Gli alberi ricoprenti minimi

Gianpiero Cabodi, Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Alberi ricoprenti minimi

Dato $G=(V,E)$ grafo non orientato, pesato con pesi positivi $w: E \rightarrow \boldsymbol{R^+}$ e connesso, estrarre da G un

Albero ricoprente minimo

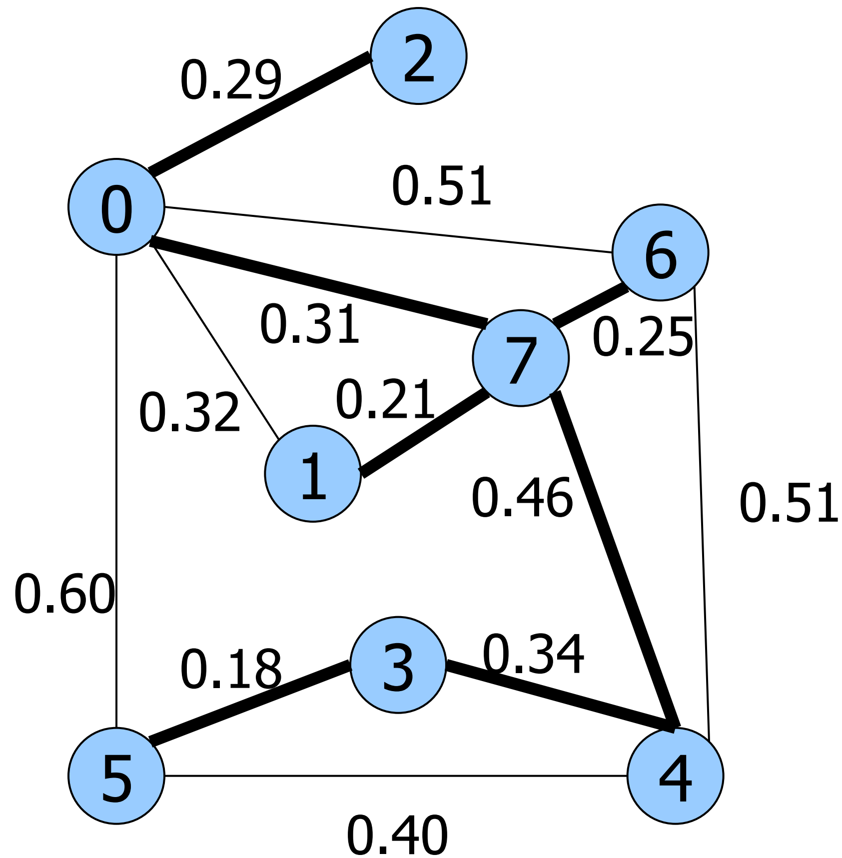
(Minimum-weight Spanning Tree – MST) :

- grafo $G'=(V, T)$ dove $T \subseteq E$
- aciclico
- minimizza $w(T)=\Sigma\ w(u,v)$.

Aciciclità && copertura di tutti i vertici $\Rightarrow$ G' è un albero.

L'albero MST è unico se e solo se tutti i pesi sono distinti.

Esempio



Rappresentazione

ADT grafo non orientato e pesato: estensione dell'ADT grafo non orientato:

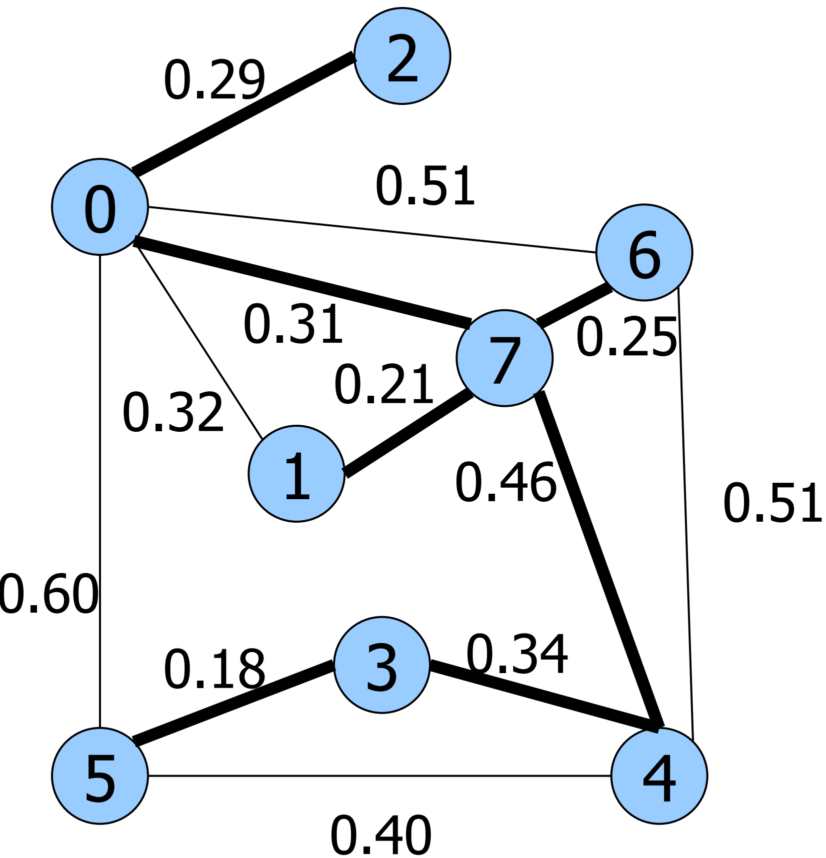
- lista delle adiacenze
- matrice delle adiacenze

Valore-sentinella per indicare l'assenza di un arco (peso inesistente):

- `maxWT` (idealmente $+\infty$), soluzione scelta nell'algoritmo di Prim
- 0 se non sono ammessi archi a peso 0
- -1 se non sono ammessi archi a peso negativo.

Per semplicità si considerano pesi interi e non reali.

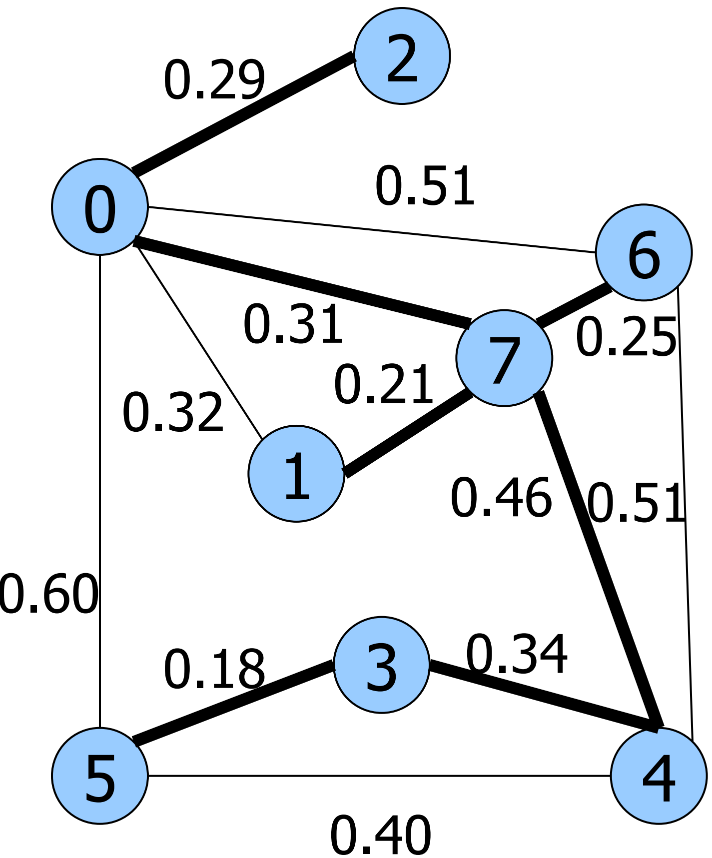
Rappresentazione degli MST

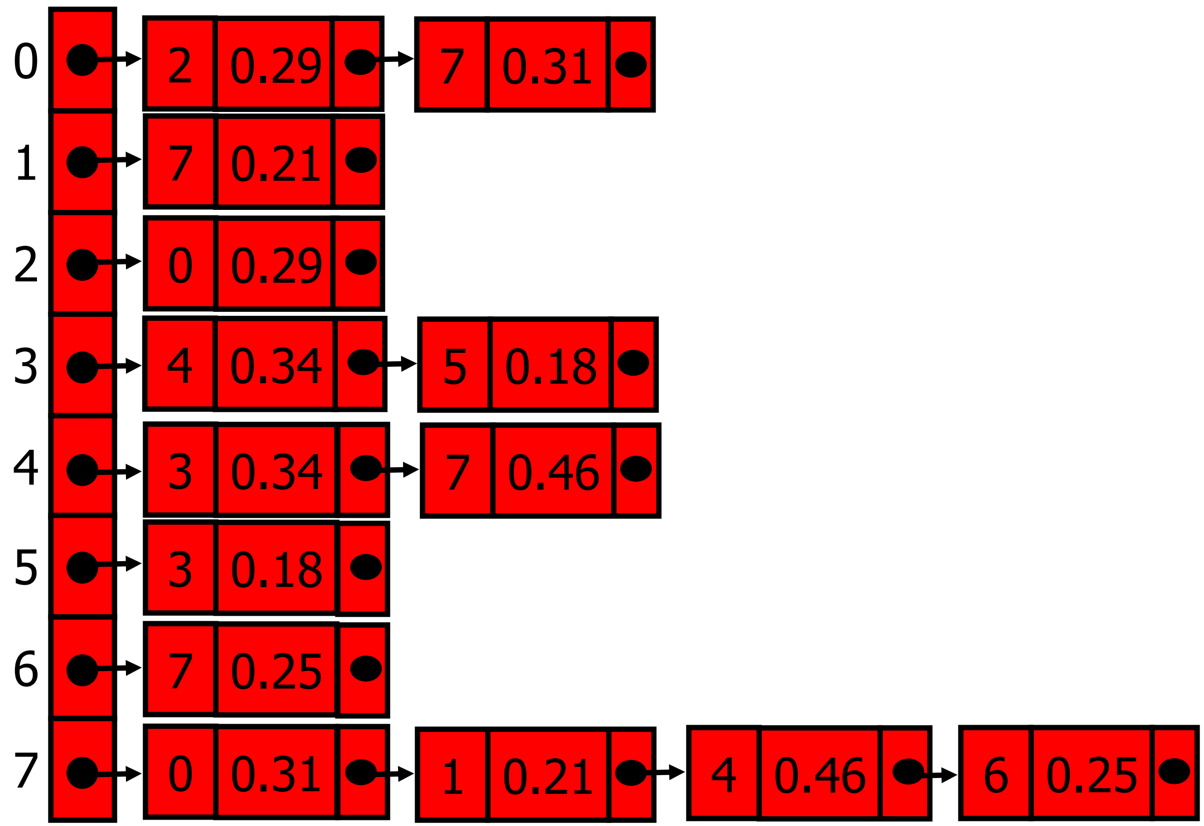


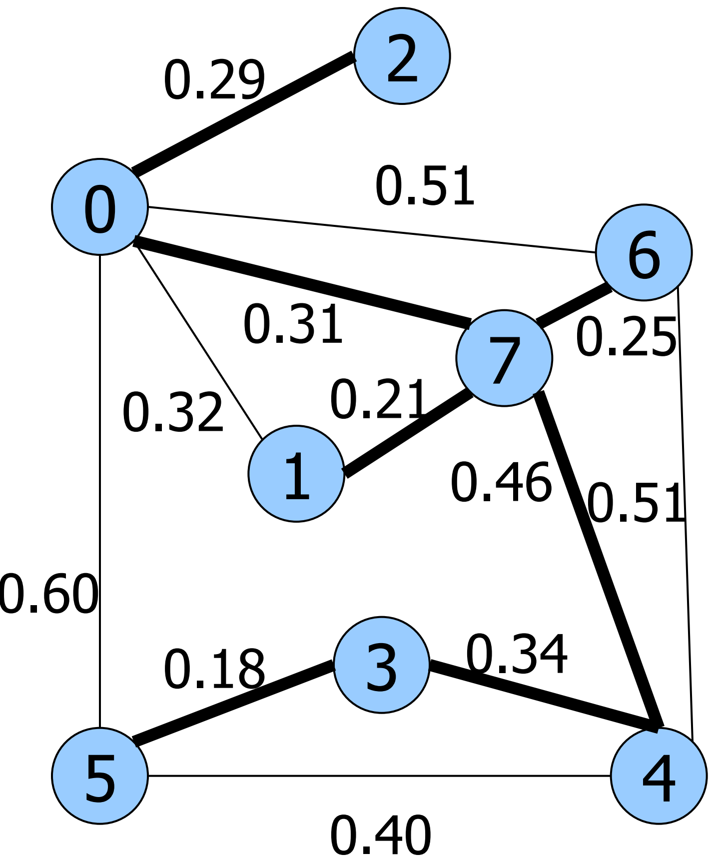
Algortimo di Kruskal: elenco di archi, memorizzato in un vettore di archi `mst[maxE]`

0-2	0.29
4-3	0.34
5-3	0.18
7-4	0.46
7-0	0.31
7-6	0.25
7-1	0.21

Grafo come lista di adiacenze:







Algoritmo di Prim: vettore `st` dei padri e `wt` dei pesi

	0	1	2	3	4	5	6	7
`st`	0	7	0	4	7	3	7	0
`wt`	0	.21	.29	.34	.46	.18	.25	.31

Approccio greedy

Approccio greedy:

- ad ogni passo, scelta della soluzione localmente ottima
- non garantisce soluzione globalmente ottima.

Algoritmo generico

- A (=insieme di archi) = sottoinsieme di albero ricoprente minimo, inizialmente vuoto
- fintanto che A non è un albero ricoprente minimo, aggiungi ad A un arco "sicuro"

Invarianza: l'arco (u,v) è *sicuro* se e solo se aggiunto ad un sottoinsieme di un albero ricoprente minimo produce ancora un sottoinsieme di un albero ricoprente minimo.

Tagli e archi

G=(V,E) grafo non orientato, pesato, connesso.

Taglio = partizione di V in S e V-S

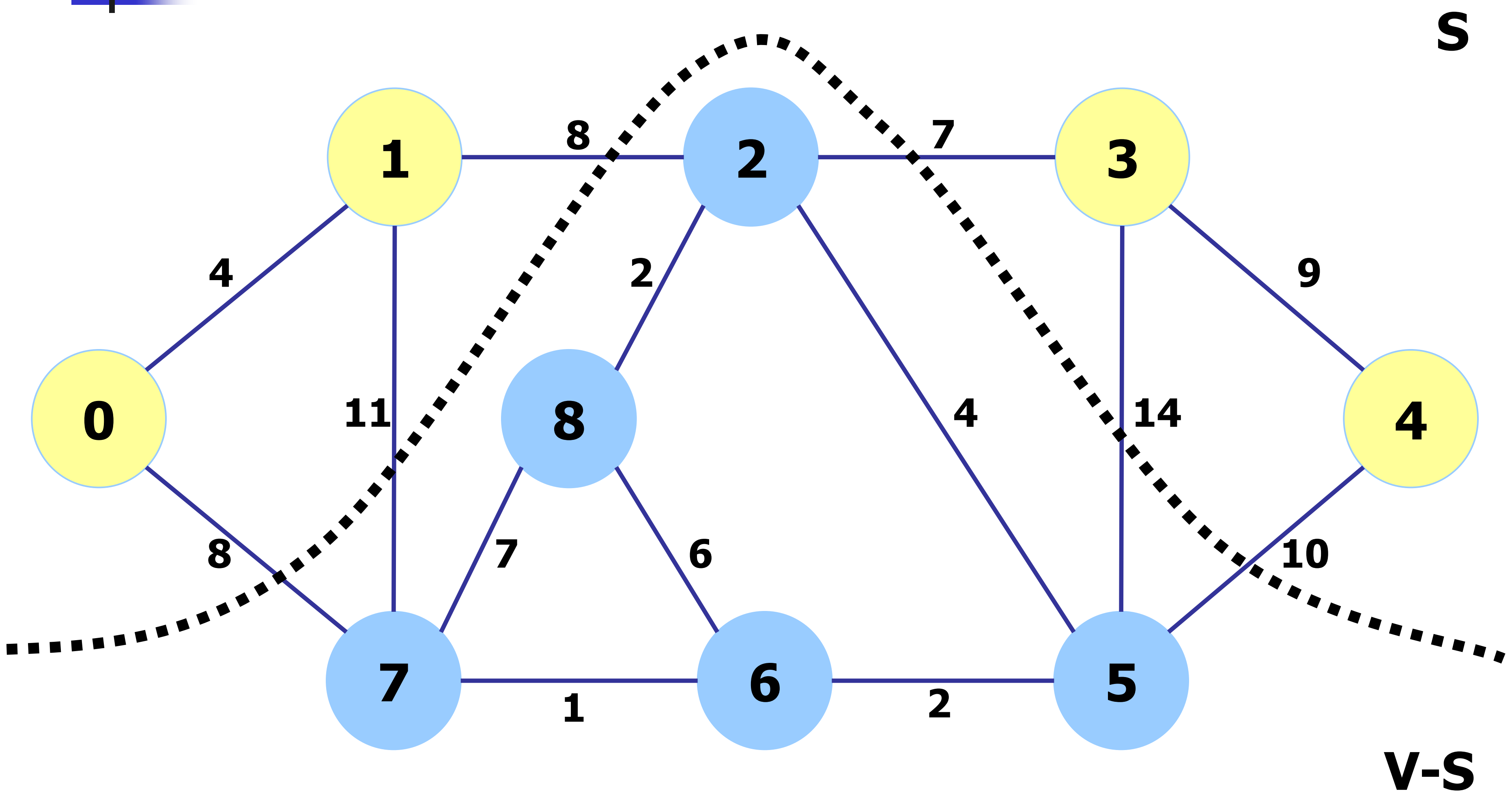
$$V = S \cup V\text{-}S \;\; \&\& \;\; S \cap V\text{-}S = \emptyset$$

$(u,v) \in E$ attraversa il taglio $\Leftrightarrow u \in S \;\&\&\; v \in V\text{-}S$ (o viceversa).

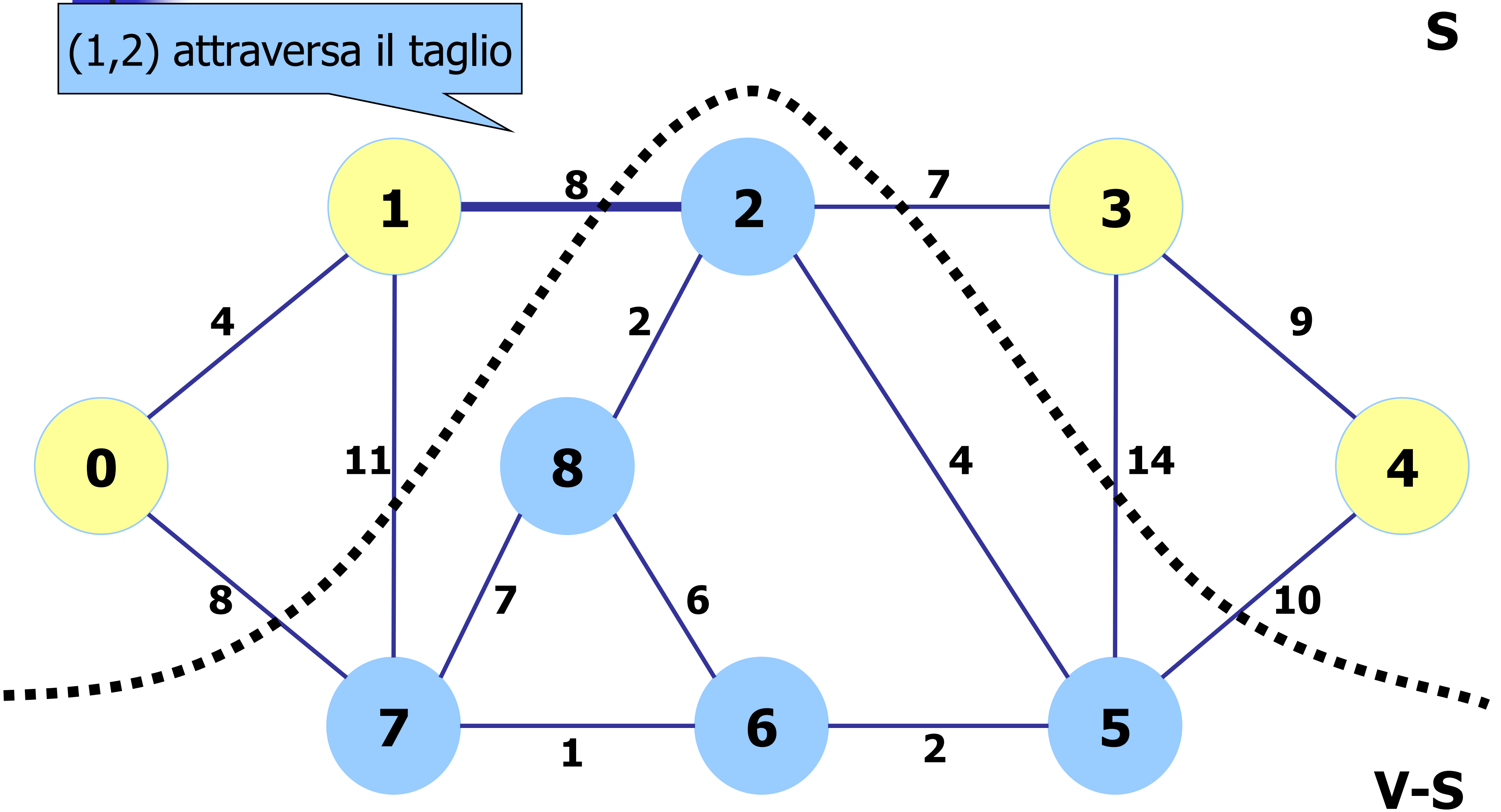
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

Un arco si dice leggero se ha peso minimo tra gli archi che attraversano il taglio.

Esempio

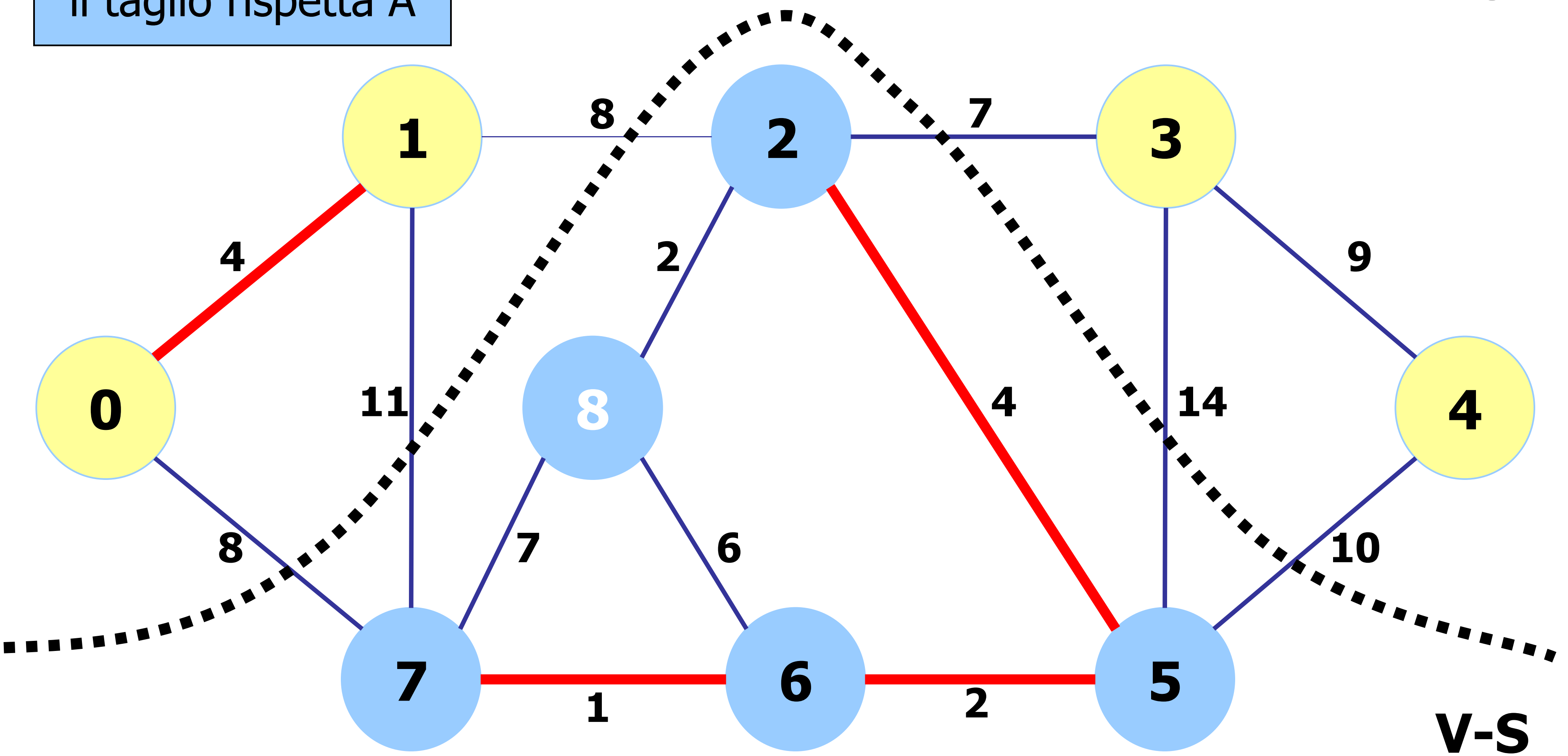


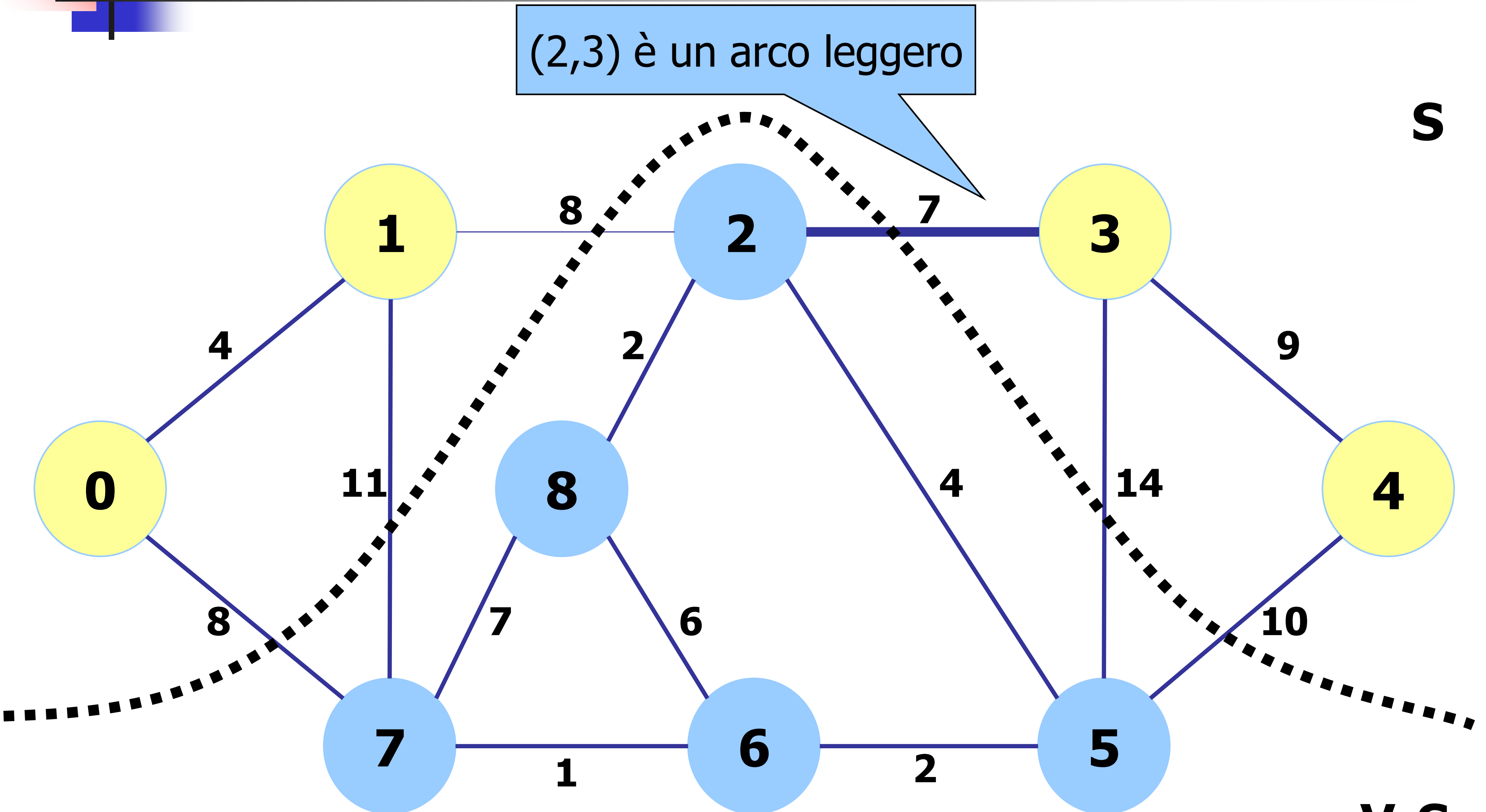
(1,2) attraversa il taglio



Posto che A sia A = {(0,1), (2,5), (5,6), (6,7)}

il taglio rispetta A




(2,3) è un arco leggero
S
1
8
2
7
3
4
2
9
0
11
8
4
14
4
8
7
6
10
7
1
6
2
5
V-S

Archi sicuri: teorema

G=(V,E) grafo non orientato, pesato, connesso, A sottoinsieme degli archi. Se:

- A $\subseteq$ E contenuto in un qualche albero ricoprente minimo di G. Inizialmente A è vuoto
- (S,V-S) taglio qualunque che rispetta A
- (u,v) un arco leggero che attraversa (S,V-S)

$\Rightarrow$ (u,v) è sicuro per A.

Archi sicuri: corollario

$G=(V,E)$ grafo non orientato, pesato, connesso ,
A sottoinsieme degli archi. Se:

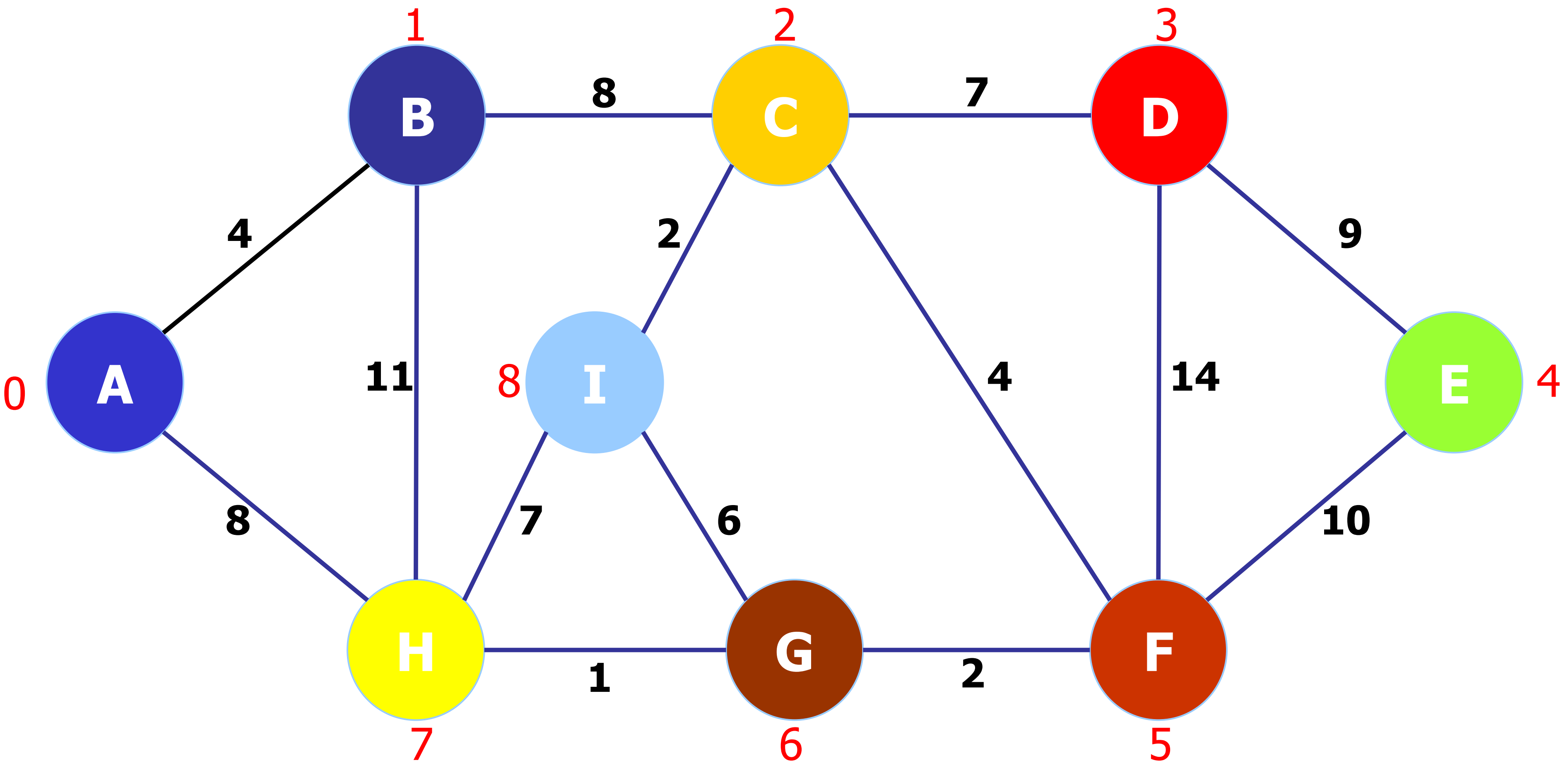
- $A \subseteq E$ contenuto in un qualche albero ricoprente minimo di G. Inizialmente A è vuoto
- C albero nella foresta $G_A= (V,A)$
- (u,v) un arco leggero che connette C ad un altro albero in G_A

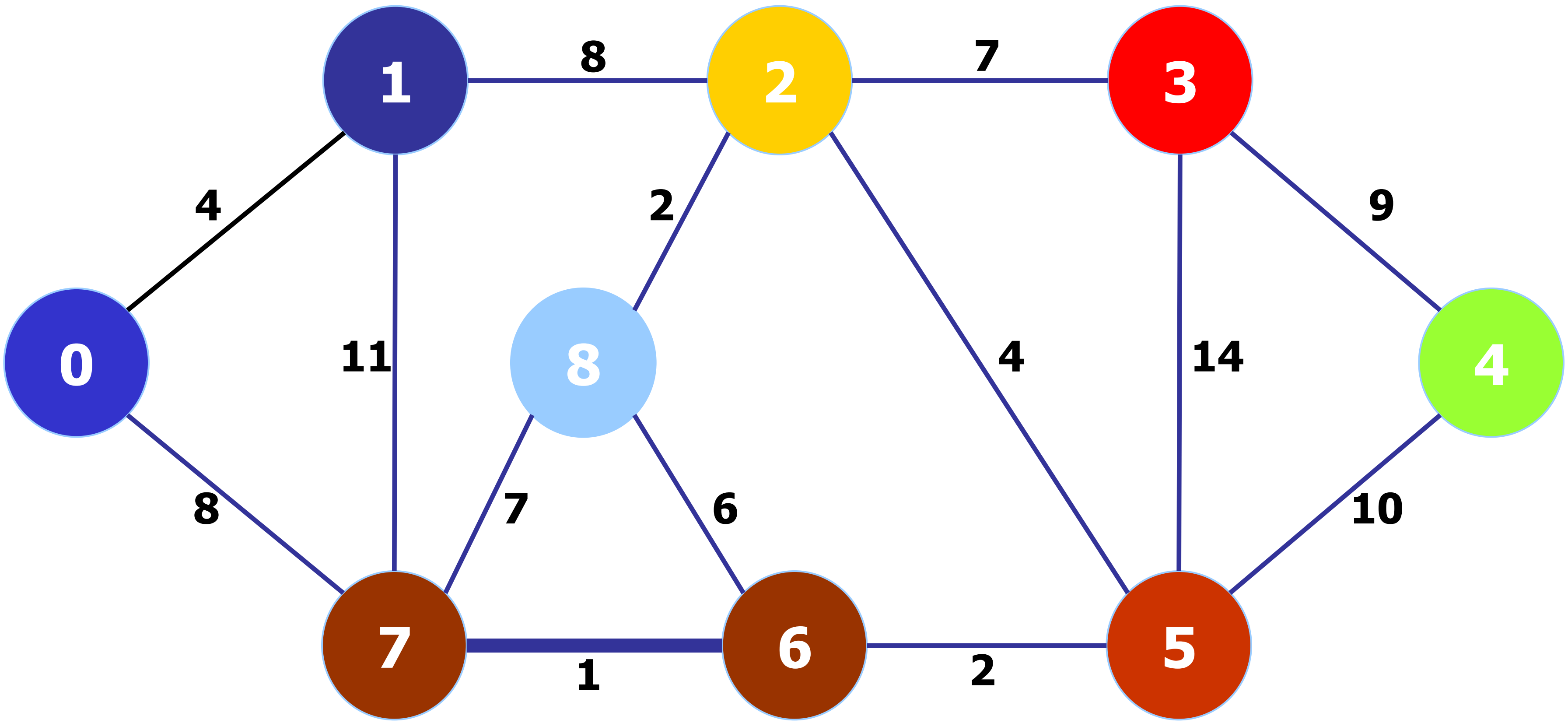
$\Rightarrow$ (u,v) è sicuro per A.

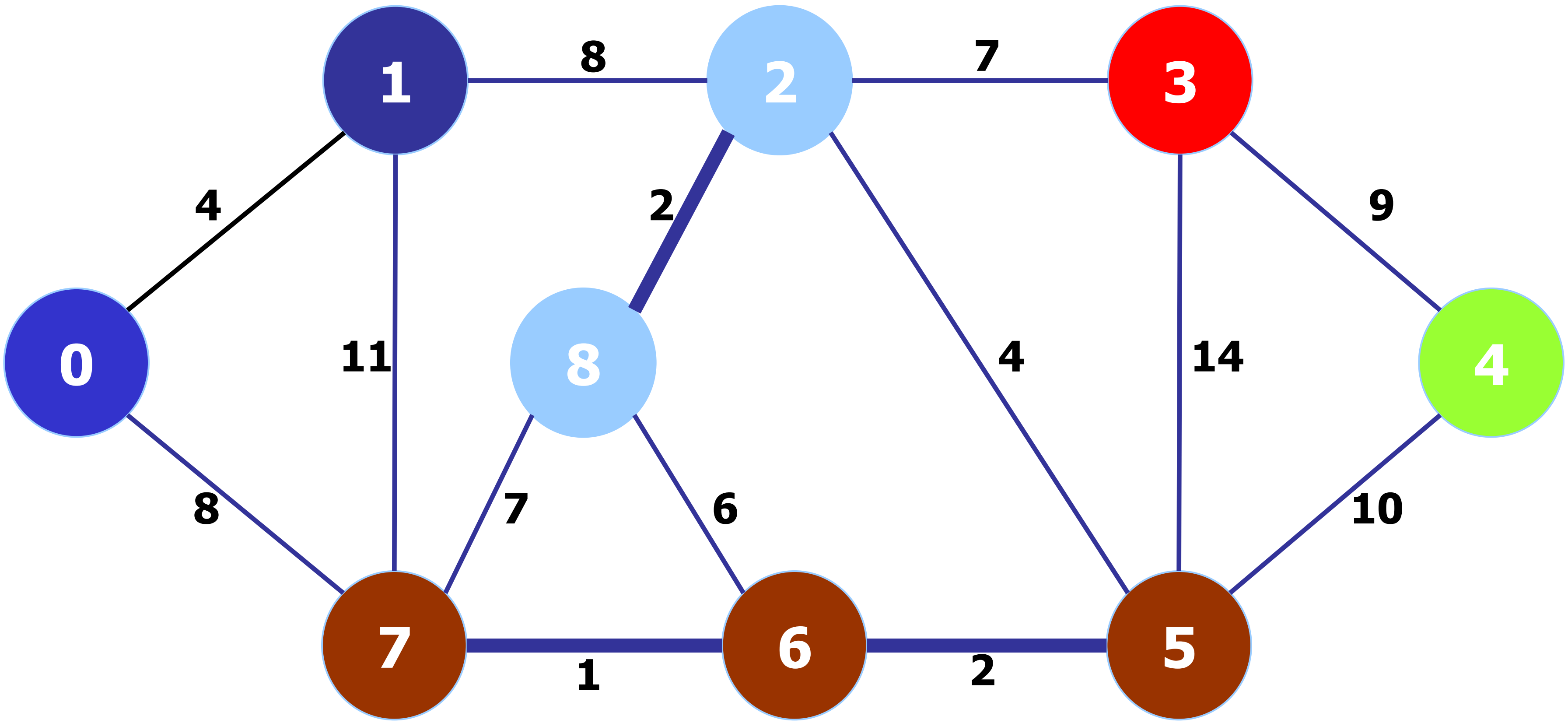
Algoritmo di Kruskal (1956)

- basato su algoritmo generico
- uso del corollario per determinare l’arco sicuro:
  - foresta di alberi, inizialmente vertici singoli
  - ordinamento degli archi per pesi crescenti
  - iterazione: selezione di un arco sicuro: arco di peso minimo che connette 2 alberi generando un albero (Union-Find)
  - terminazione: considerati tutti i vertici.

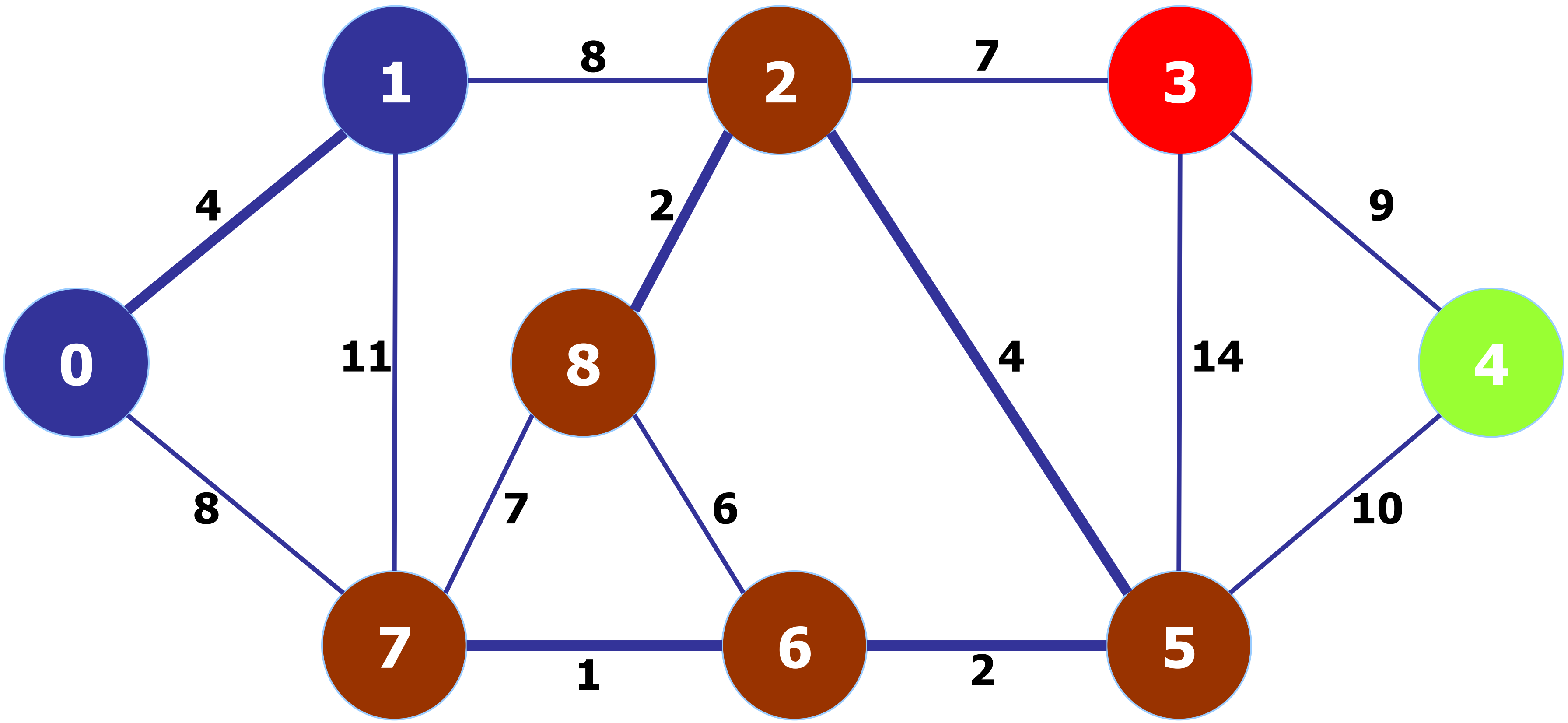
Esempio

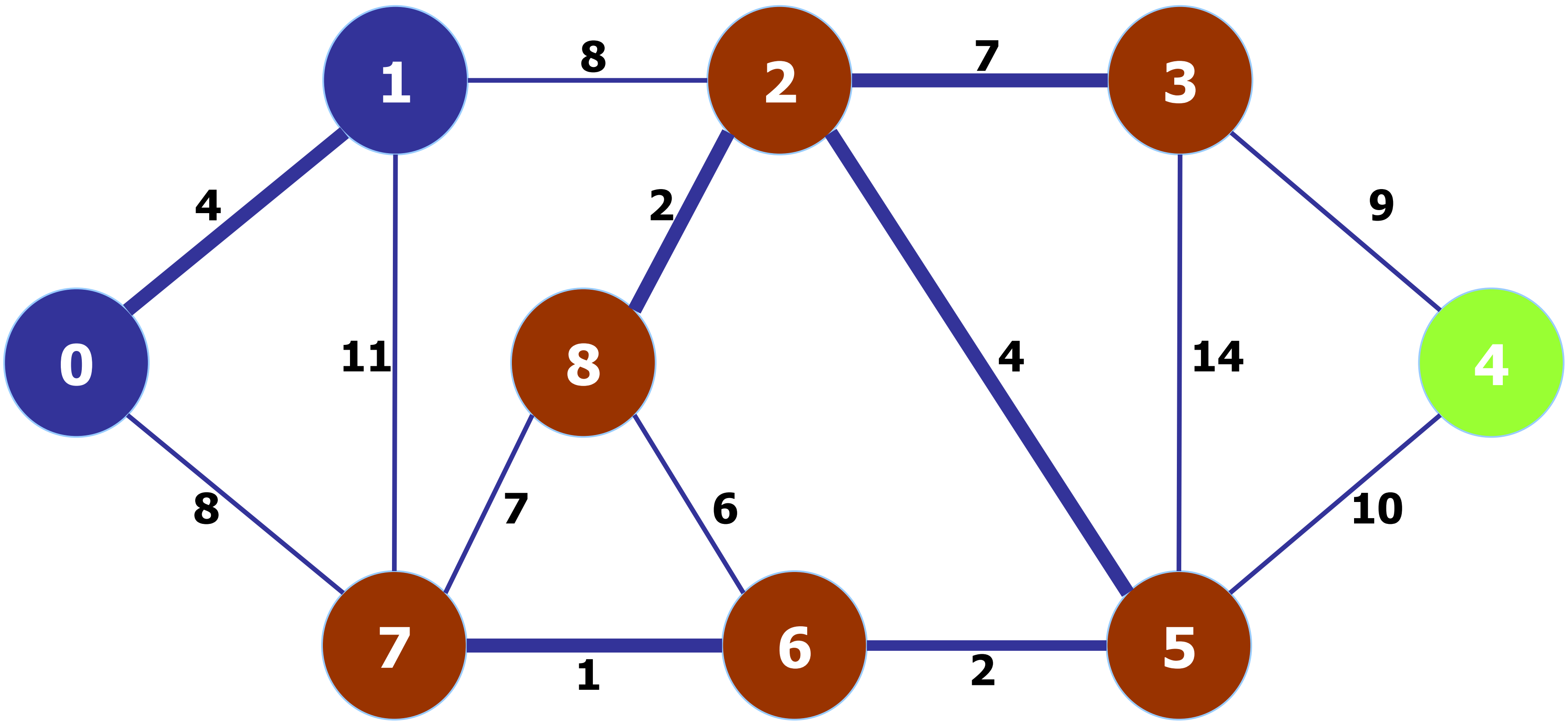


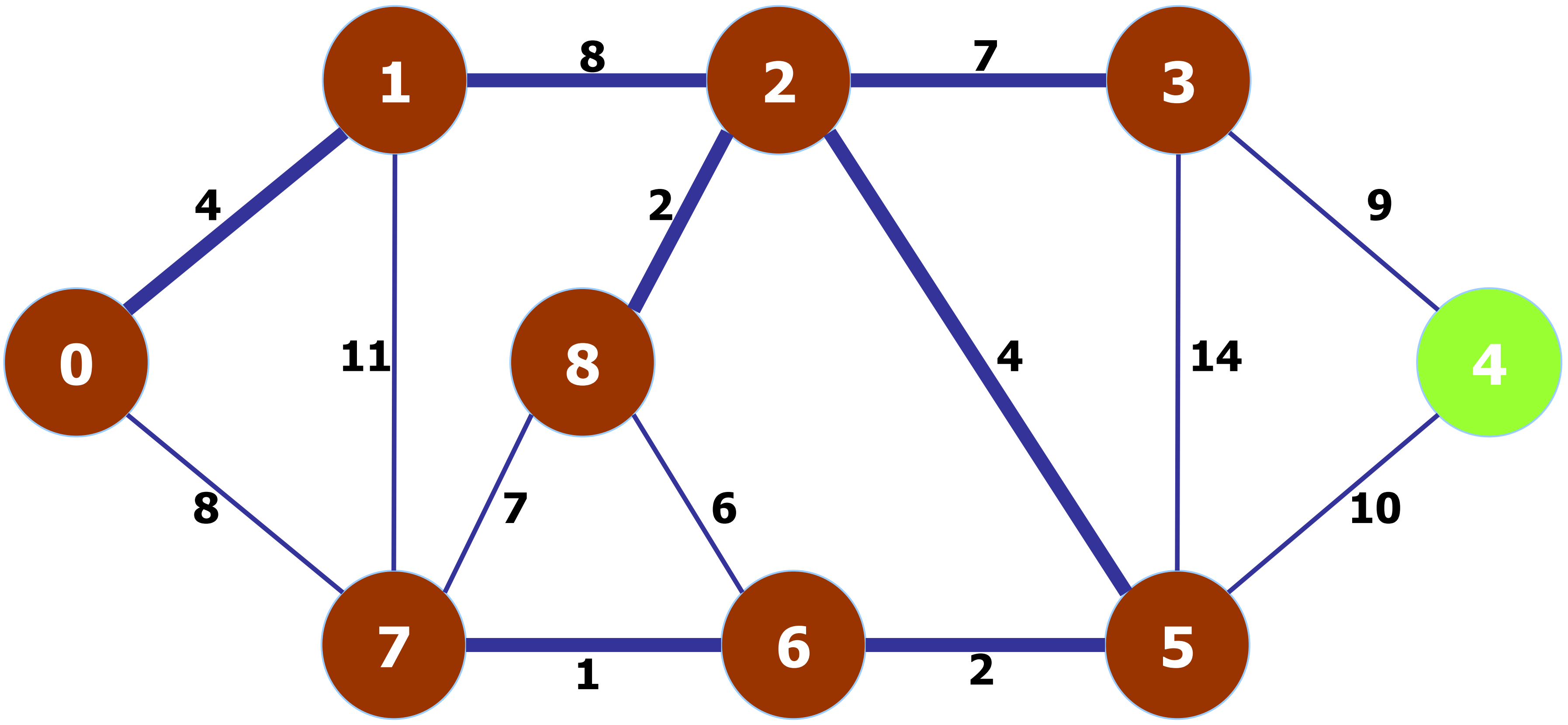

8
7
1
2
3
4
2
9
0
11
8
4
14
4
8
7
6
10
7
6
5
1
2

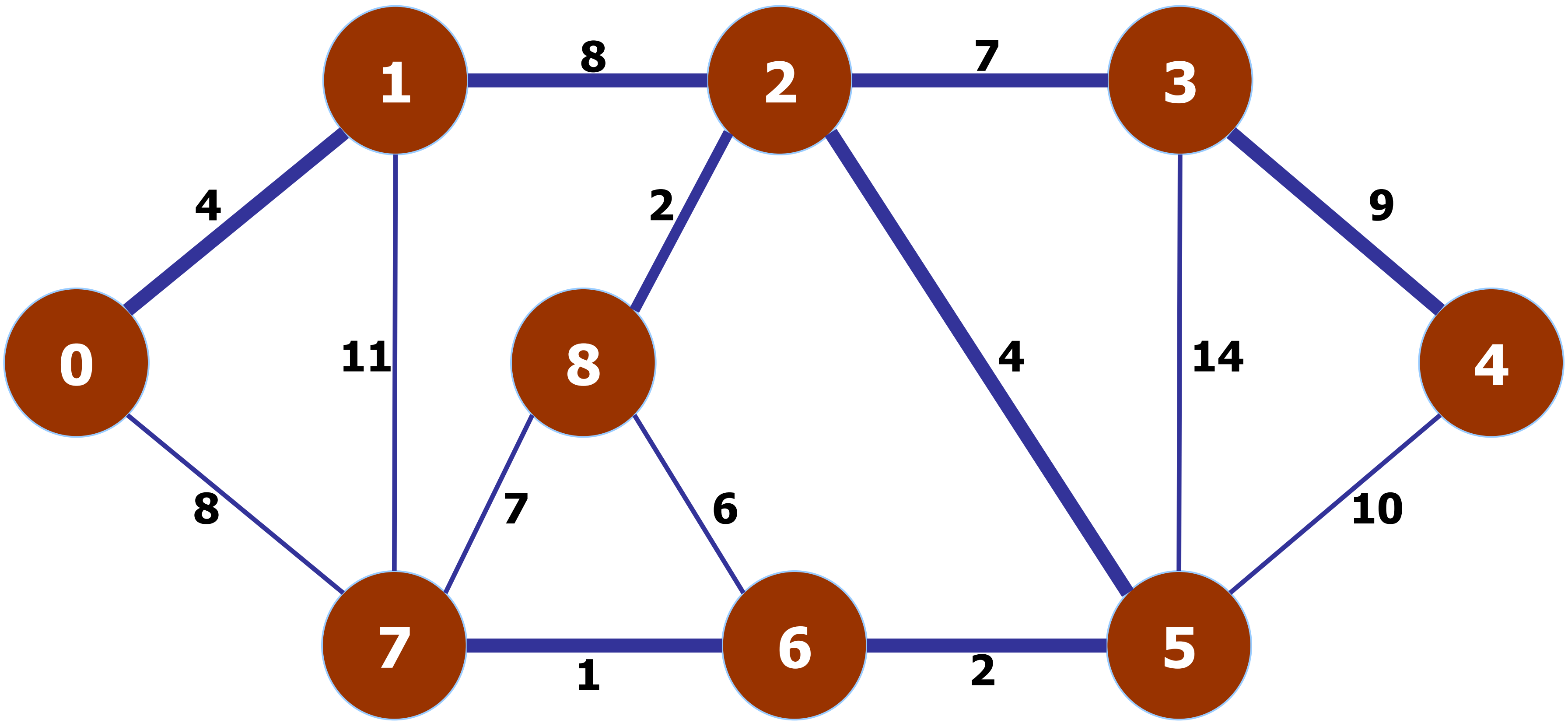


1
2
3
0
8
4
7
6
5
8
7
4
2
11
4
14
9
8
7
6
10
1
2


1
2
3
0
8
4
7
6
5
8
7
4
2
9
11
4
14
8
7
6
10
1
2


1
8
2
7
3
4
2
9
0
11
8
4
14
4
8
7
6
10
7
1
6
2
5


1
8
2
7
3
4
2
9
0
11
8
4
14
4
8
7
6
10
7
1
6
2
5



1
2
3
0
8
4
7
6
5
8
7
4
2
9
11
4
14
8
7
6
10
1
2

ADT di I cat. UF

UF.h

```
void  UFinit(int);
int   UFfind(int, int);
void  UFunion(int, int);
```

UF.c

```
#include <stdlib.h>
#include "UF.h"
static int *id, *sz;
void UFinit(int N) {
  int i;
  id  = malloc(N*sizeof(int));
  sz  = malloc(N*sizeof(int));
  for(i=0; i<N; i++) {
    id[i] = i; sz[i] = 1;
  }
}
```

```
static int find(int x) {
  int i = x;
  while (i!= id[i]) i = id[i];
  return i;
}
int UFfind(int p, int q) { return(find(p) == find(q)); }

void UFunion(int p, int q) {
  int i = find(p), j = find(q);
  if (i == j) return;
  if (sz[i] < sz[j]) {
    id[i] = j; sz[j] += sz[i];
  }
  else {
    id[j] = i; sz[i] += sz[j];
  }
}
```

```c
int mstE(Graph G, Edge *mst) {
  int i, k;
  Edge a[G->E];
  GRAPHedges(G, a);
  sort(a, 0, G->E-1);
  UFinit(G->V);
  for ( i=0, k=0; i < G->E && k < G->V-1; i++ )
    if (!UFfind(a[i].v, a[i].w)) {
      UFunion(a[i].v, a[i].w);
      mst[k++]=a[i];
  }

return k;
}
```

wrapper

```
void GRAPHmstK(Graph G) {
  int i, k, weight = 0;
  Edge *mst = malloc(G->E * sizeof(Edge));

  k = mstE(G, mst);

  printf("\nEdges in the MST: \n");
  for (i=0; i < k; i++) {
    printf("(%s - %s) \n", STretrieve(G->tab, mst[i].v),
                     STretrieve(G->tab, mst[i].w));
    weight += mst[i].wt;
  }
  printf("minimum weight: %d\n", weight);
}
```

Complessità

- Dipende dalle strutture dati utilizzate.
- Con strutture efficienti $T(n) = (|E| \lg |E|)$.

Algoritmo di Prim (1930-1959)

- basato su algoritmo generico
- uso del teorema per determinare l'arco sicuro:
  - inizialmente S = ∅, poi S = {vertice di partenza}
  - iterazione:
    - determinare gli archi che attraversano il taglio
    - tra questi, selezionare l'arco di peso minimo e aggiungilo alla soluzione
    - in base al vertice in cui arriva l'arco, aggiornare S e aggiornare l'insieme degli archi che attraversano il taglio
  - terminazione: considerati tutti i vertici.

Struttura dati

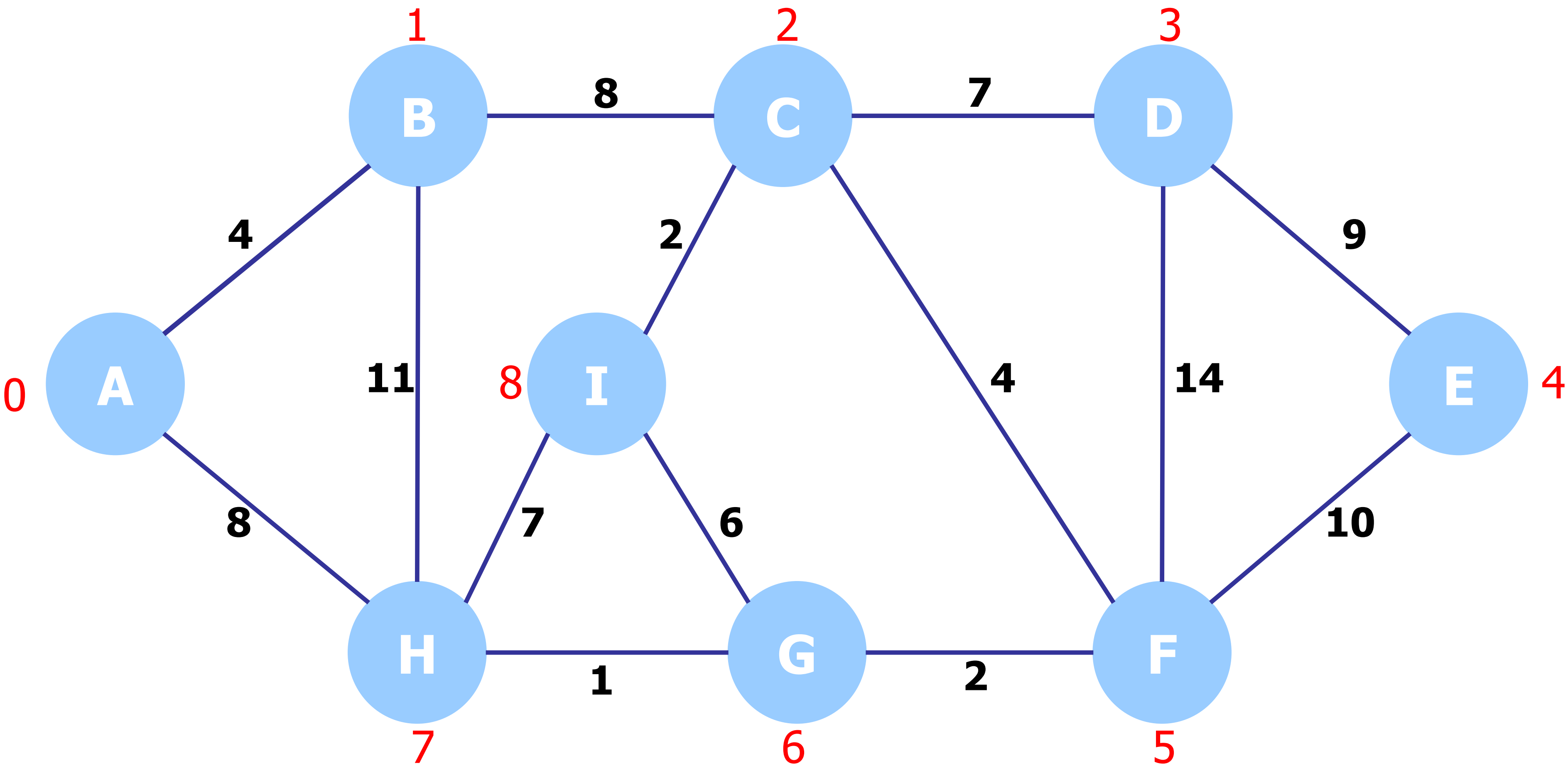
- Vettore `st` per registrare il padre di un vertice che appartiene ad S
- Vettore `fr` per registrare per ogni vertice di V-S quale è il vertice di S più vicino. E' dichiarato static in `Graph.c`
- Vettore `wt` per registrare:
  - per vertici di S il peso dell'arco al padre
  - per vertici di V-S il peso dell'arco verso il vertice di S più vicino
- variabile `min` per il vertice in V-S più vicino a vertici di S

Quando si aggiunge alla soluzione un nuovo arco e un nuovo vertice ad S:

- si controlla se il nuovo arco ha portato qualche vertice di V-S più vicino a vertici di S
- si determina il prossimo arco da aggiungere.

Esempio

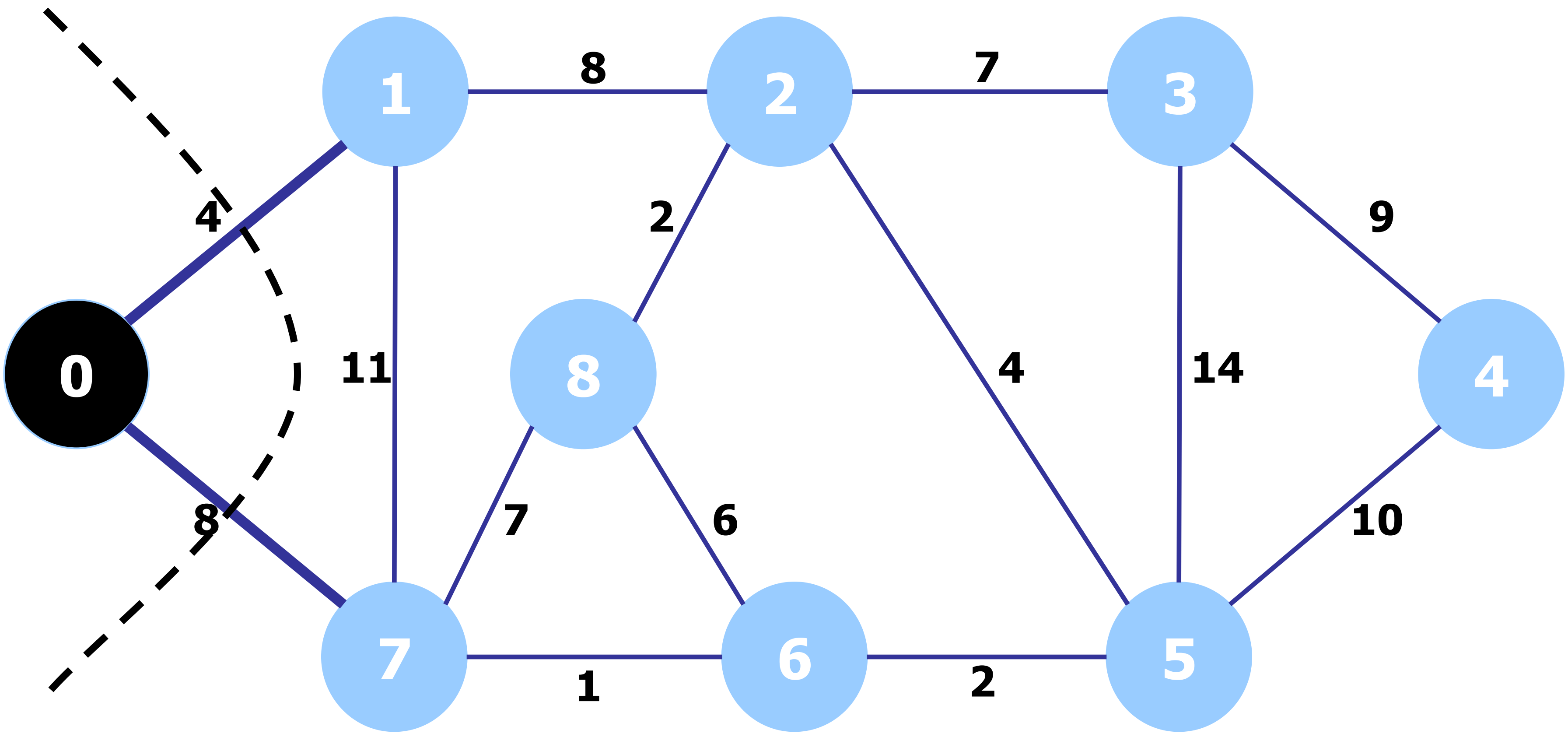
	0	1	2	3	4	5	6	7	8
st	-1	-1	-1	-1	-1	-1	-1	-1	-1
wt	∞	∞	∞	∞	∞	∞	∞	∞	∞
fr	0	1	2	3	4	5	6	7	8



$S = \emptyset$ $V\text{-}S = \{0,1,2,3,4,5,6,7,8\}$

	0	1	2	3	4	5	6	7	8
st	0	-1	-1	-1	-1	-1	-1	-1	-1
wt	0	4	∞	∞	∞	∞	∞	8	∞
fr	0	0	2	3	4	5	6	0	8

min = 1

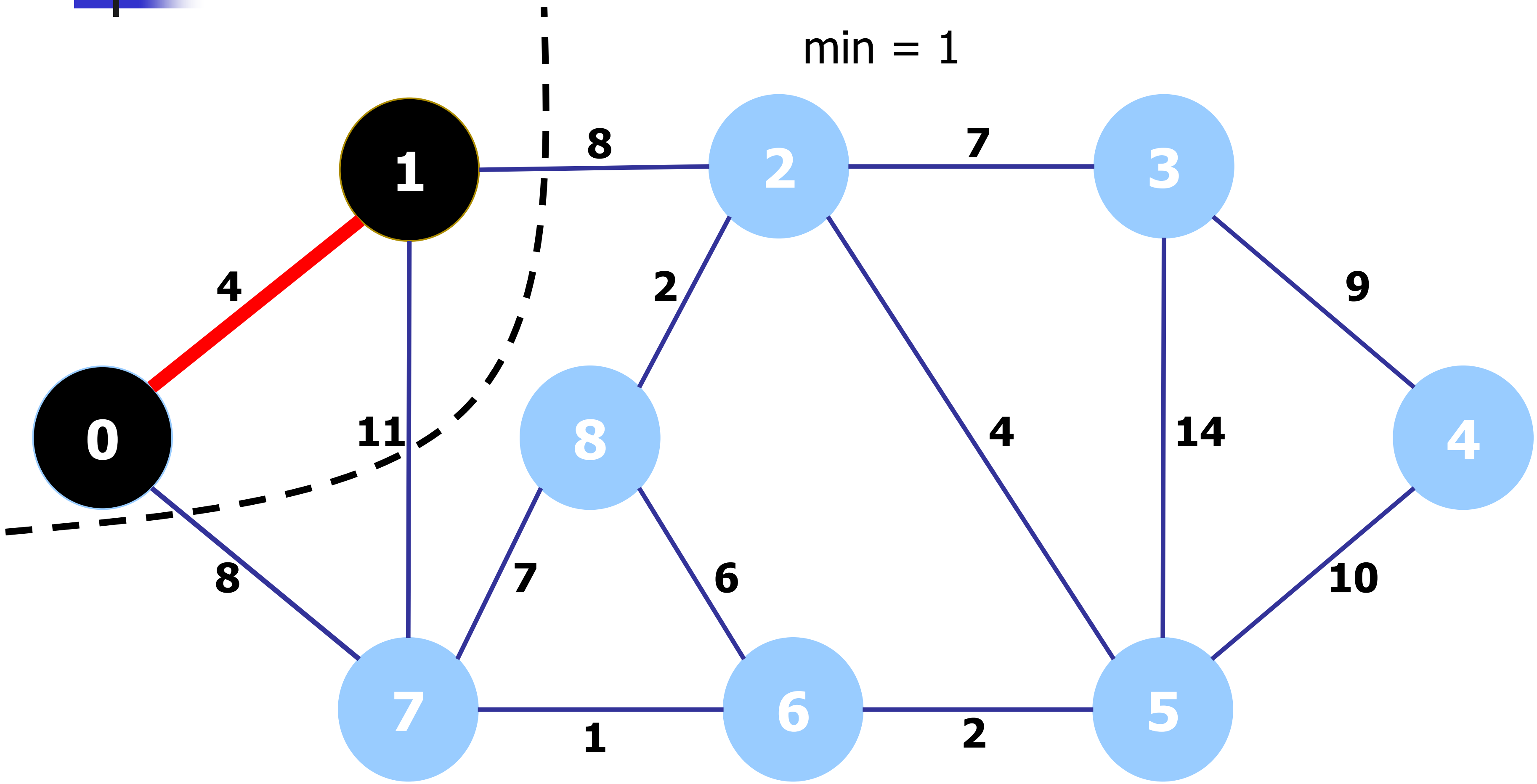


S = {0}

V-S = {1,2,3,4,5,6,7,8}

	0	1	2	3	4	5	6	7	8
st	0	0	-1	-1	-1	-1	-1	-1	-1
wt	0	4	∞	∞	∞	∞	∞	8	∞
fr	0	0	2	3	4	5	6	0	8

min = 1

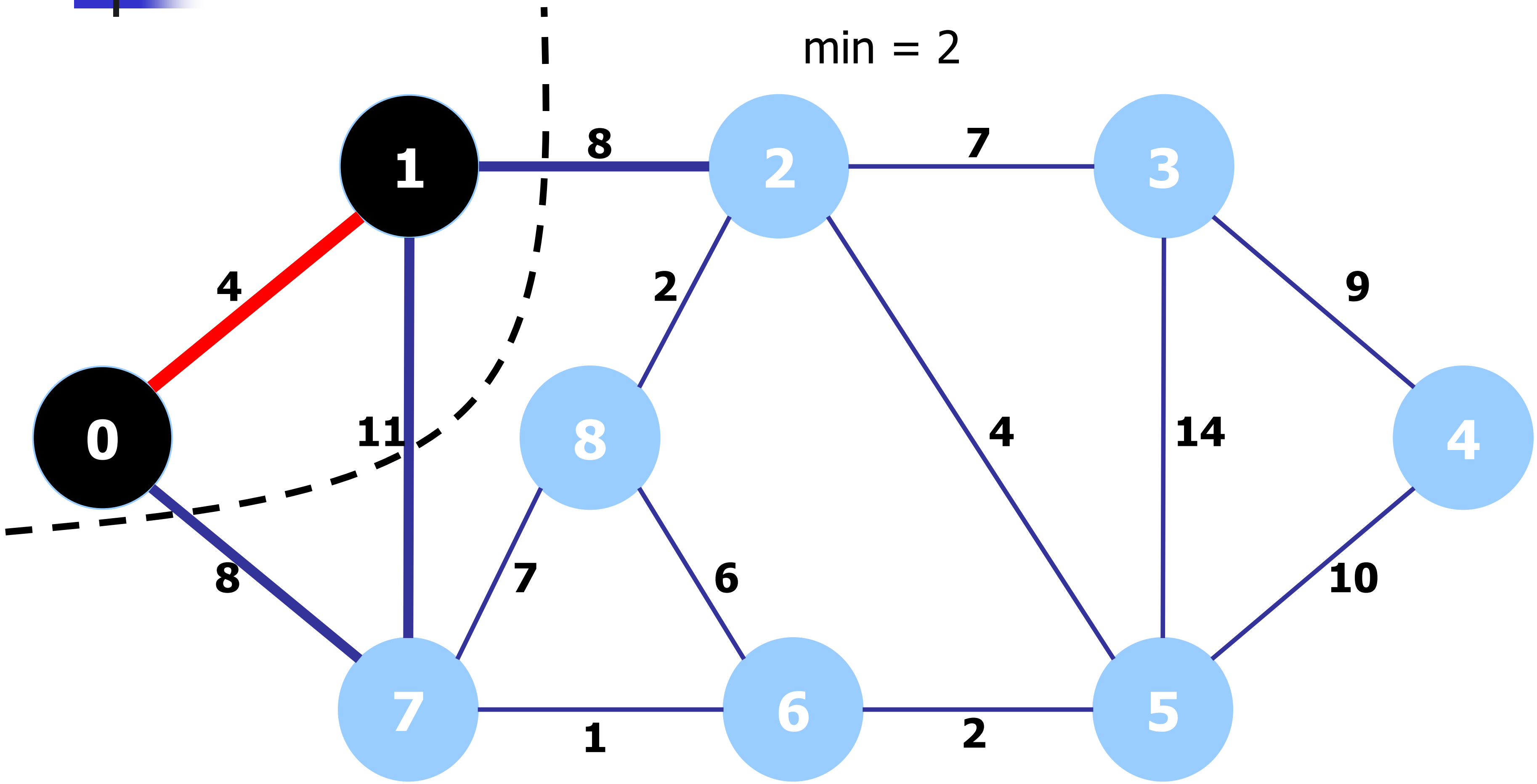


S = {0,1}

V-S = {2,3,4,5,6,7,8}

	0	1	2	3	4	5	6	7	8
st	0	0	-1	-1	-1	-1	-1	-1	-1
wt	0	4	8	∞	∞	∞	∞	8	∞
fr	0	0	1	3	4	5	6	0	8

min = 2

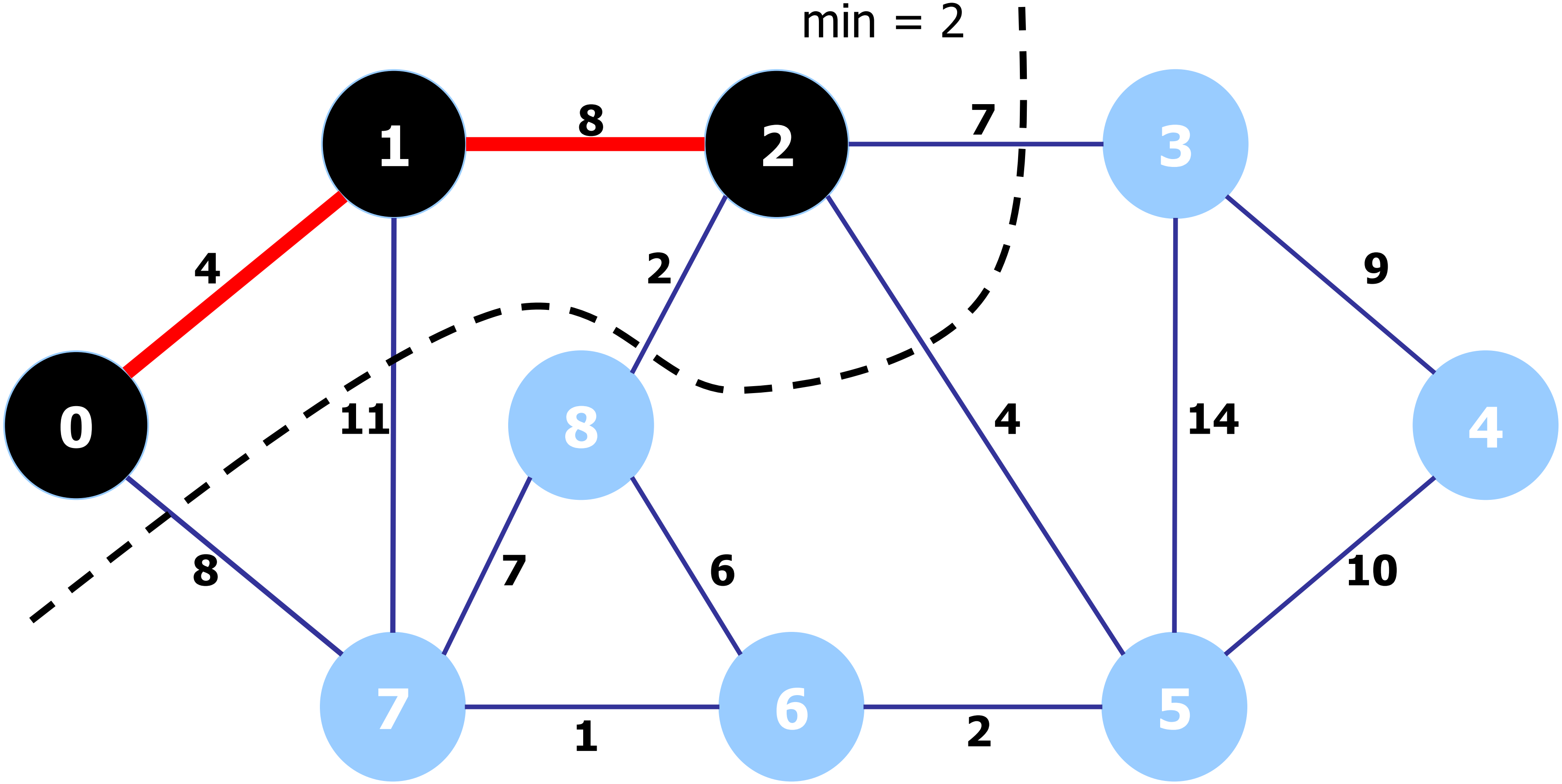


S = {0,1}

V-S = {2,3,4,5,6,7,8}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	-1	-1	-1	-1
wt	0	4	8	∞	∞	∞	∞	8	∞
fr	0	0	1	3	4	5	6	0	8

min = 2

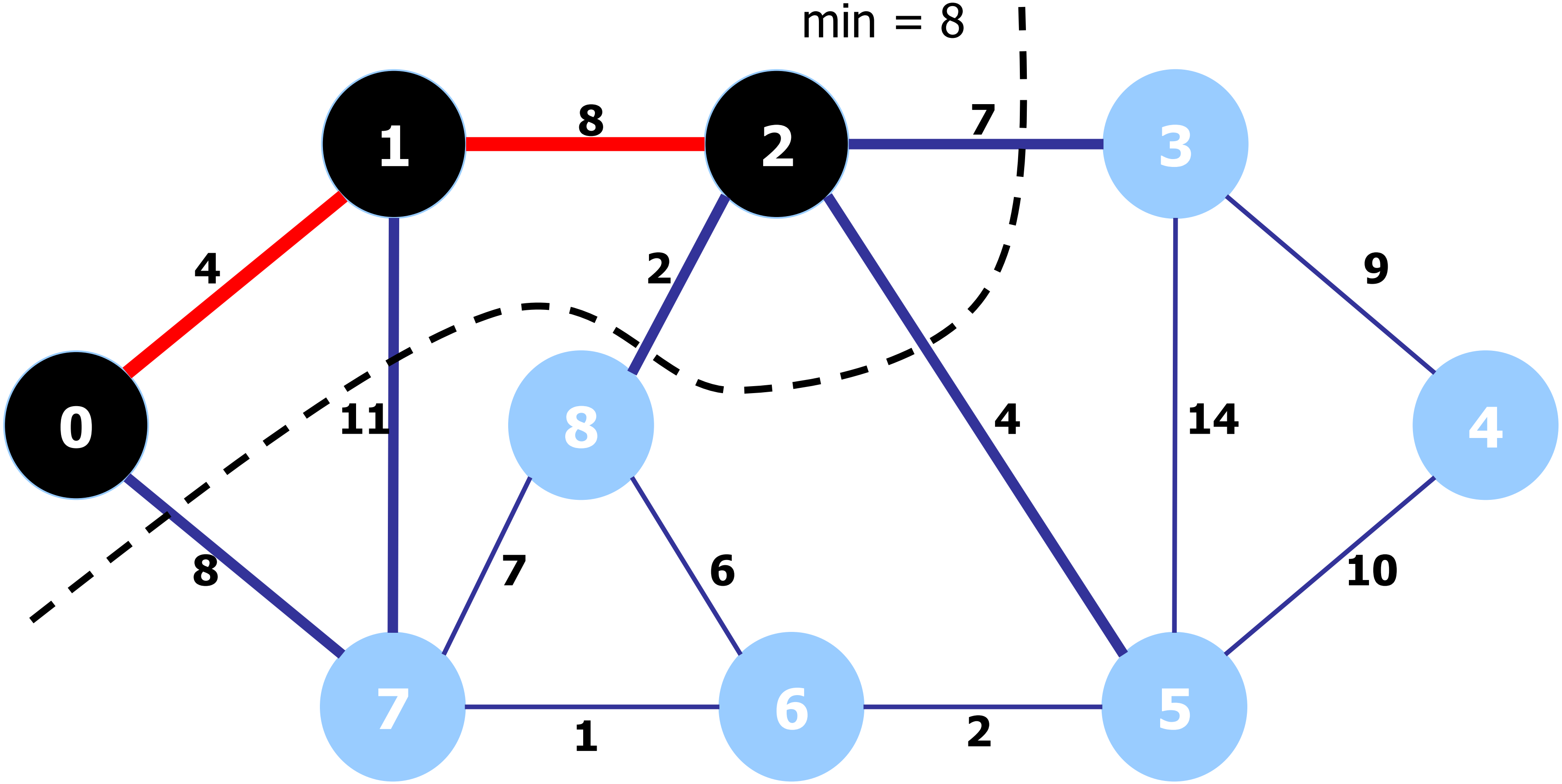


S = {0,1,2}

V-S = {3,4,5,6,7,8}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	-1	-1	-1	-1
wt	0	4	8	7	∞	4	∞	8	2
fr	0	0	1	2	4	2	6	0	2

min = 8

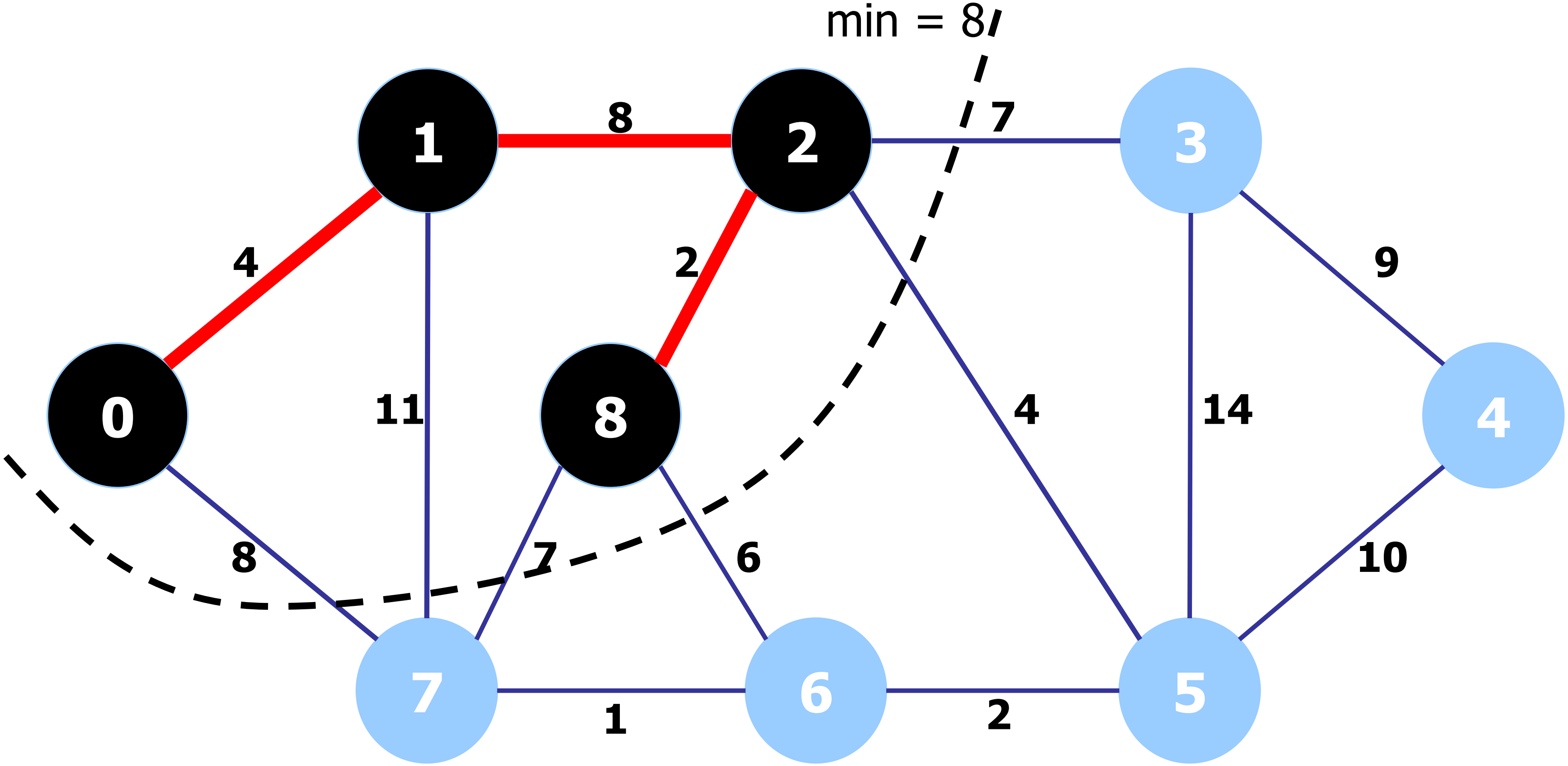


S = {0,1,2}

V-S = {3,4,5,6,7,8}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	-1	-1	-1	2
wt	0	4	8	7	∞	4	∞	8	2
fr	0	0	1	2	4	2	6	0	2

min = 8

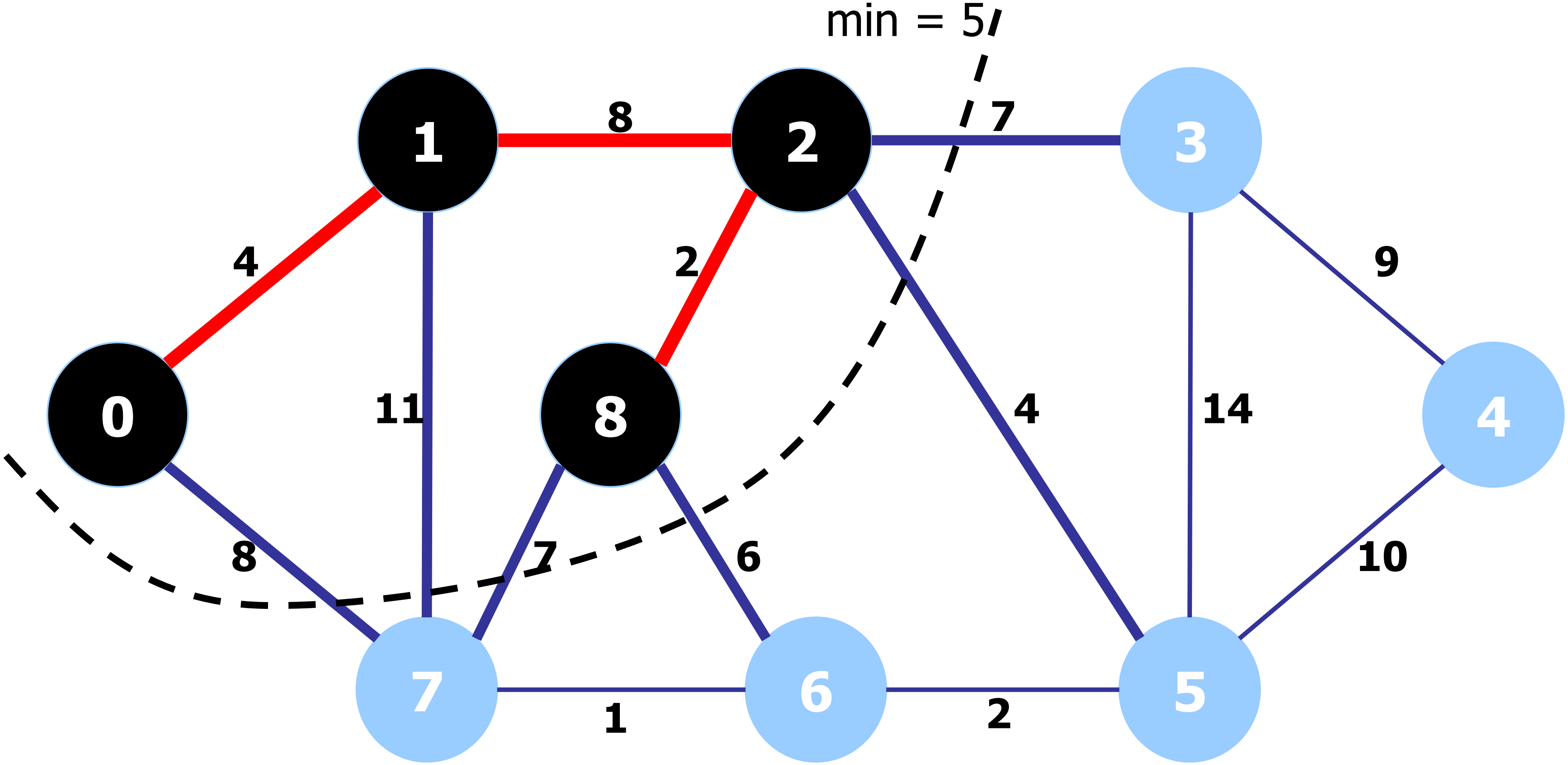


S = {0,1,2,8}

V-S = {3,4,5,6,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	-1	-1	-1	2
wt	0	4	8	7	∞	4	6	7	2
fr	0	0	1	2	4	2	8	8	2

min = 5

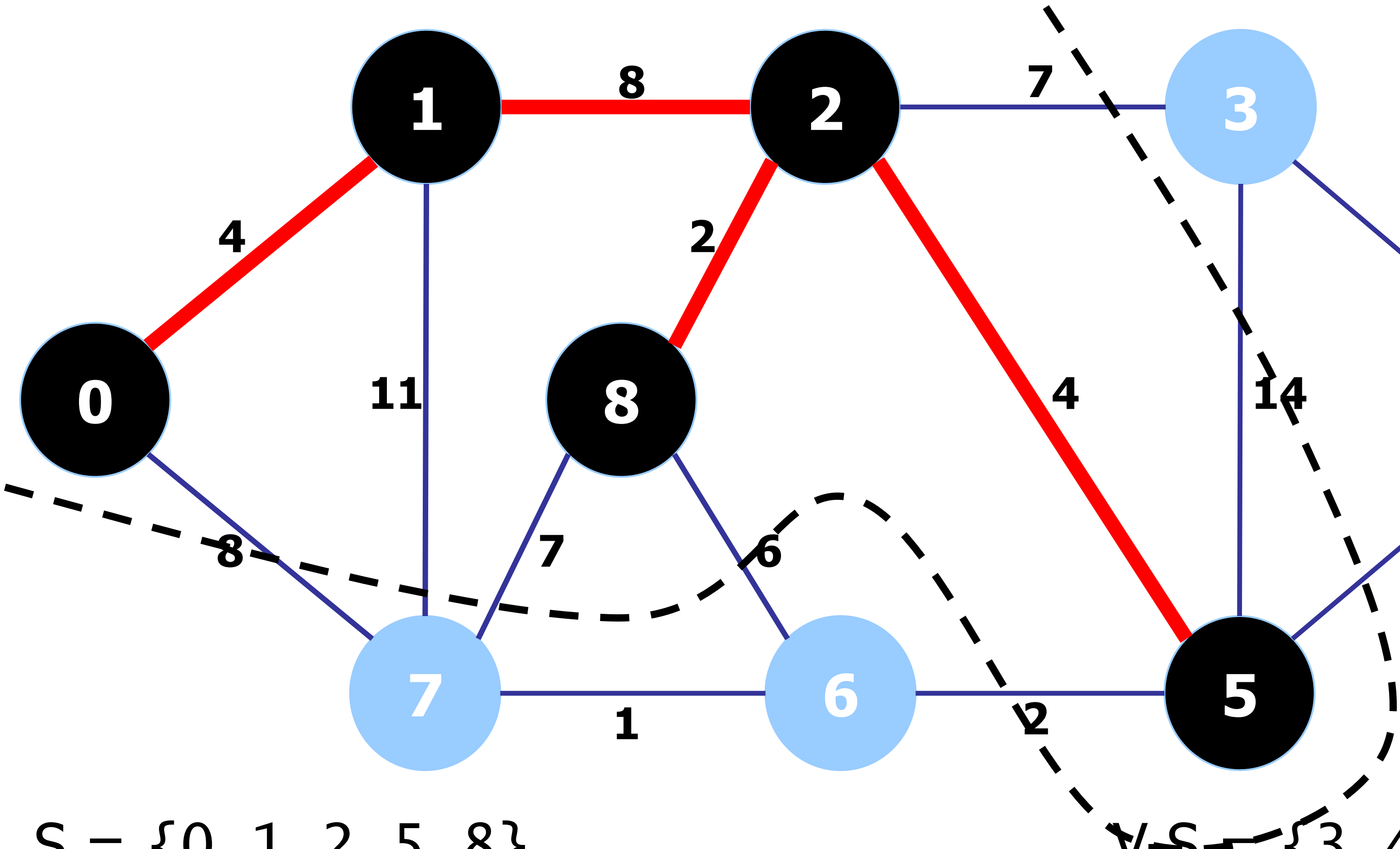


S = {0,1,2,8}

V-S = {3,4,5,6,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	-1	-1	2
wt	0	4	8	7	∞	4	6	7	2
fr	0	0	1	2	4	2	8	8	2

min = 5

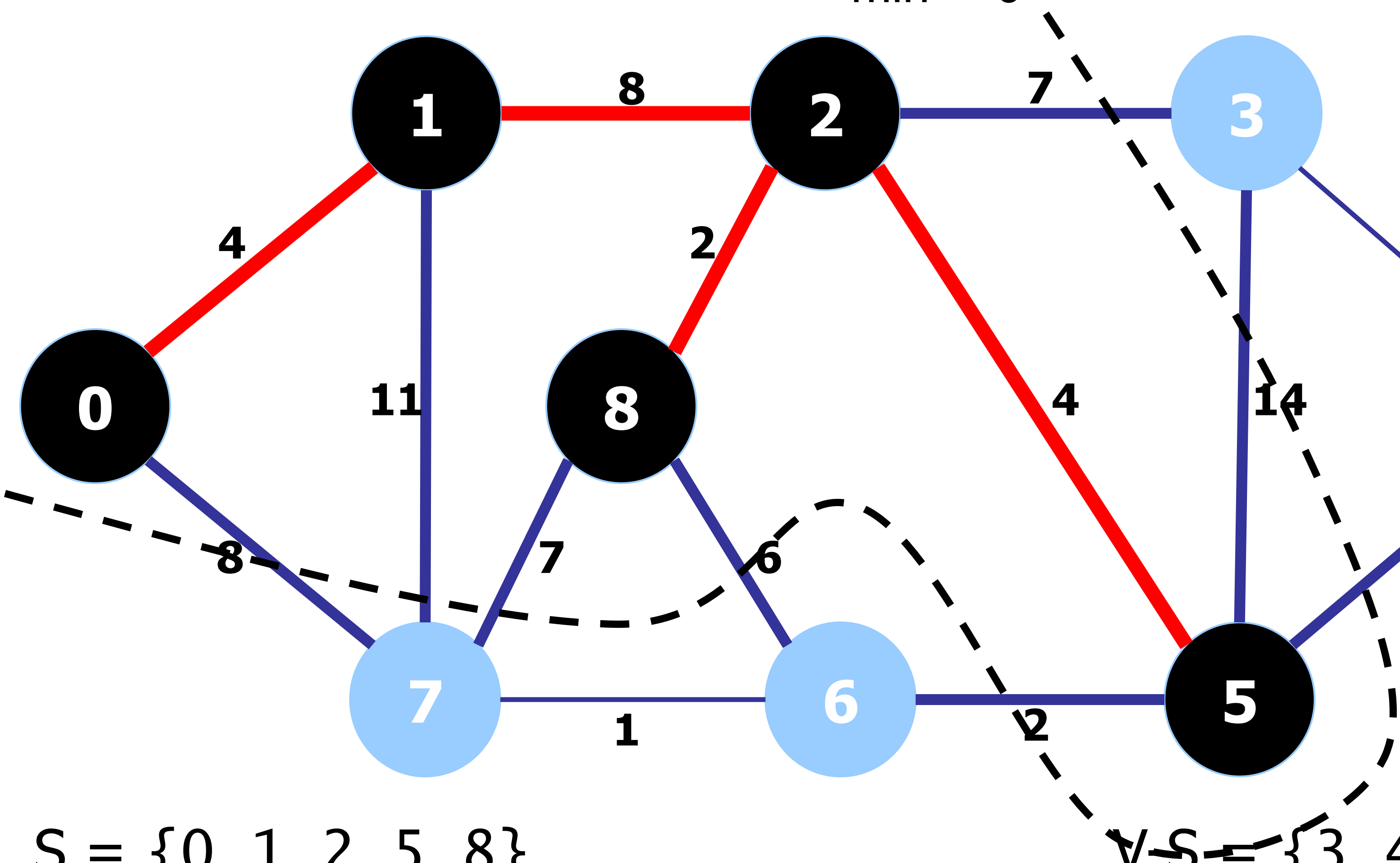


S = {0,1,2,5,8}

V-S = {3,4,6,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	-1	-1	2
wt	0	4	8	7	10	4	2	7	2
fr	0	0	1	2	5	2	5	8	2

min = 6

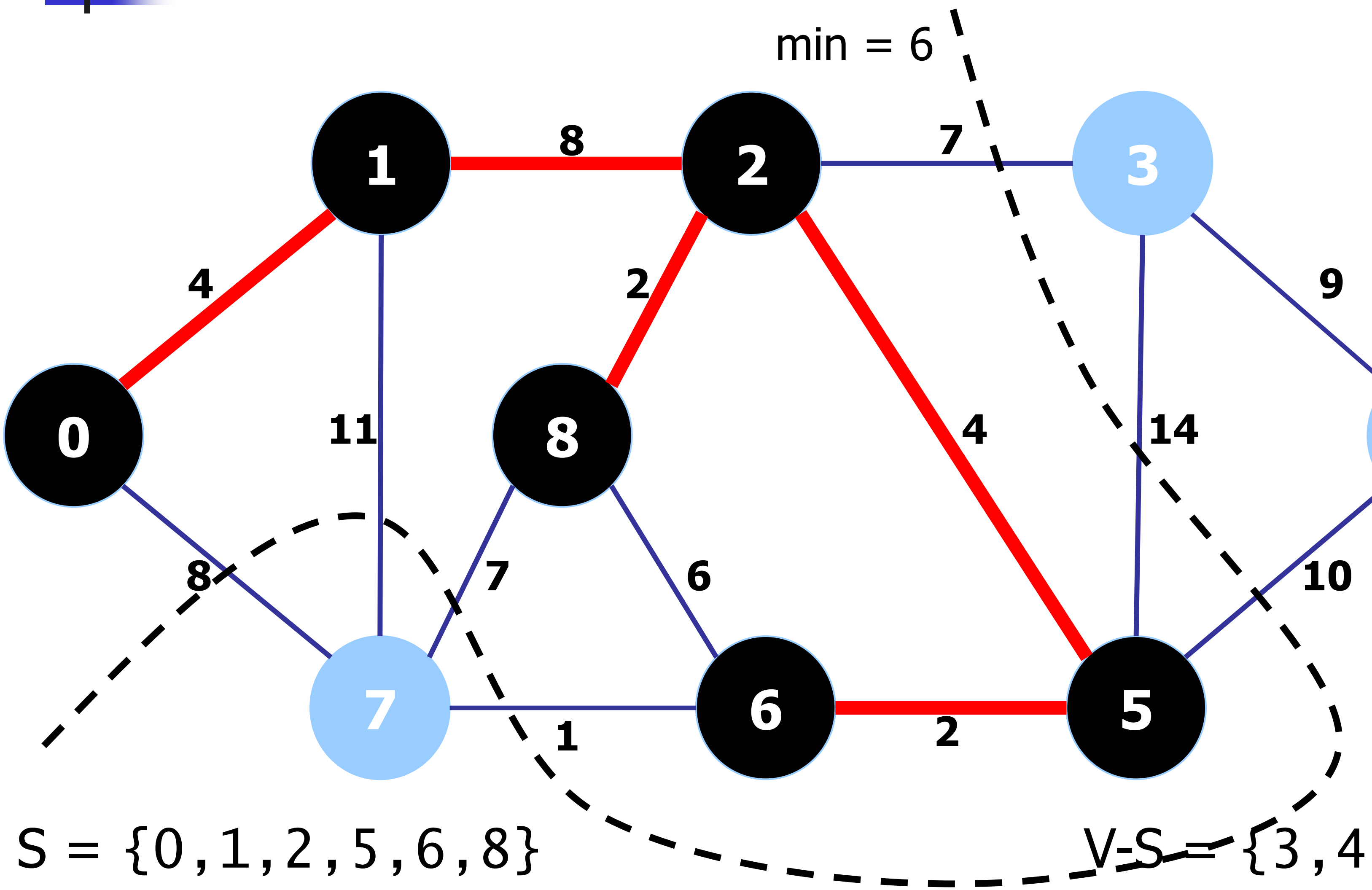


S = {0,1,2,5,8}

V-S = {3,4,6,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	5	-1	2
wt	0	4	8	7	10	4	2	7	2
fr	0	0	1	2	5	2	5	8	2

min = 6

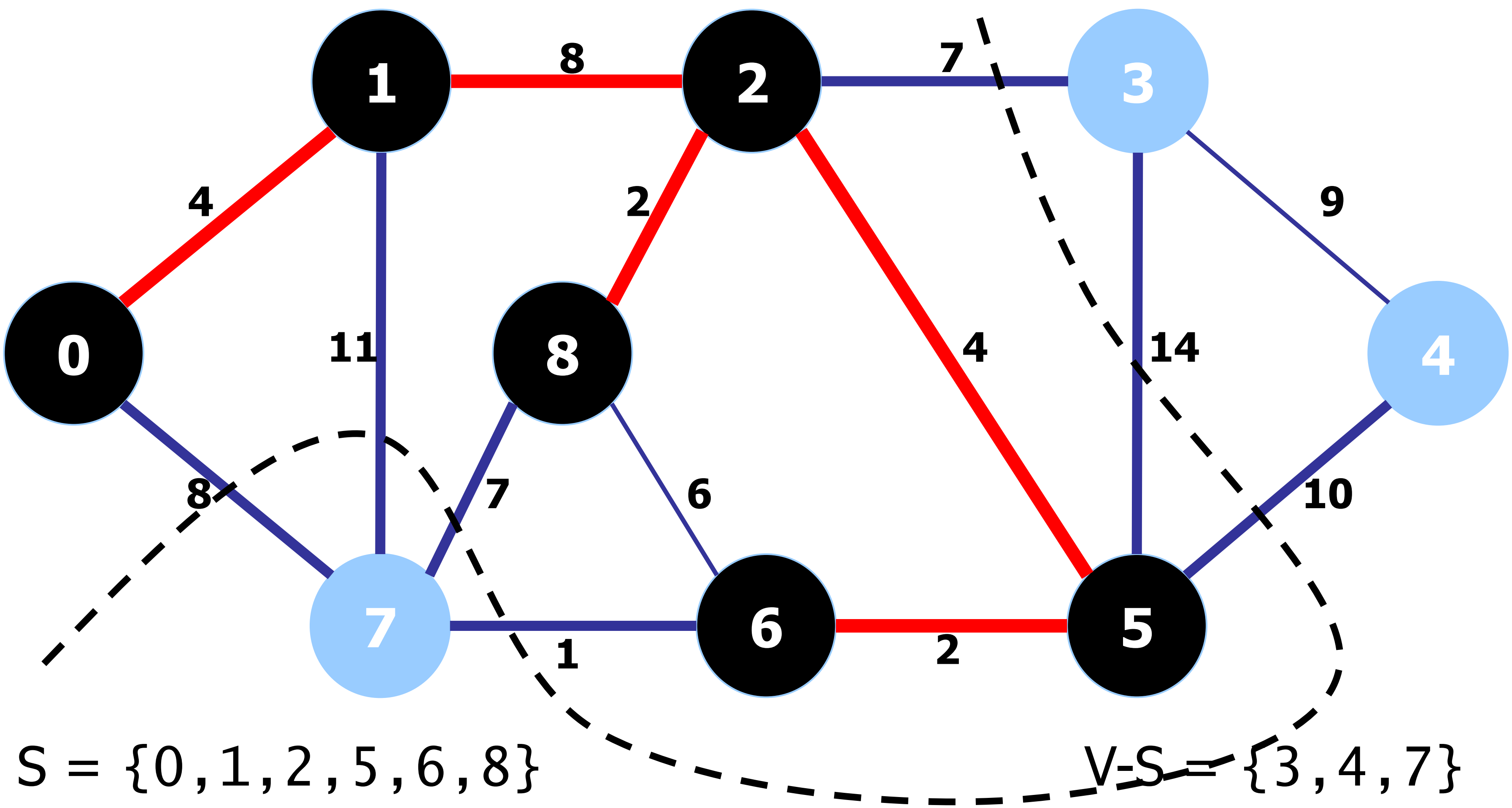


S = {0,1,2,5,6,8}

V-S = {3,4,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	5	-1	2
wt	0	4	8	7	10	4	2	1	2
fr	0	0	1	2	5	2	5	6	2

min = 7

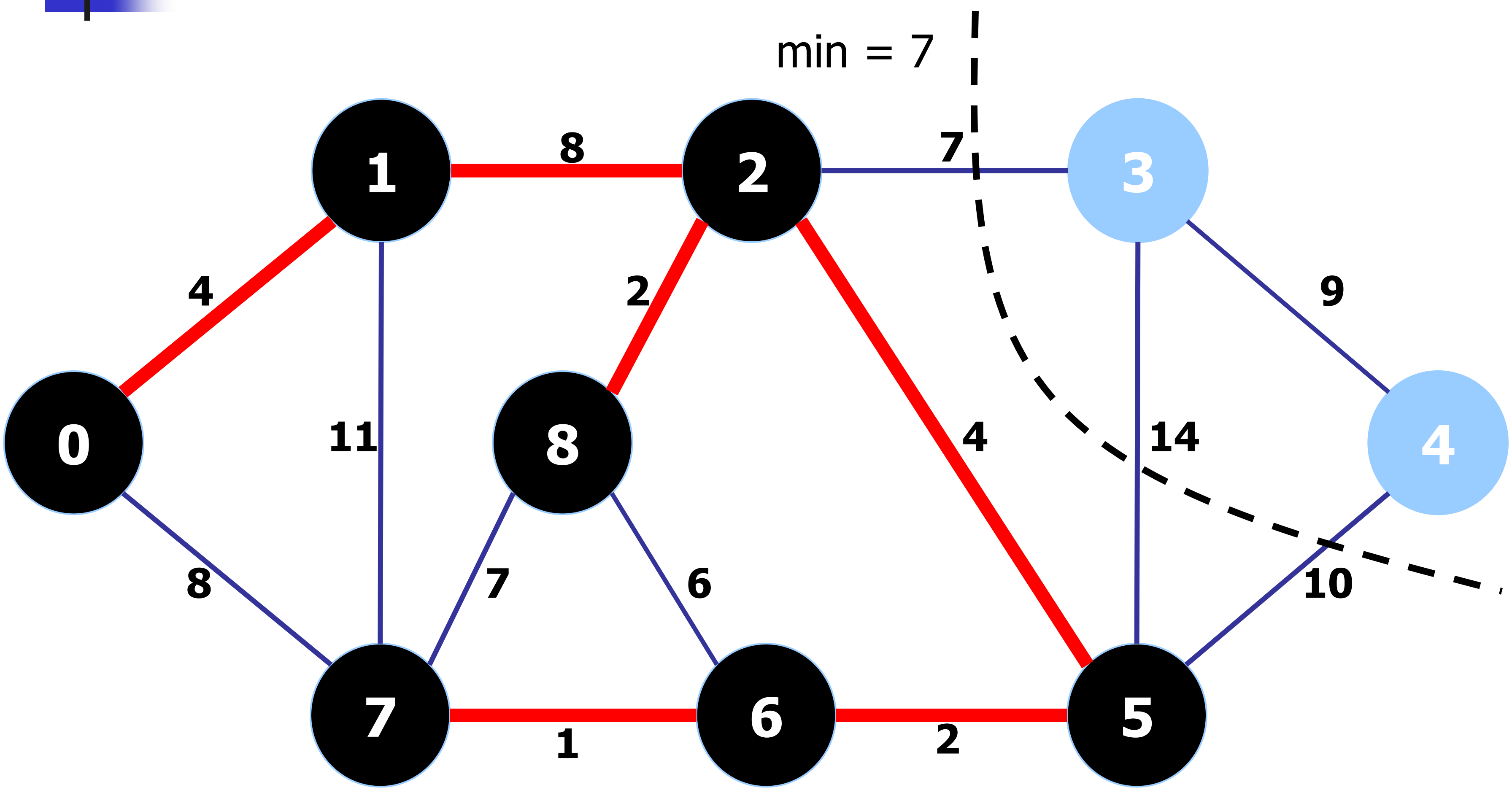


S = {0,1,2,5,6,8}

V-S = {3,4,7}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	5	6	2
wt	0	4	8	7	10	4	2	1	2
fr	0	0	1	2	5	2	5	6	2

min = 7

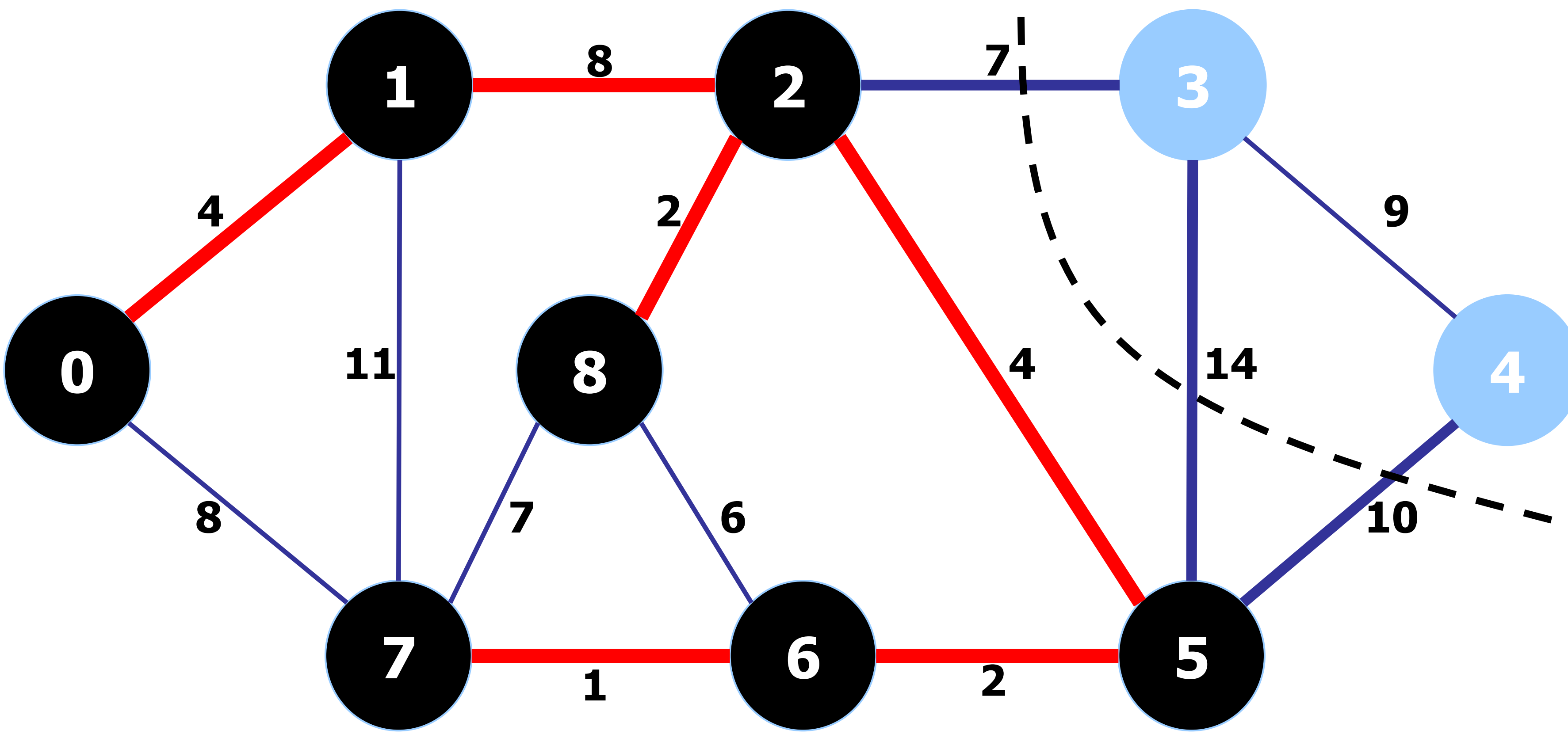


S = {0,1,2,5,6,7,8}

V-S = {3,4}

	0	1	2	3	4	5	6	7	8
st	0	0	1	-1	-1	2	5	6	2
wt	0	4	8	7	10	4	2	1	2
fr	0	0	1	2	5	2	5	6	2

min = 3

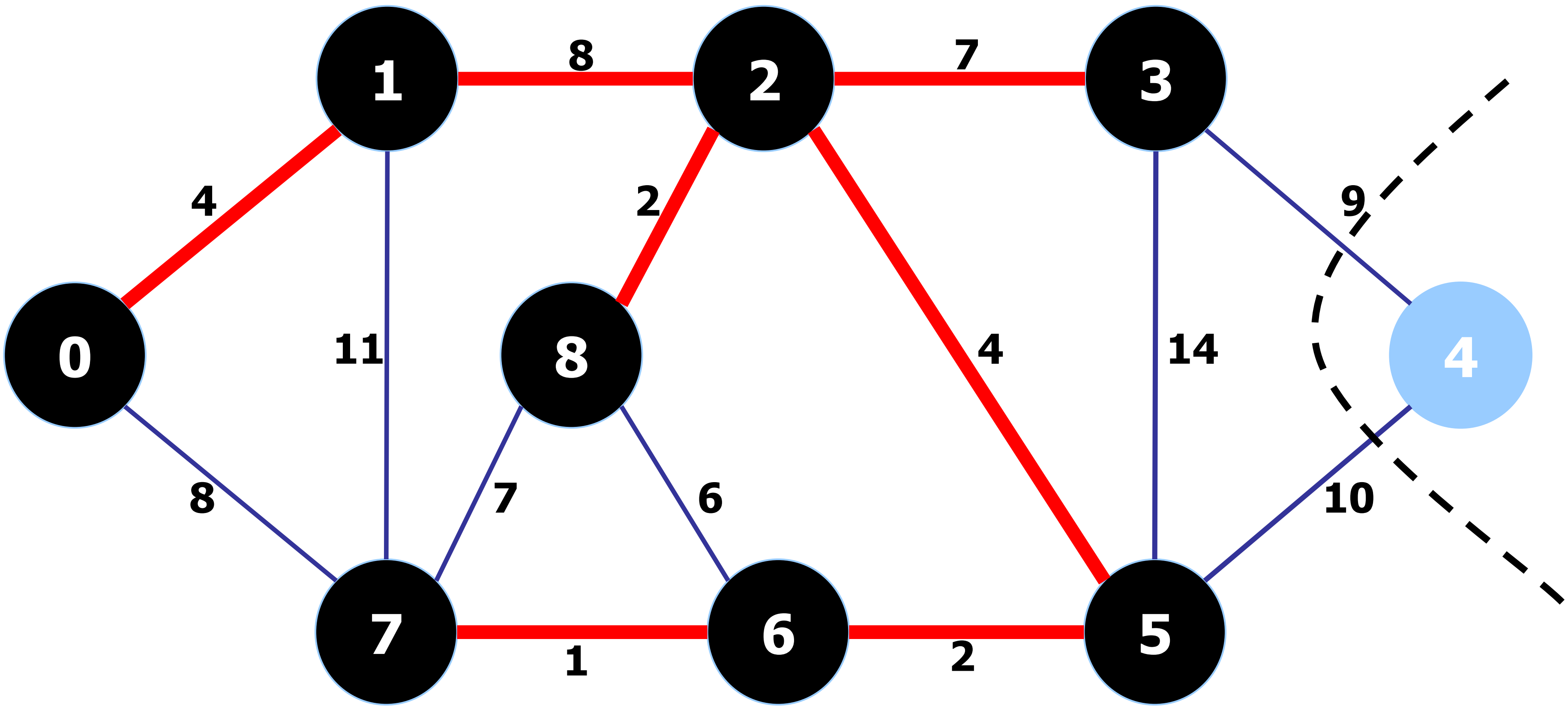


S = {0,1,2,5,6,7,8}

V-S = {3,4}

	0	1	2	3	4	5	6	7	8
st	0	0	1	2	-1	2	5	6	2
wt	0	4	8	7	10	4	2	1	2
fr	0	0	1	2	5	2	5	6	2

min = 3

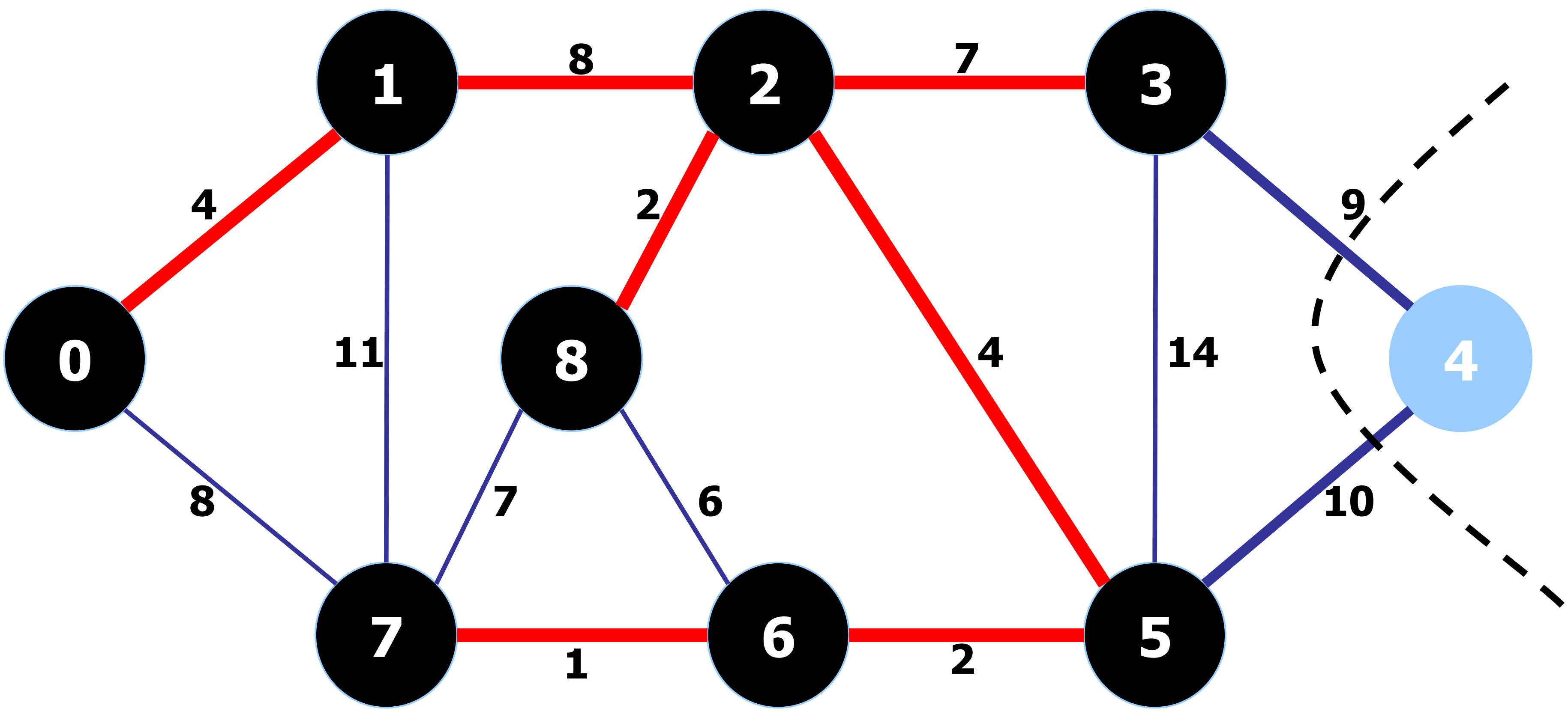


S = {0,1,2,3,5,6,7,8}

V-S = {4}

	0	1	2	3	4	5	6	7	8
st	0	0	1	2	-1	2	5	6	2
wt	0	4	8	7	9	4	2	1	2
fr	0	0	1	2	3	2	5	6	2

min = 9

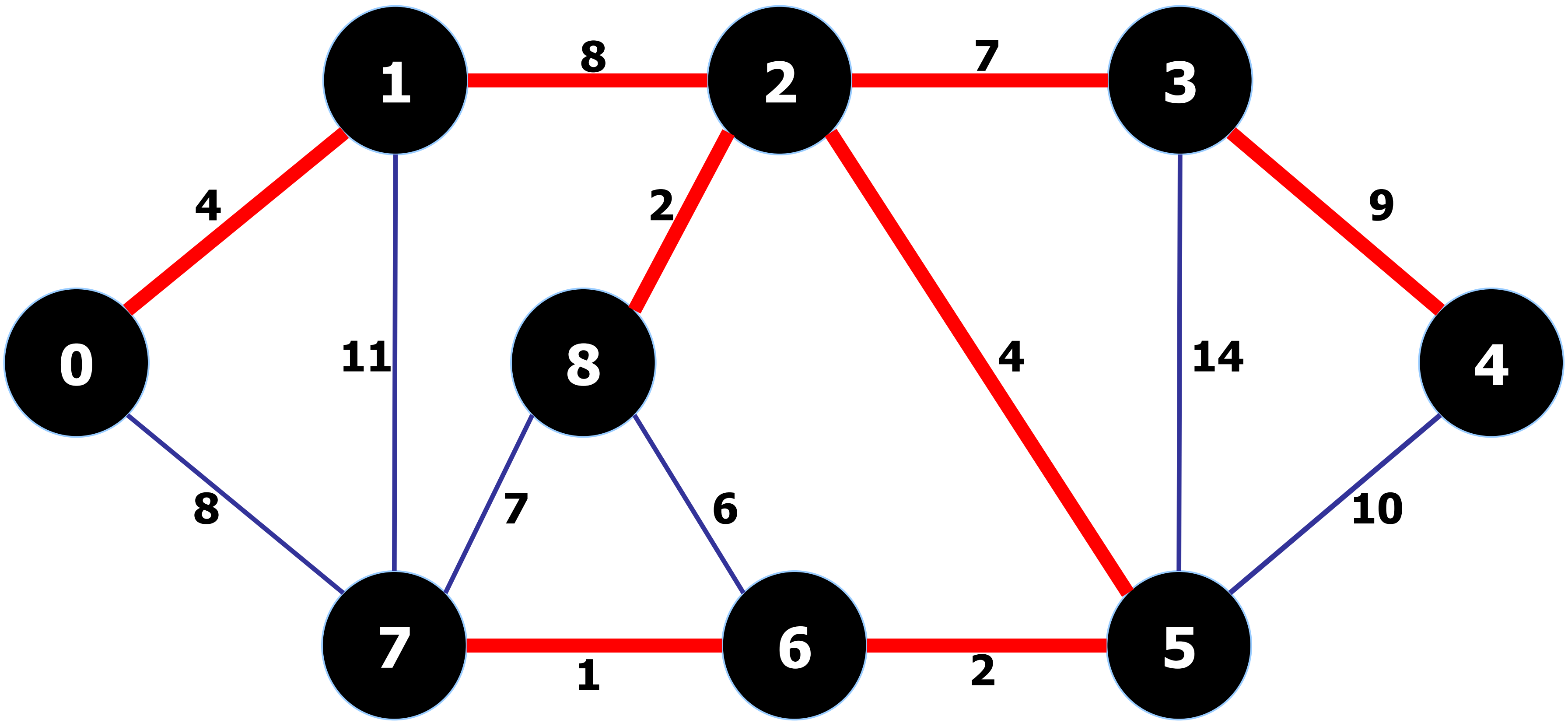


S = {0,1,2,3,5,6,7,8}

V-S = {4}

	0	1	2	3	4	5	6	7	8
st	0	0	1	2	3	2	5	6	2
wt	0	4	8	7	9	4	2	1	2
fr	0	0	1	2	3	2	5	6	2

min = 9



$S = \{0,1,2,3,4,5,6,7,8\}$ $V\text{-}S = \emptyset$

```c
void mstV(Graph G, int *st, int *wt) {
  int v, w, min, *fr = malloc(G->V*sizeof(int));
  for ( v=0; v < G->V; v++) {
    st[v] = -1; fr[v] = v;  wt[v] = maxWT;
  }
  st[0] = 0;   wt[0] = 0;   wt[G->V] = maxWT;
  for ( min = 0; min != G->V; ) {
    v = min;  st[min] = fr[min];
    for (w = 0, min = G->V; w < G->V; w++)
      if (st[w] == -1) {
        if (G->adj[v][w] < wt[w]) {
          wt[w] = G->adj[v][w]; fr[w] = v;
        }
        if (wt[w] < wt[min]) min = w;
      }
  }
}
```

wrapper

```
void GRAPHmstP(Graph G) {
  int v, *st, *wt, weight = 0;
  st = malloc(G->V*sizeof(int));
  wt = malloc((G->V+1)*sizeof(int));

  mstV(G, st, wt);

  printf("\nEdges in the MST: \n");
  for ( v=0; v < G->V; v++) {
    if (st[v] != v) {
      printf("(%s-%s)\n",STretrieve(G->tab,st[v]),
                STretrieve(G->tab,v));
      weight += wt[v];
    }
  }
  printf("\nminimum weight: %d\n", weight);
}
```

Complessità

$$T(n) = O(|V| \lg |V| + |E| \lg |V|)$$
$$= O(|E| \lg |V|).$$

Con strutture dati particolari (heap di Fibonacci)

$$T(n) = O(|E| + |V| \lg |V|)$$

Riferimenti

- Rappresentazione:
  - Sedgewick Part 5 20.1
- Principi:
  - Sedgewick Part 5 20.2
  - Cormen 24.1
- Algoritmo di Kruskal
  - Sedgewick Part 5 20.4
  - Cormen 24.2
- Algoritmo di Prim
  - Sedgewick Part 5 20.3
  - Cormen 24.2