Gli alberi ricoprenti minimi

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

Alberi ricoprenti minimi

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

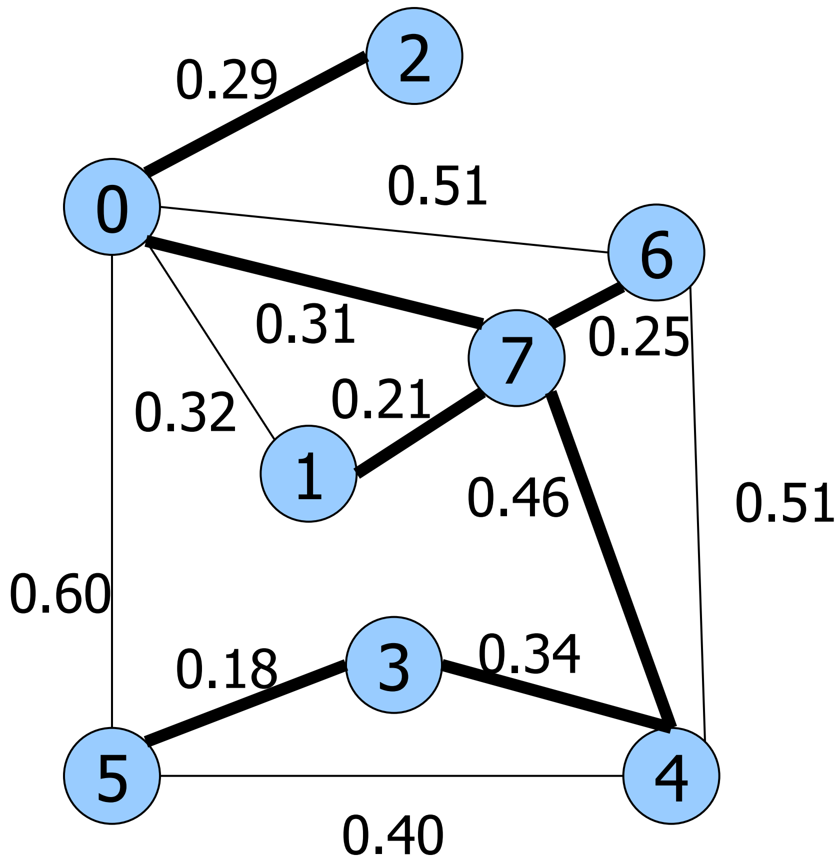
Albero ricoprente minimo - Minimum-weight Spanning Tree.

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

Acicicltà && 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

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 $+\infty$)
- 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.

Interfaccia

```
typedef struct { int v; int w; int wt;} Edge;
Edge EDGE(int, int, int);

typedef struct graph *Graph;

Graph GRAPHinit(int);
void  GRAPHinsertE(Graph, Edge);
void  GRAPHremoveE(Graph, Edge);
int   GRAPHedges(Edge [], Graph G);
void  GRAPHshow(Graph G);
```

Implementazione (matrice)

```
#include <stdlib.h>
#include <stdio.h>
#include "Graph.h"

struct graph {int V; int E; int **adj; };

Edge EDGE(int v, int w, int wt)
{
    Edge e;
    e.v = v;
    e.w = w;
    e.wt = wt;
    return e;
}
```

```
int **MATRIXint(int r, int c, int val) {
    int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i=0; i < r; i++)
        t[i] = malloc(c * sizeof(int));
    for (i=0; i < r; i++)
        for (j=0; j < c; j++)
            t[i][j] = val;
    return t;
}

Graph GRAPHinit(int V) {
    Graph G = malloc(sizeof *G);
    G->V = V;
    G->E = 0;
    G->adj = MATRIXint(V, V, 0);
    return G;
}
```

```
void  GRAPHinsertE(Graph G, Edge e) {
    int v = e.v, w = e.w, wt = e.wt;
    if (G->adj[v][w] == 0)
        G->E++;
    G->adj[v][w] = wt;
    G->adj[w][v] = wt;
}
```

Attenzione: si possono generare cappi!

```
void  GRAPHremoveE(Graph G, Edge e) {
    int v = e.v, w = e.w;
    if (G->adj[v][w] != 0)
        G->E--;
    G->adj[v][w] = 0;
    G->adj[w][v] = 0;
}
```

```
int   GRAPHedges(Edge a[], Graph G) {
    int v, w, E = 0;
    for (v=0; v < G->V; v++)
        for (w=v+1; w < G->V; w++)
            if (G->adj[v][w] != 0)
                a[E++] = EDGE(v, w, G->adj[v][w]);
    return E;
}

void  GRAPHshow(Graph G) {
    int i, j;
    printf("%d vertices, %d edges \n", G->V, G->E);
    for (i=0; i < G->V; i++) {
        printf("%2d:", i);
        for (j=0; j < G->V; j++)
            printf("%2d", G->adj[i][j]);
        printf("\n");
    }
}
```

Implementazione (lista)

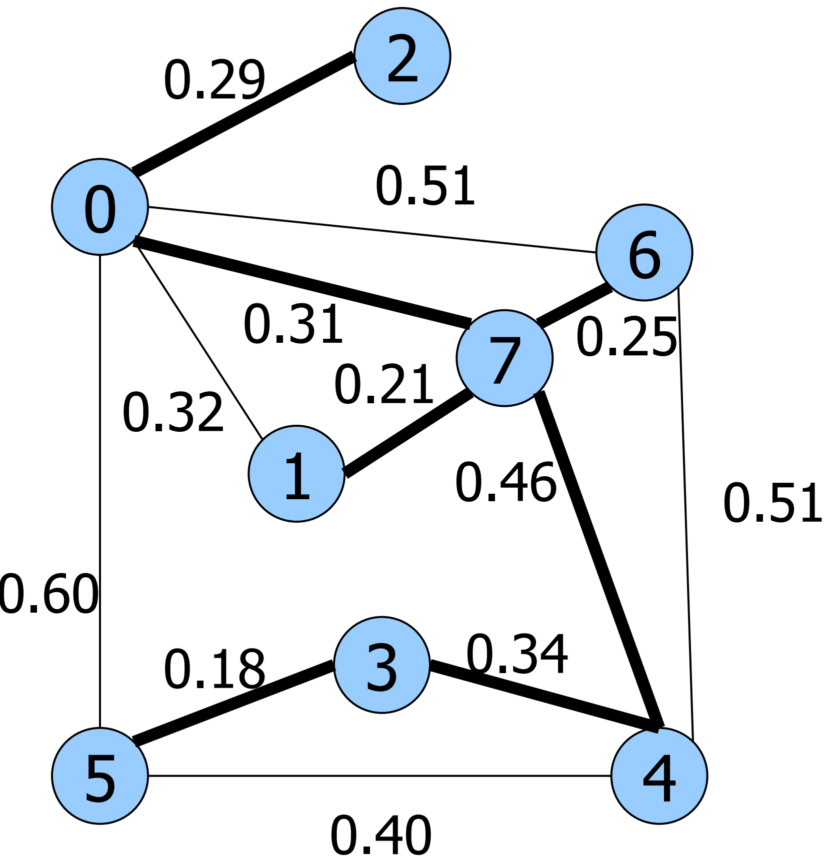
```
#include <stdlib.h>
#include <stdio.h>
#include "Graph.h"
typedef struct node *link;
struct node { int v; int wt; link next; } ;
struct graph { int V; int E; link *adj; } ;

link NEW(int v, int wt; link next) {
    link x = malloc(sizeof *x);
    x->v = v; x->wt = wt; x->next = next;
    return x;
  }
Edge EDGE(int v, int w, int wt) {
    Edge e;
    e.v = v;
    e.wt = wt;
    return e;
}
```

```
Graph GRAPHinit(int V) {
    int v;
    Graph G = malloc(sizeof *G);
    G->V = V;
    G->E = 0;
    G->adj = malloc( V  * sizeof(link));
    for (v = 0; v < V; v++)
        G->adj[v] = NULL;
    return G;
}
void  GRAPHinsertE(Graph G, Edge e)
{
    int v = e.v, w = e.w, wt = e.wt;
    if (v == w) return;
    G->adj[v] = NEW(w, wt, G->adj[v]);
    G->adj[w] = NEW(v, wt, G->adj[w]);
    G->E++;
}
```

Attenzione: si possono generare archi ripetuti!

```
int   GRAPHedges(Edge a[], Graph G) {
    int v, E = 0; link t;
    for (v=0; v < G->V; v++)
        for (t=G->adj[v]; t != NULL; t = t->next)
            if (v < t->v)
                a[E++] = EDGE(v, t->v, t->wt);
    return E;
}
void  GRAPHshow(Graph G) {
    int v; link t;
    printf("%d vertices, %d edges \n", G->V, G->E);
    for (v=0; v < G->V; v++) {
        printf("%2d:", v);
        for (t=G->adj[v]; t != NULL; t = t->next)
            printf(" %d/%d", t->v, t->wt);
        printf("\n");
    }
}
```

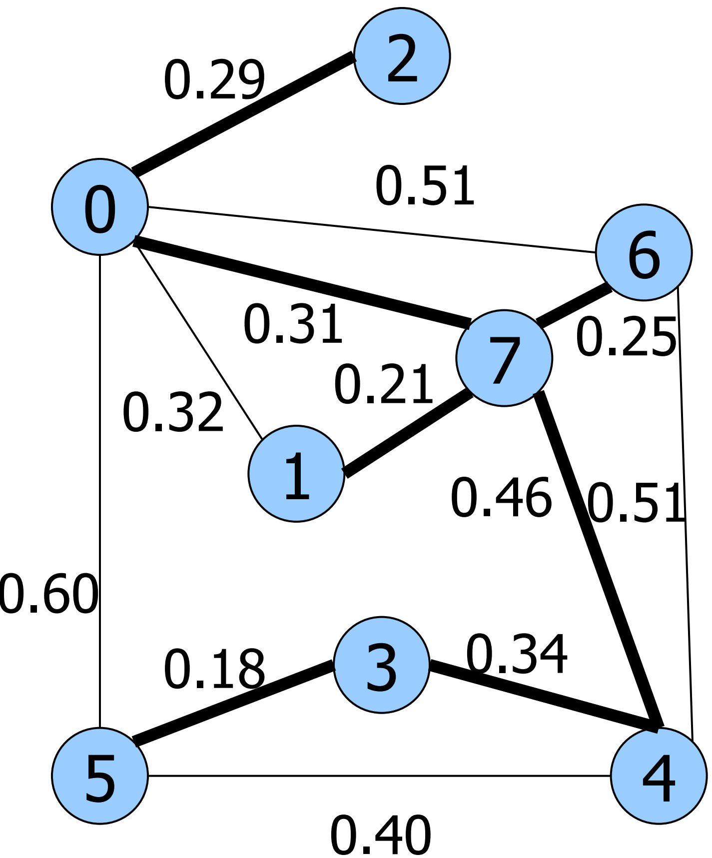
Rappresentazione degli MST

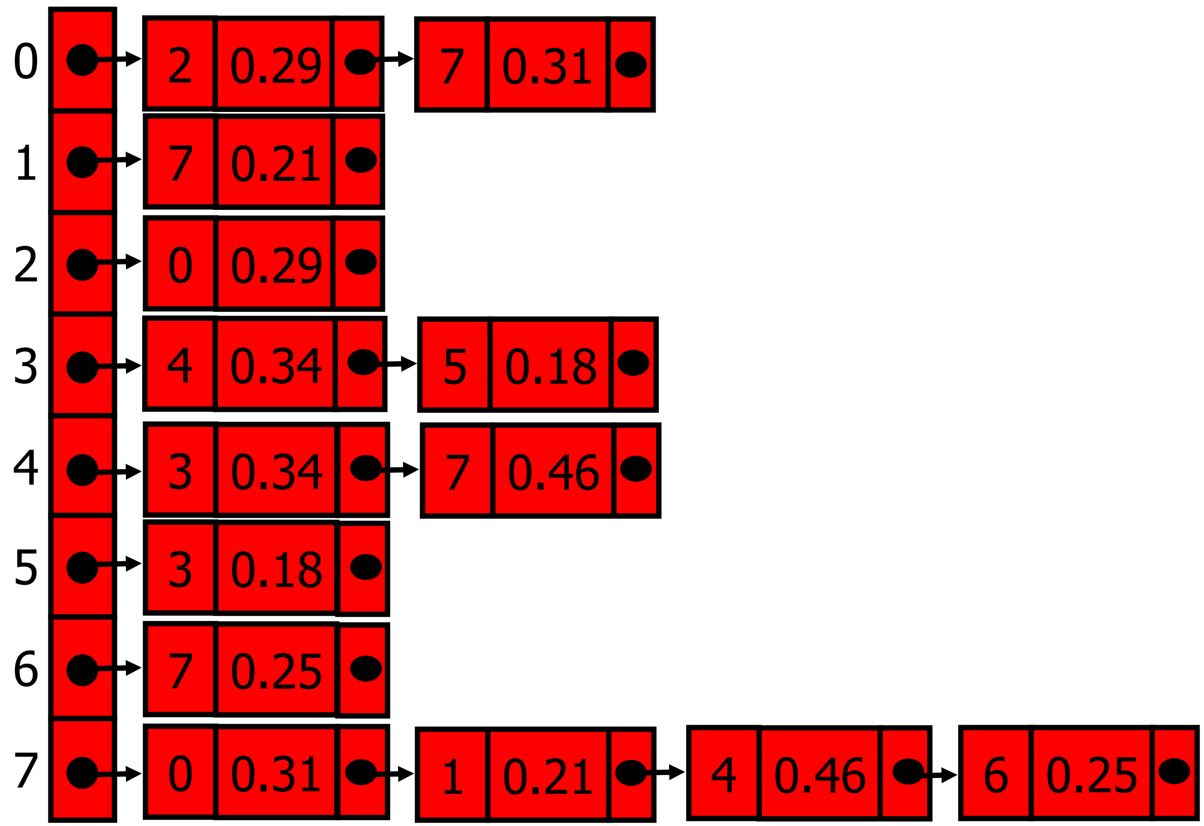


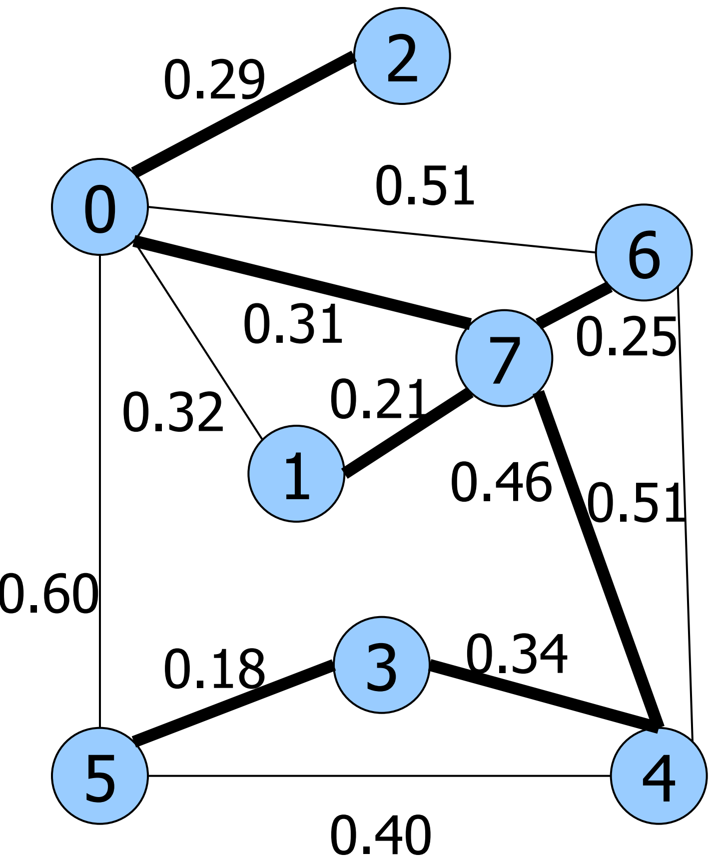
Elenco di archi, eventualmente memorizzato in un vettore

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:







Vettore `st` dei padri e `val` dei pesi

	0	1	2	3	4	5	6	7
`st`	0	7	0	4	7	3	7	0
`val`	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 = 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

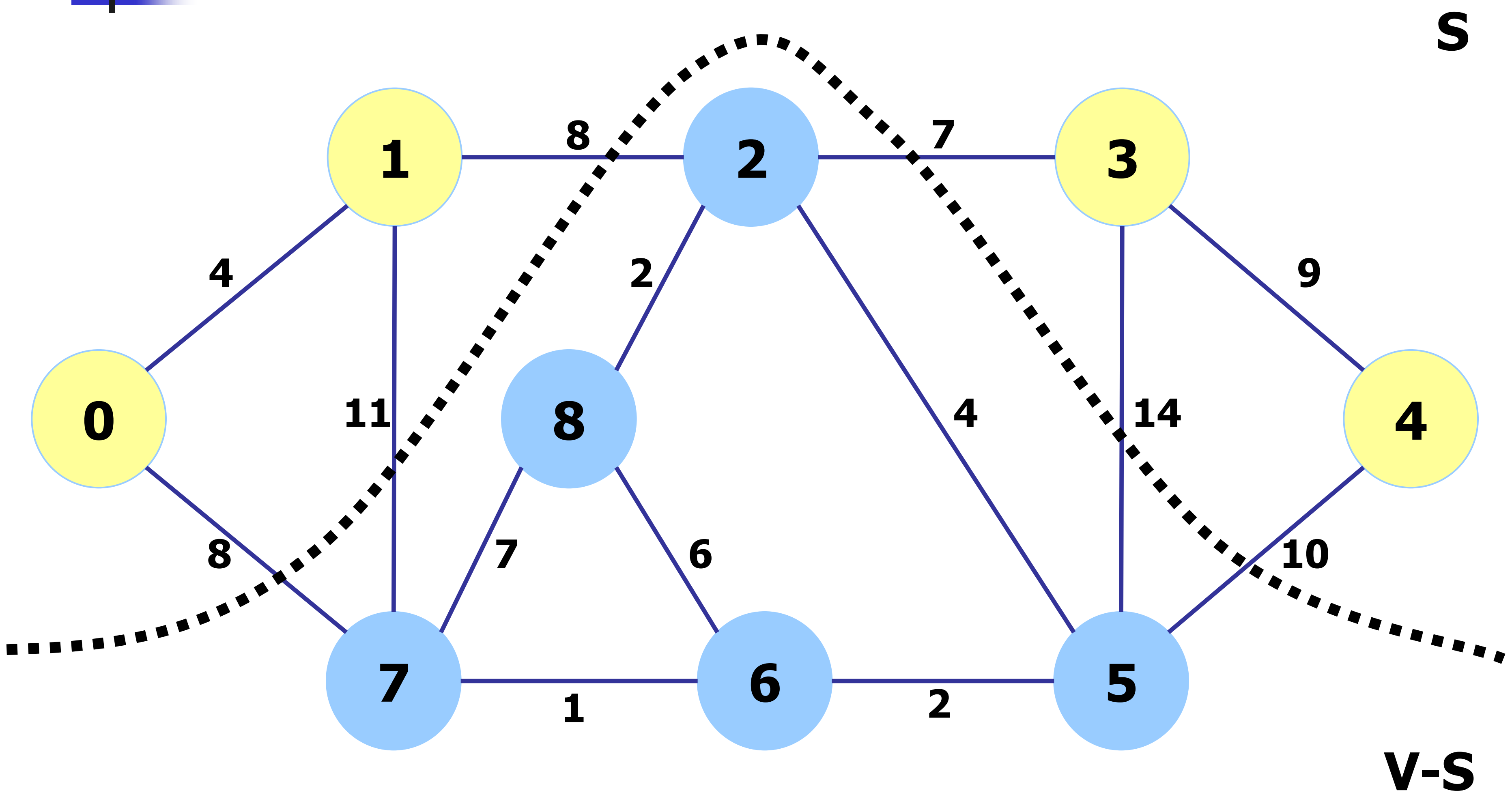
$$V = S \cup V\text{-}S \ \ \&\& \ S \cap V\text{-}S = \emptyset$$

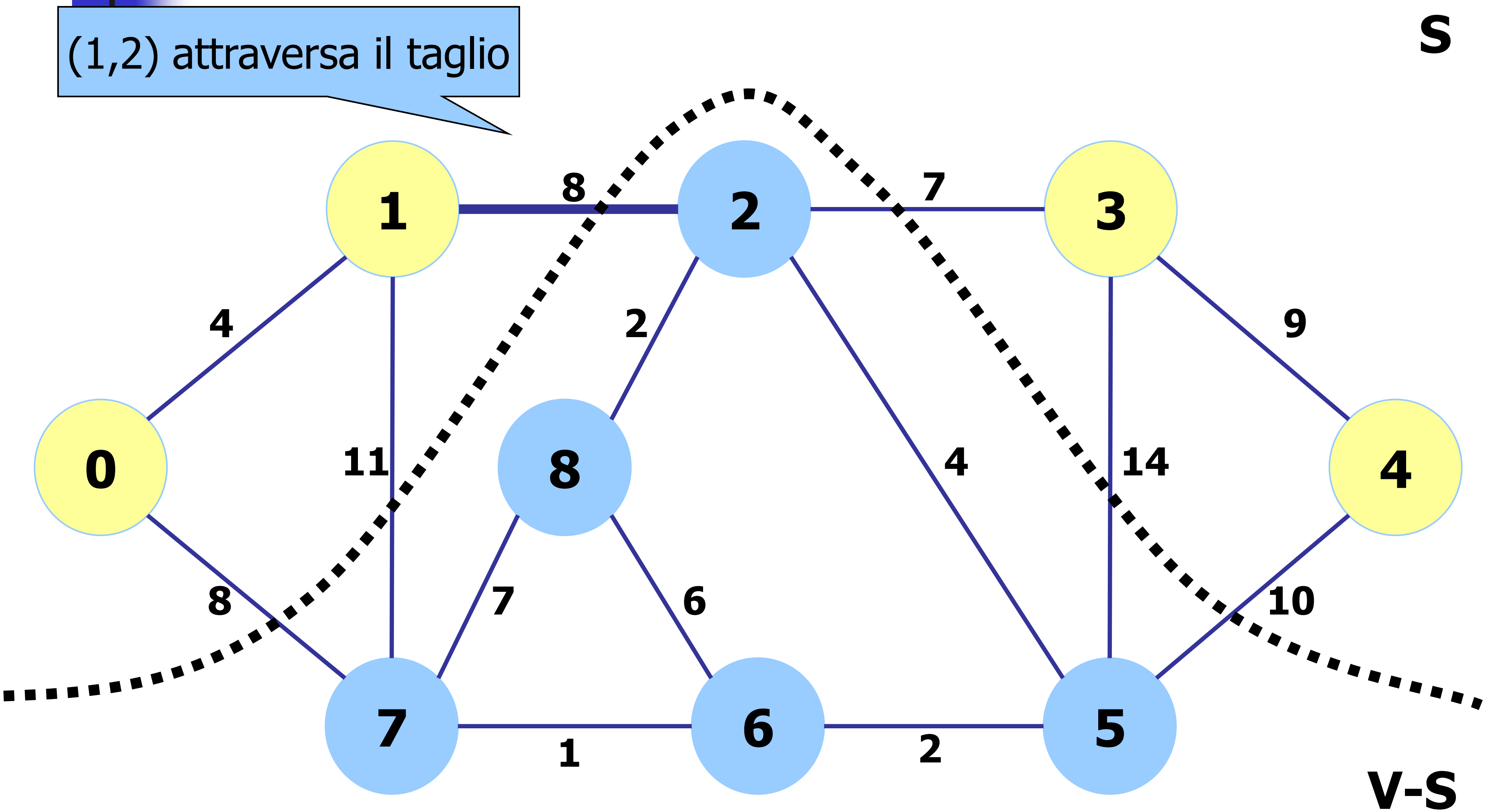
(u,v) $\in$ E attraversa il taglio $\Leftrightarrow$ u$\in$S && v$\in$V-S (o viceversa).

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

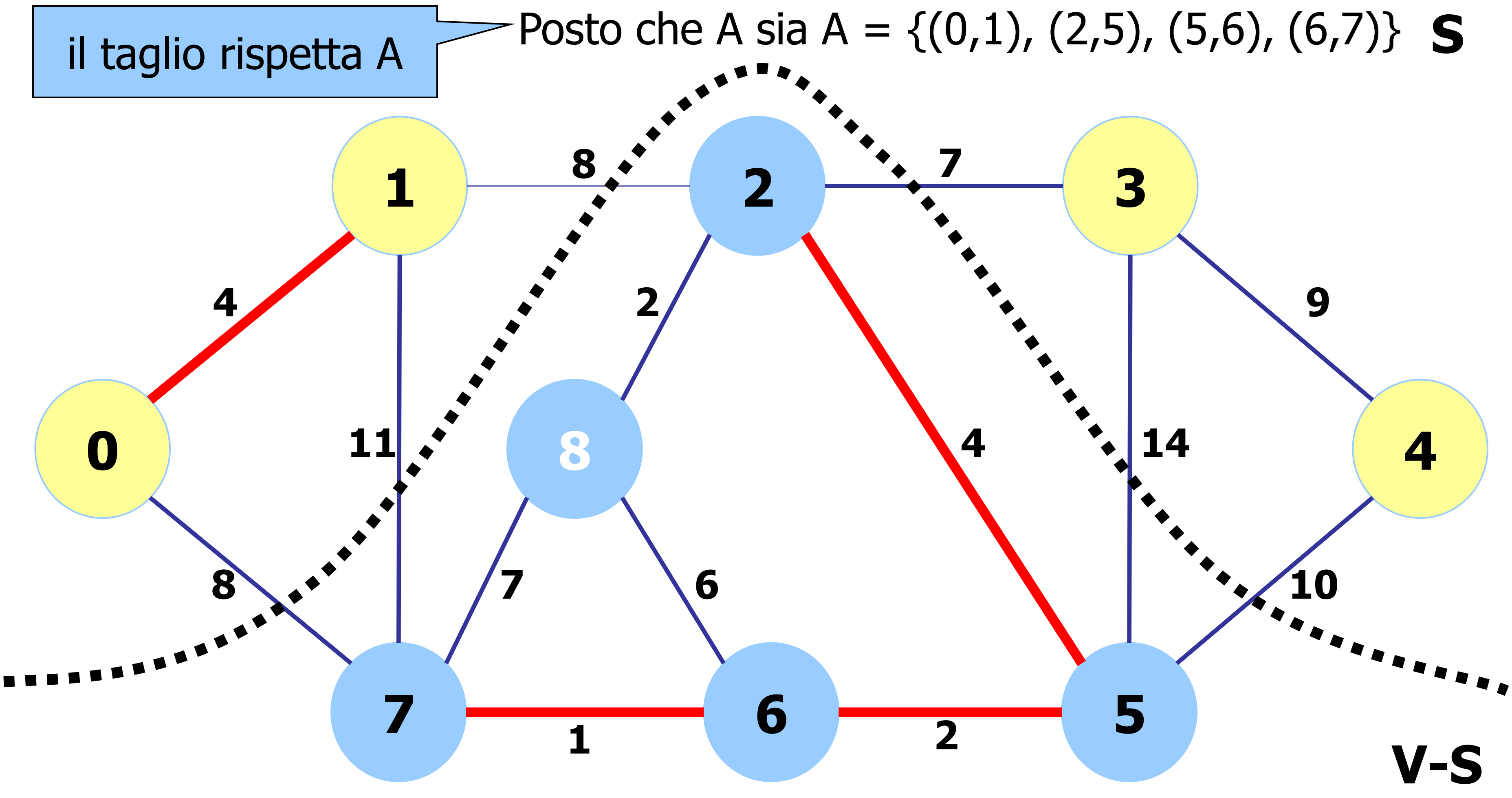
Un arco si dice leggero se ha peso minimo tra gli archi che attraversano il taglio.

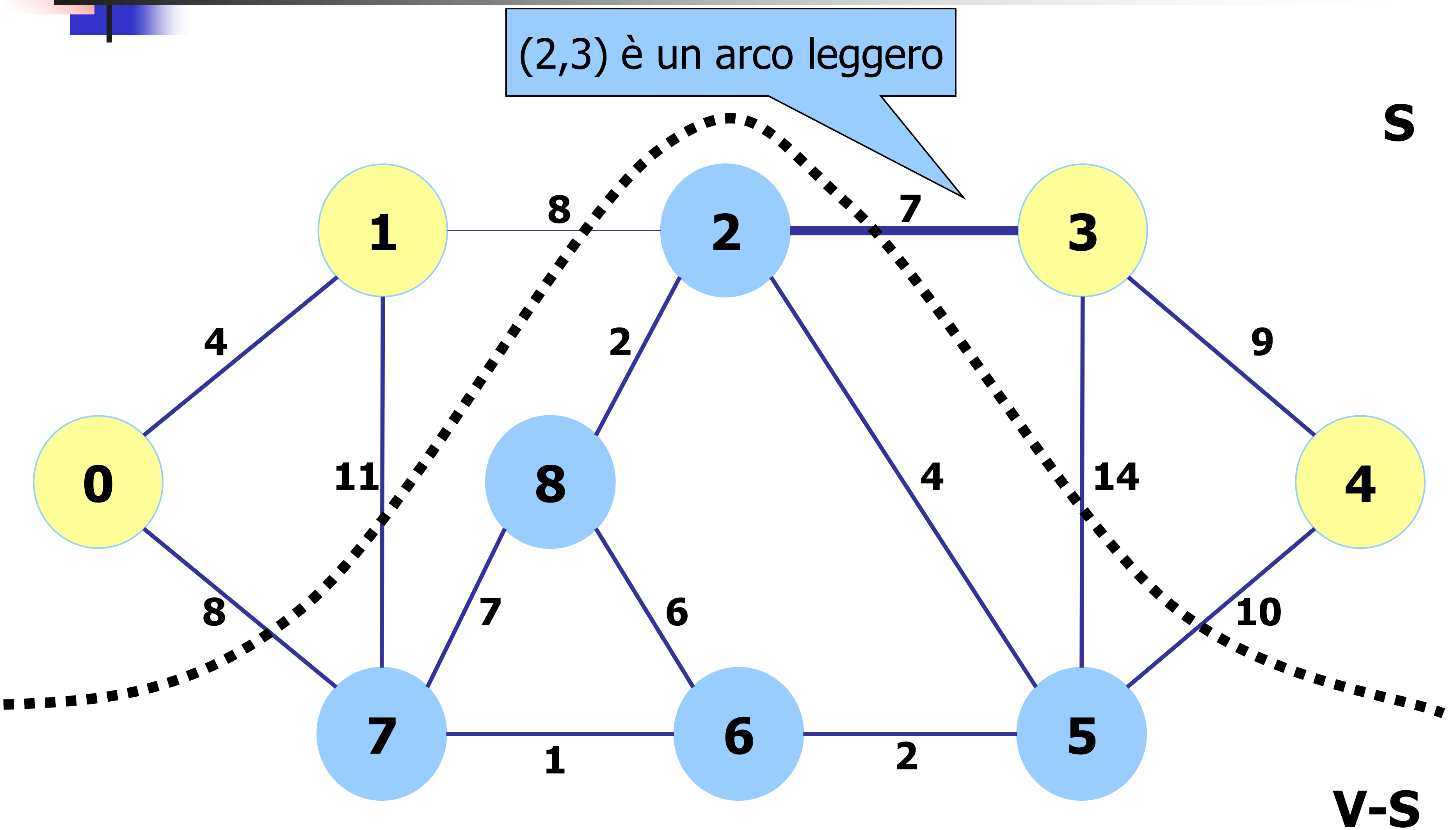
Esempio




(1,2) attraversa il taglio
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

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





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

Archi sicuri: teorema

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

- A $\subseteq$ E contenuto in un qualche albero ricoprente minimo di G
- (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.

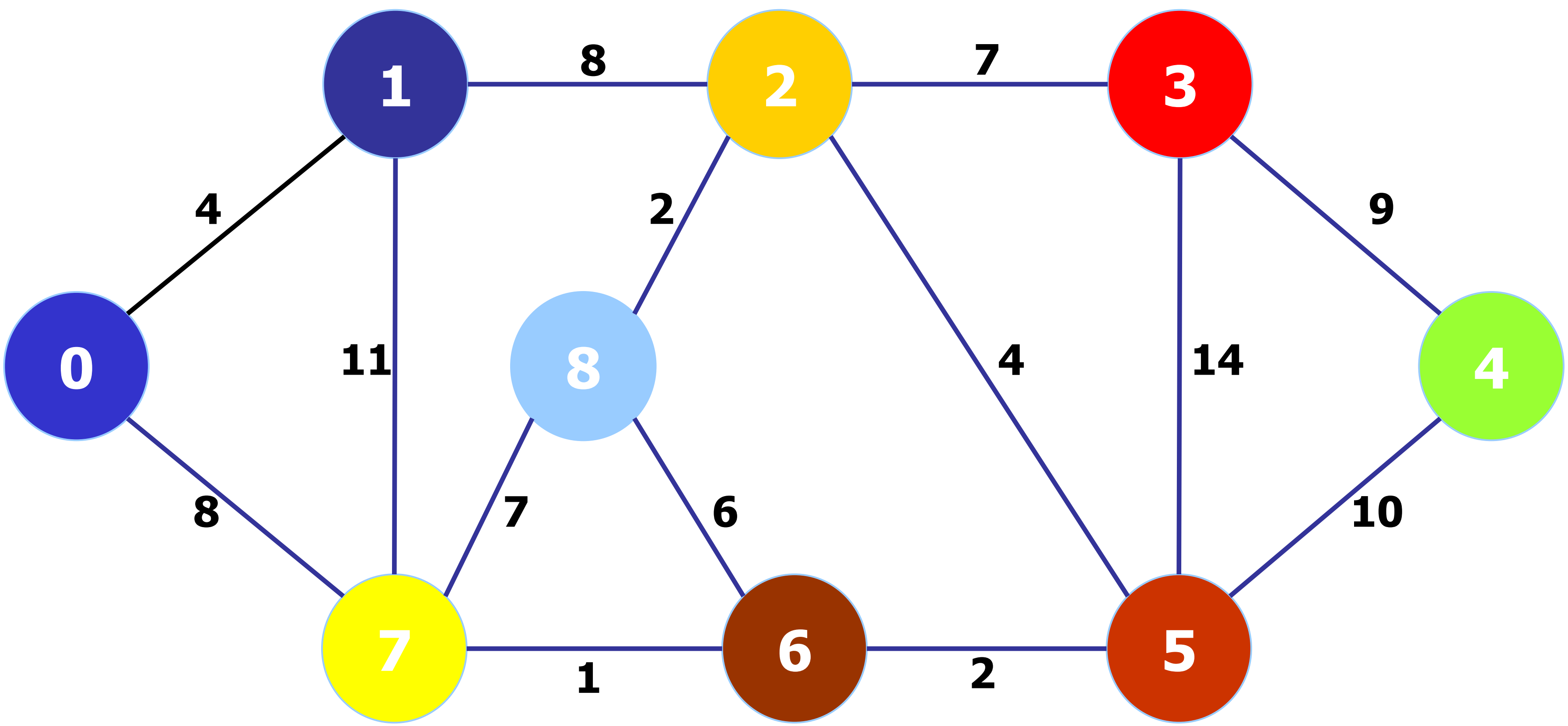
- A $\subseteq$ E contenuto in un qualche albero ricoprente minimo di G
- C albero nella foresta G_A= (V,A)
- (u,v) un arco leggero che connette C ad un altro albero in G_A

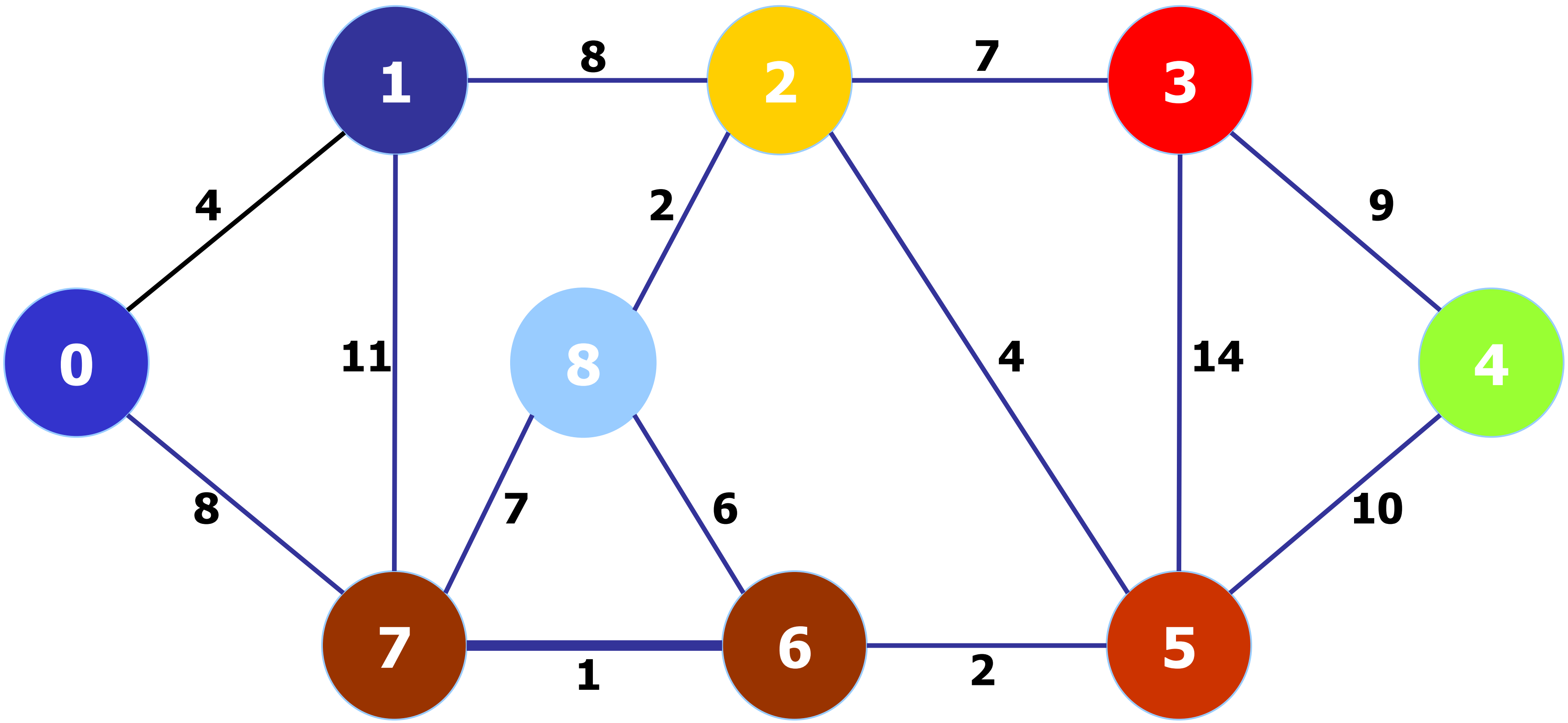
$\Rightarrow$ (u,v) è sicuro per A.

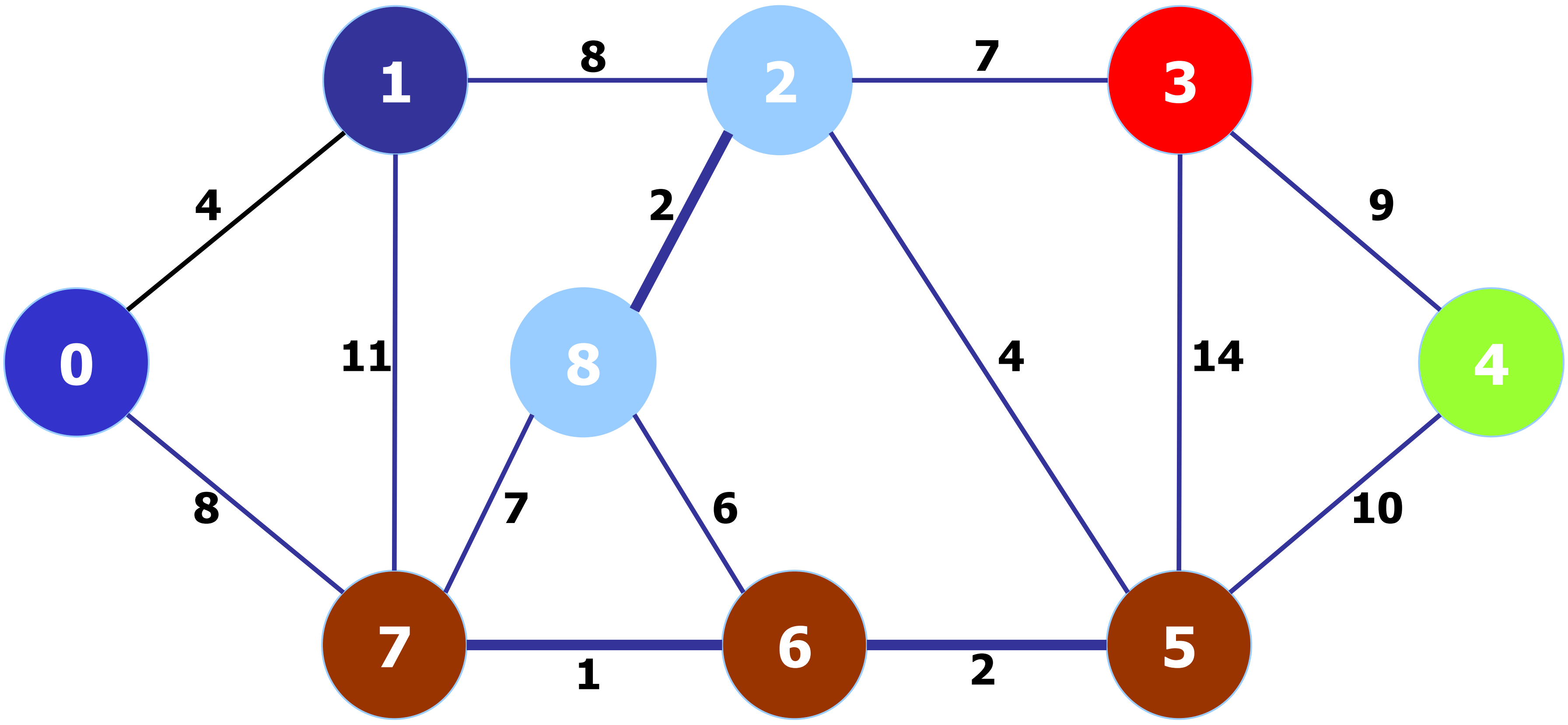
Algoritmo di Kruskal (1956)

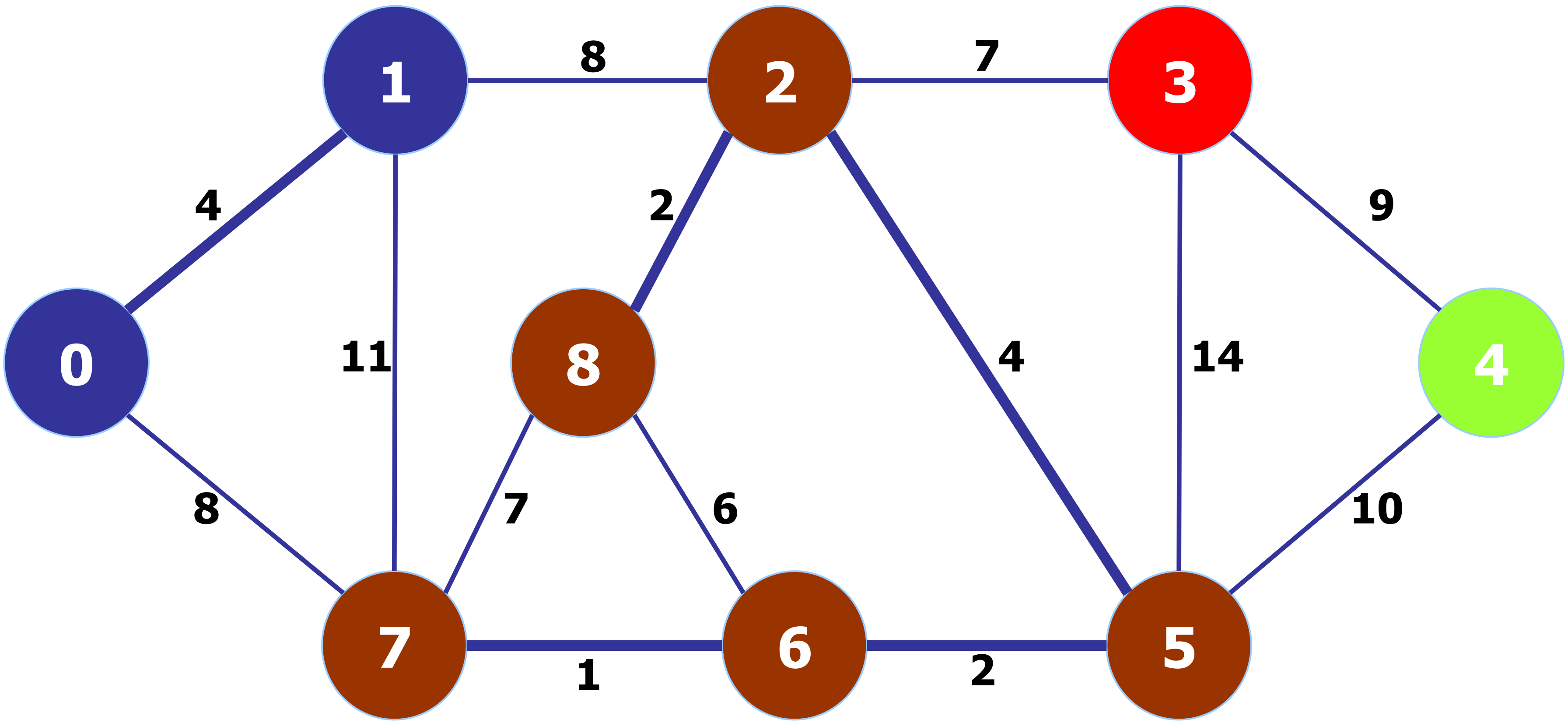
- basato su algoritmo generico
- uso del corollario per determinare l’arco sicuro:
  - foresta di alberi, inizialmente vertici singoli
  - ordinamento degli archi per pesi crescenti
  - iterazione: selezione di un arco sicuro: arco di peso minimo che connette 2 alberi generando un albero (Union-Find)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

Esempio

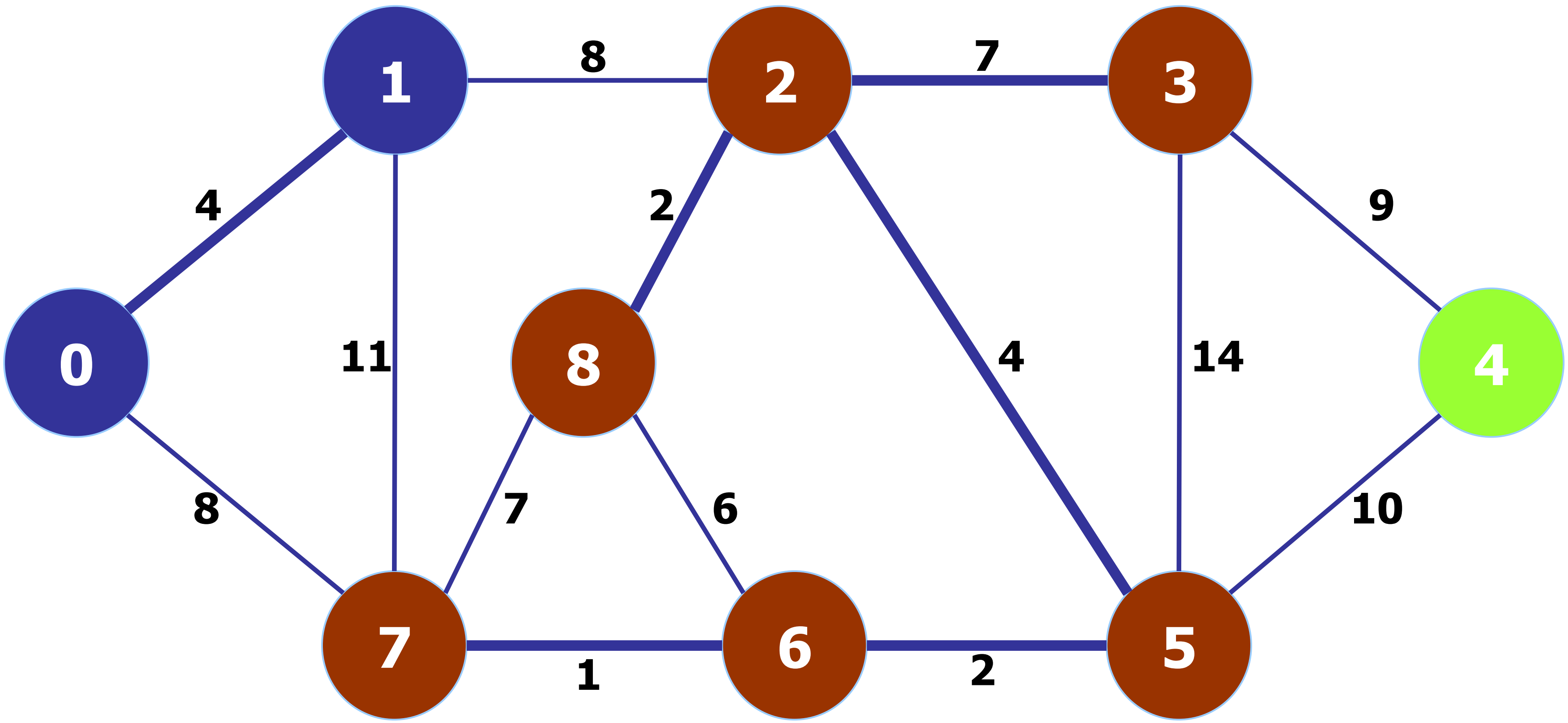



0
1
2
3
4
5
6
7
8
8
7
4
2
9
11
4
14
8
7
6
10
1
2

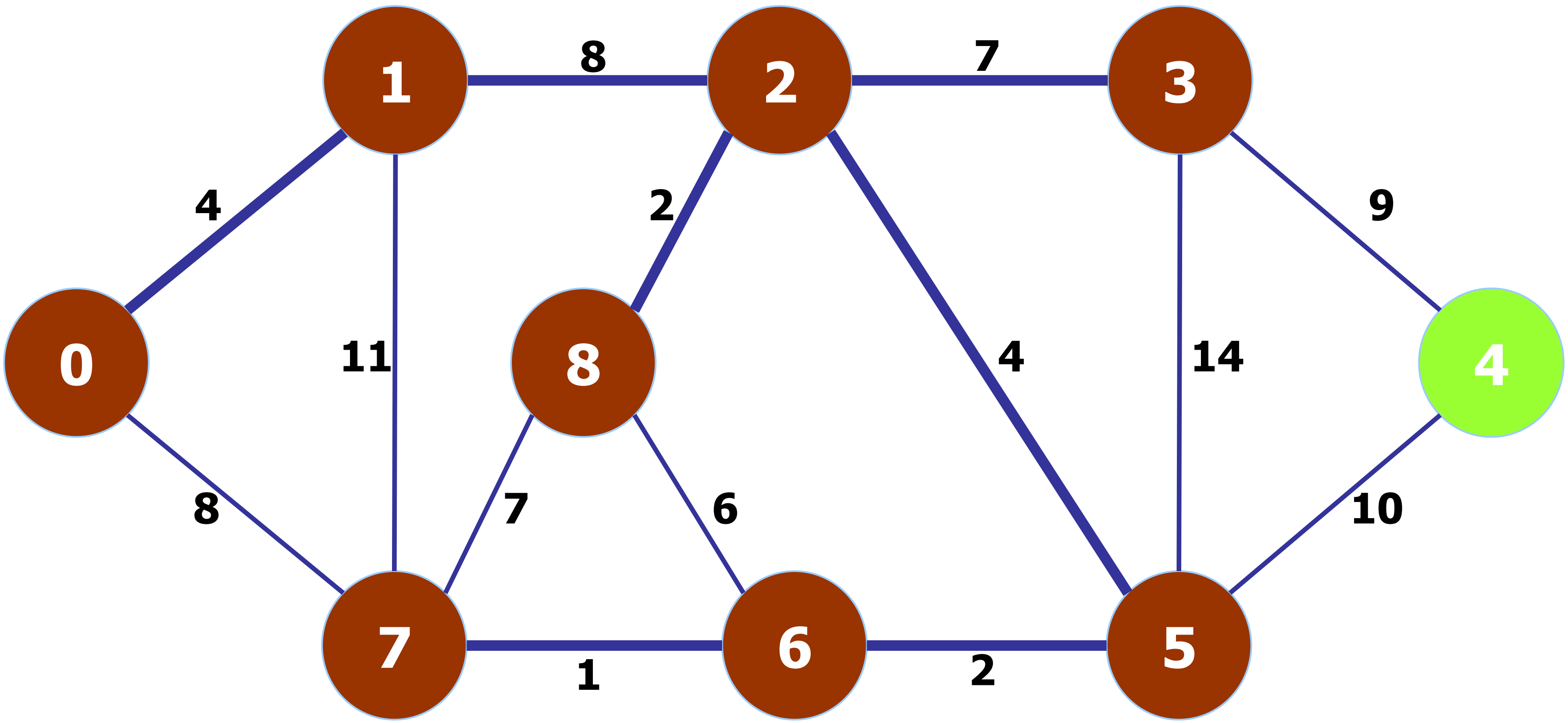

1
2
3
0
8
4
7
6
5
8
7
4
2
9
11
4
14
8
7
6
10
1
2

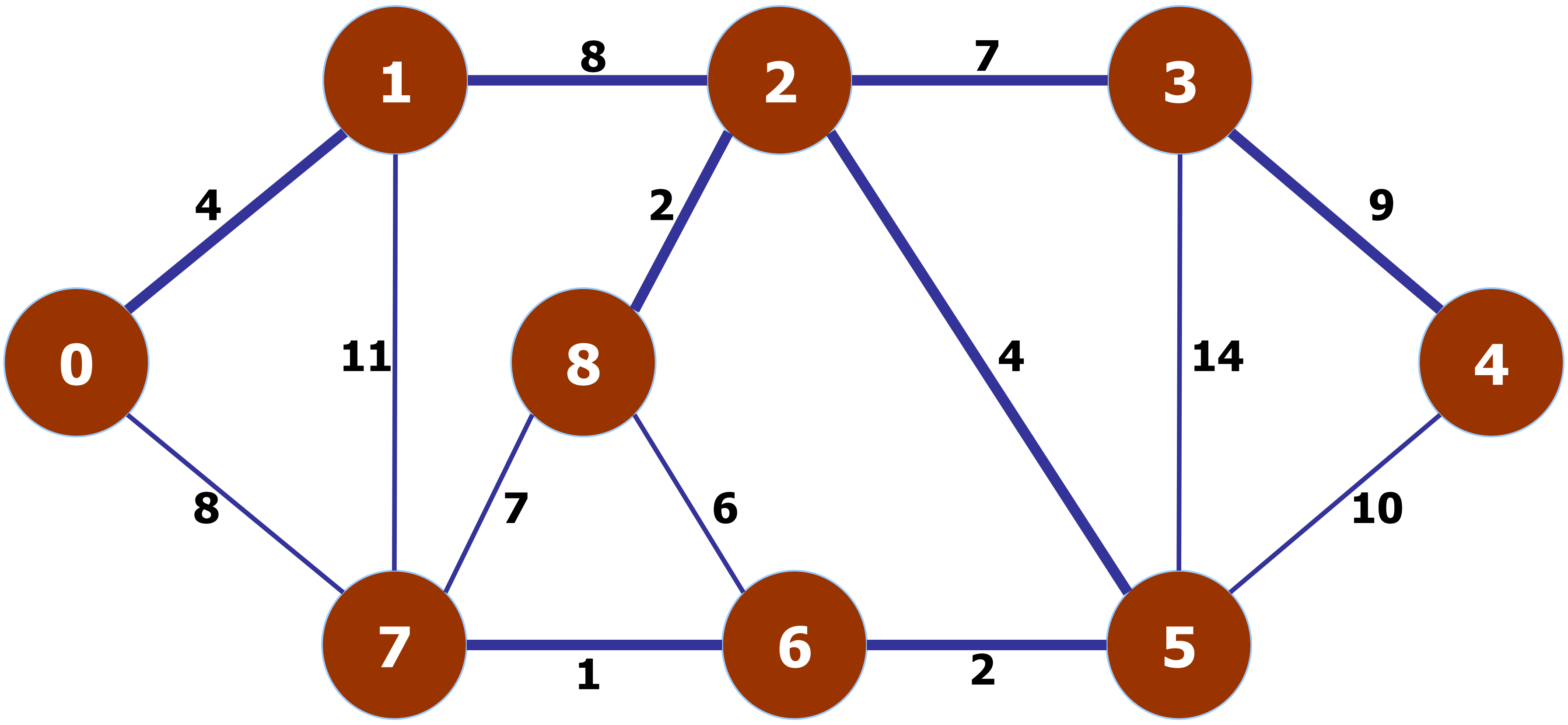


1
2
3
0
8
4
7
6
5
8
7
4
2
9
11
4
14
8
7
6
10
1
2



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


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


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

Interfaccia per Union-Find

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

Implementazione per Union-Find

```
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];
  }
}
```

```
int GRAPHmstE(Graph G, Edge mst[]) {
  int i, k;
  Edge a[maxE];

  E = GRAPHedges(a, G);

  sort(a, 0, E-1);

  UFinit(G->V);

  for ( i=0, k=0; i < 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;
}
```

```c
void GRAPHmstK(Graph G) {
    int i, k, weight = 0;
    k = GRAPHmstE(G, mst);
    printf("the list of edges in the MST is: \n");
    for (i=0; i < k; i++) {
      printf("(%d - %d) \n", mst[i].v, 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 A = {r}
  - iterazione:
    - determina gli archi che attraversano il taglio
    - tra questi, seleziona l’arco di peso minimo e aggiungilo ad A
    - in base al vertice in cui arriva, aggiorna l’insieme degli archi che attraversano il taglio
  - terminazione: considerati tutti i vertici.

Struttura dati

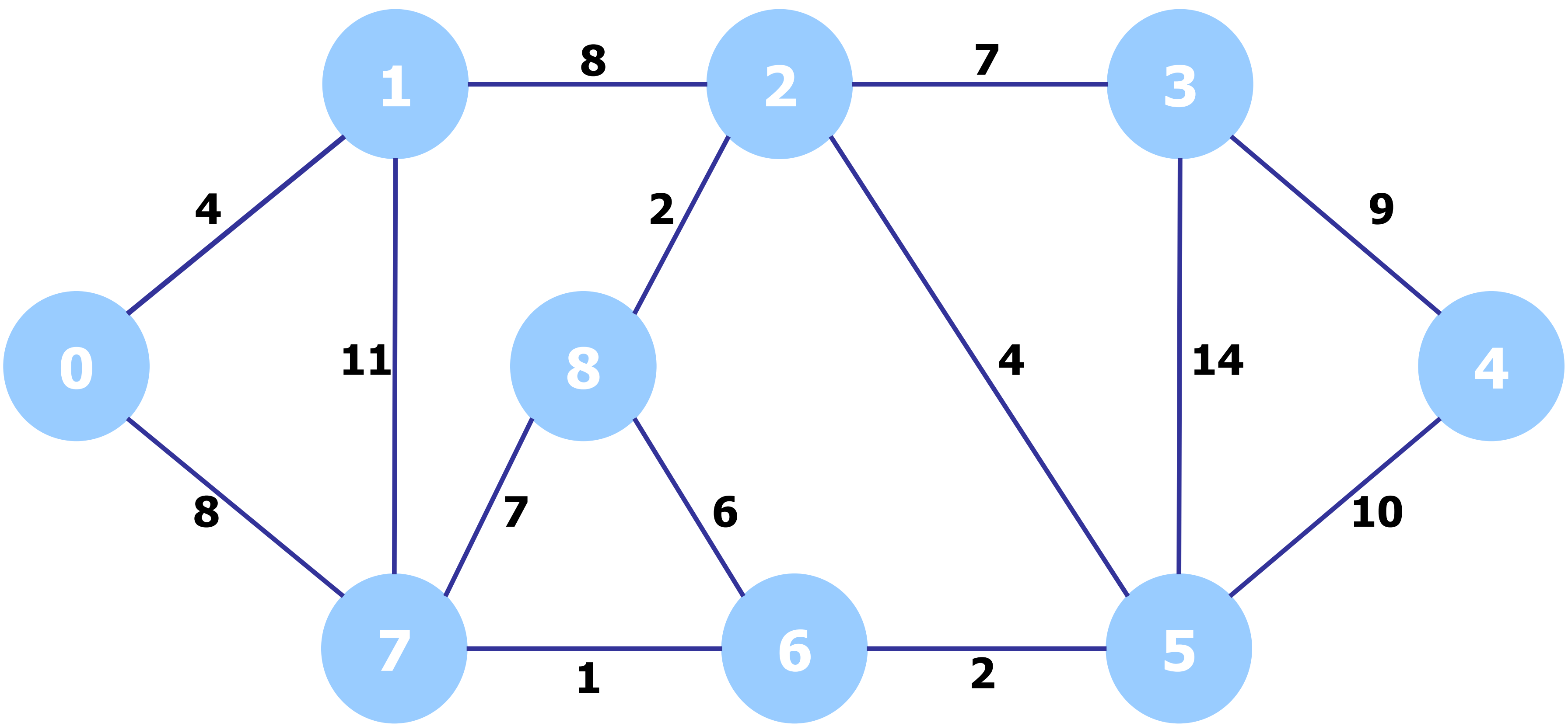
- Vettore `st` per registrare il padre di un vertice che appartiene ad A
- Vettore `fr` per registrare per ogni vertice di V-A quale è il vertice di A più vicino
- Vettore `wt` per registrare:
  - per vertici di A il peso dell'arco al padre
  - per vertici di V-A il peso dell'arco verso il vertice di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

Quando si aggiunge ad A un nuovo arco (e un nuovo vertice):

- si controlla se il nuovo arco ha portato qualche vertice di V-A più vicino a vertici di A
- 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

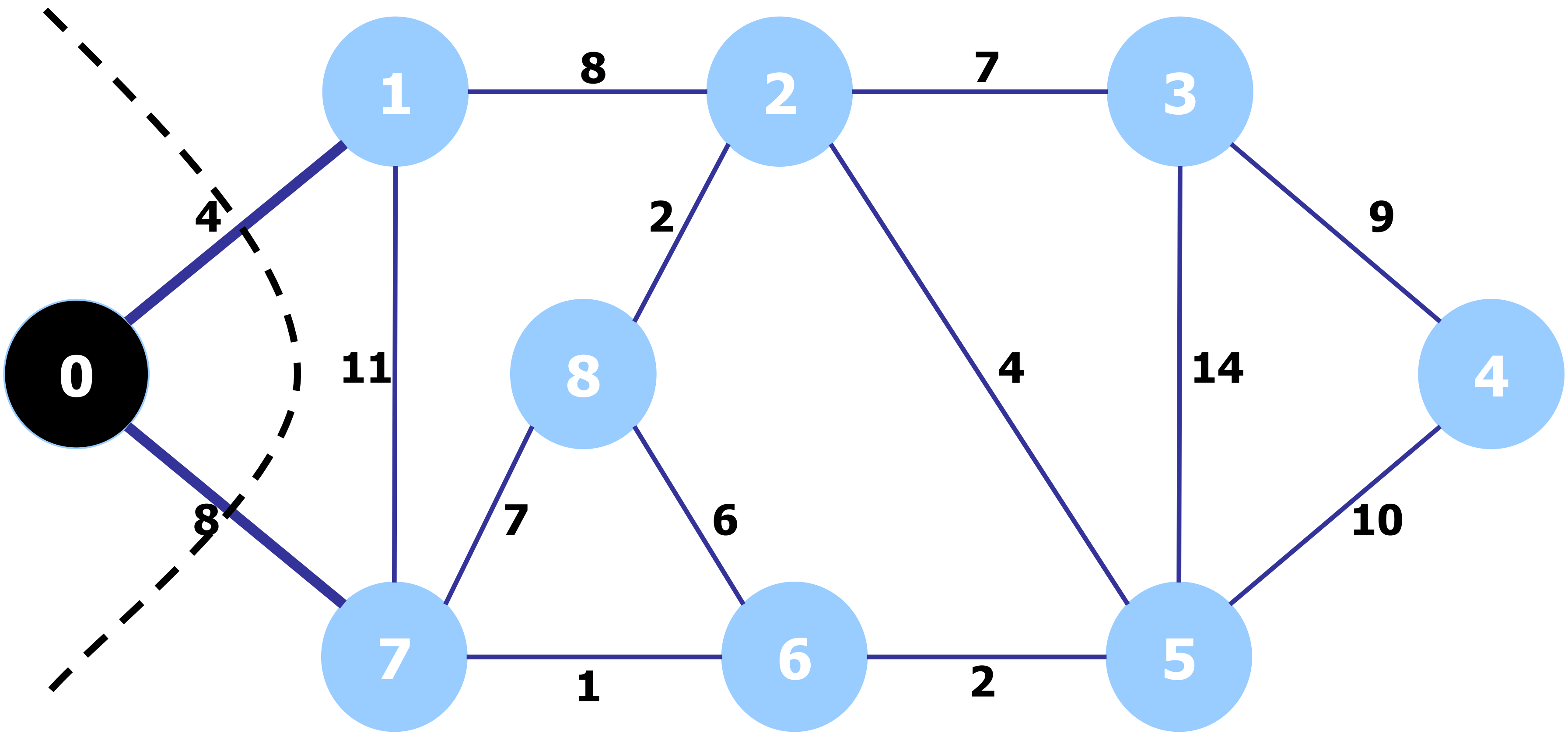


$A = \emptyset$

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

	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

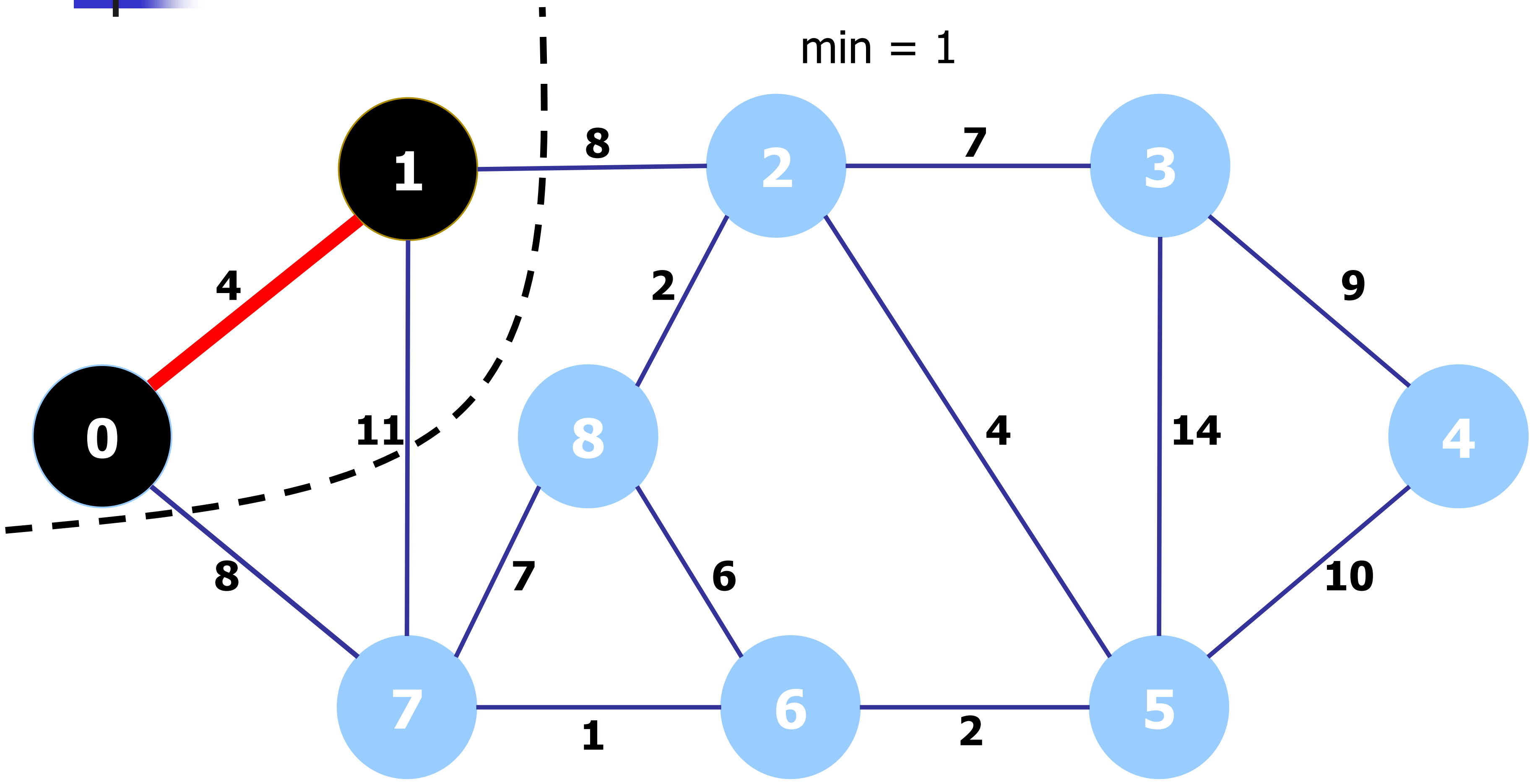


A = {0}

V-A = {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

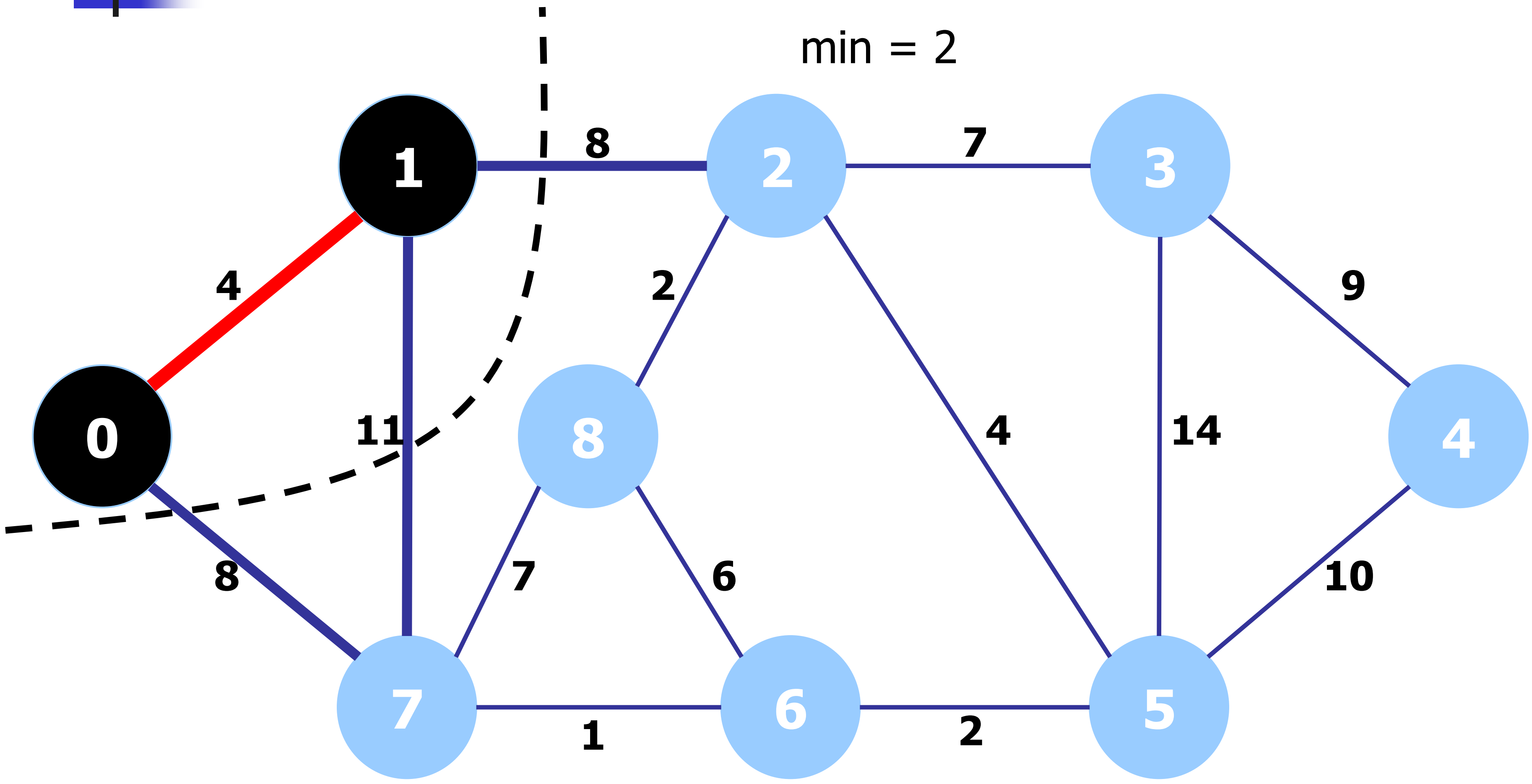


A = {0,1}

V-A = {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

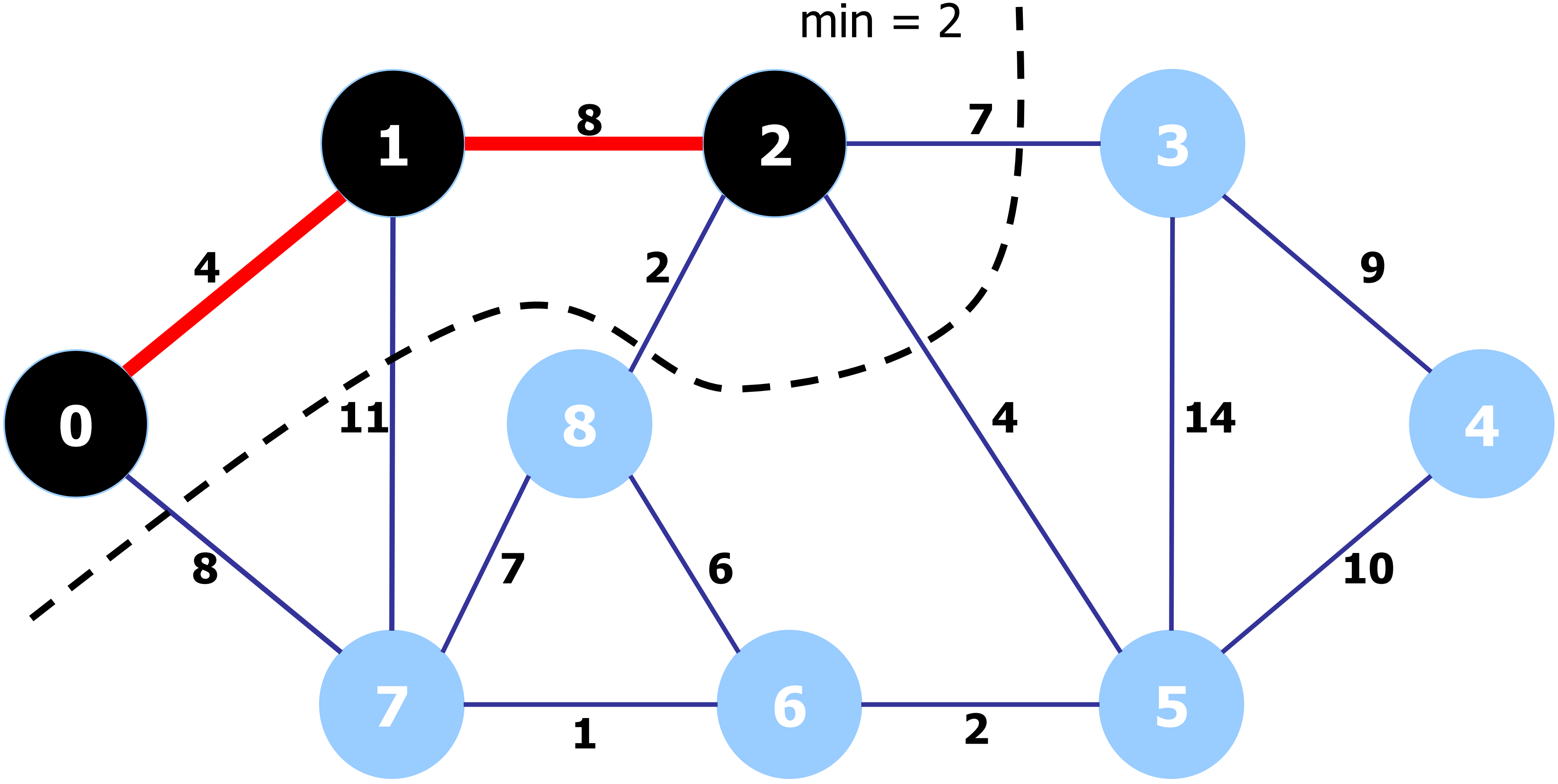


A = {0,1}

V-A = {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

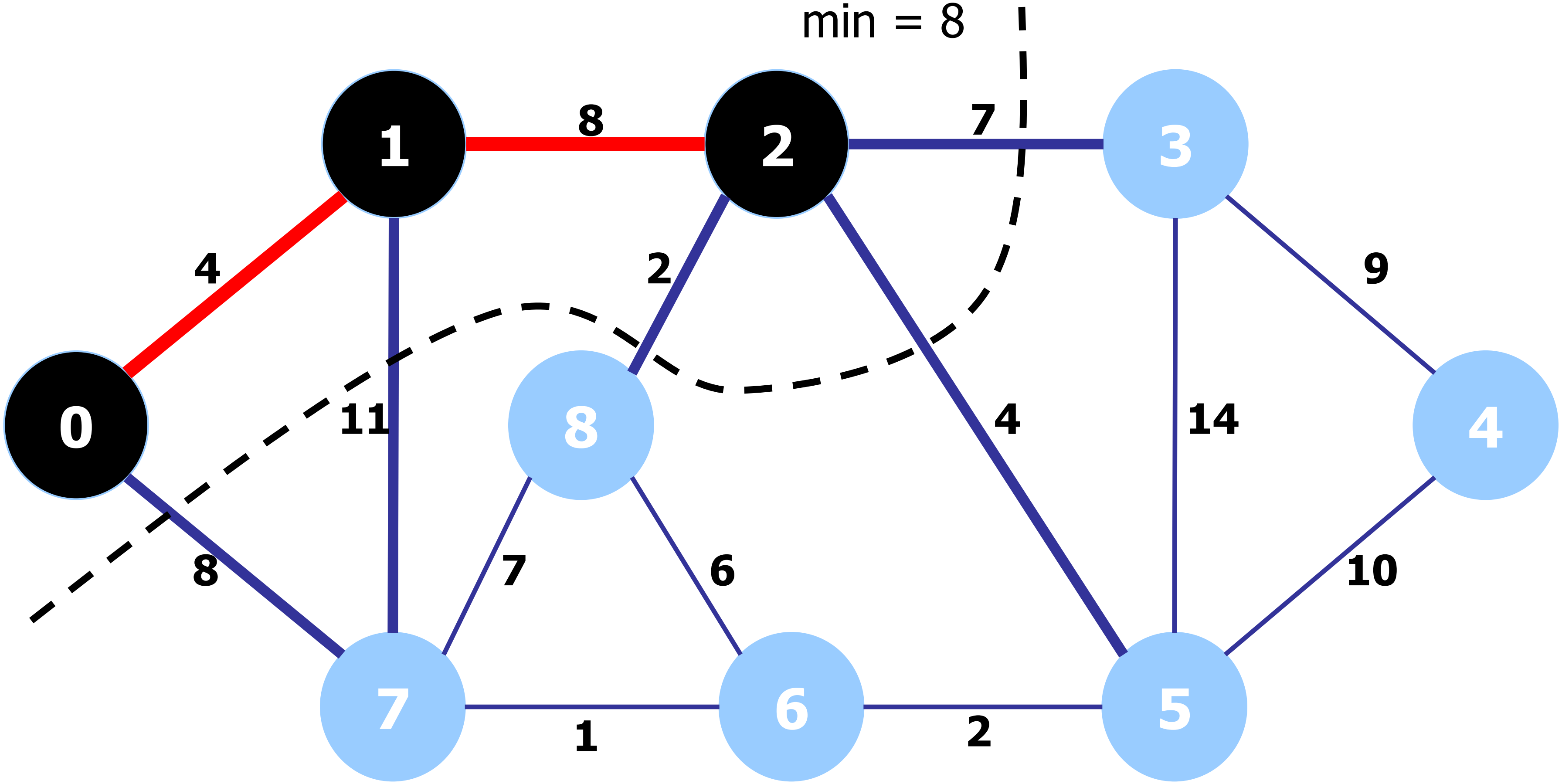


A = {0,1,2}

V-A = {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

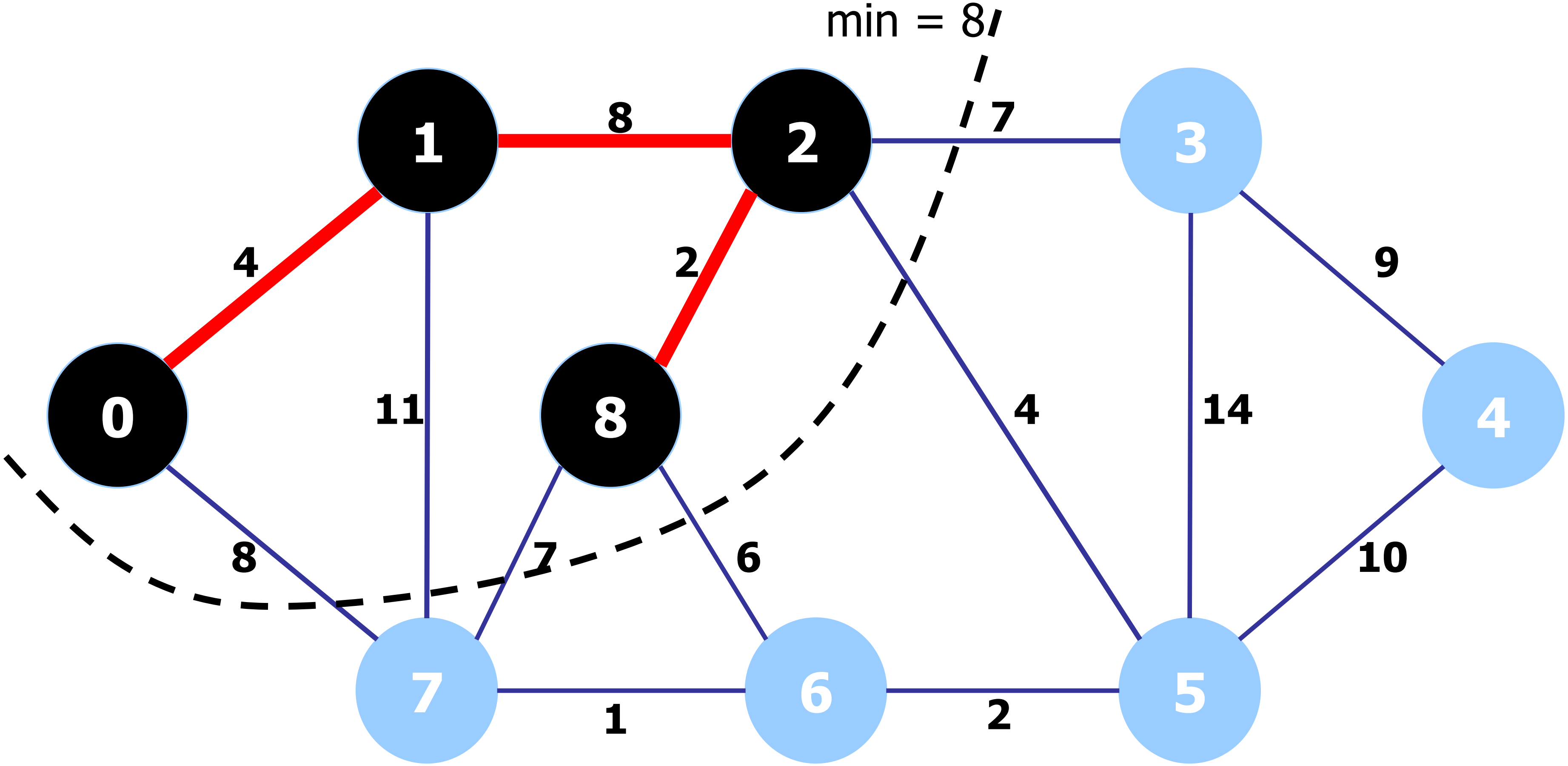


A = {0,1,2}

V-A = {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

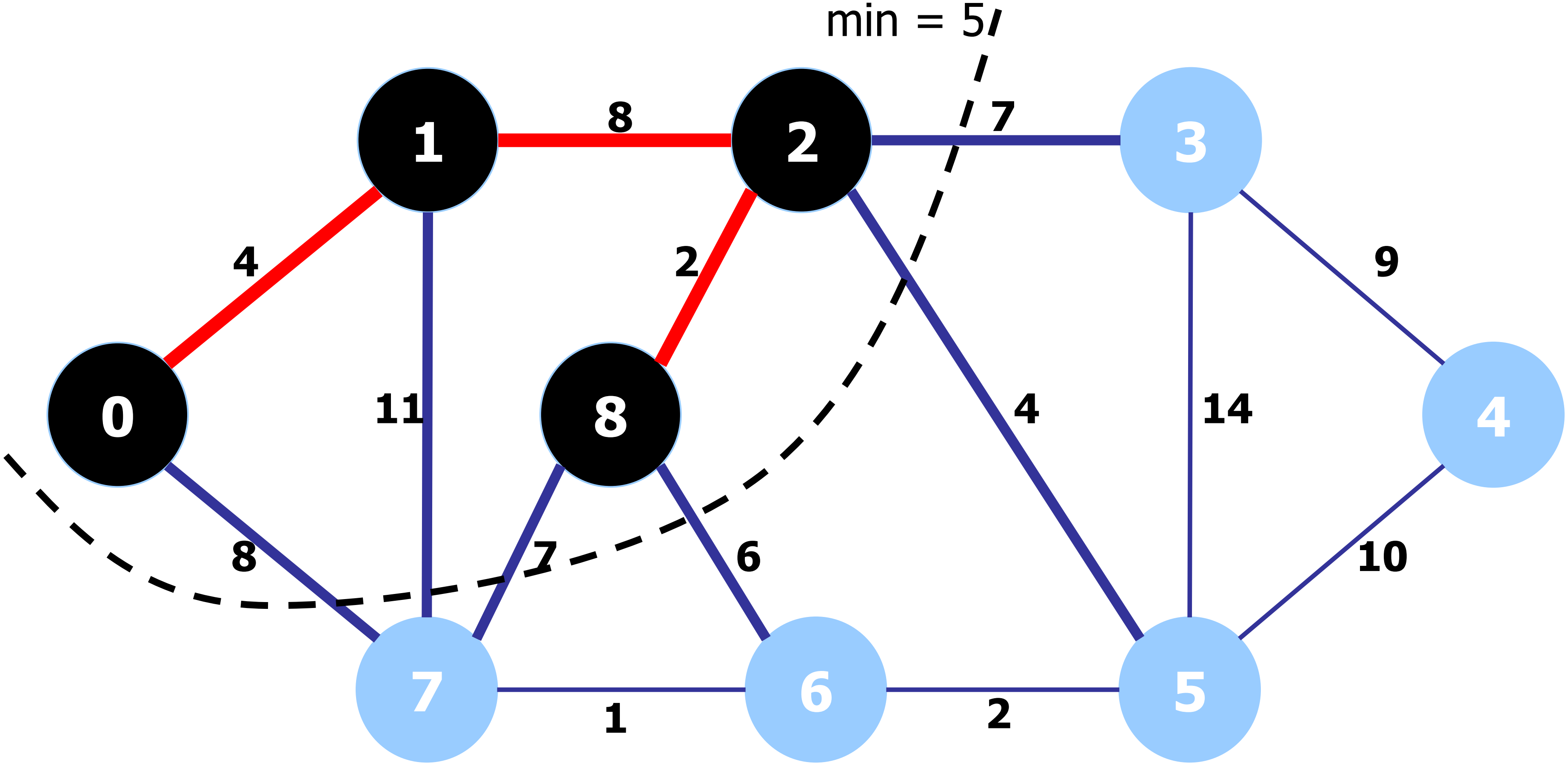


A = {0,1,2,8}

V-A = {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

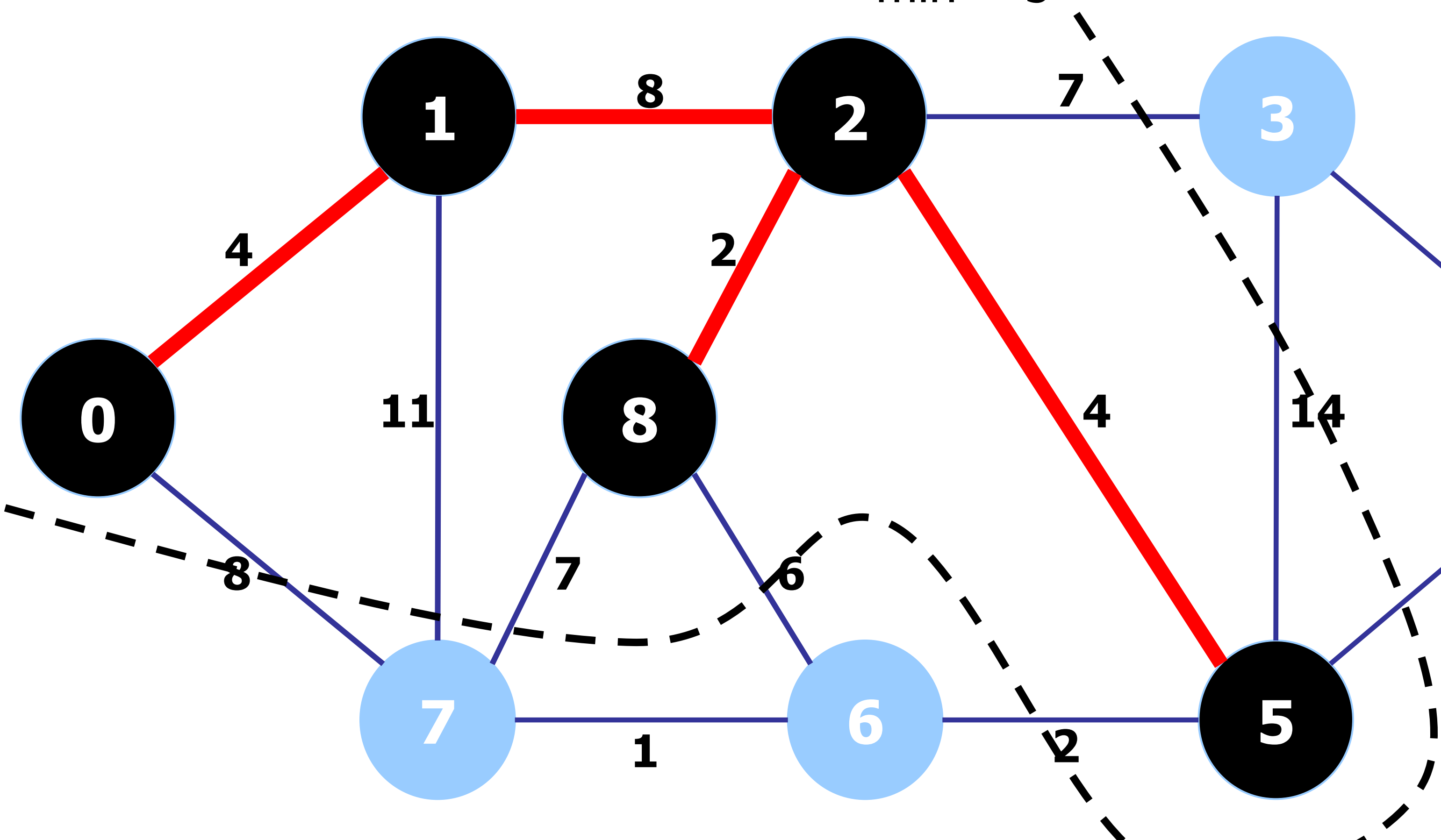


A = {0,1,2,8}

V-A = {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

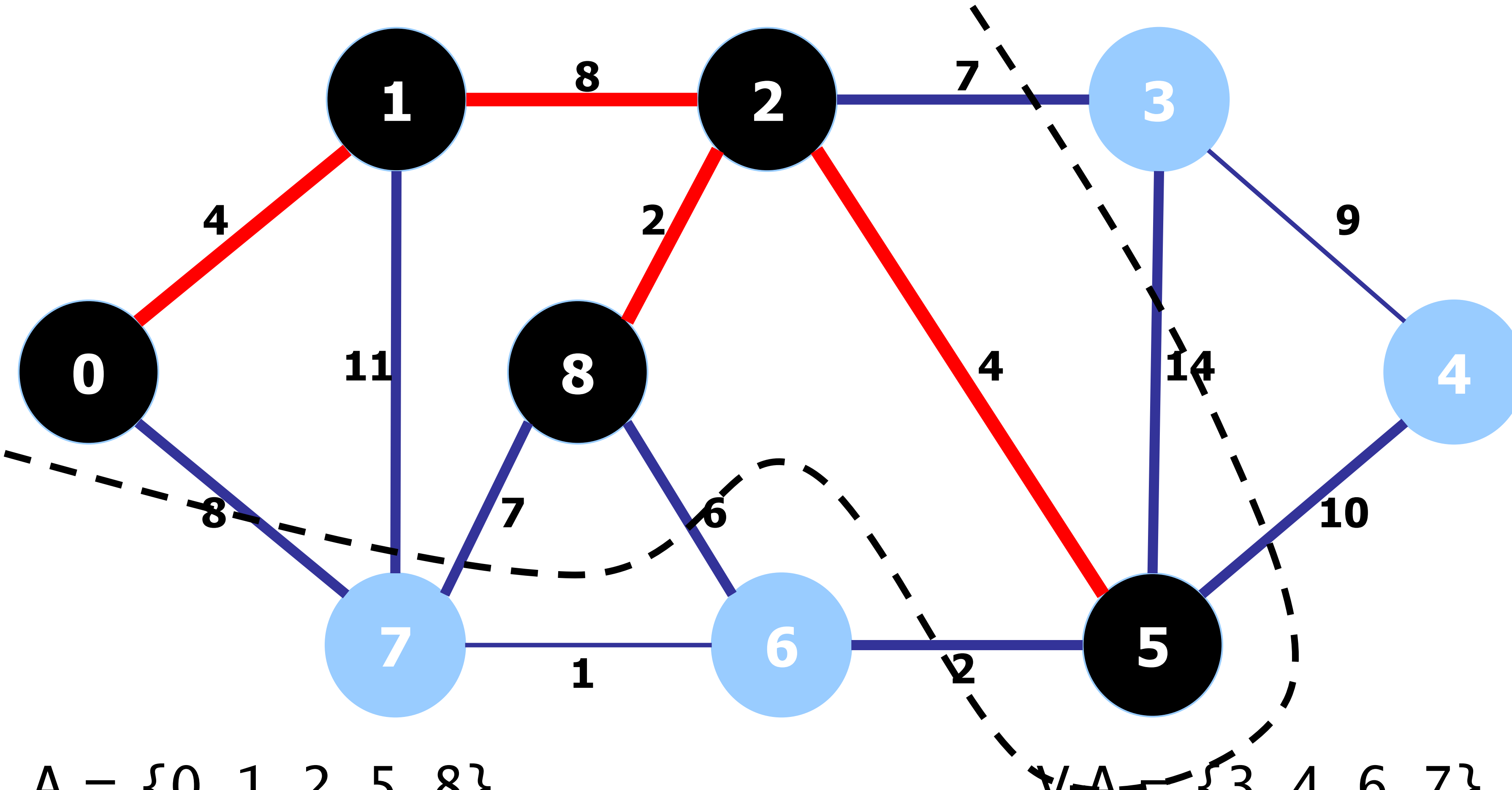


A = {0,1,2,5,8}

V-A = {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

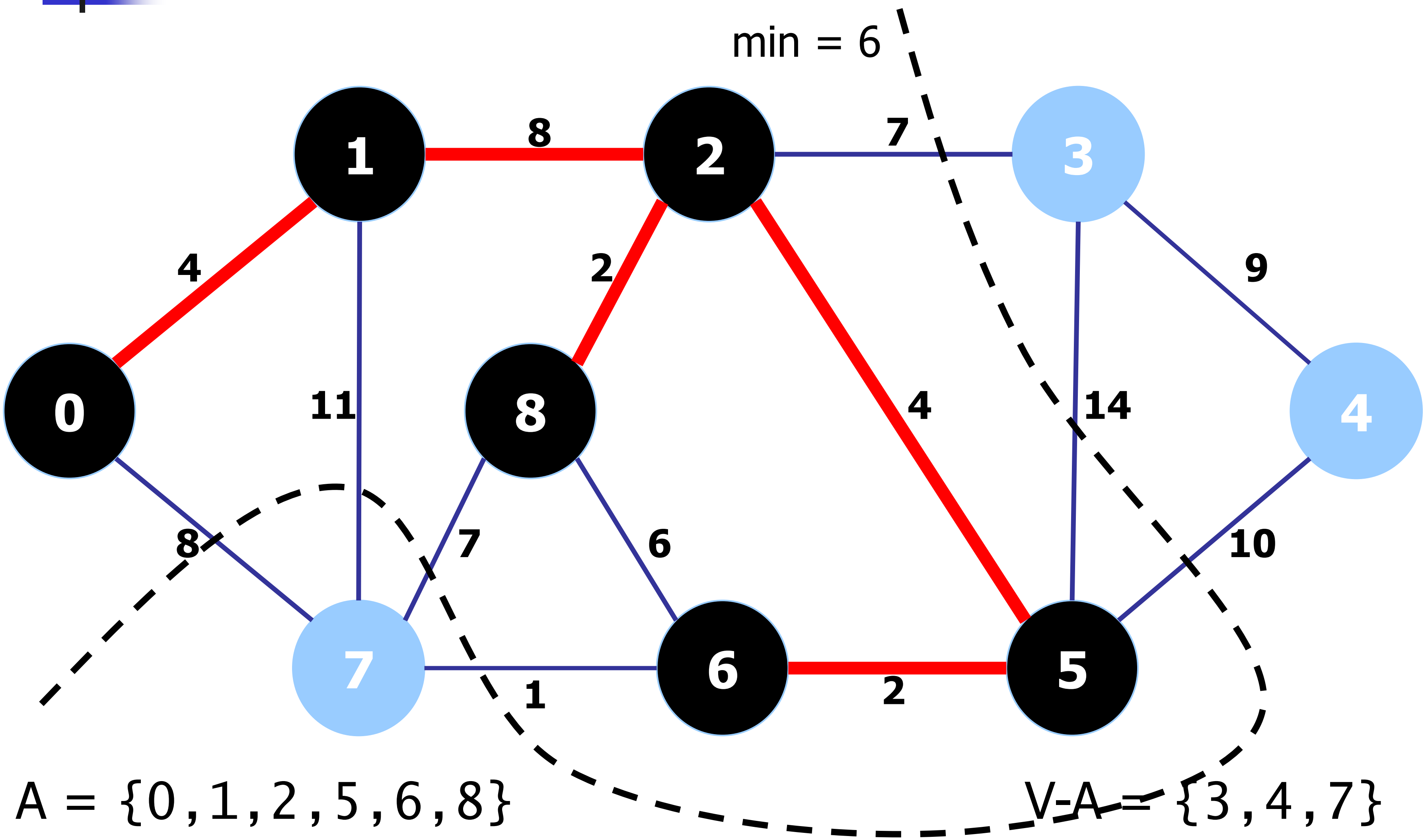


A = {0,1,2,5,8}

V-A = {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

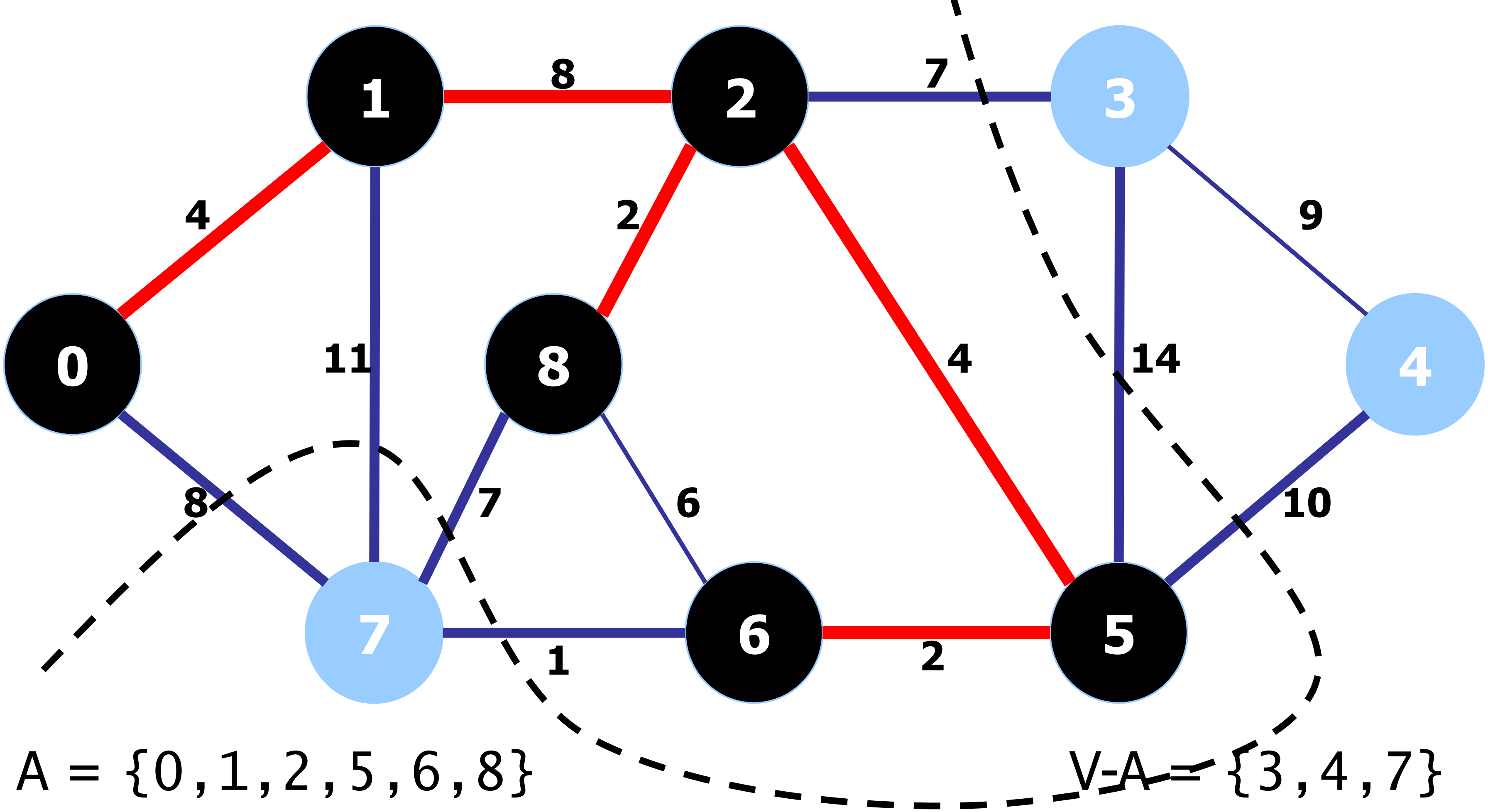


A = {0,1,2,5,6,8}

V-A = {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

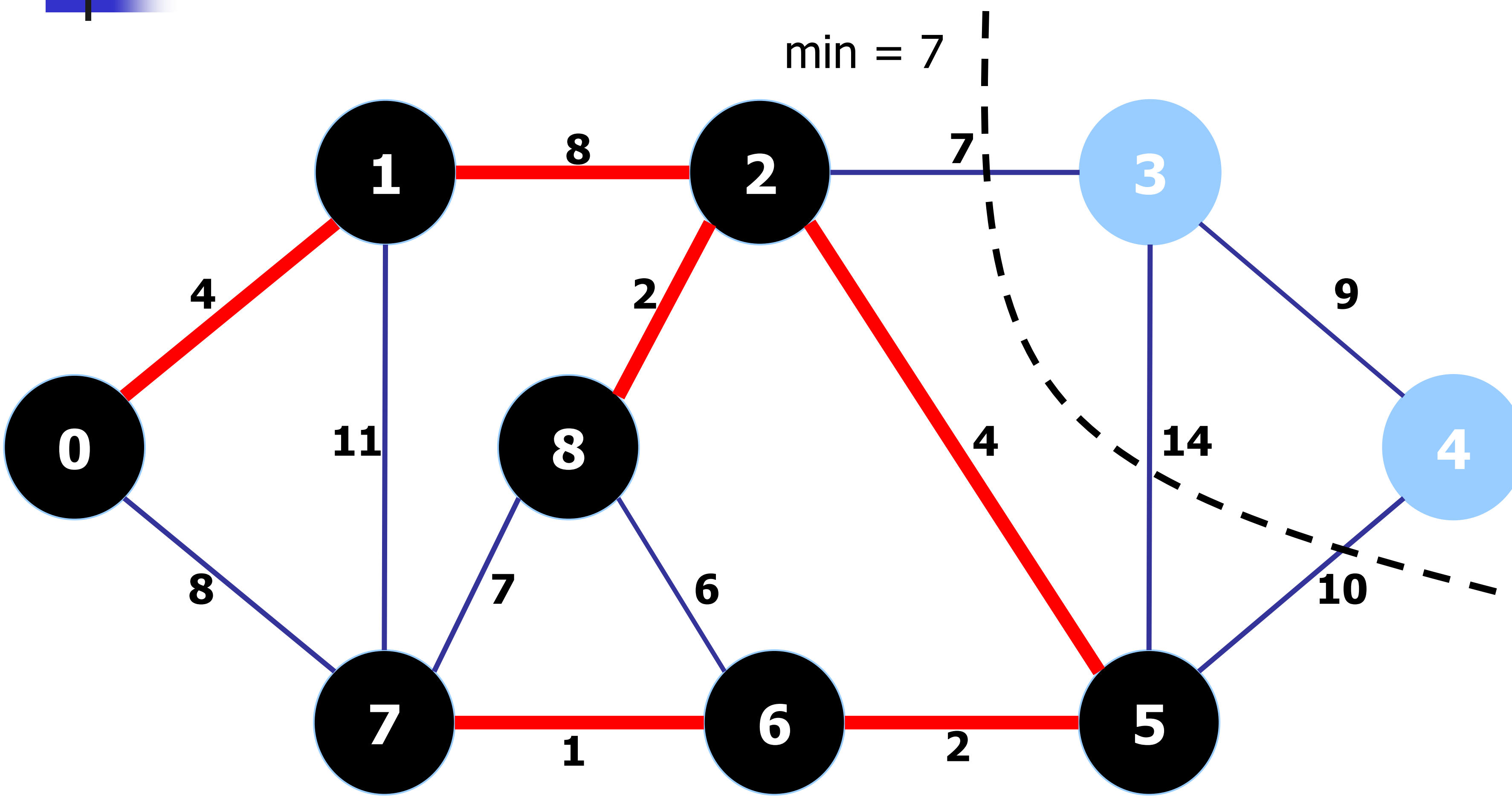


A = {0,1,2,5,6,8}

V-A = {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

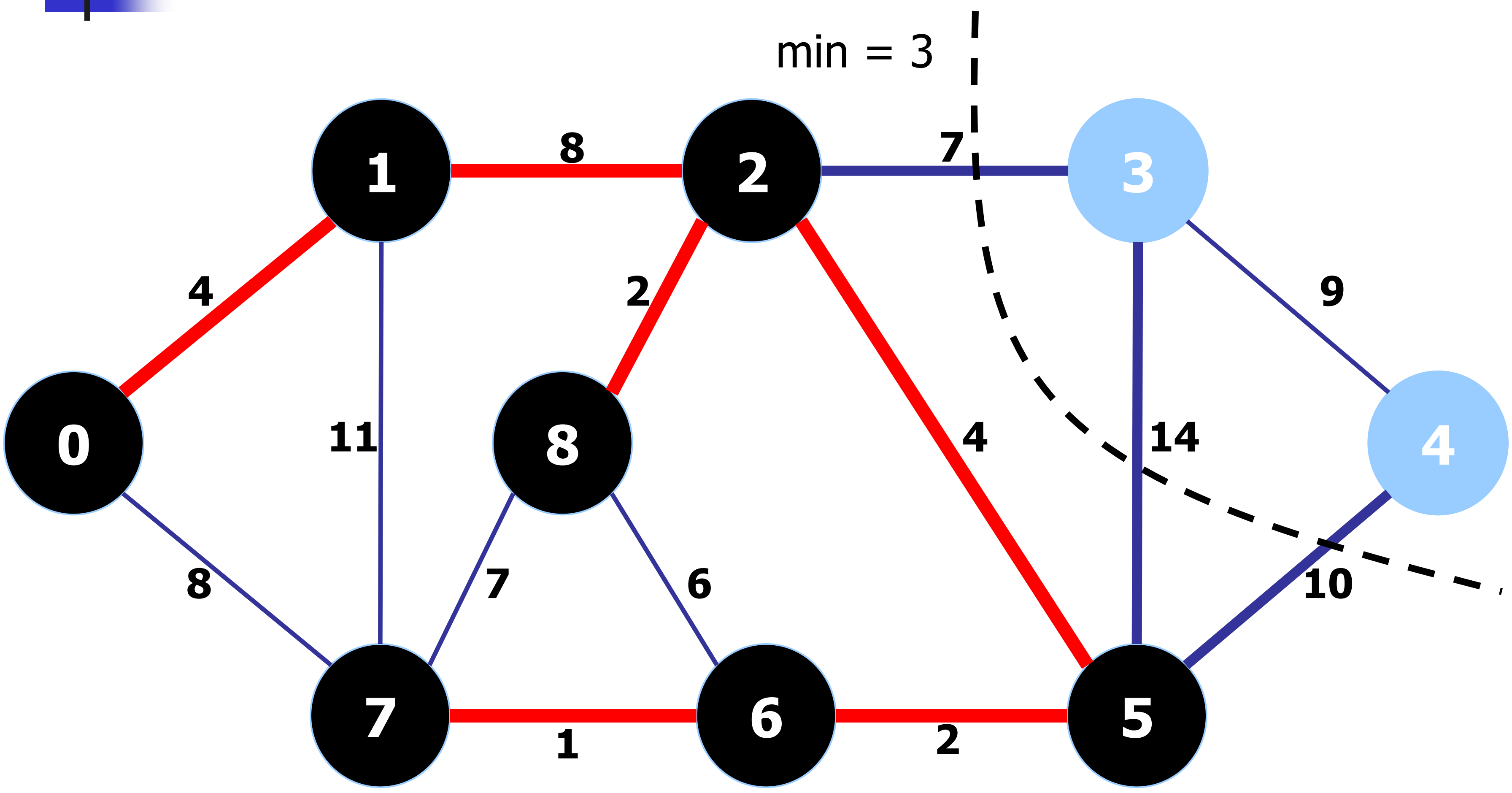


A = {0,1,2,5,6,7,8}

V-A = {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

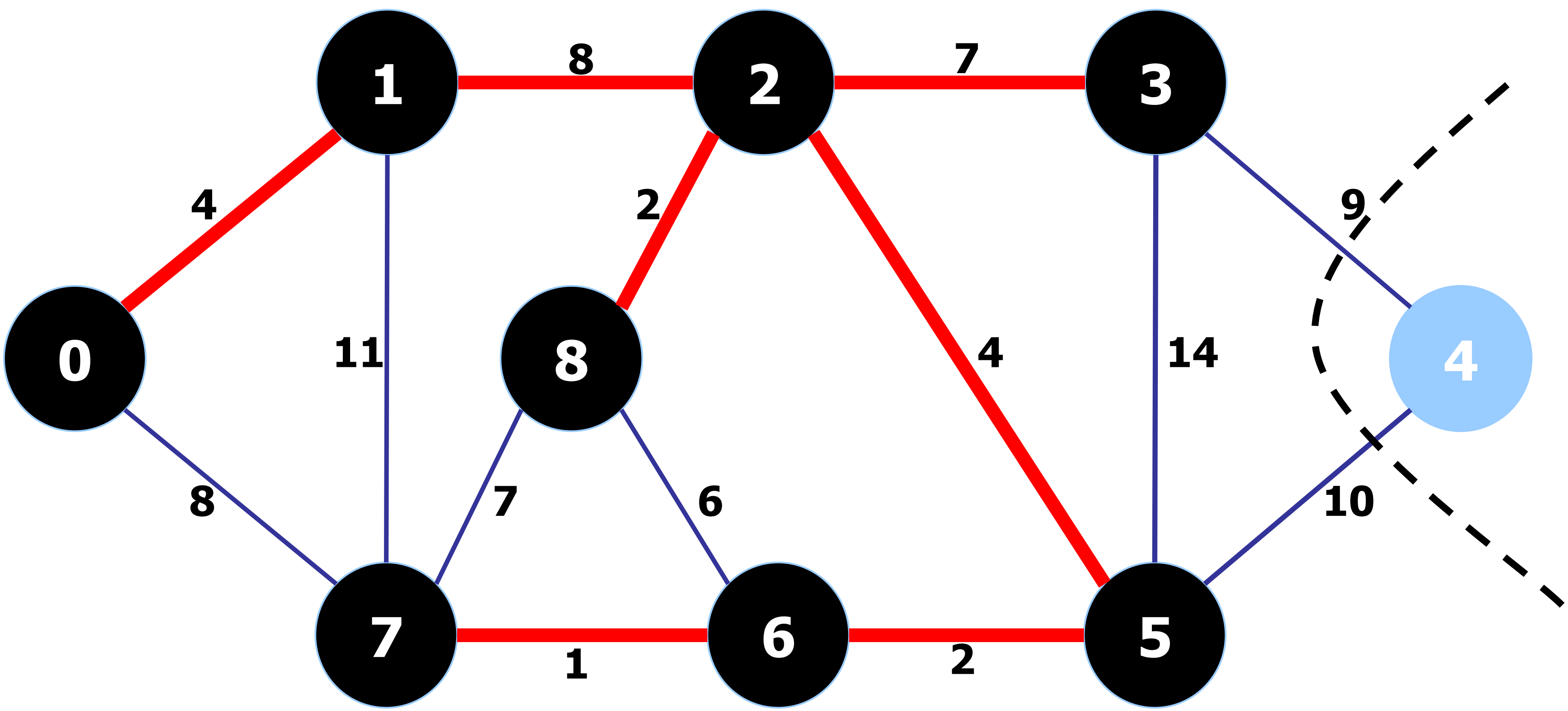


A = {0,1,2,5,6,7,8}

V-A = {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

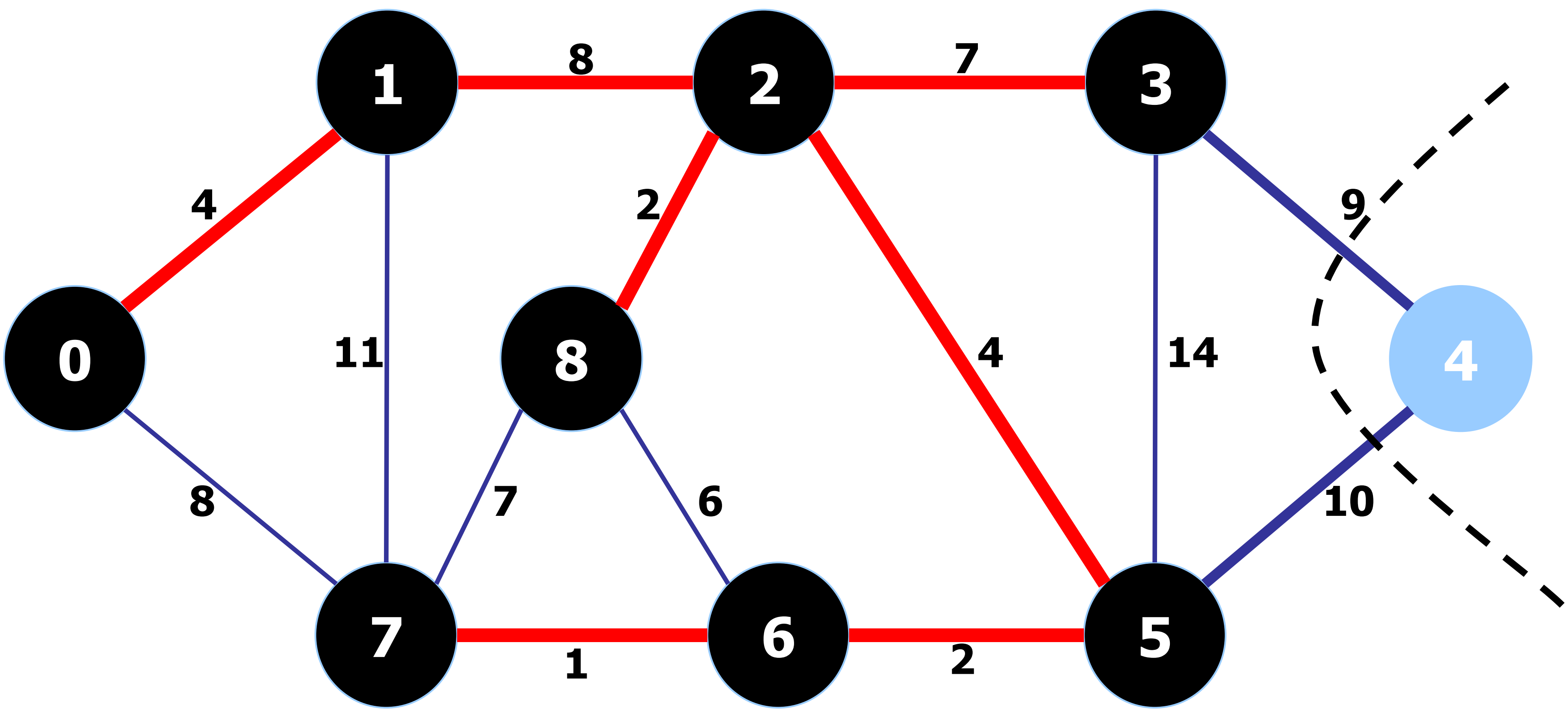


A = {0,1,2,3,5,6,7,8}

V-A = {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

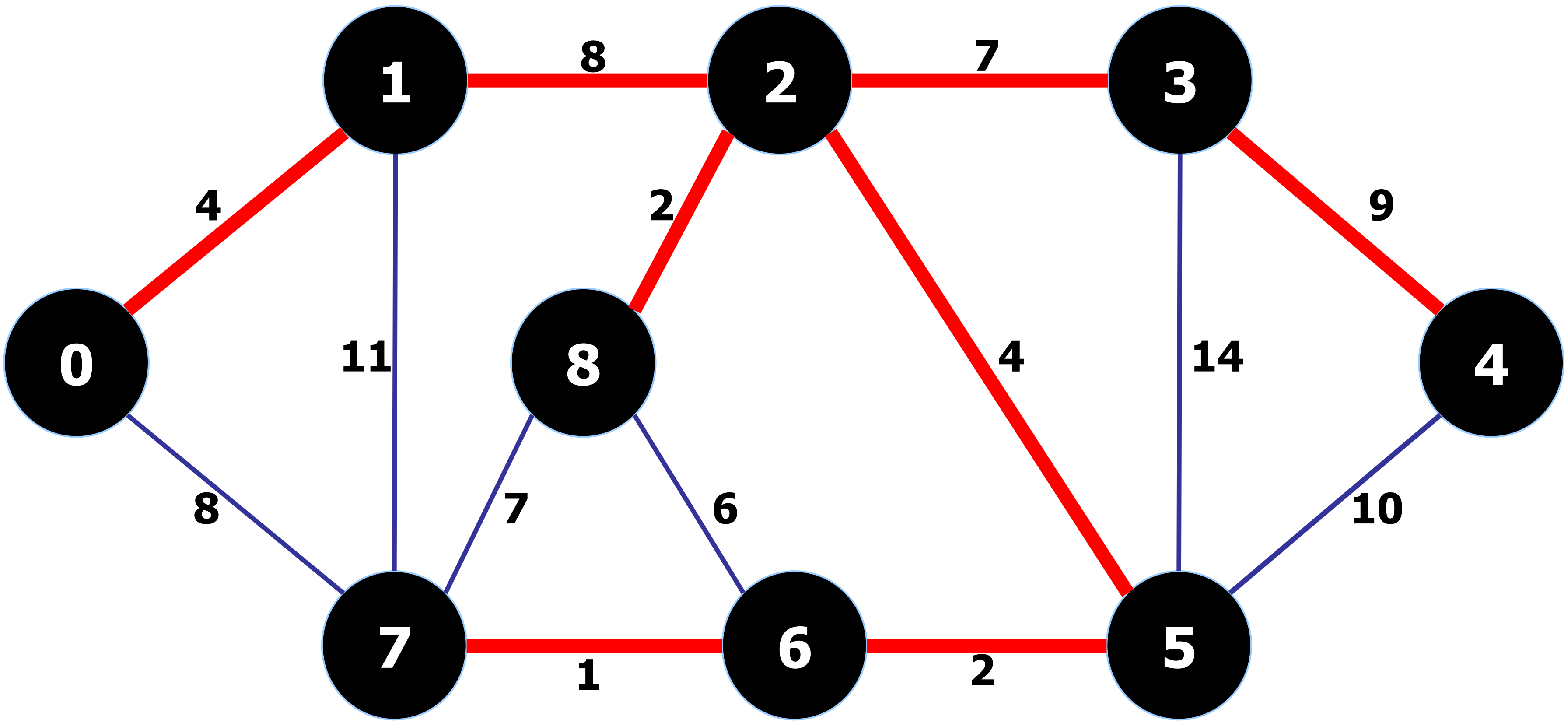


A = {0,1,2,3,5,6,7,8}

V-A = {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



A = {0,1,2,3,4,5,6,7,8}

V-A = ∅

```
void GRAPHmstV(Graph G, int st[], int wt[]) {
  int v, w, min;
  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;
      }
  }
}
```

```
void GRAPHmstP(Graph G) {
  int v, st[maxV], wt[maxV];
  GRAPHmstV(G, st, wt);
  printf("st \n");
  for ( v=0; v < G->V; v++)
    printf("%3d", v);
  printf("\n");
  for ( v=0; v < G->V; v++)
    printf("%3d", st[v]);
  printf("\n");
  printf("wt \n");
  for ( v=0; v < G->V; v++)
    printf("%3d", wt[v]);
  printf("\n");
}
```

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