5,5], [0,0,9,0,0,8,0,7], [0,9,0,7,9,11,0,0],
[0,0,7,0,8,0,0,0], [0,0,9,8,0,8,0,4], [0,8,11,0,8,0,0,0], [5,0,0,0,0,0,0,5],
[5,7,0,0,4,0,5,0]], [5,0,0,0,0,0,0,5], [5,7,0,0,4,0,5,0]]
```

Programmation Xcas

On veut écrire une fonction Prim(G) :

- Input : le graphe G sous la forme matricielle précédente.
- Output : l'arbre couvrant de poids minimal sous la forme d'une liste d'arêtes (chaque arête étant donnée par ses deux sommets extrémités) et le poids de l'arbre.

Programmation Xcas – Initialisation

On utilisera :

Xcas

```
n :=rowdim(G); // n est le nombre de sommets du graphe.
s :=hasard(n); // sommet de départ, choisi au hasard.
dans_arbre :=[faux$ n];//dans_arbre[j]==vrai signifiera que le sommet (0 ⩽ j ⩽ n − 1) est déjà intégré dans l'arbre
pere :=["vide"$ n];// pere[j]=k signifiera que l'arête d'extrémités k–j est dans l'arbre, k étant intégré avant j
//le sommet s est mis dans l'arbre :
 dans_arbre[s] :=vrai;
```

Programmation Xcas

Exercice. Écrire le programme.

Programmation Xcas

Xcas

```
pour nbetape de 1 jusque n-1 faire
//parcourir les arêtes frontières pour repérer une arête de poids min
mini :=+infinity ;
pour j de 0 jusque n-1 faire // pour chaque sommet j
 si non(dans_arbre[j]) alors // j n'étant pas encore dans l'arbre
   pour k de 0 jusque n-1 faire // pour chaque sommet k voisin de j et
dans l'arbre
   si G[k,j]>0 et dans_arbre[k] et G[k,j]<mini alors mini :=G[k,j] ;
elu :=j ; predecesseur :=k ;fsi ;
   fpour ;  fsi ;  fpour ;
// On met elu dans l'arbre :
dans_arbre[elu] :=vrai ; pere[elu] :=predecesseur ;
fpour ;
```

Programmation Xcas

Xcas

```
// On parcourt les sommets pour afficher les arêtes de l'arbre et on
calcule le poids total :
arbre :=[ ] ;
poidstotal :=0 ;
pour j de 0 jusque n-1 faire
si j !=s alors
  arbre :=append(arbre,[j,pere[j]]) ;
  poidstotal :=poidstotal+G[j,pere[j]] ;
fsi ;
fpour ;
retourne (arbre,poidstotal) ;
// fin de Prim(G) } : ;
```

Complexité

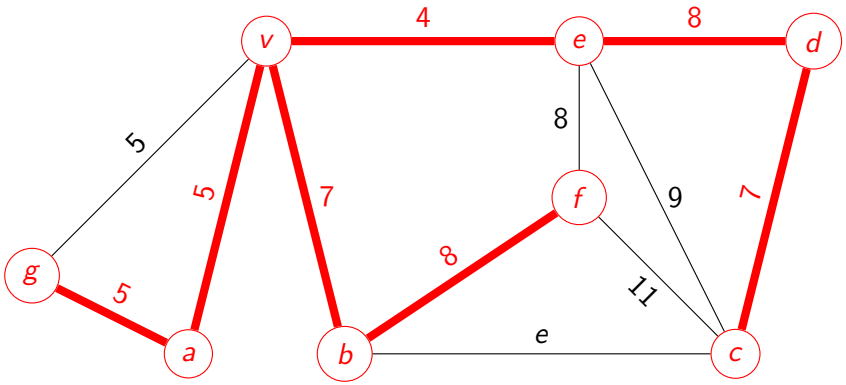
Une implémentation naïve comme celle qui précède mène à une complexité en S^3 (où S est le nombre de sommets) : 3 boucles imbriquées dans la boucle "tant que" principale.

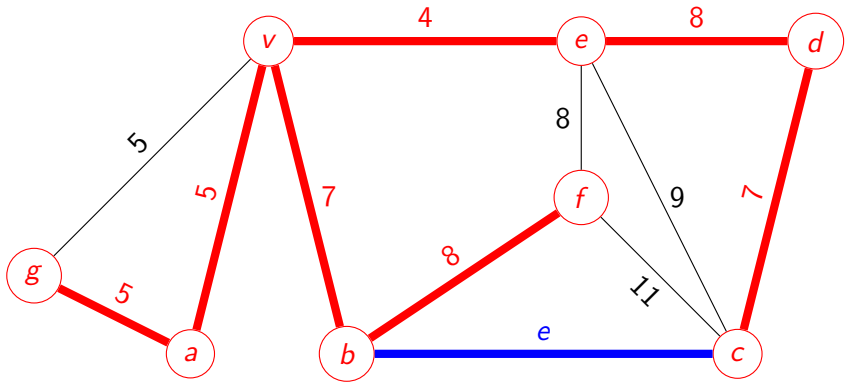
Il est en fait possible de l'implémenter en $O(A + S\log(S))$ (où A est le nombre d'arêtes et S le nombre de sommets).

Validité de l'algorithme de Prim

Deux résultats classiques sur les arbres couvrants :

Lemme du cycle.
« Si $\mathcal{A}$ est un arbre couvrant d'un graphe connexe G et si e est une arête de $G - \mathcal{A}$ alors le graphe $\mathcal{A} + e$ contient un unique cycle (e est une arête de ce cycle). »

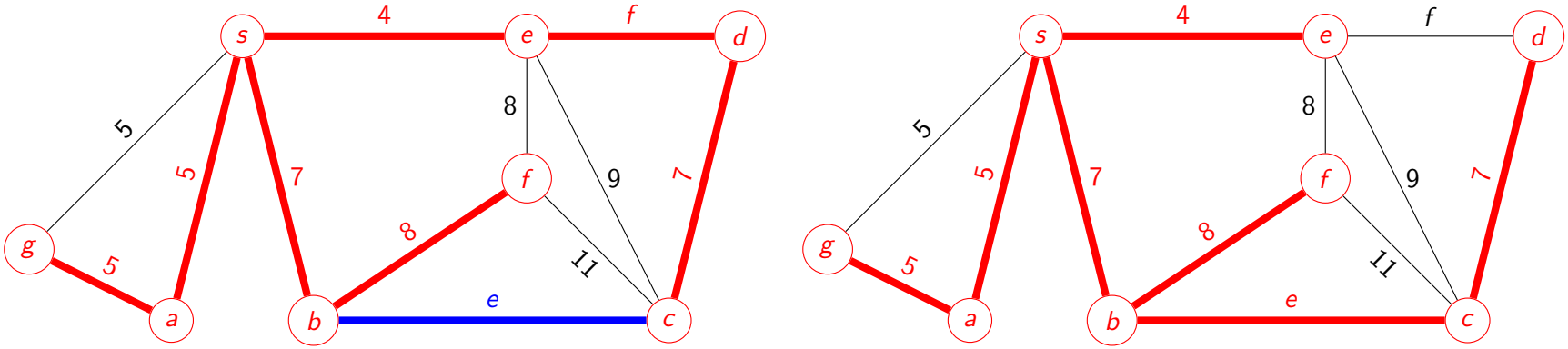




Validité de l'algorithme de Prim

Lemme d'échange.

« Soit $\mathcal{A}$ un arbre couvrant du graphe connexe G, e une arête n'appartenant pas à $\mathcal{A}$ et f une arête de l'unique cycle de $\mathcal{A} + e$ alors $\mathcal{A}' = \mathcal{A} + e - f$ est un arbre couvrant de G ».



Validité de l'algorithme de Prim

Exercice. Prouver l'algorithme de Prim en établissant l'invariant de boucle suivant :

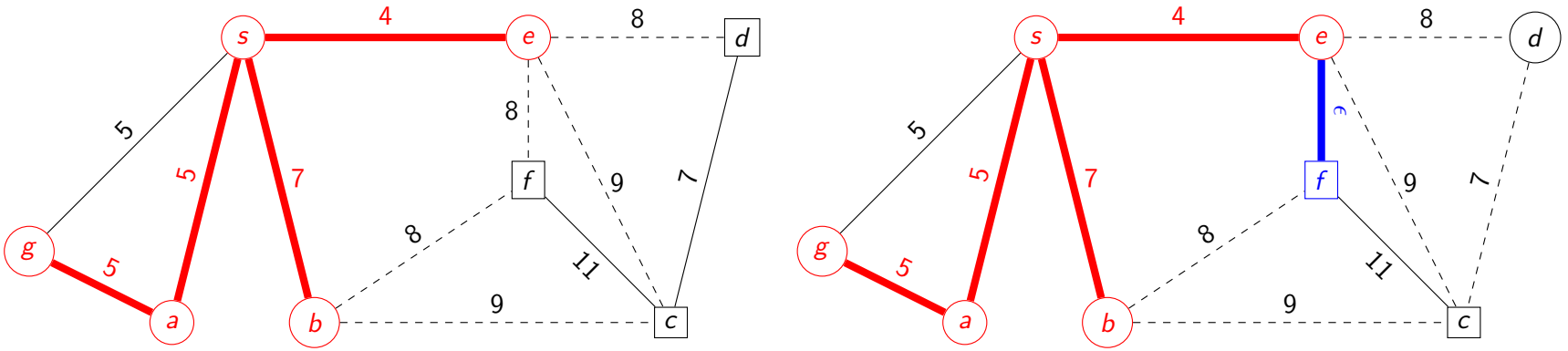
T_k l'arbre obtenu après k itérations de l'algorithme de Prim sur un graphe connexe G est contenu dans un arbre couvrant de G de poids minimal.

A l'initialisation, la condition est évidemment satisfaite : pour $k = 0$, l'arbre T_0 est réduit au sommet source s et tout arbre de poids minimal (existe évidemment) contient s.

Validité de l'algorithme de Prim

Supposons que pour un k $(0 \leqslant k \leqslant |V_G| - 2)$, T_k soit sous-arbre d'un arbre couvrant $\mathcal{A}$ de poids minimal.

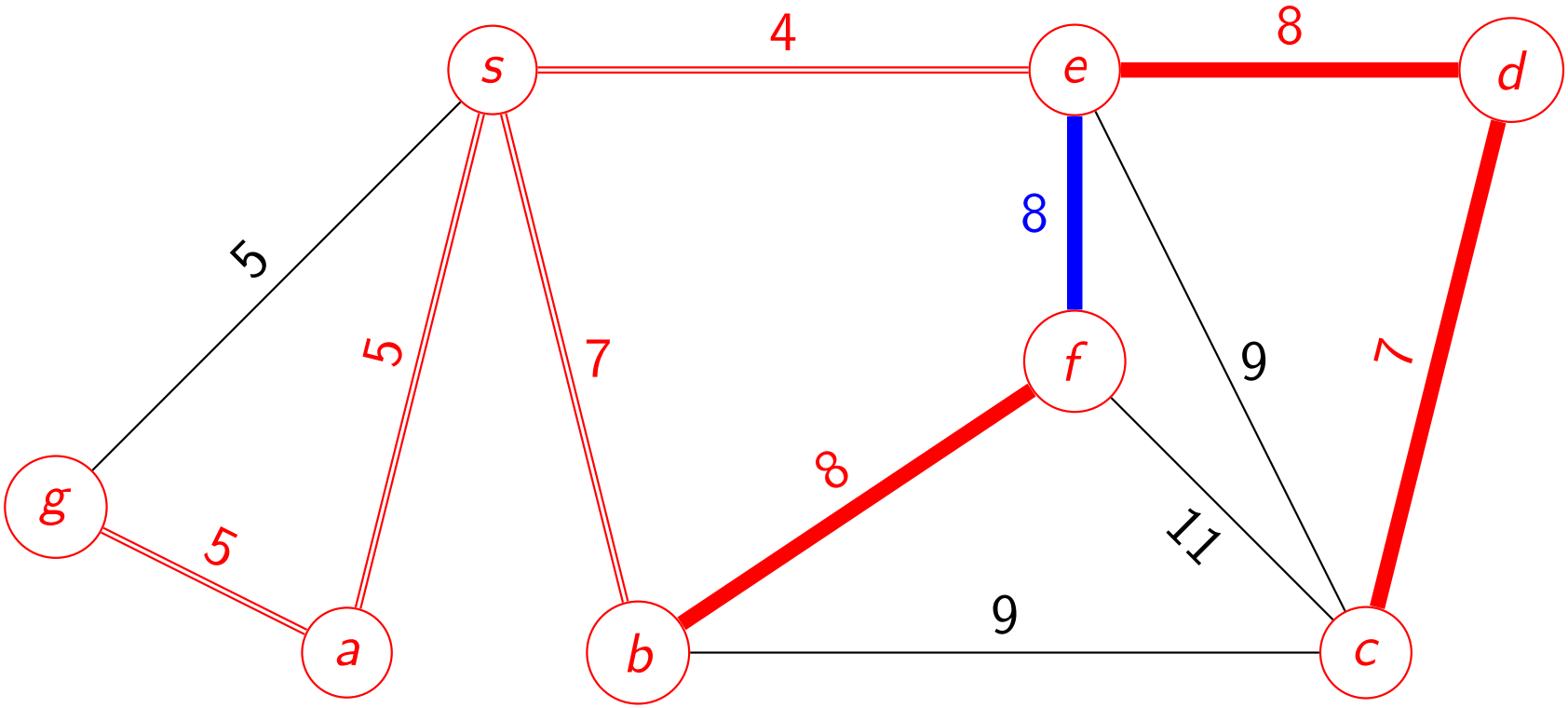
Notons ϵ l'arête de G qui est ajouté à T_k pour obtenir T_{k+1}, et e le sommet de ϵ appartenant à T_k, f le sommet de ϵ n'appartenant pas à T_k.



Si ϵ est une arête de $\mathcal{A}$, alors T_{k+1} est un sous-arbre de $\mathcal{A}$, donc est sous-arbre d'un arbre de poids minimal.

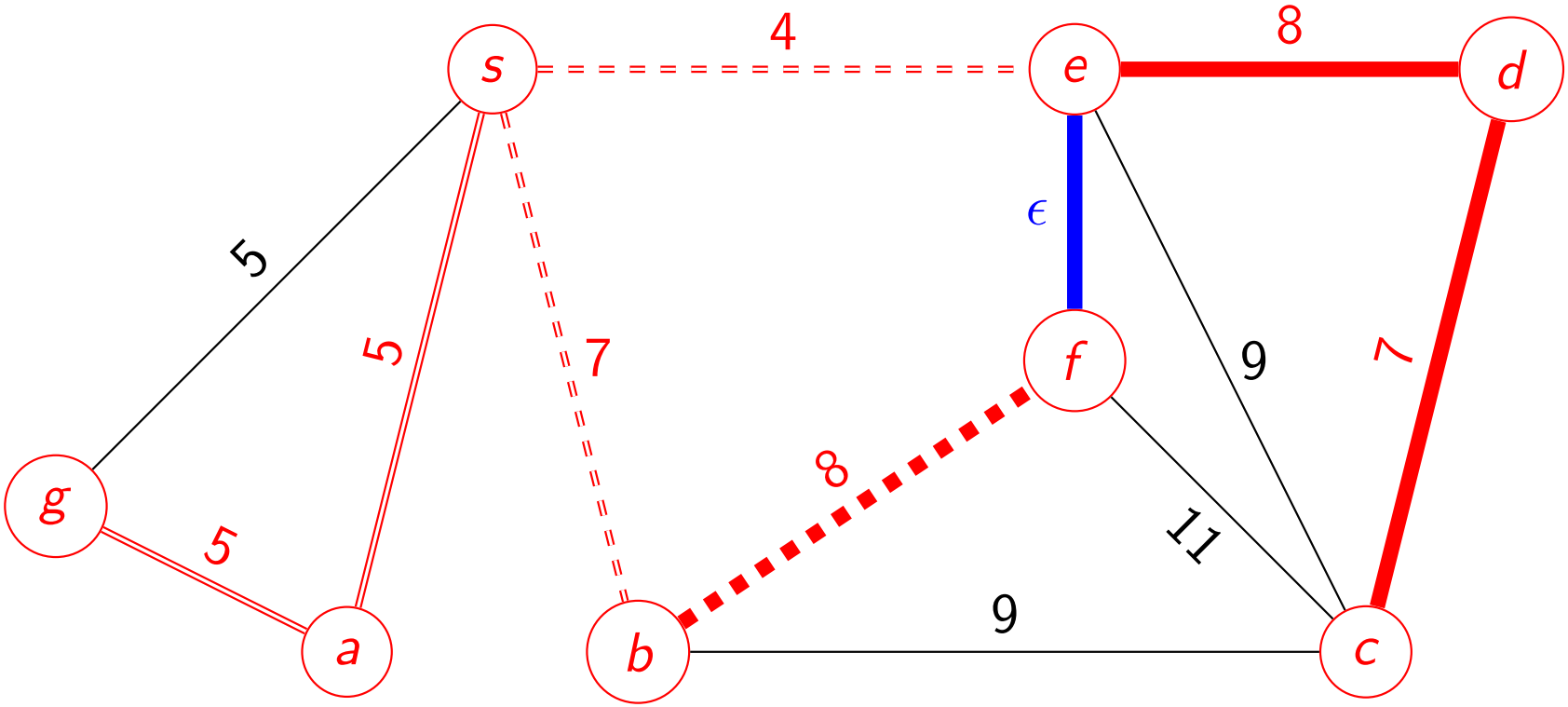
Validité de l'algorithme de Prim

Si ϵ n'est pas une arête de l'arbre minimal $\mathcal{A}$, on construit un autre arbre couvrant $\mathcal{A}'$ de poids minimal et contenant T_{k+1}.



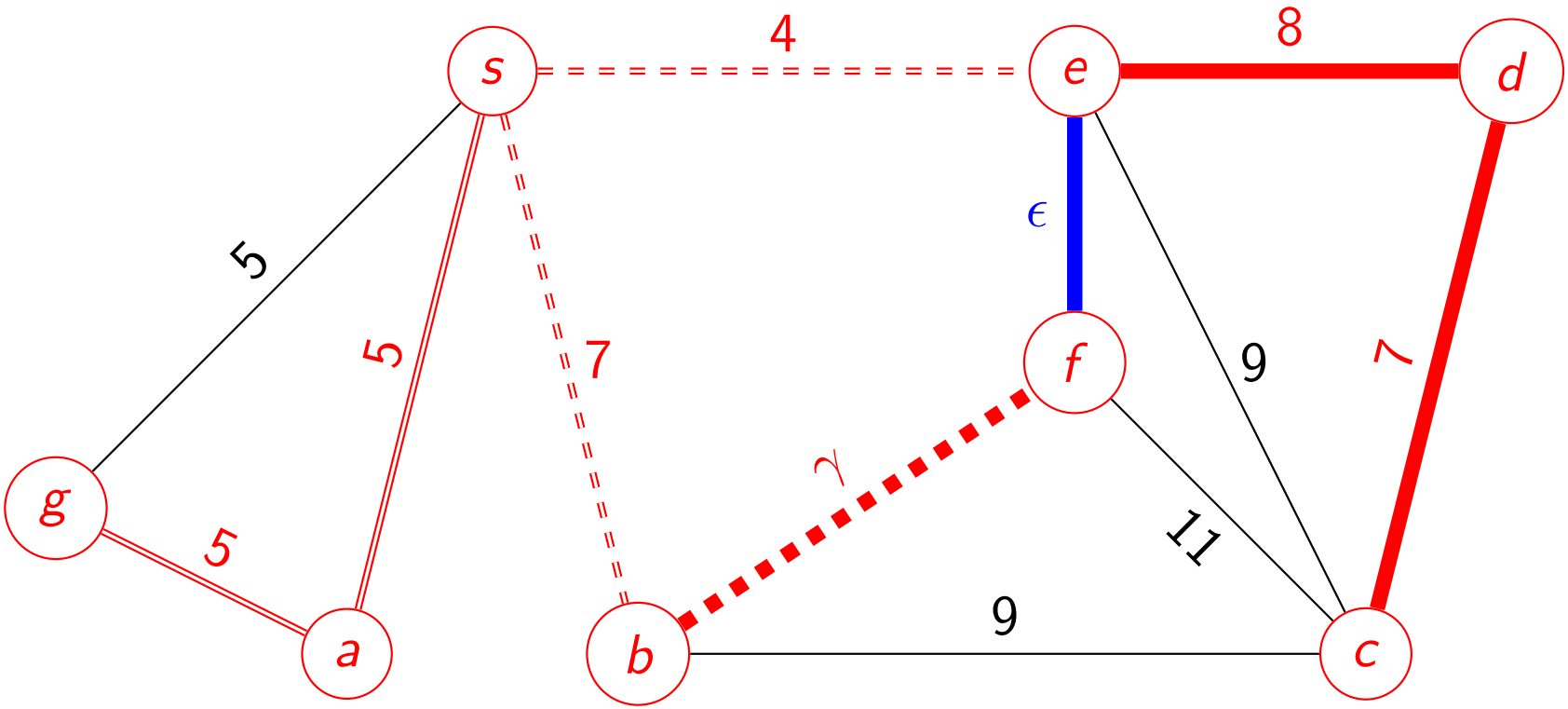
Validité de l'algorithme de Prim

Comme ϵ n'est pas une arête de $\mathcal{A}$, il existe un unique cycle dans le graphe $\mathcal{A} + \epsilon$ (et ϵ en est une arête). Considérons le chemin élémentaire de e à f "complémentaire de ϵ dans ce cycle".



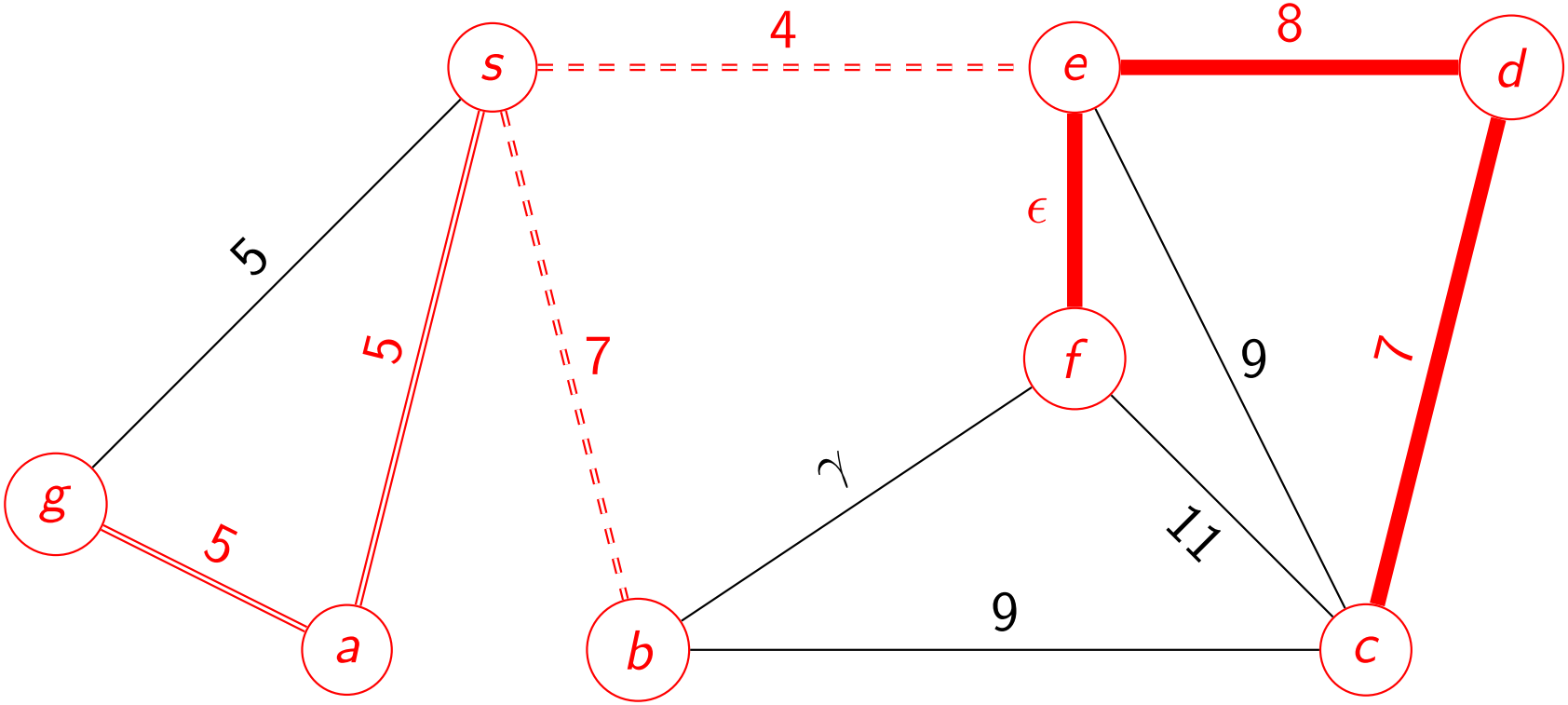
Validité de l'algorithme de Prim

Comme e est dans T_k et f ne l'est pas, il existe une arête γ dans ce chemin joignant un sommet de T_k à un sommet hors de T_k. γ est donc une arête frontière à la fin de l'étape k.



Validité de l'algorithme de Prim

Comme l'algorithme a sélectionné ϵ, on en déduit : poids(ϵ) $\leqslant$ poids(γ).
L'arbre $T' = T + \epsilon - \gamma$ est un arbre couvrant de G, contient T_{k+1} et poids(T') = poids(T) + poids(ϵ) − poids(γ) $\leqslant$ poids(T).



Arborescence des distances à un sommet source

Les poids des arêtes sont maintenant considérés comme des distances entre sommets. Les poids des arêtes sont en particulier supposés positifs. On aimerait connaître la distance minimale du sommet source s à tout autre sommet.

On aimerait donc que l'arbre couvrant T obtenu soit tel que pour tout sommet x de G, la chaîne de l'arbre T reliant la source s à x soit une chaîne de longueur minimale parmi les chaînes de s à x du graphe G.

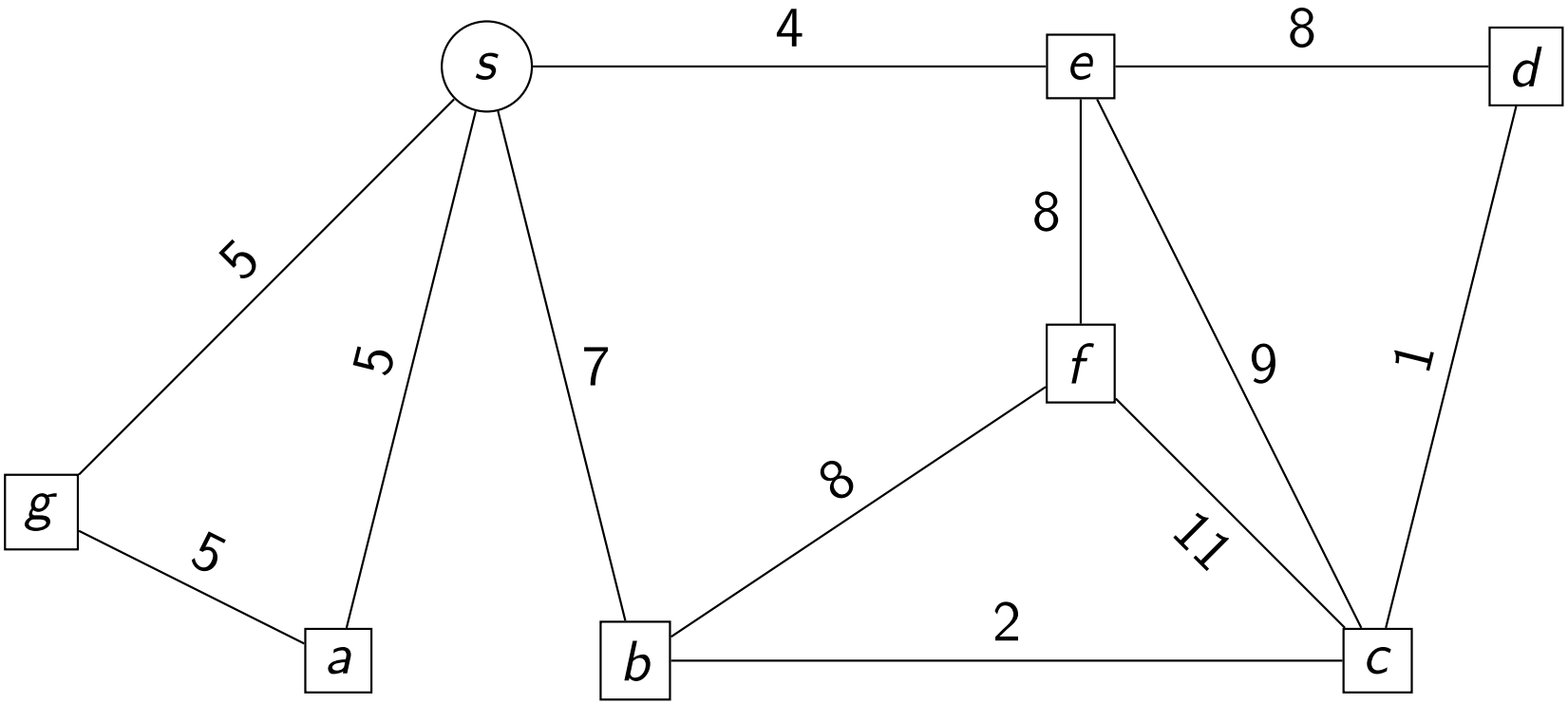
Principe de l'algorithme de Dijkstra

On initialise l'arbre T avec le seul sommet s.
On marque s d'un 0 (qui signifie que s est à la distance 0 de s).
On initialise l'ensemble $\mathcal{F}$ des arêtes frontières à l'ensemble des arêtes incidentes à s.
Tant que $\mathcal{F} \neq \varnothing$

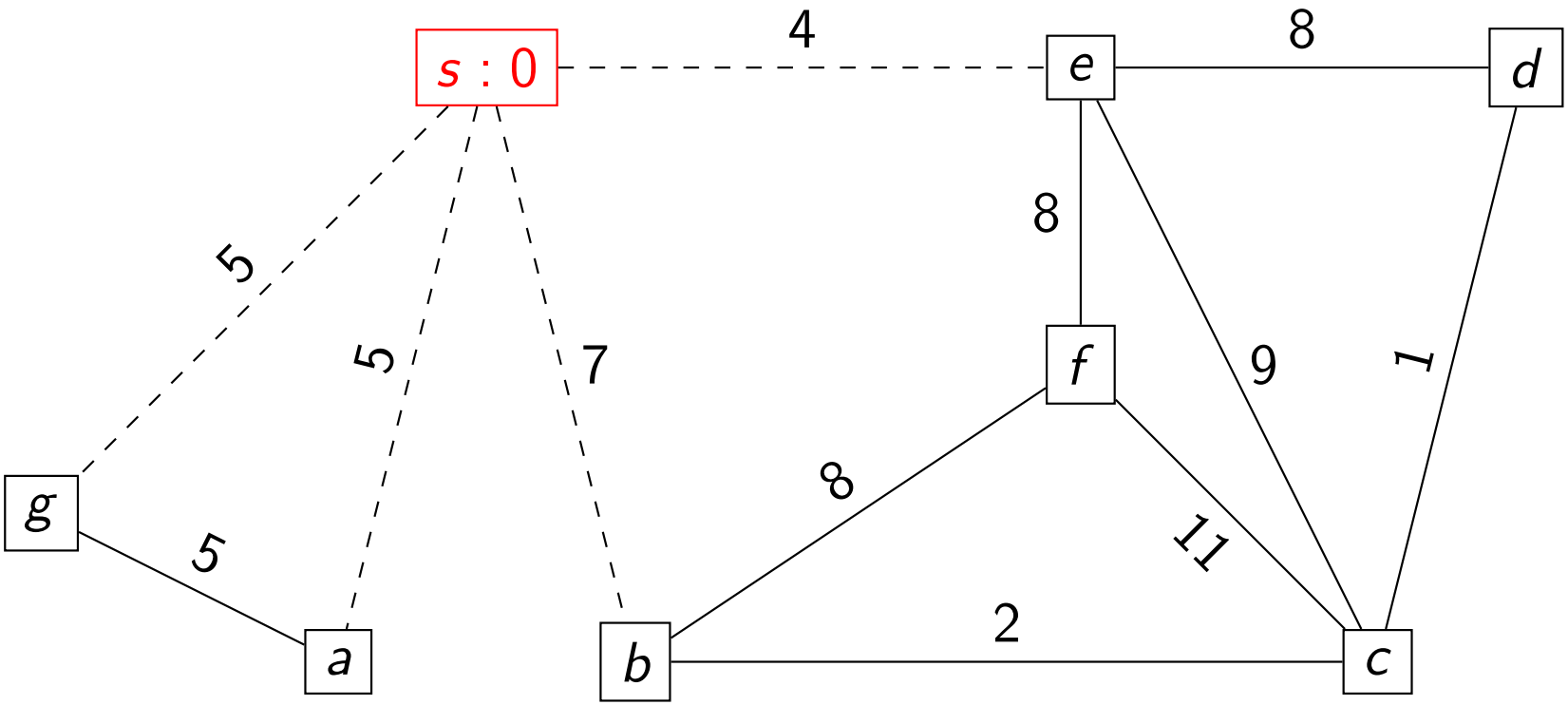
- On marque chaque sommet $v \in (G - T) \cap \text{Voisinage}(T)$ par la plus petite des sommes "marque du sommet u + poids de l'arête u—v" pour $u \in T \cap \text{Voisinage}(v)$. On choisit alors comme ProchaineArêteFrontière δ une arête incidente à un sommet extérieur à T de marque minimum, δ étant une arête réalisant ce minimum.
- $T := T$ auquel on ajoute δ et le sommet de $G - T$ incident à δ.
- On met à jour l'ensemble $\mathcal{F}$ des arêtes frontières.

Algorithme de Dijskstra

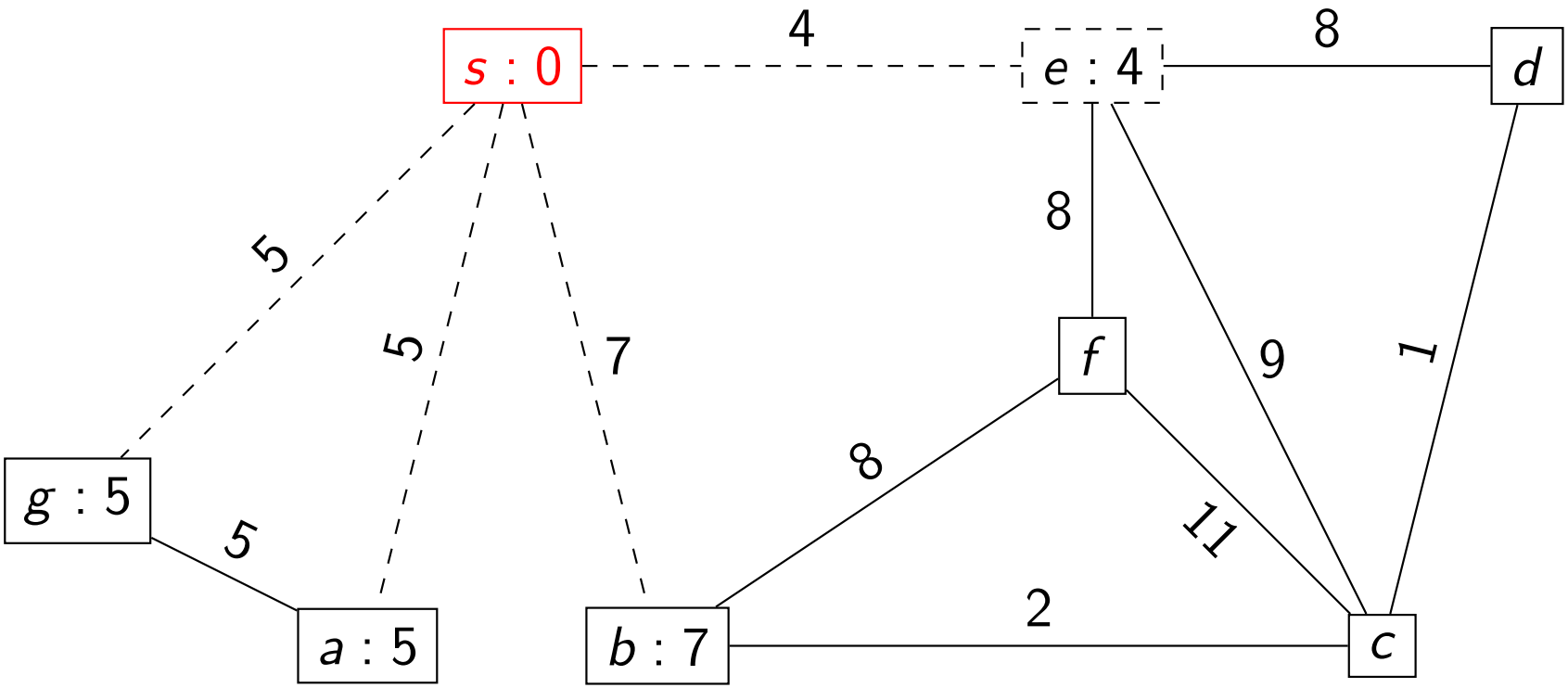
Appliquer cet algorithme au graphe ci-dessous.



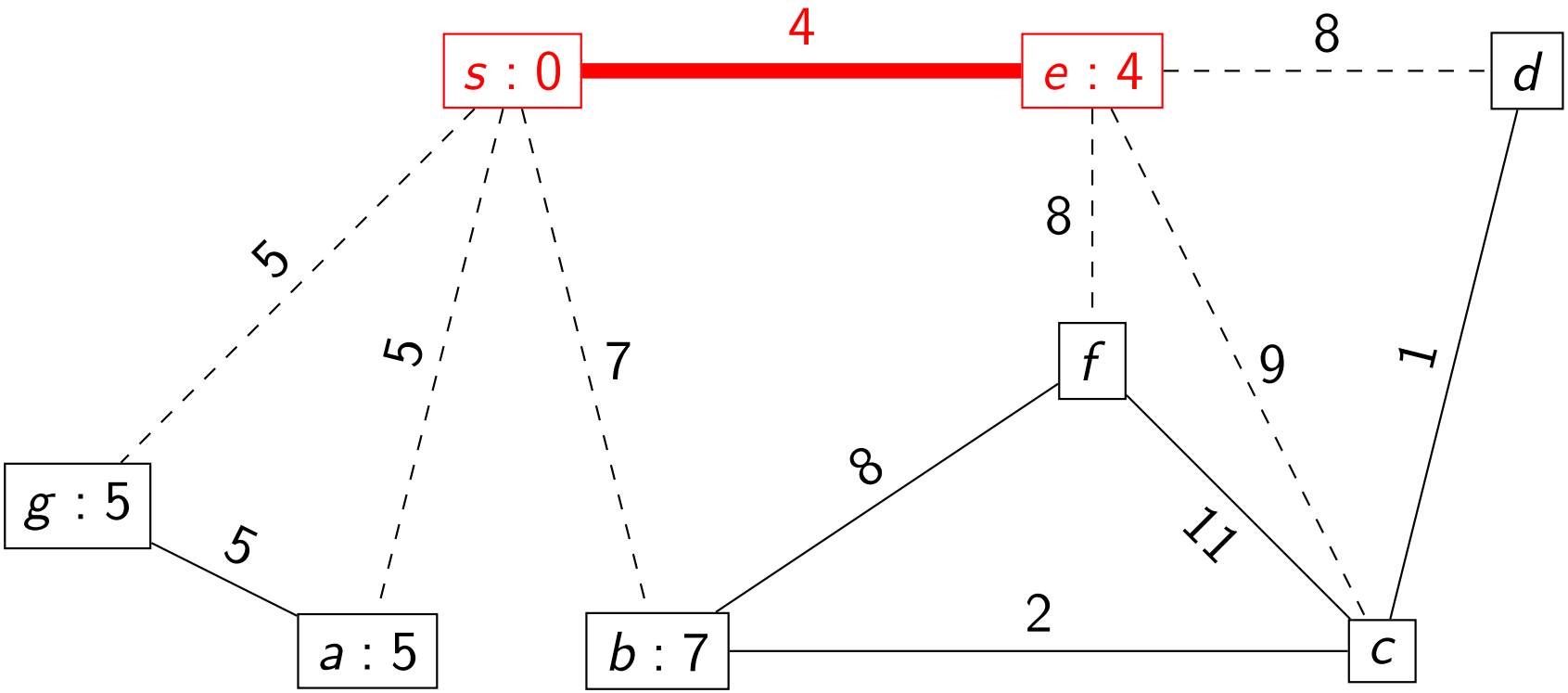
Algorithme de Dijskstra



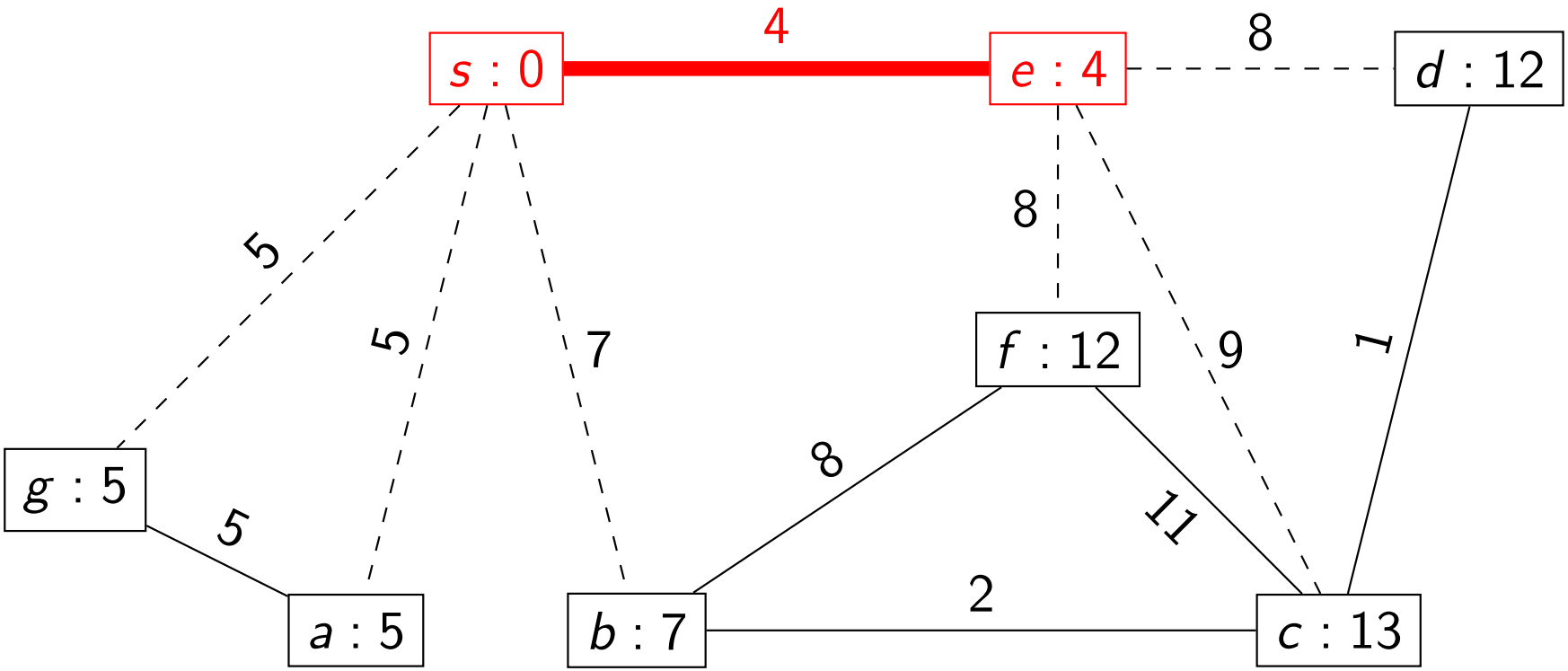
Algorithme de Dijskstra



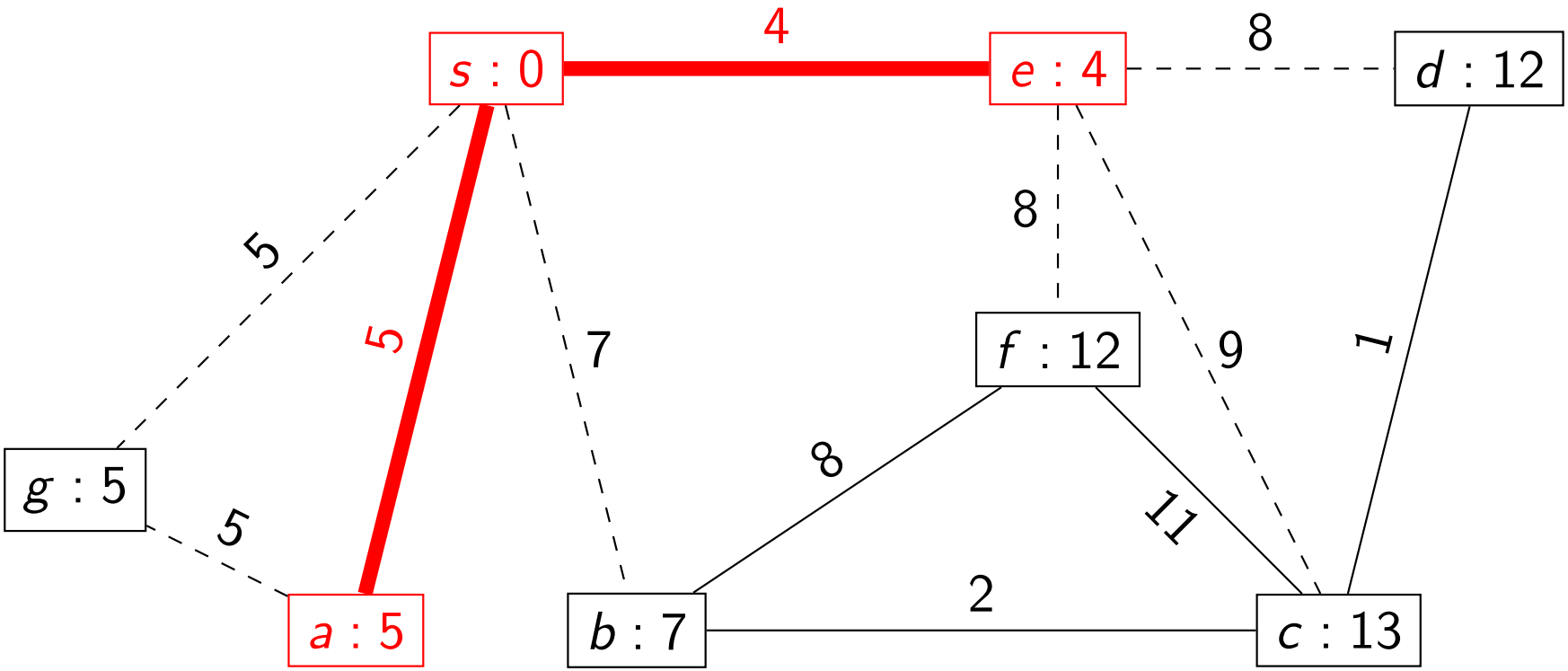
Algorithme de Dijskstra



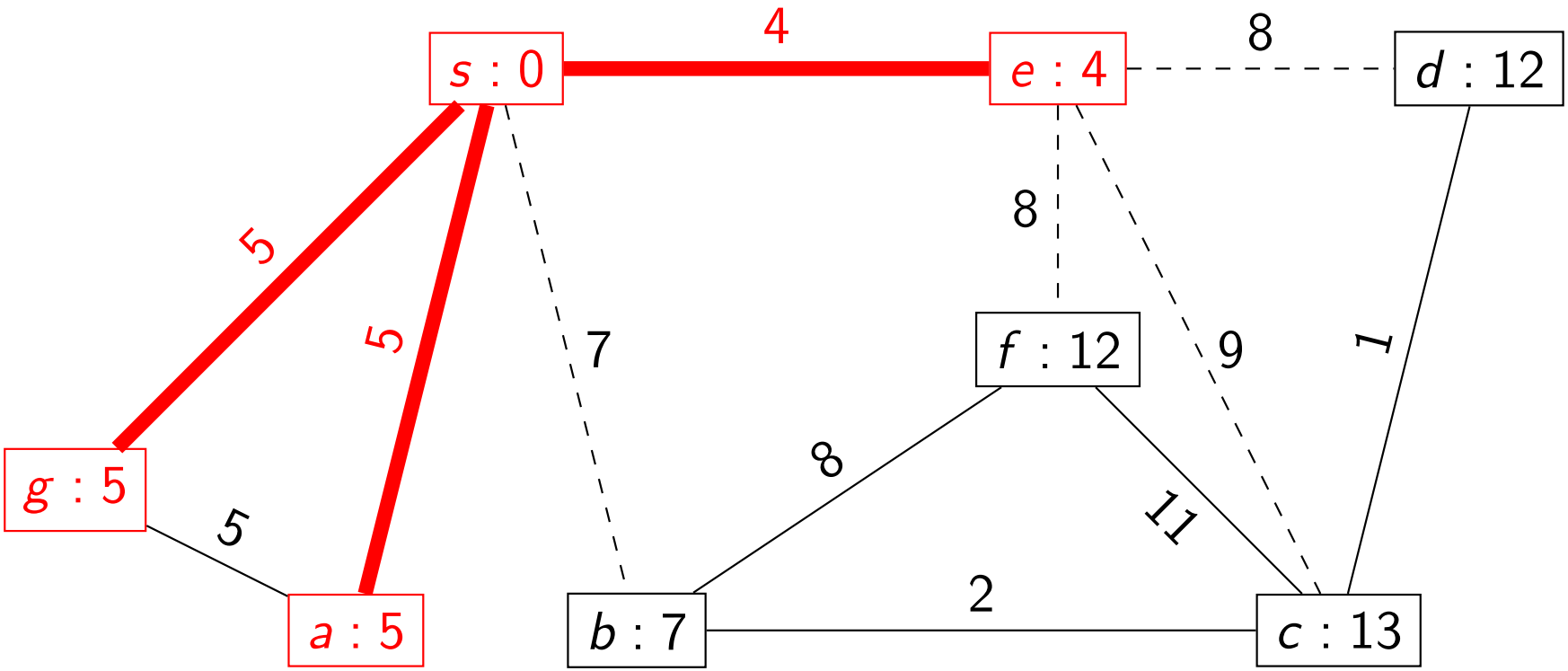
Algorithme de Dijskstra



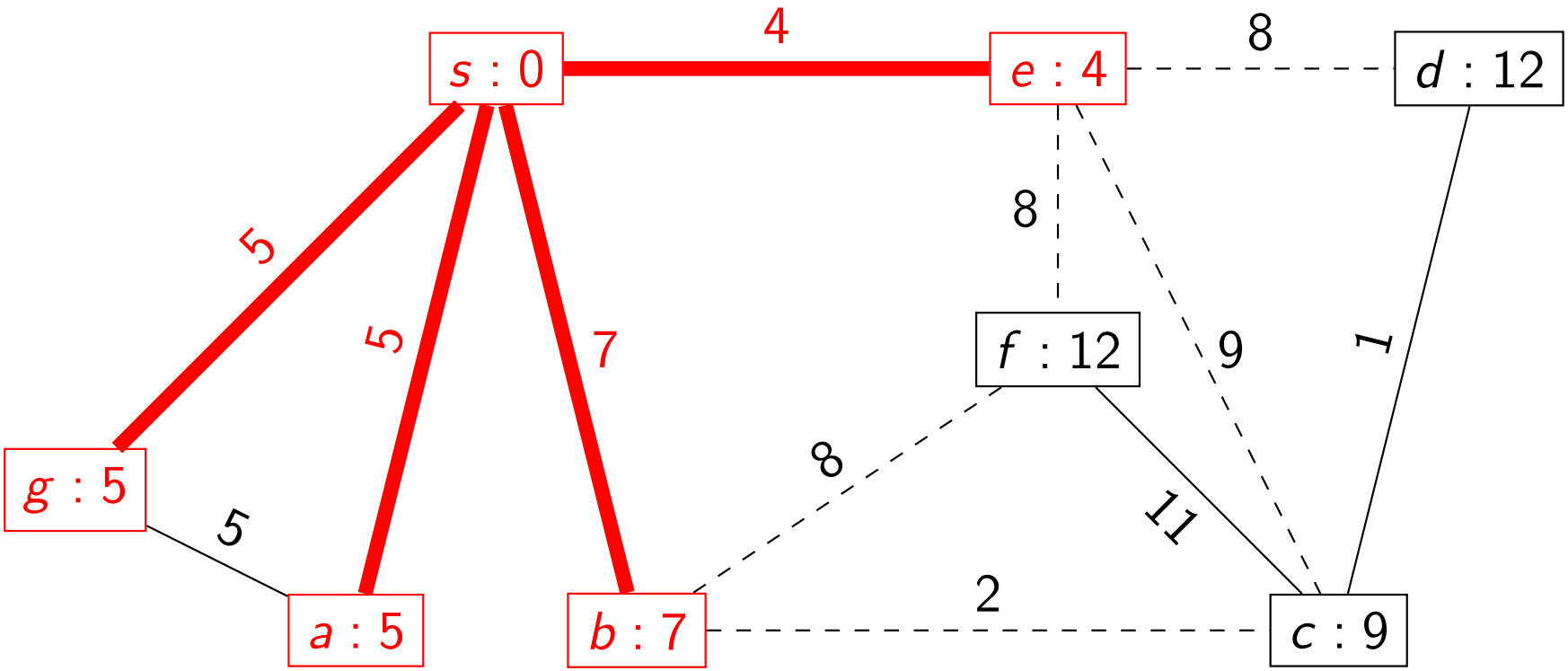
Algorithme de Dijskstra



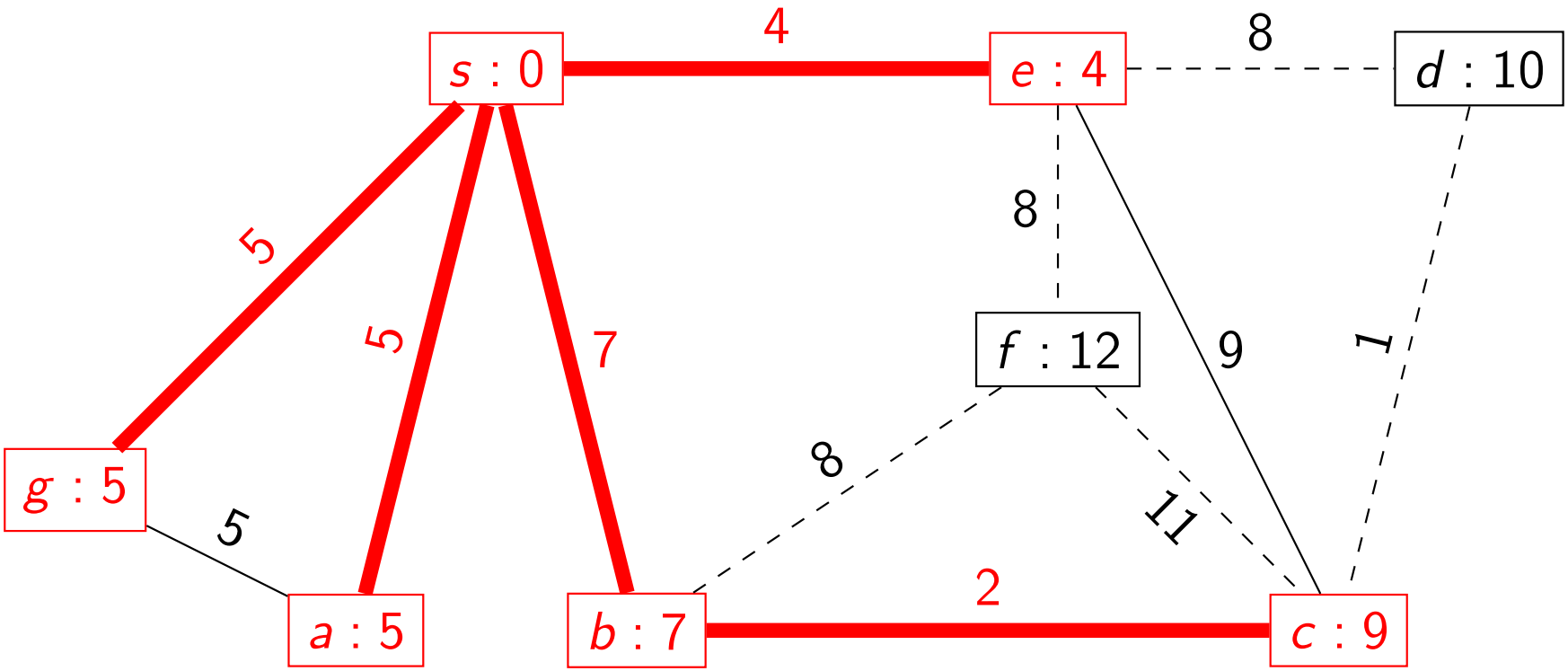
Algorithme de Dijskstra



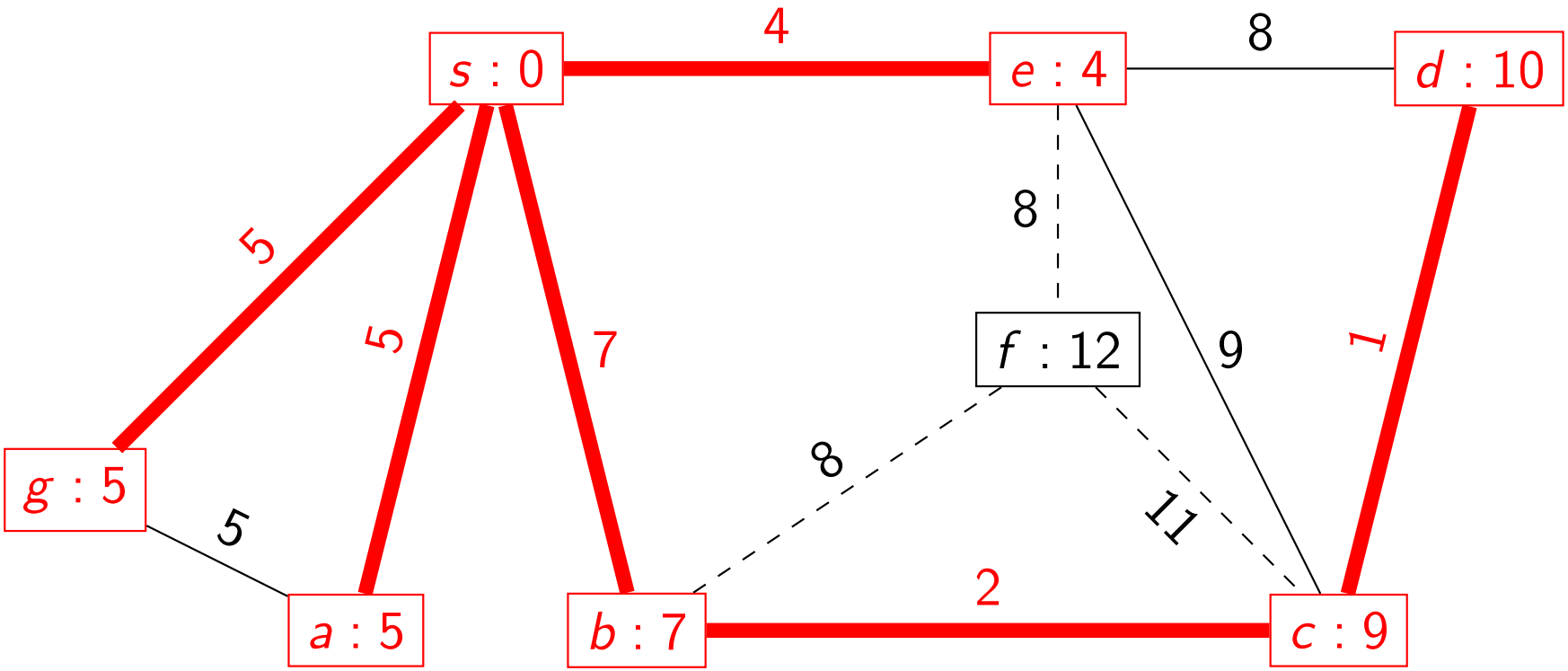
Algorithme de Dijskstra



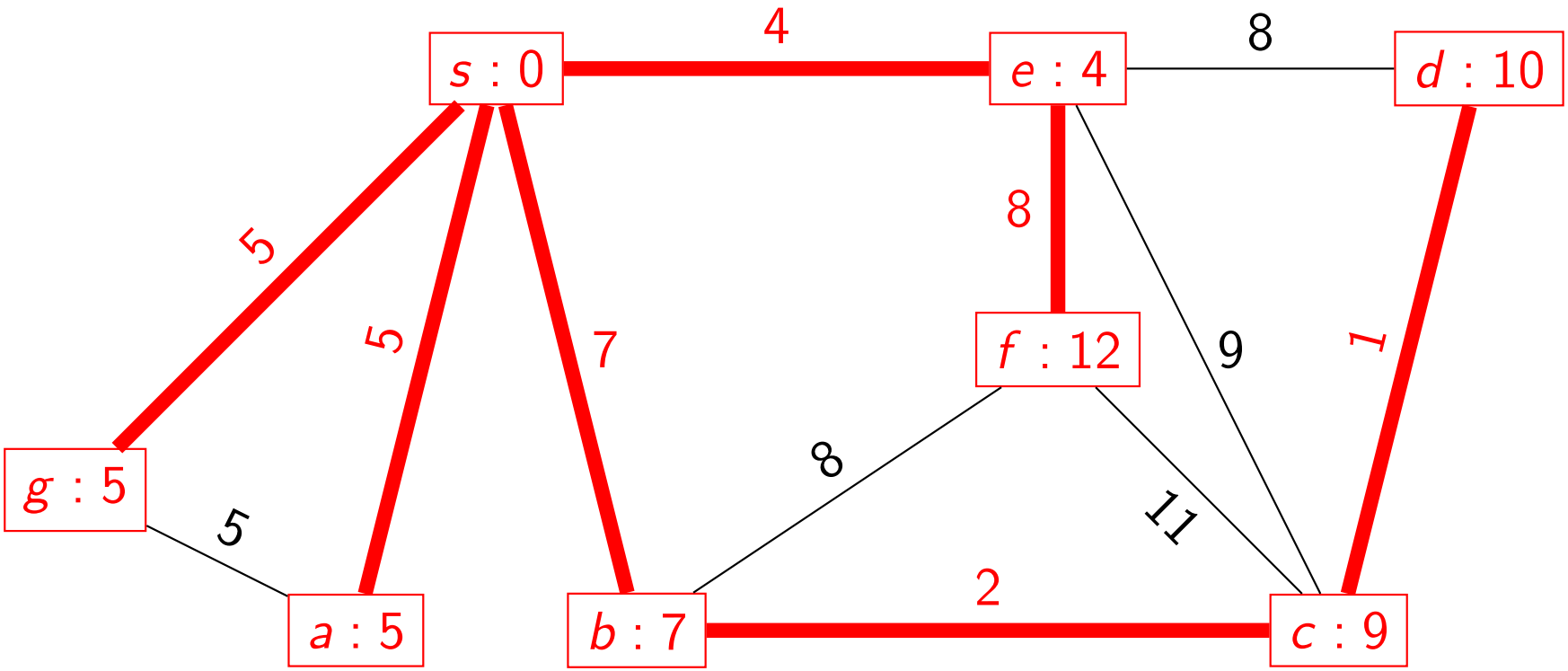
Algorithme de Dijskstra



Algorithme de Dijskstra



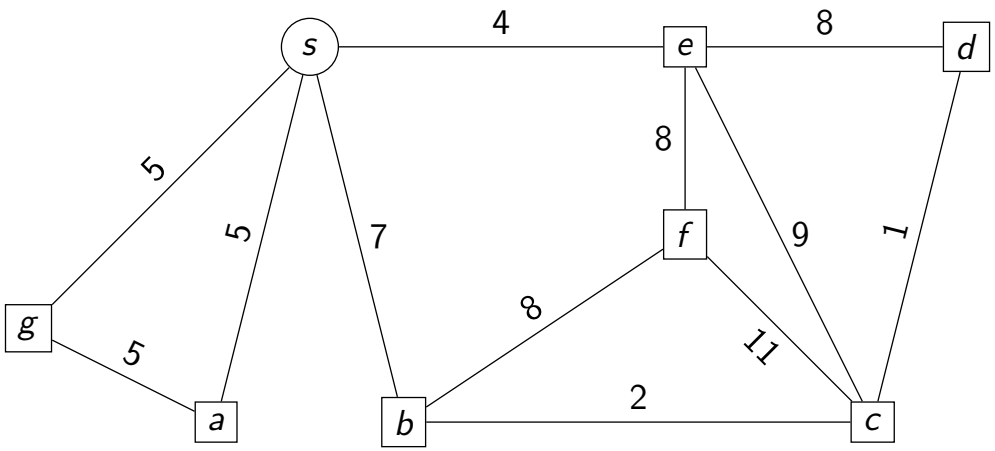
Algorithme de Dijskstra



Programmation

Pour programmer l'algorithme, on décide de représenter le graphe précédent par une matrice :

0	0	0	0	0	0	5	5	a
0	0	2	0	0	8	0	7	b
0	2	0	1	9	11	0	0	c
0	0	1	0	8	0	0	0	d
0	0	9	8	0	8	0	4	e
0	8	11	0	8	0	0	0	f
5	0	0	0	0	0	0	5	g
5	7	0	0	4	0	5	0	s
a	b	c	d	e	f	g	s	



Codage Xcas

Xcas

```
G      :=[[0,0,0,0,0,0,5,5],      [0,0,2,0,0,8,0,7],      [0,2,0,1,9,11,0,0],
[0,0,1,0,8,0,0,0], [0,0,9,8,0,8,0,4], [0,8,11,0,8,0,0,0], [5,0,0,0,0,0,0,5],
[5,7,0,0,4,0,5,0]]
```

Codage Xcas

On veut écrire une fonction Dijkstr(G,s,f) :

- Input : le graphe G sous la forme matricielle précédente, s le sommet source, f un sommet but (pour simplifier, on arrête dès que le sommet f est dans l'arbre).
- Output : on ne renverra que le chemin de s à f.

Remarque : lorsque cette fonction est écrite, il est facile de la modifier pour obtenir l'arbre en entier.

Codage Xcas – Initialisation

On utilisera :

Xcas

```
n :=rowdim(G);/* n est le nombre de sommets du graphe.*/
DP :=[(+infinity) $ n]// tableau des n Distances (Provisoires puis Pérennes) attribuées à chaque sommet.
dsArbre :=[faux $ n]// dsArbre[j]==vrai signifiera que le sommet j est dans l'arbre.
pere :=["vide" $ n]// pere[j]==k signifiera que l'arête k—j (k étant atteint avant j) est dans l'arbre.
//Au départ, la longueur de s à s vaut 0 et ce sommet s est mis dans l'arbre :
DP[s] :=0; dsArbre[s] :=vrai;
```

Codage Xcas

Exercice. Écrire la boucle principale mettant à jour les marques sur chaque sommet ainsi que les sommets prédécesseurs tant que f n'est pas dans l'arbre.

Codage Xcas

Xcas

```
elu :=s ;
//Tant que le sommet final f n'est pas achevé, on continue
tantque non(dsArbre[f]) faire
  pour j de 0 jusque n-1 faire
  //on regarde si les DP des voisins (non achevés) du sommet elu
peuvent être plus courtes :
  si G[elu,j]>0 et DP[elu]+G[elu,j]<DP[j] et non(dsArbre[j]) alors
  DP[j] :=DP[elu]+G[elu,j] ;// diminution de la DP du sommet j
  pere[j] :=elu ;// mise à jour de l'arête incidente à j
  fsi ;
  fpour ;
```

Codage Xcas

Xcas

```
/*On cherche le prochain elu, c'est à dire le sommet non achevé de plus petite DP : */
mini :=+infinity;
pour j de 0 jusque n-1 faire
  si non(dsArbre[j]) et DP[j]<mini alors
  mini :=DP[j];
  elu :=j;
  fsi;
fpour;
dsArbre[elu] :=vrai; //elu est mis dans l'arbre, le P de DP passe de Provisoire à Permanent.
ftantque;
```

Codage Xcas

On parcourt le chemin à l'envers de f à s et on affiche ce chemin.

Xcas

```
k :=f ; res :=[f] ;
tantque k !=s faire
k :=pere[k] ;
res :=append(res,k) ;
ftantque ;
res :=revlist(res) ;
retourne "Le chemin le plus court du sommet "+ s+ " au sommet "+
f+" coûte " + DP[f] + " et s'effectue en passant par les sommets "
+ res ;
// fin de la fonction Dijsktr(G,s,f) } : ;
```

Complexité

Le programme précédent a une boucle principale en S^2 (où S est le nombre de sommets). Si on ajoute une boucle pour traiter tous les sommets (en ne s'arrêtant pas à f), on aurait donc ici une implémentation en S^3.

On peut en fait implémenter l'algorithme de Dijkstra avec une complexité en $O(S \log(S) + A)$ où S est le nombre de sommets et A le nombre d'arêtes.

Validité

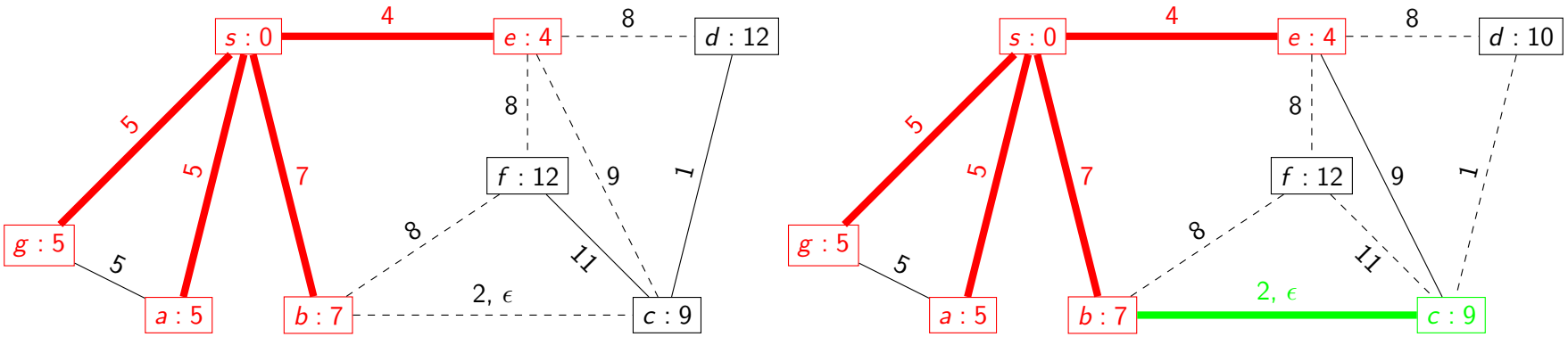
Exercice.
Établir la validité de l'algorithme précédent en démontrant l'invariant de boucle suivant :

Soit T_j l'arbre obtenu après j itérations de l'algorithme.
Pour chaque sommet w de T_j, l'unique chaîne élémentaire du sommet source s à w dans T_j est une plus courte chaîne de s à w dans G.

Initialisation : pour $j = 0$, T_0 est réduit à s et la propriété est satisfaite.

Validité

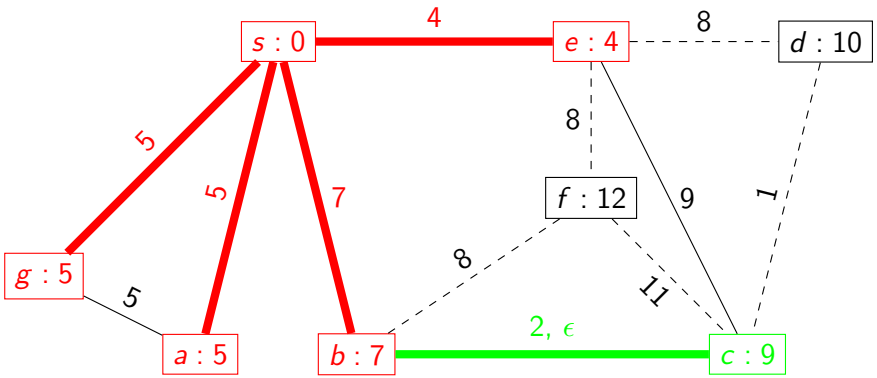
Supposons qu'elle soit vraie pour un certain j ($0 \leqslant j \leqslant |V_G| - 2$). Notons ϵ d'extrémités $b \in T_j$ et $c \notin T_j$ l'arête ajoutée à T_j pour obtenir T_{j+1}.

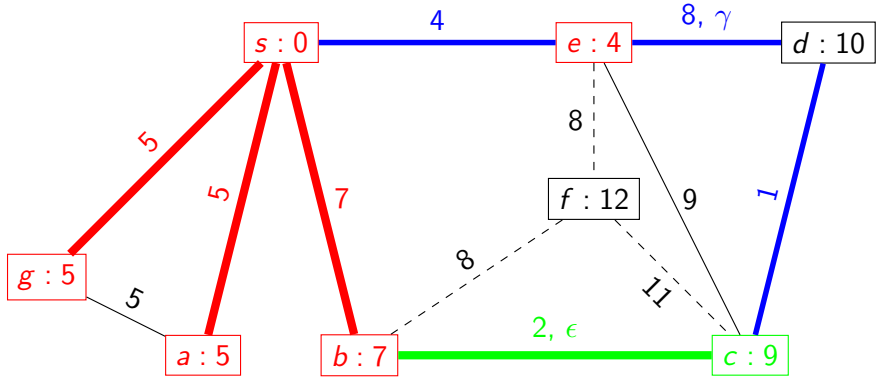


Il s'agit d'établir que l'unique chaîne élémentaire Q de T_{j+1} reliant s à c est une plus courte chaîne dans G. Cette chaîne Q a pour longueur la somme de la longueur de l'unique chaîne de s à b dans T_j et du poids de l'arête ϵ, c'est à dire (compte tenu de l'hypothèse de récurrence) $\text{dist}_G(s, b) + \text{poids}(\epsilon)$.

Validité

Notons R une chaîne de s à c dans G et cherchons à établir que la longueur de cette chaîne est au moins celle de Q.





Notons γ d'extrémités e et d la première arête dans le parcours de cette chaîne R de s vers c qui ne soit pas une arête de l'arbre T_j.
Notons également R' la sous-chaîne de R de d à c.

Validité

A la fin de l'itération j, les arêtes γ et ϵ sont des arêtes frontières. Comme l'algorithme sélectionne ϵ, on a :
$\text{dist}_G(s,e) + \text{poids}(\gamma) \geqslant \text{dist}_G(s,b) + \text{poids}(\epsilon)$. D'où :

$$\begin{aligned}\text{longueur}(R) &= \text{dist}_G(s,e) + \text{poids}(\gamma) + \text{longueur}(R') \\ &\geqslant \text{dist}_G(s,b) + \text{poids}(\epsilon) + \text{longueur}(R') \\ &\geqslant \text{longueur}(Q)\end{aligned}$$

Référence

Introduction à l'algorithmique.
Cormen, Leiserson, Rivest, Stein.
Éditions Dunod.