POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Gli alberi ricoprenti minimi

Gianpiero Cabodi e Paolo Camurati

Alberi ricoprenti minimi

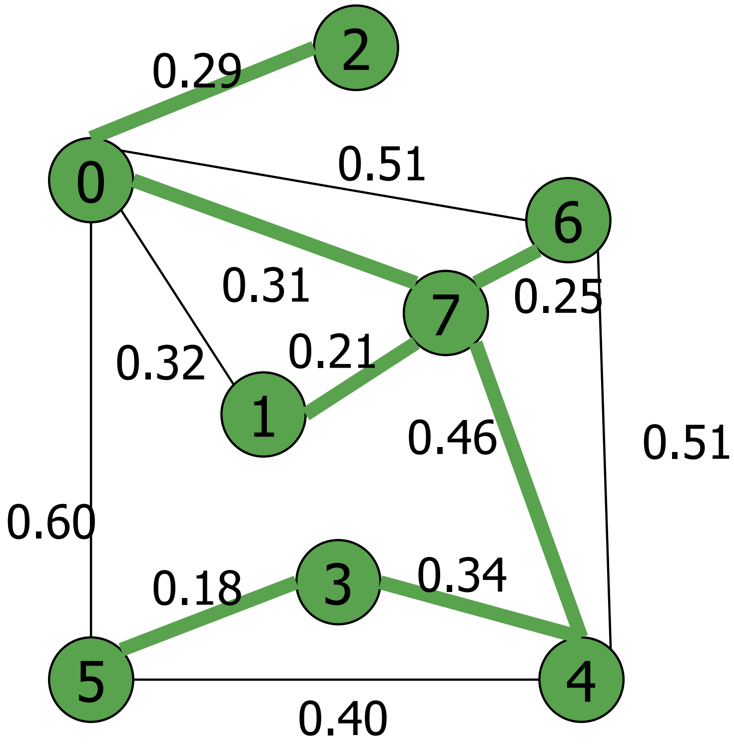
Dato $G=(V,E)$ grafo non orientato, pesato con pesi reali $w: E \rightarrow \boldsymbol{R}$ e connesso, estrarre da G un **albero ricoprente minimo** (Minimum-weight Spanning Tree – MST) , definito come:

- grafo $G'=(V, A)$ dove $A \subseteq E$
- aciclico
- minimizza $w(A)=\Sigma_{(u,v) \in T}\ w(u,v)$.

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:

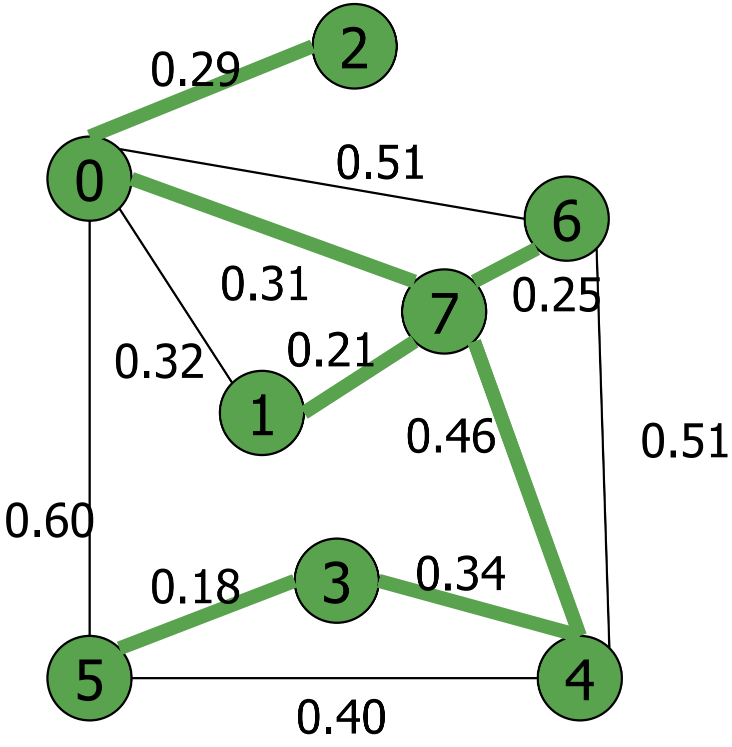
- lista delle adiacenze
- matrice delle adiacenze

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: soluzione 1



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

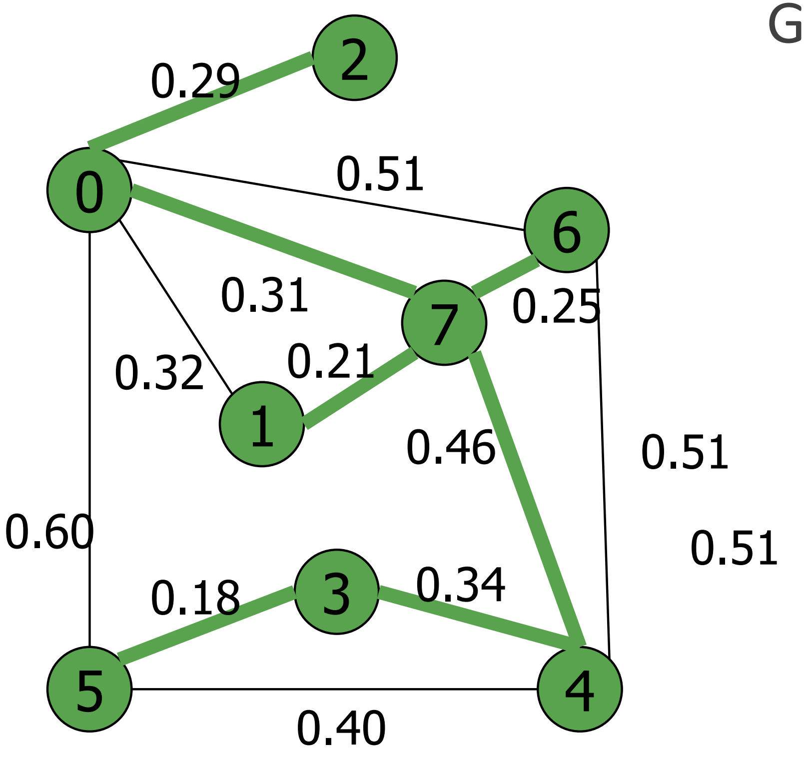
7-0 0.31

7-6 0.25

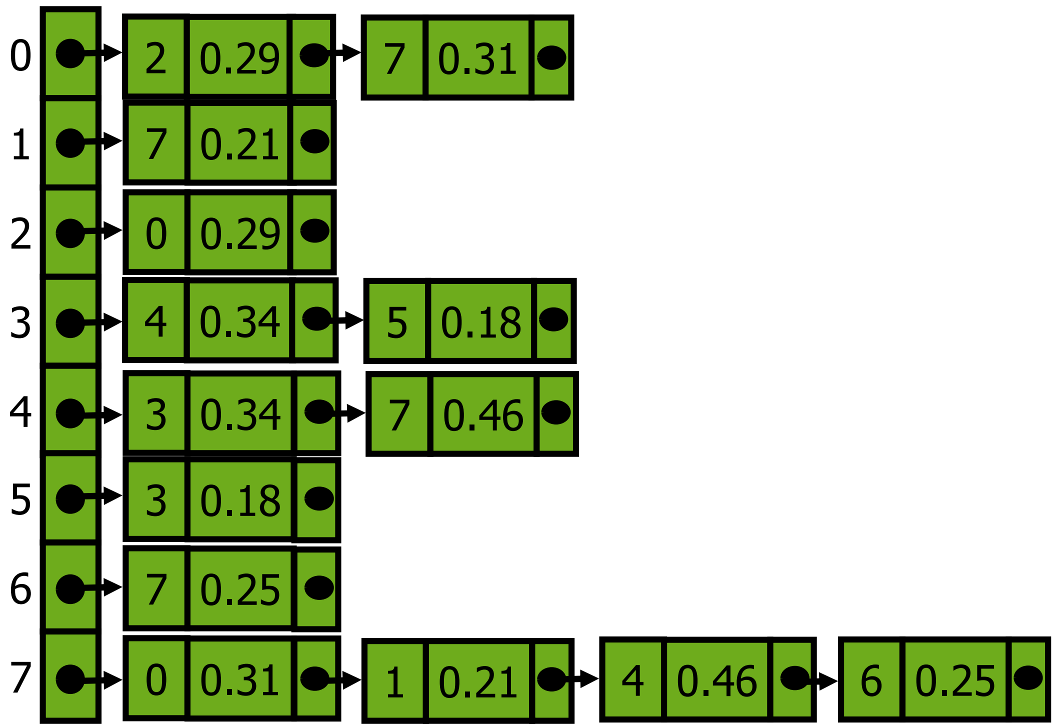
7-1 0.21

Soluzione usata nell'Algoritmo di Kruskal.

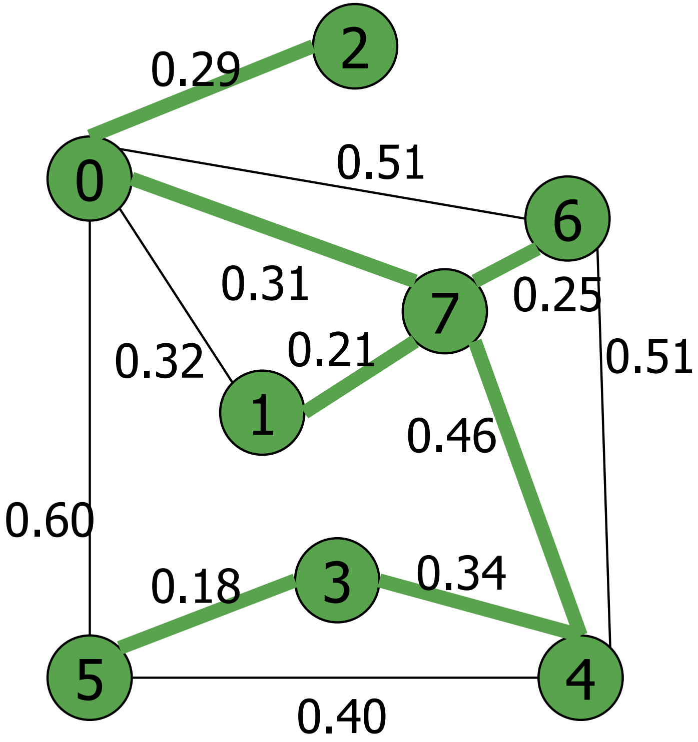
Rappresentazione degli MST: soluzione 2



Grafo come lista di adiacenze:



Rappresentazione degli MST: soluzione 3



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

Soluzione usata nell'Algoritmo di Prim.

Approccio completo

- Dati V vertici, l'alberi ricoprente avrà esattamente V-1 archi
- si esplorano tutte le maniere di raggruppare V-1 archi scelti dagli E archi del grafo
  - l'ordine non conta (combinazioni)
  - condizione di accettazione: verifica di copertura e di aciclicità
  - problema di ottimizzazione: si considerano tutte le soluzioni e si sceglie la migliore
- costo: esponenziale.

Approccio greedy

In generale:

- ad ogni passo, si sceglie la soluzione localmente ottima
- non garantisce necessariamente una soluzione globalmente ottima.

Per gli MST:

- si parte da un algoritmo incrementale, generico e greedy dove la scelta è ottima localmente
- si può dimostrare che la soluzione è globalmente ottima.

Algoritmo incrementale, generico e greedy

- La soluzione A corrente è un sottoinsieme degli archi di un albero ricoprente minimo.
- inizialmente A è l’insieme vuoto
- ad ogni passo si aggiunge ad A un arco “sicuro”
- fino a quando A non diventa un albero ricoprente minimo.

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.

Un teorema e un corollario permettono di identificare gli archi sicuri..

Tagli e archi

Dato $G=(V,E)$ grafo non orientato, pesato, connesso, si definisce **taglio** una partizione di V in S e V-S

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

tale che se $(u,v) \in E$ attraversa il taglio allora

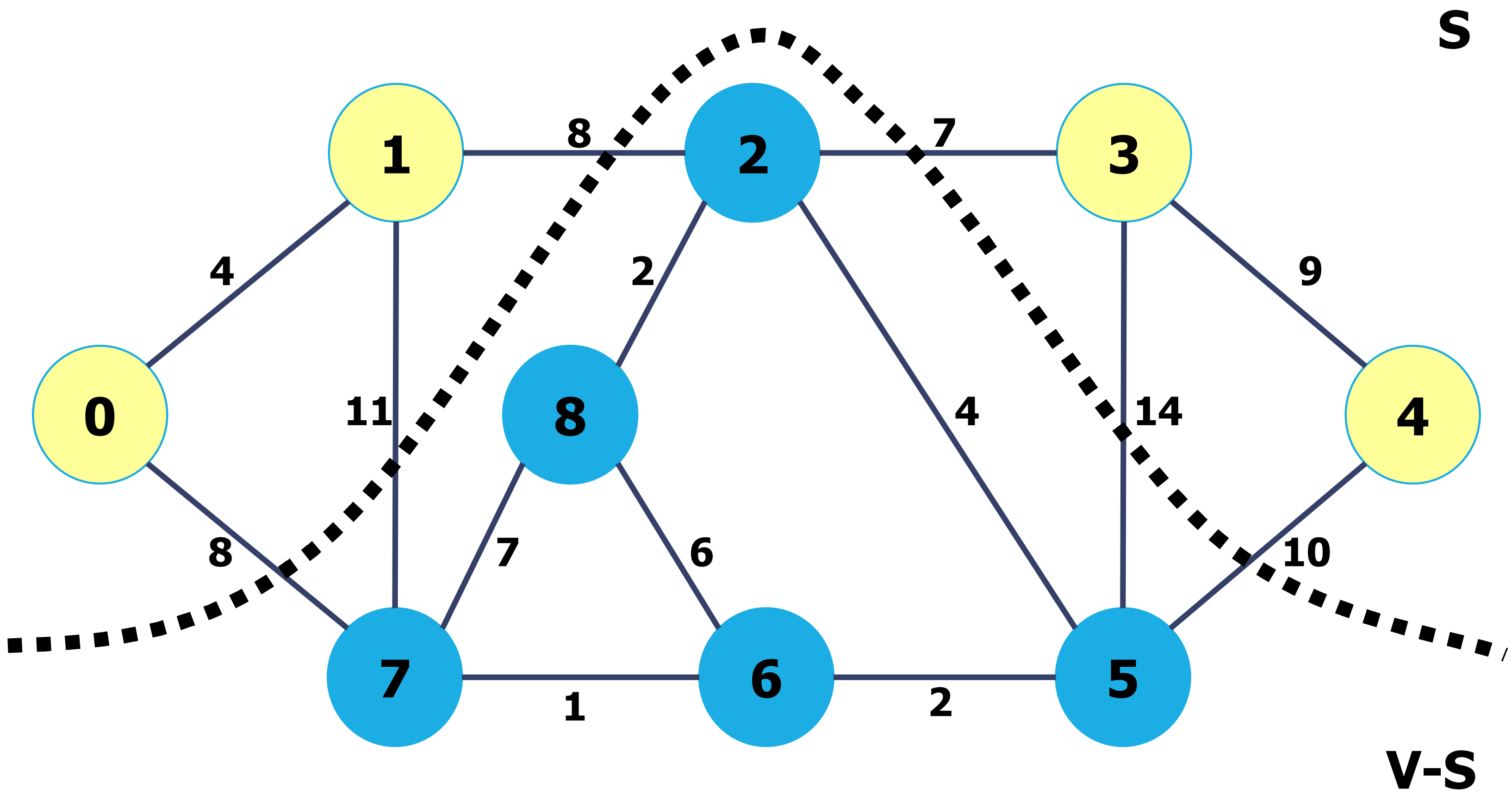
$$u \in S \;\&\&\; v \in V\text{-}S \text{ (o viceversa)}$$

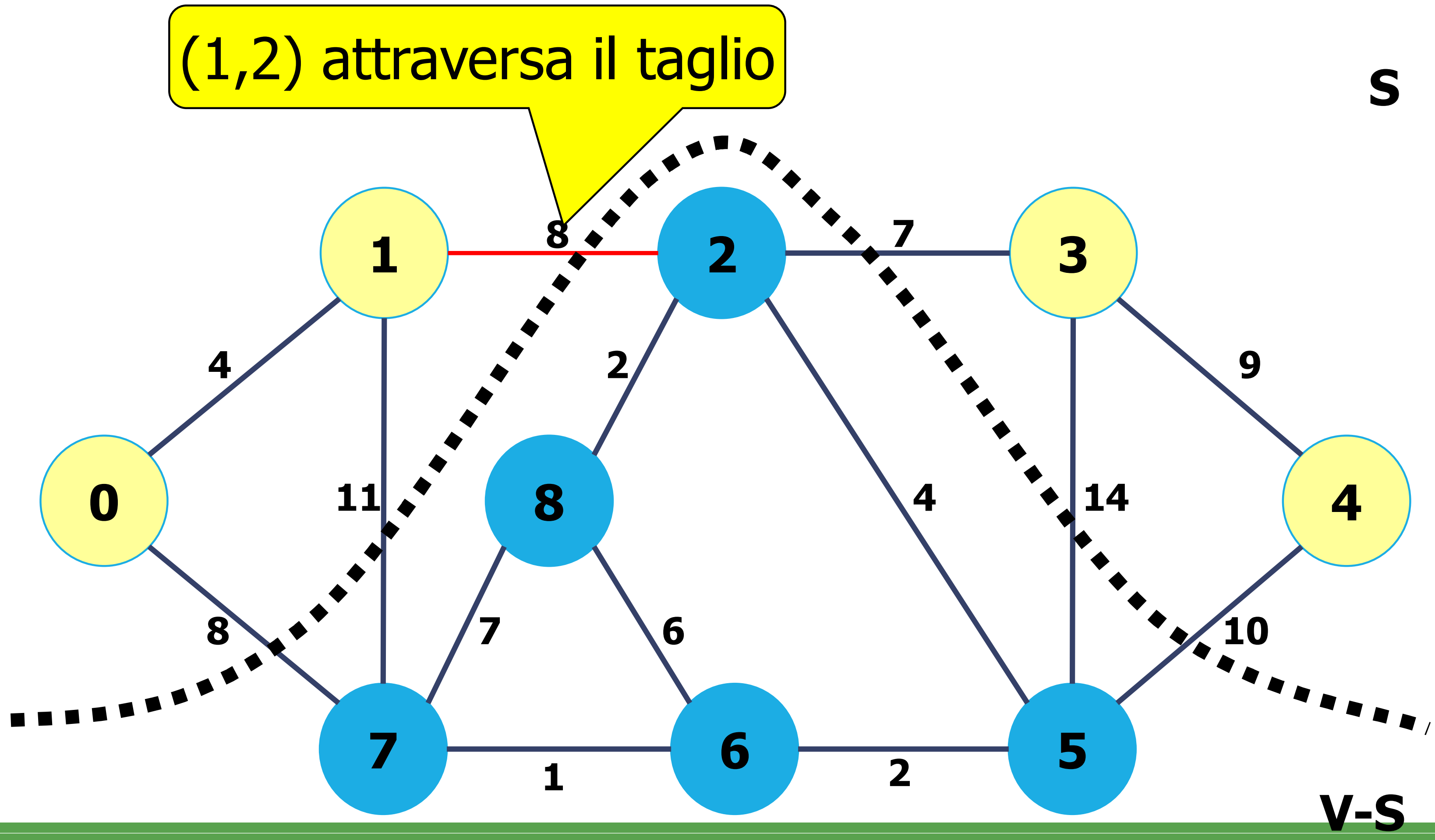
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.

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

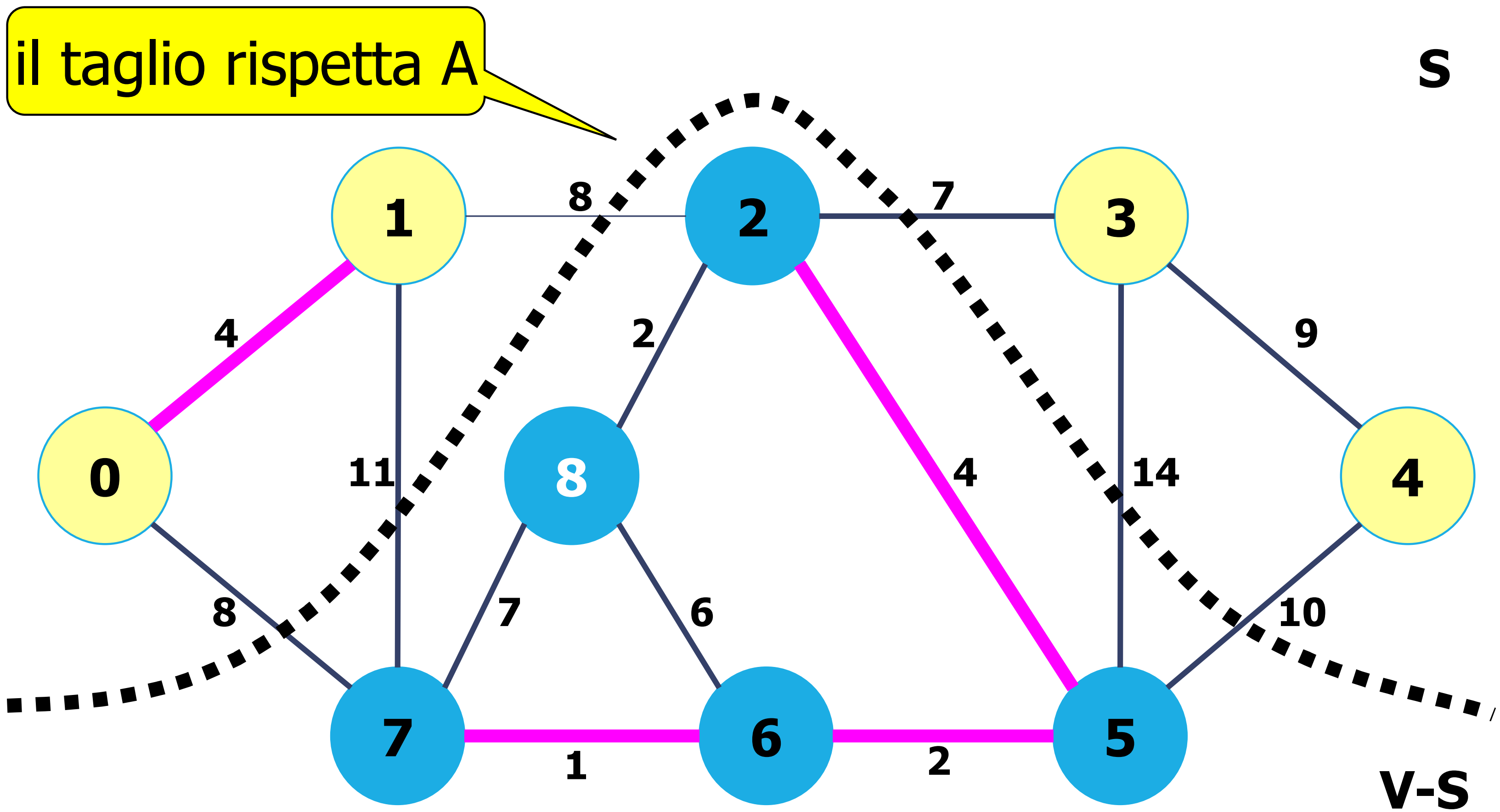
greedy: scelta localmente ottima

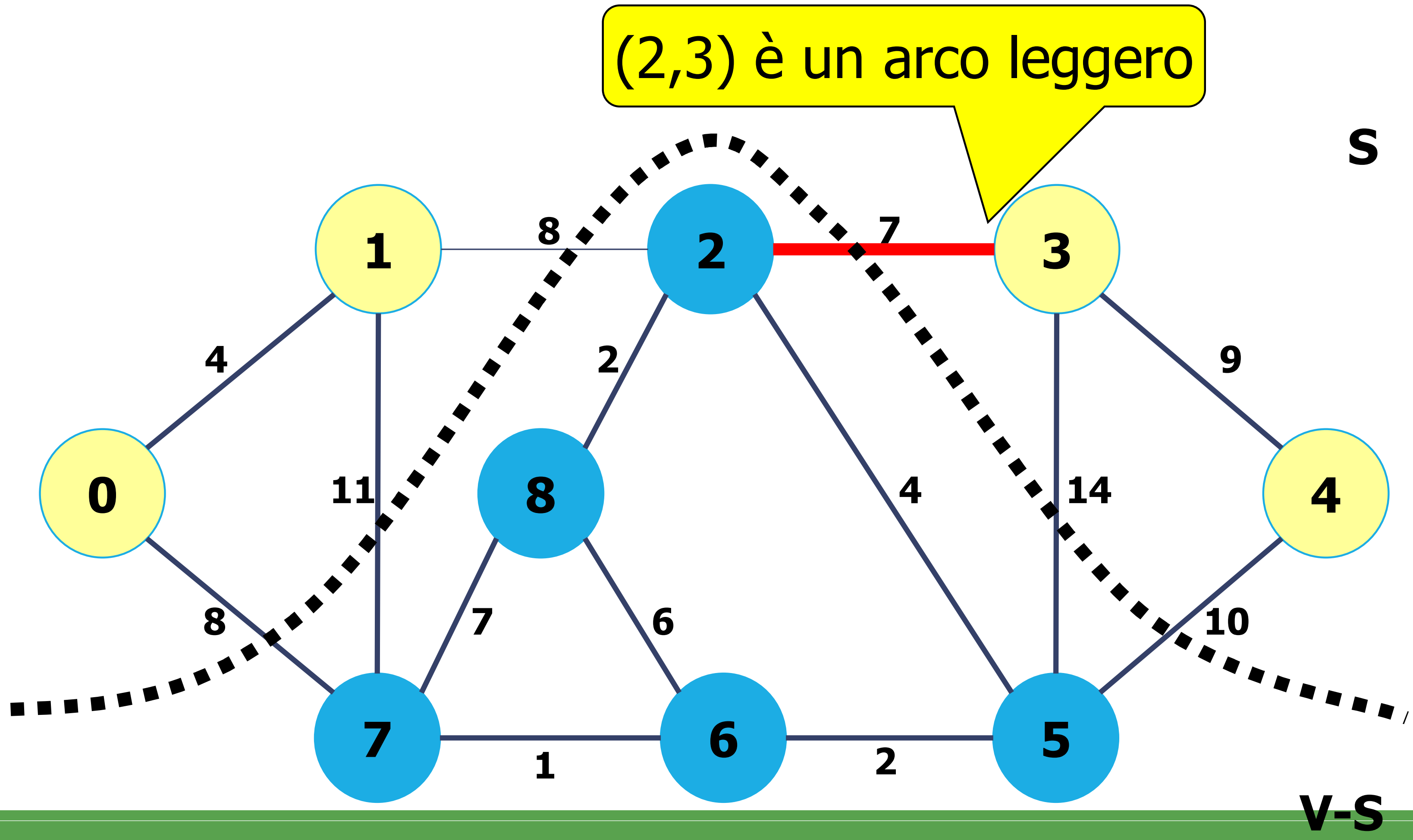
Esempio




(1,2) attraversa il taglio
S
1
2
3
0
8
4
7
6
5
8
7
4
11
2
4
14
9
8
7
6
10
1
2
V-S

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




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

Archi sicuri: teorema

Dati:

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

se:

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

allora

(u,v) è sicuro per A.

Archi sicuri: corollario

Dati:

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

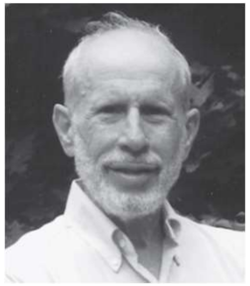
se:

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

allora

(u,v) è sicuro per A.

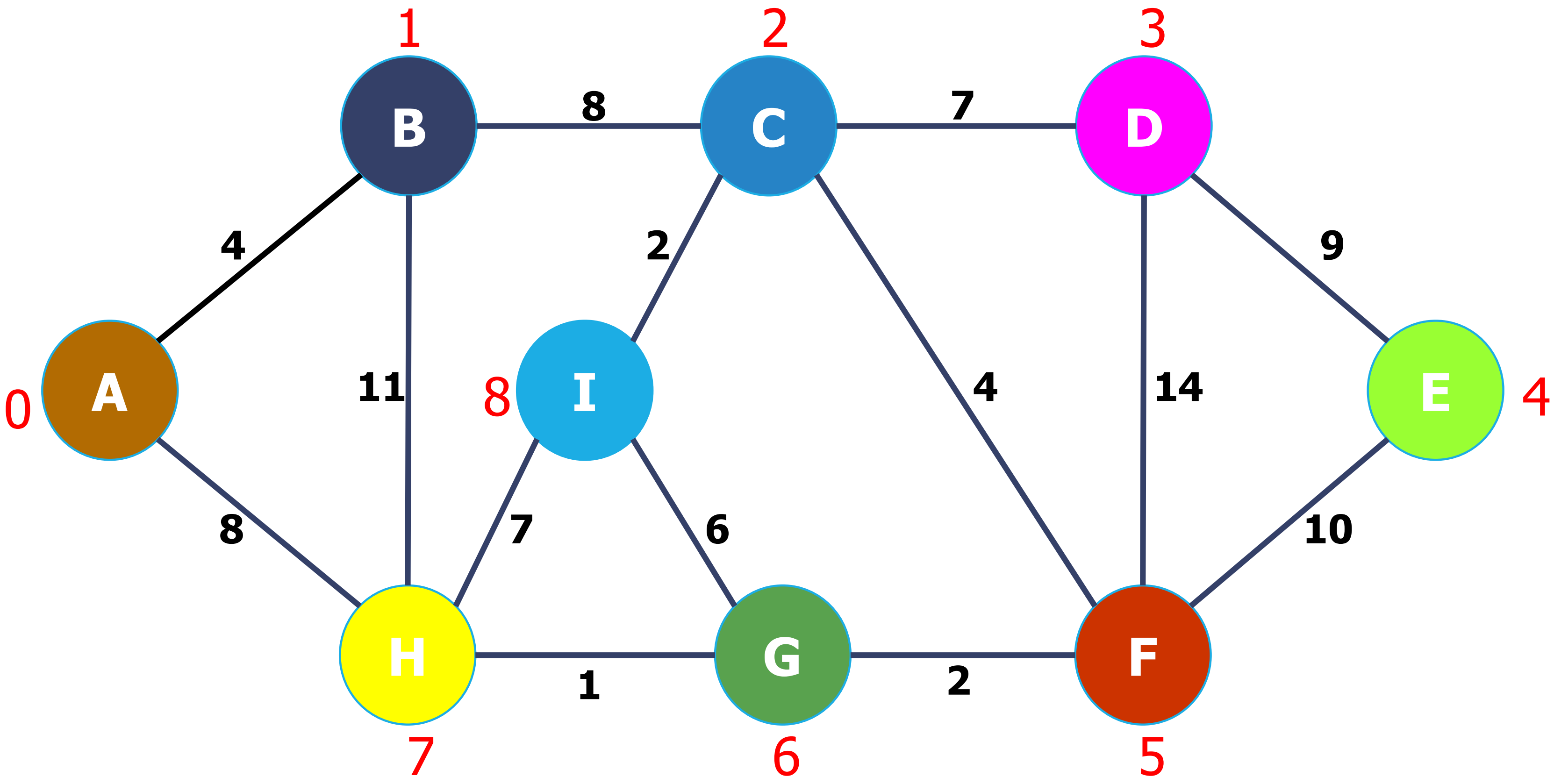
Algoritmo di Kruskal (1956)

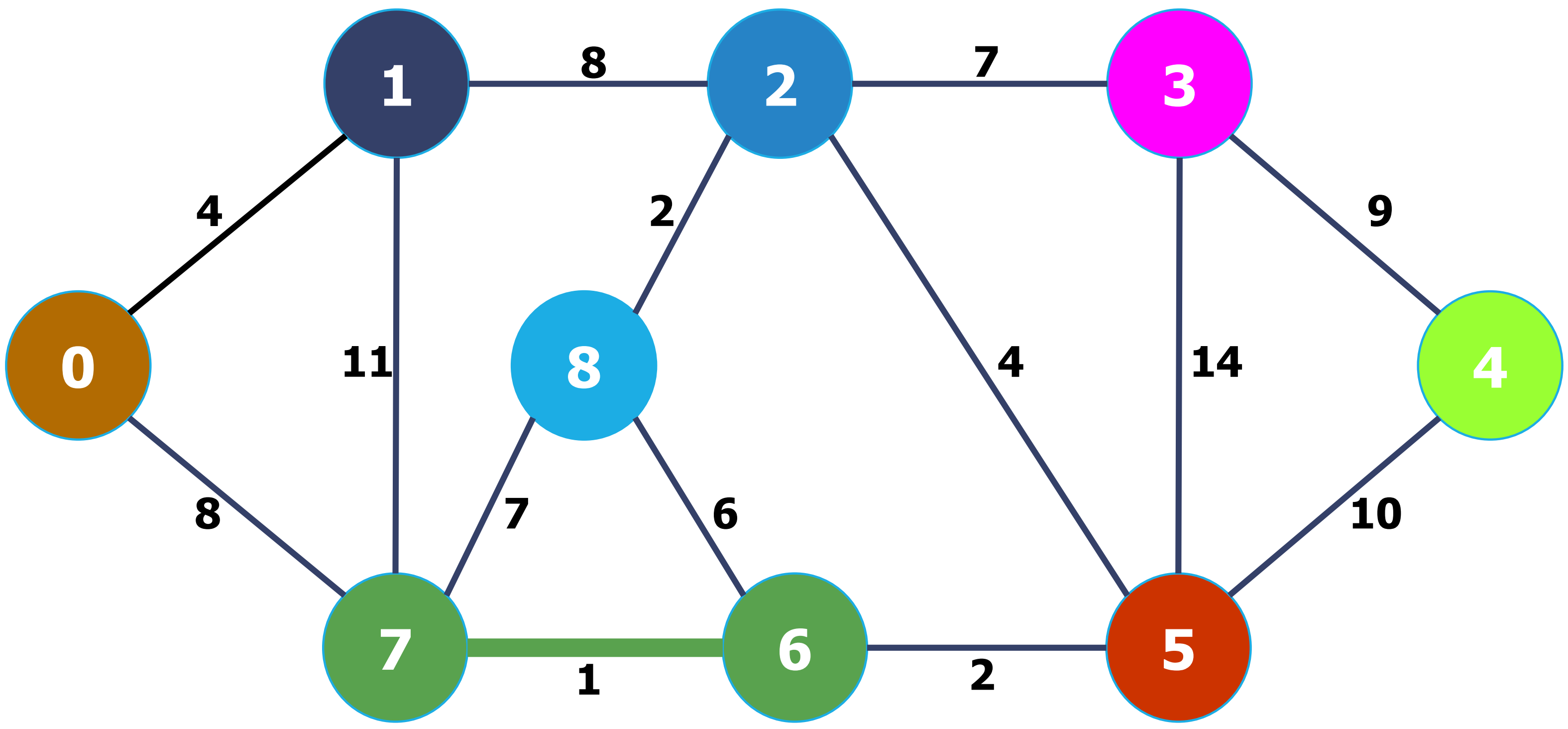


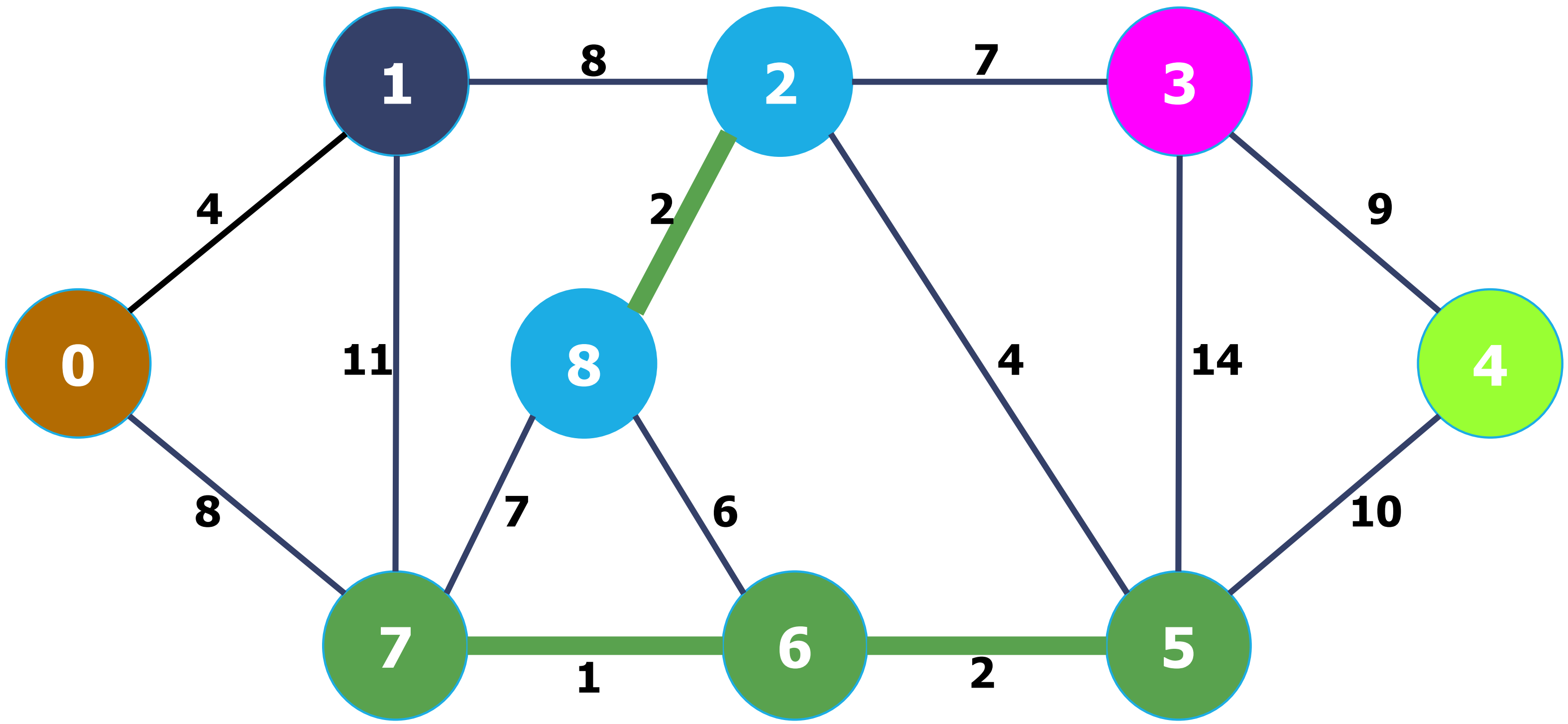
- basato su algoritmo generico
- uso del corollario per determinare l'arco sicuro:
  - si considera una foresta di alberi, inizialmente formati dai singoli vertici
  - si ordinano degli archi per pesi crescenti
  - si itera la selezione di un arco sicuro: arco di peso minimo che connette 2 alberi generando ancora un albero
  - terminazione: considerati tutti i vertici.

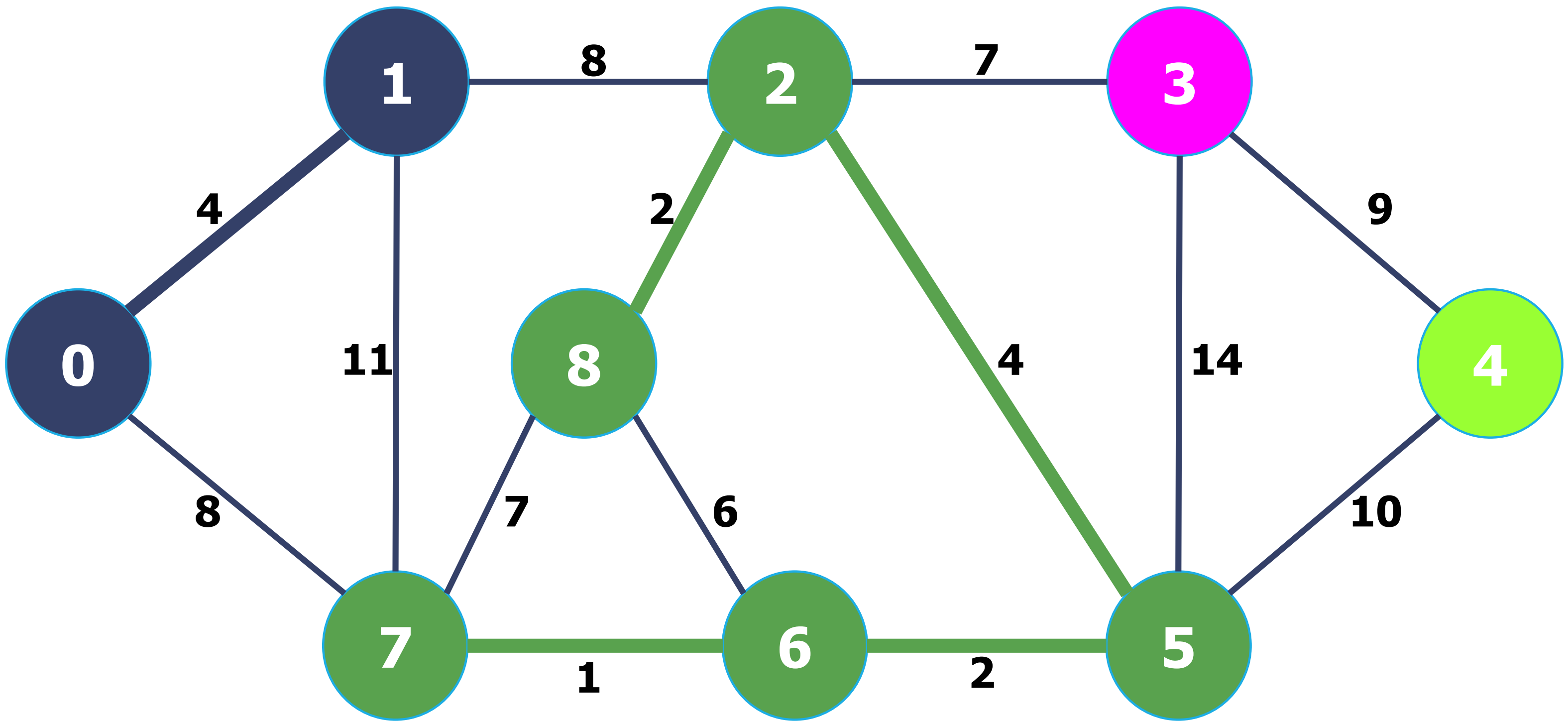
La rappresentazione degli alberi è fatta con ADT Union-Find (vedi Tecniche di Programmazione, lezione sull'Analisi della Complessità, problema dell'On-line connectivity).

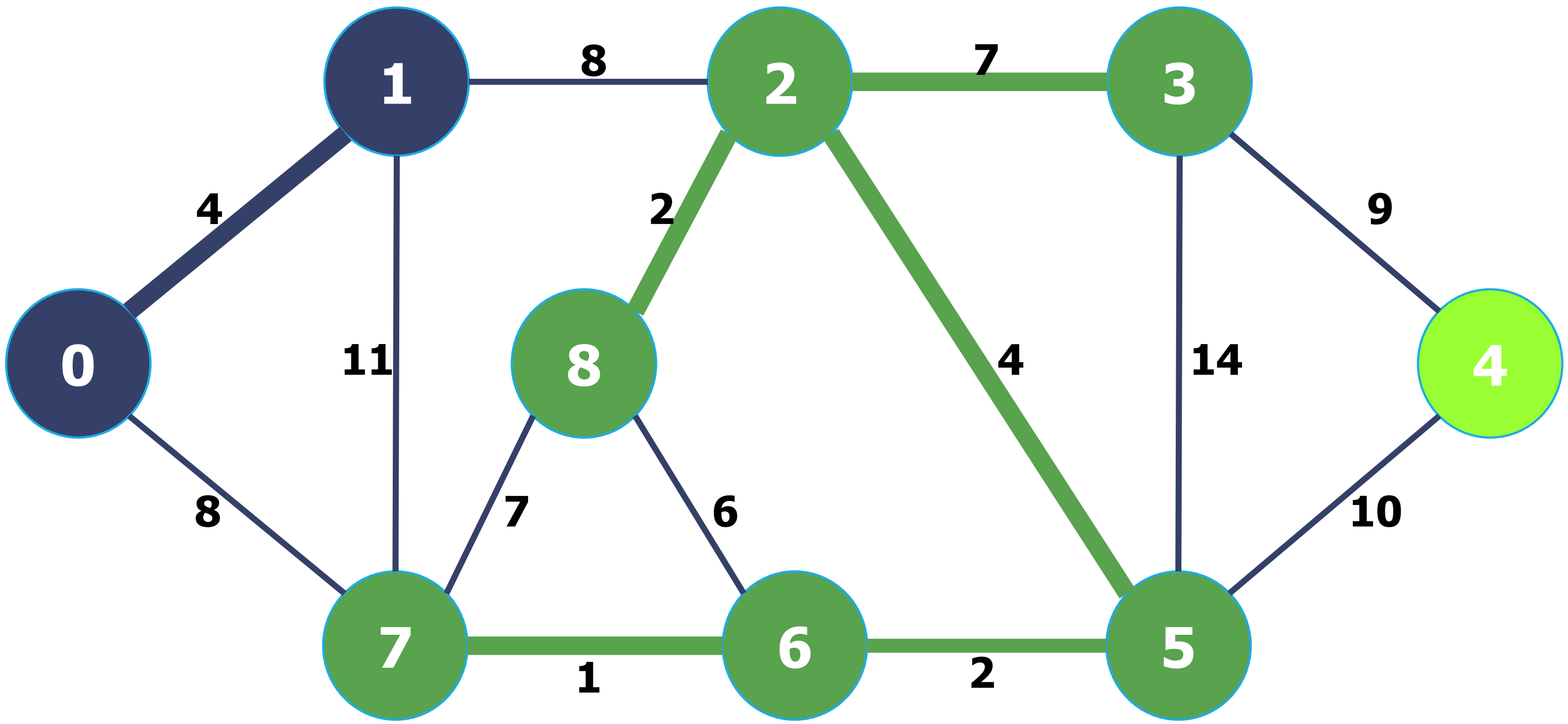
Esempio

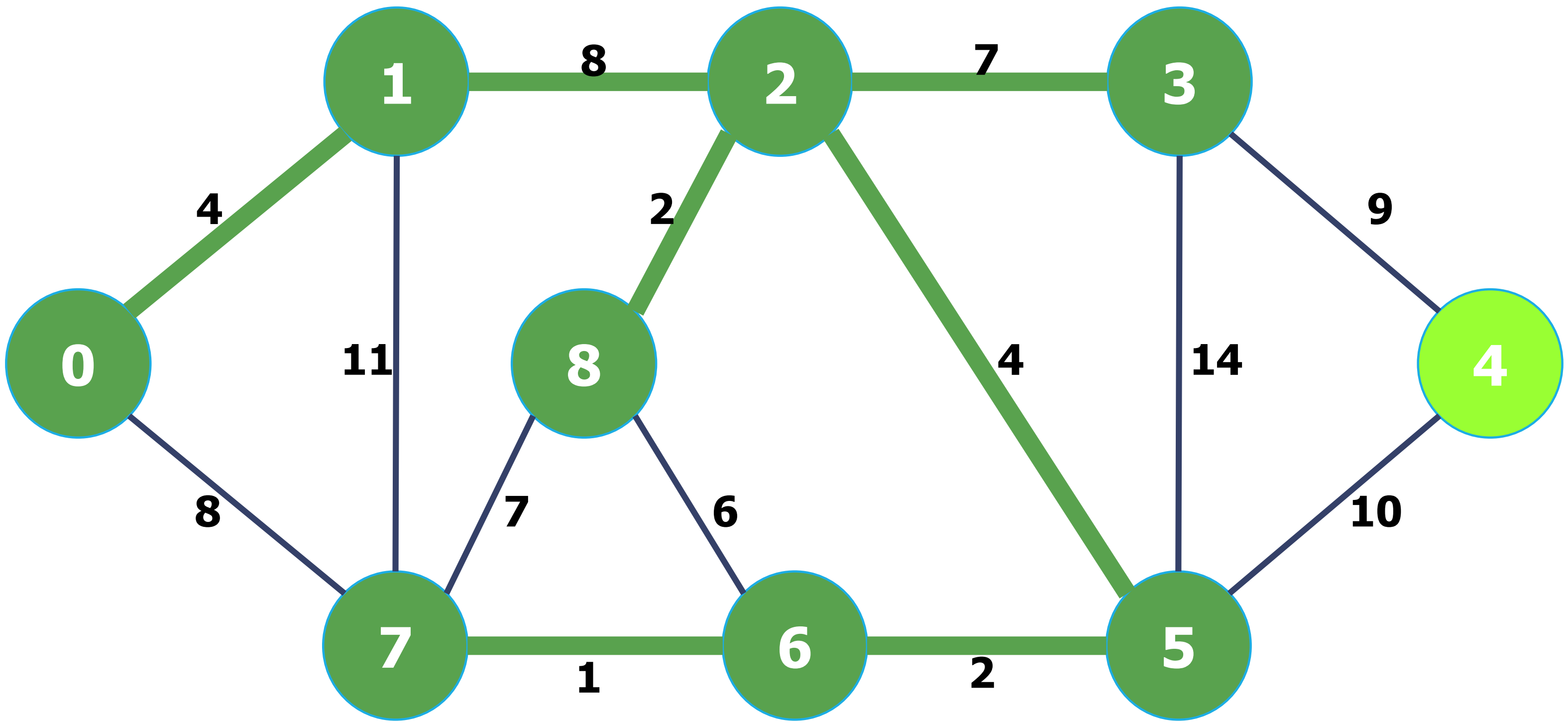


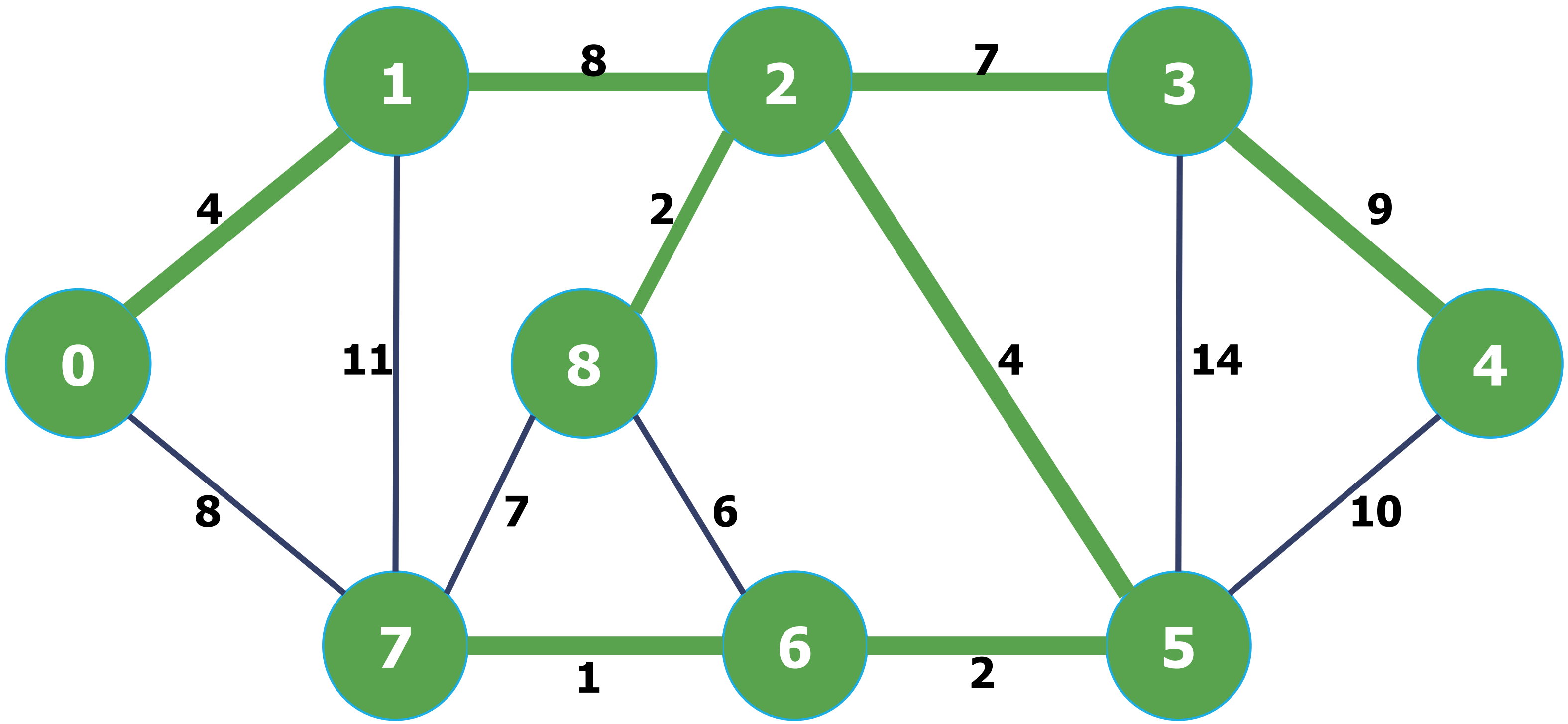

1
8
2
7
3
4
2
9
0
11
8
4
14
4
8
7
6
10
7
1
6
2
5


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


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


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

ADT Union-find (1964)

- Struttura dati per memorizzare una collezione di insiemi disgiunti, ad esempio la partizione di un insieme in più sottoinsiemi (disgiunti per definizione di partizione)
- anche nota come struttura dati **disjoint-set data** o **merge–find set**
- Operazioni:
  - UFunion: fusione di 2 sottoinsiemi
  - UFfind: verifica se 2 elementi appartengono o meno allo stesso sottoinsieme

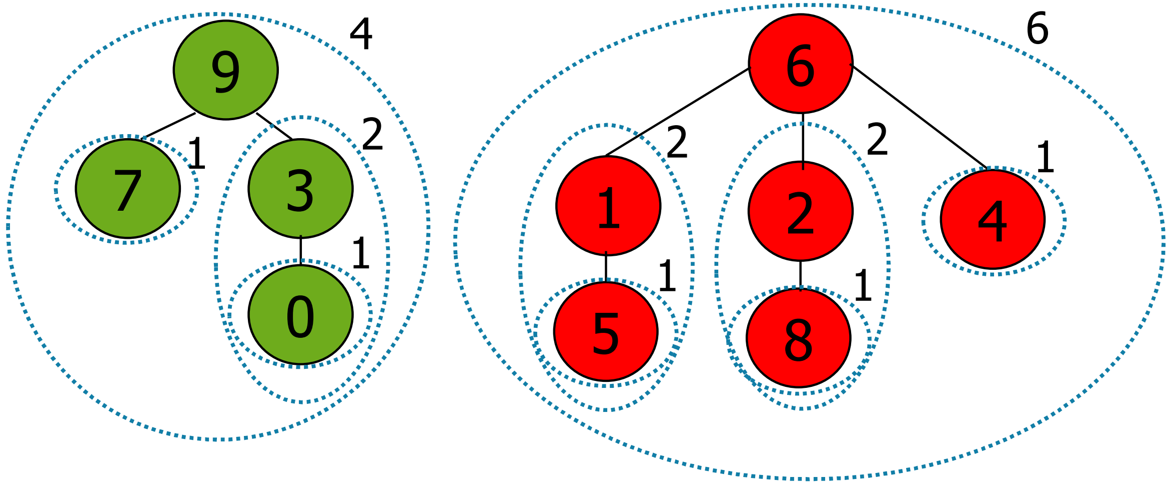
Strutture dati

Dato un insieme di N elementi denominati da 0 a N-1:

- il vettore id contiene per ogni elemento l'indice dell'elemento che lo rappresenta
- inizialmente ogni elemento rappresenta se stesso
- l'elemento che rappresenta il sottoinsieme è quello che rappresenta se stesso (indice coincide con contenuto a quell'indice)
- il vettore sz contiene la cardinalità del sottoinsieme cui appartiene ogni elemento

Esempio

L'insieme di 10 elementi denominati da 0 a 9 è partizionato in 2 sottoinsiemi {0, 3, 7, 9} e {1, 2, 4, 5, 6, 8} così rappresentati



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

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

ADT di I classe UF

- Versione con weighted quick-union
- quick-union: un elemento punta a chi lo rappresenta, quindi complessità O(1)
- find: percorrimento di una "catena" dall'elemento fino al rappresentante "ultimo" del sottoinsieme. Grazie al mantenimento dell'informazione sulla cardinalità dell'insieme che permette di fondere l'insieme a cardinalità minore in quello a cardinalità maggiore, generando un cammino di lunghezza logaritmica, la complessità è O(logN).

UF.h

```
void  UFinit(int N);
int   UFfind(int p, int q);
void  UFunion(int p, int q);
```

UF.c

```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];
  }
}
```

wrapper

```
void GRAPHmstK(Graph G) {
  int i, k, weight = 0;
  Edge *mst = malloc((G->V-1) * sizeof(Edge));
  Edge *a = malloc(G->E * sizeof(Edge));

  k = mstE(G, mst, a);

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

```
int mstE(Graph G, Edge *mst, Edge *a) {
  int i, k;

  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;
}
```

Complessità

Con l'ADT UF:

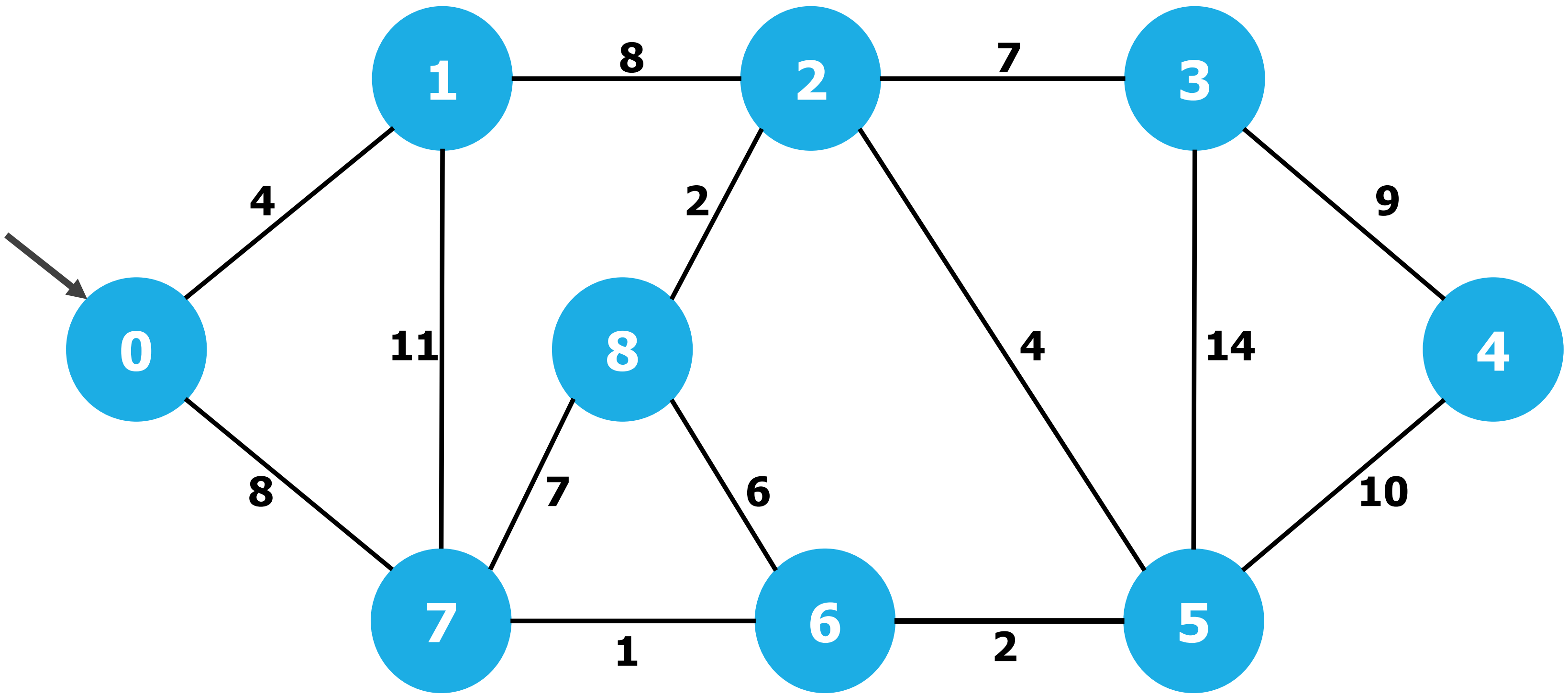
$$T(n) = O(|E| \lg |E|) = O(|E| \lg |V|)$$

in quanto, ricordando che $|E| = O(|V|^2)$ (grafo completo),
$\log|E| = O(\log|V|^2) = O(2\log|V|) = O(\log|V|)$.

Algoritmo di Prim (1930-1959)

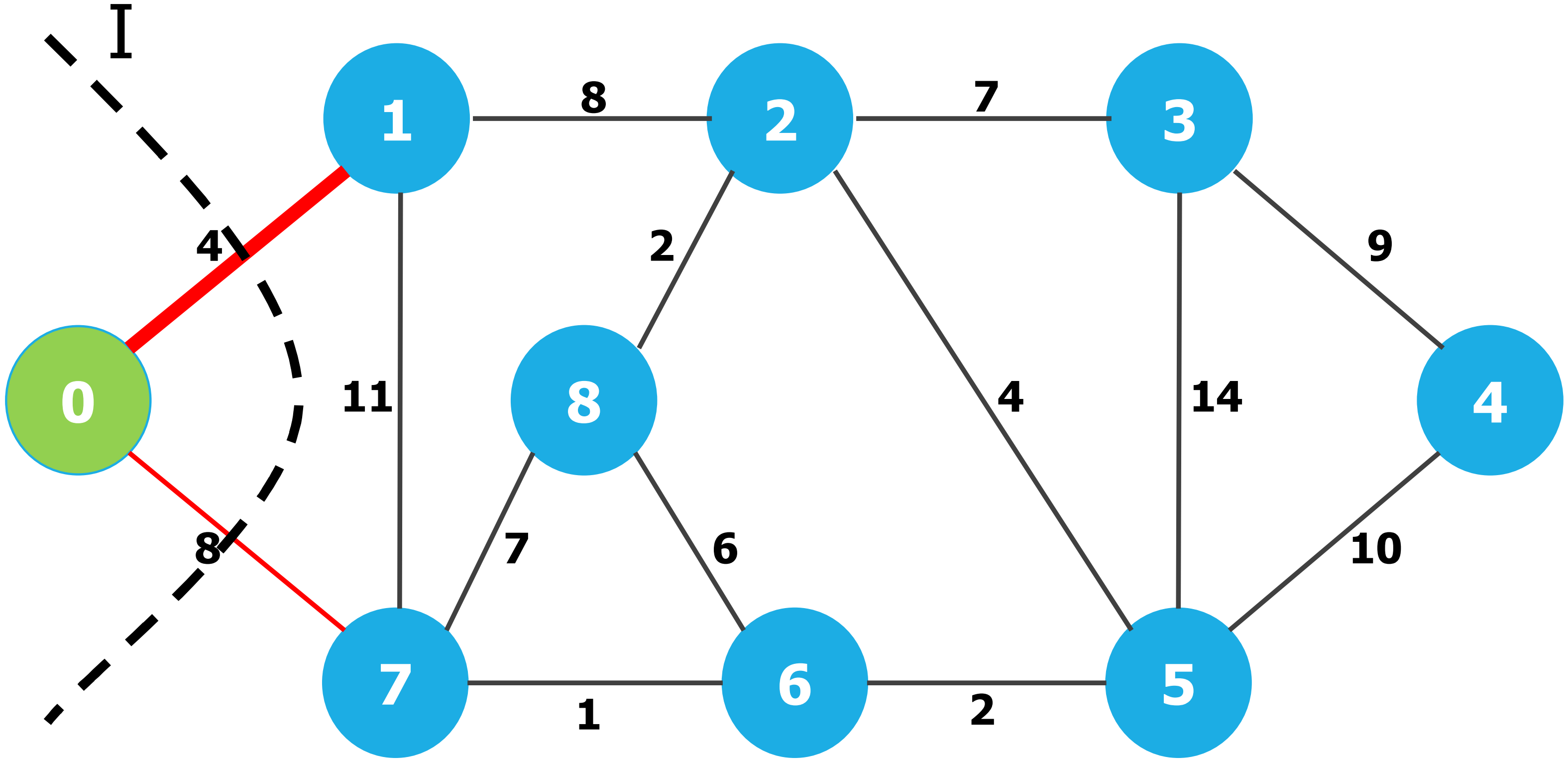
- basato su algoritmo generico
- soluzione brute-force
- uso del teorema per determinare l’arco sicuro:
  - inizialmente S = ∅, poi S = {vertice di partenza}
  - iterazione: V-1 passi in cui si aggiunge 1 arco alla soluzione
    - iterazione sugli archi per selezionarne 1:
      - selezionare quello di peso minimo tra gli archi che attraversano il taglio e aggiungerlo alla soluzione
      - in base al vertice in cui arriva l’arco, aggiornare S
  - terminazione: considerati tutti i vertici, quindi soluzione che contiene V-1 archi
  - versione semplice, ma non efficiente a causa del ciclo annidato sugli archi.

Esempio



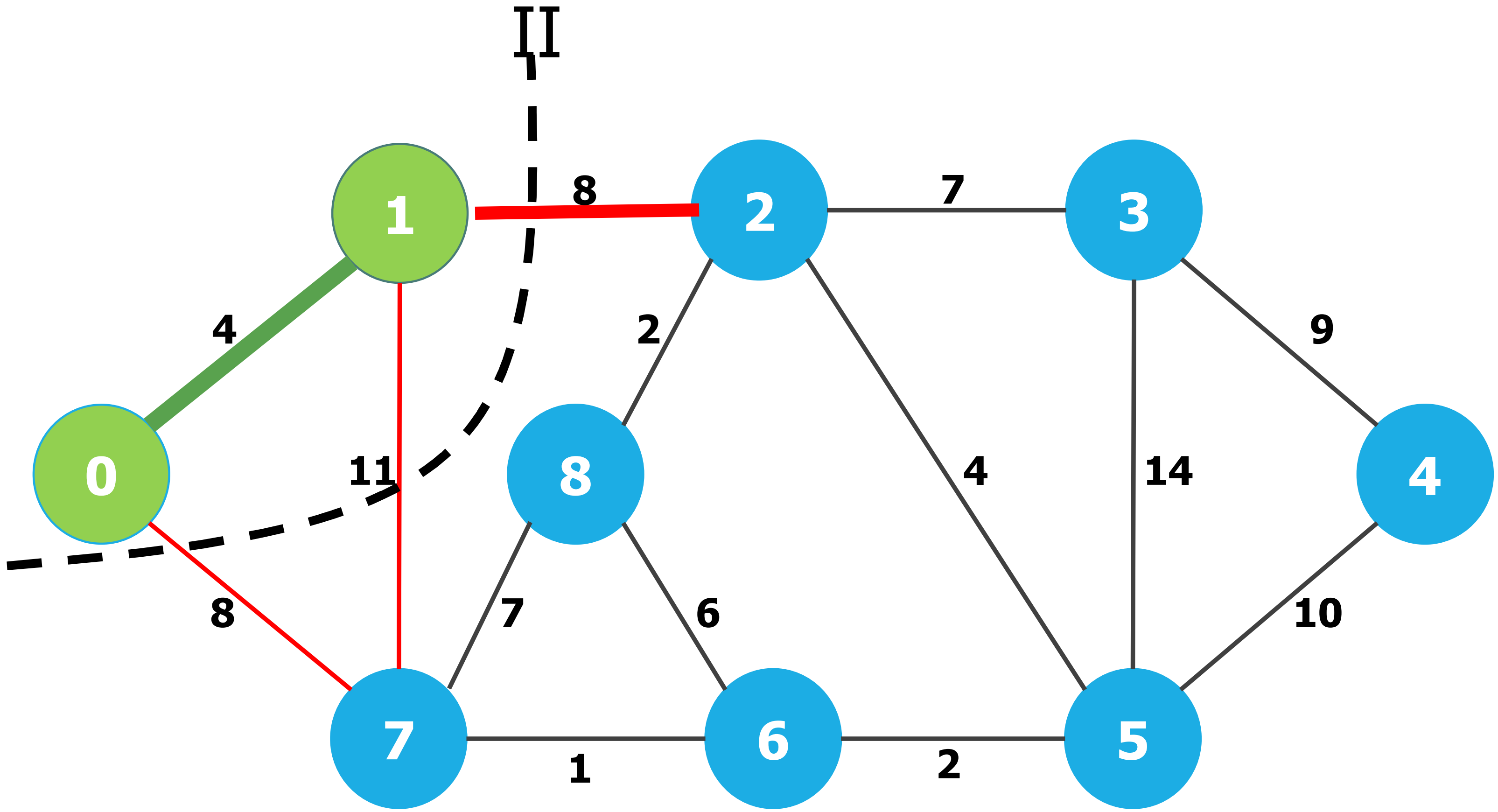
S = ∅

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



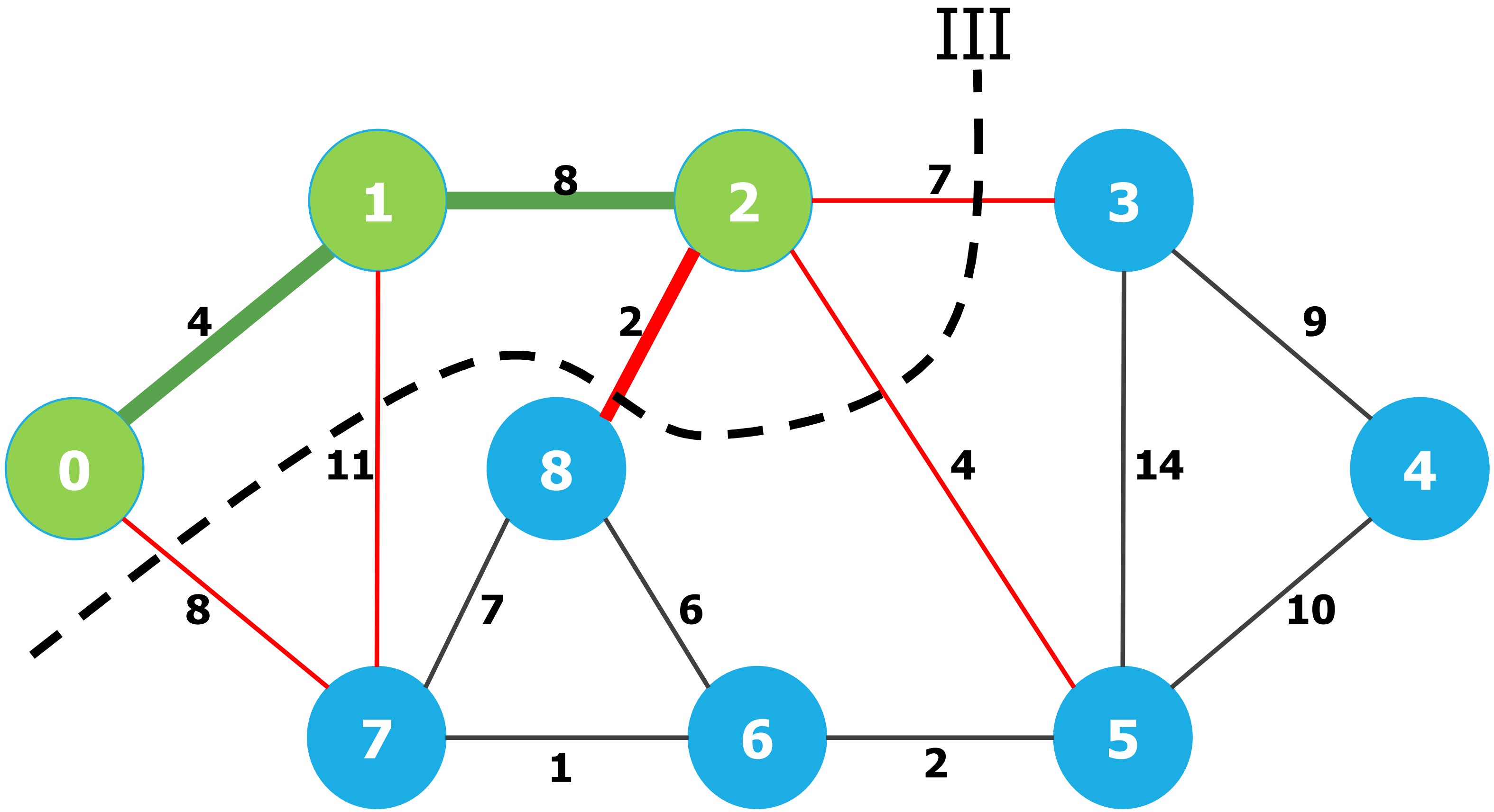
S = {0}

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



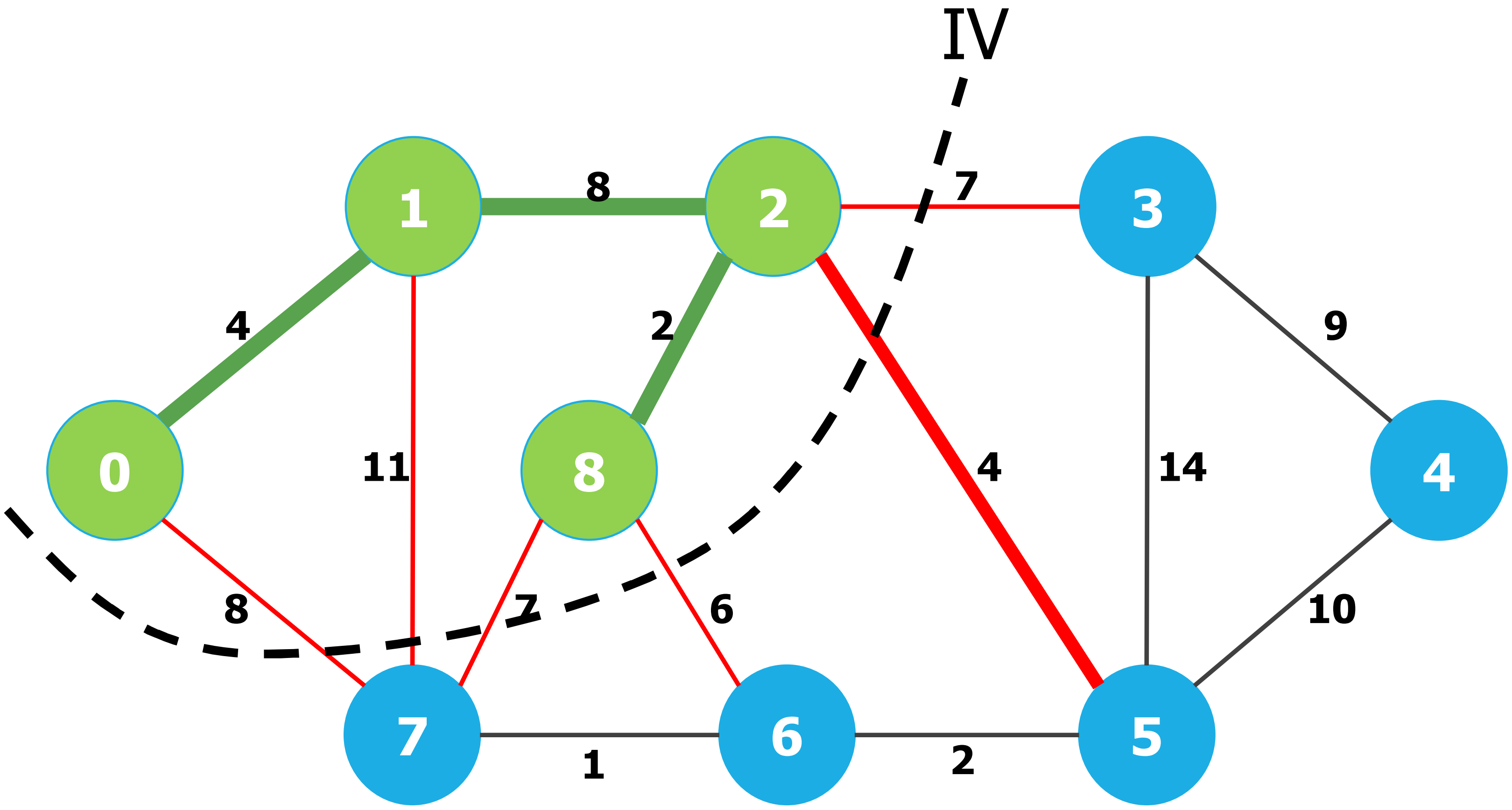
S = {0,1}

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



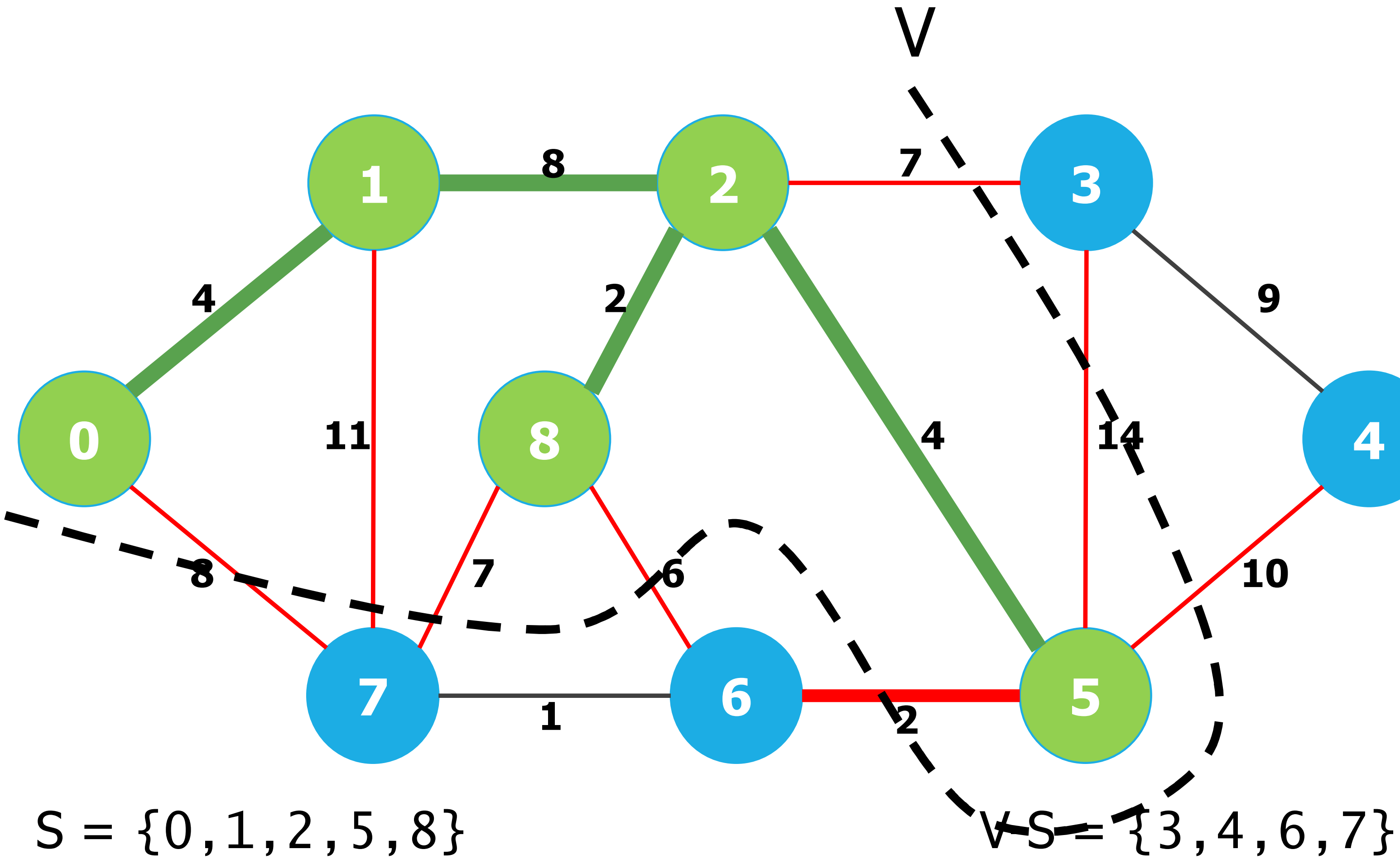
S = {0,1,2}

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



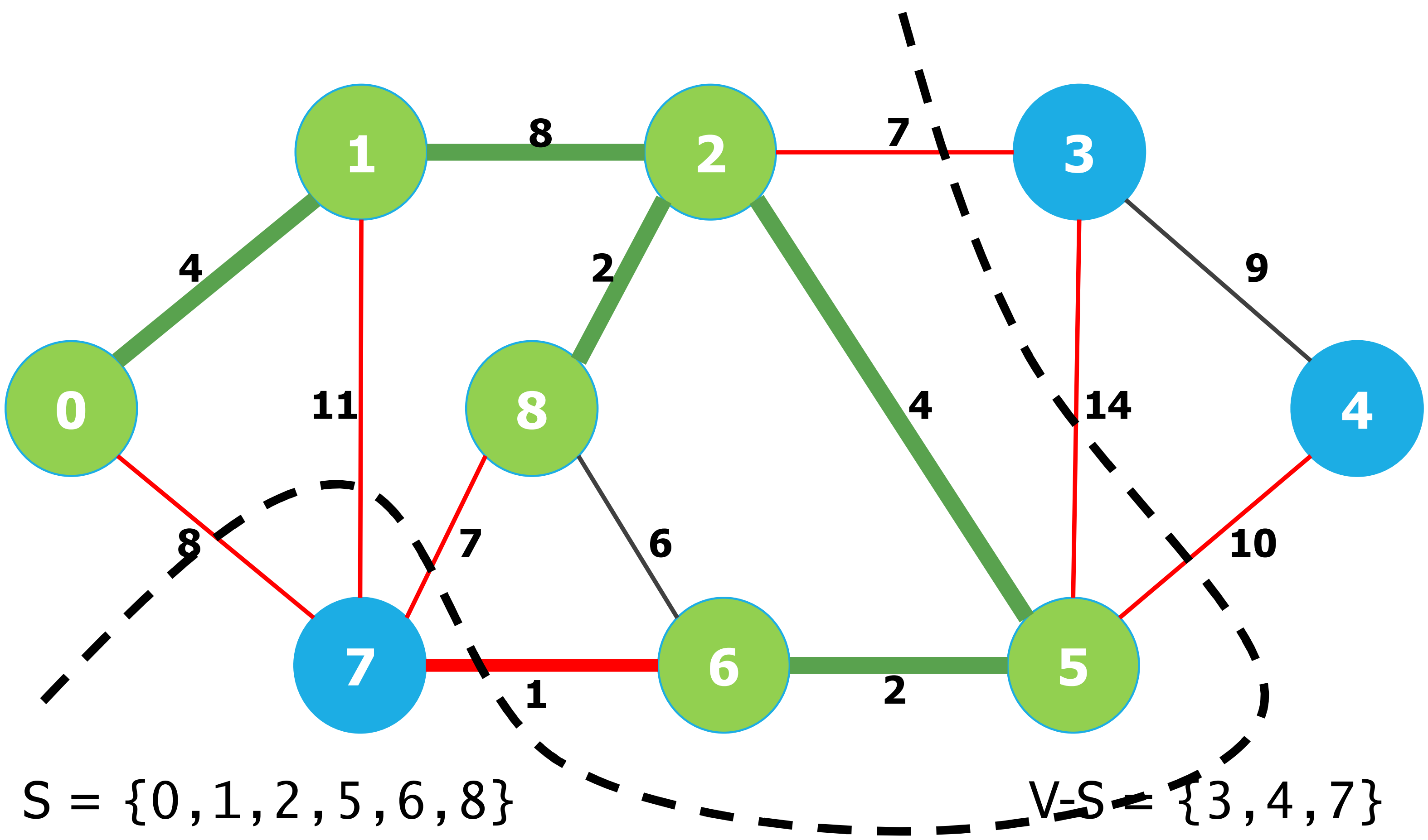
S = {0,1,2,8}

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



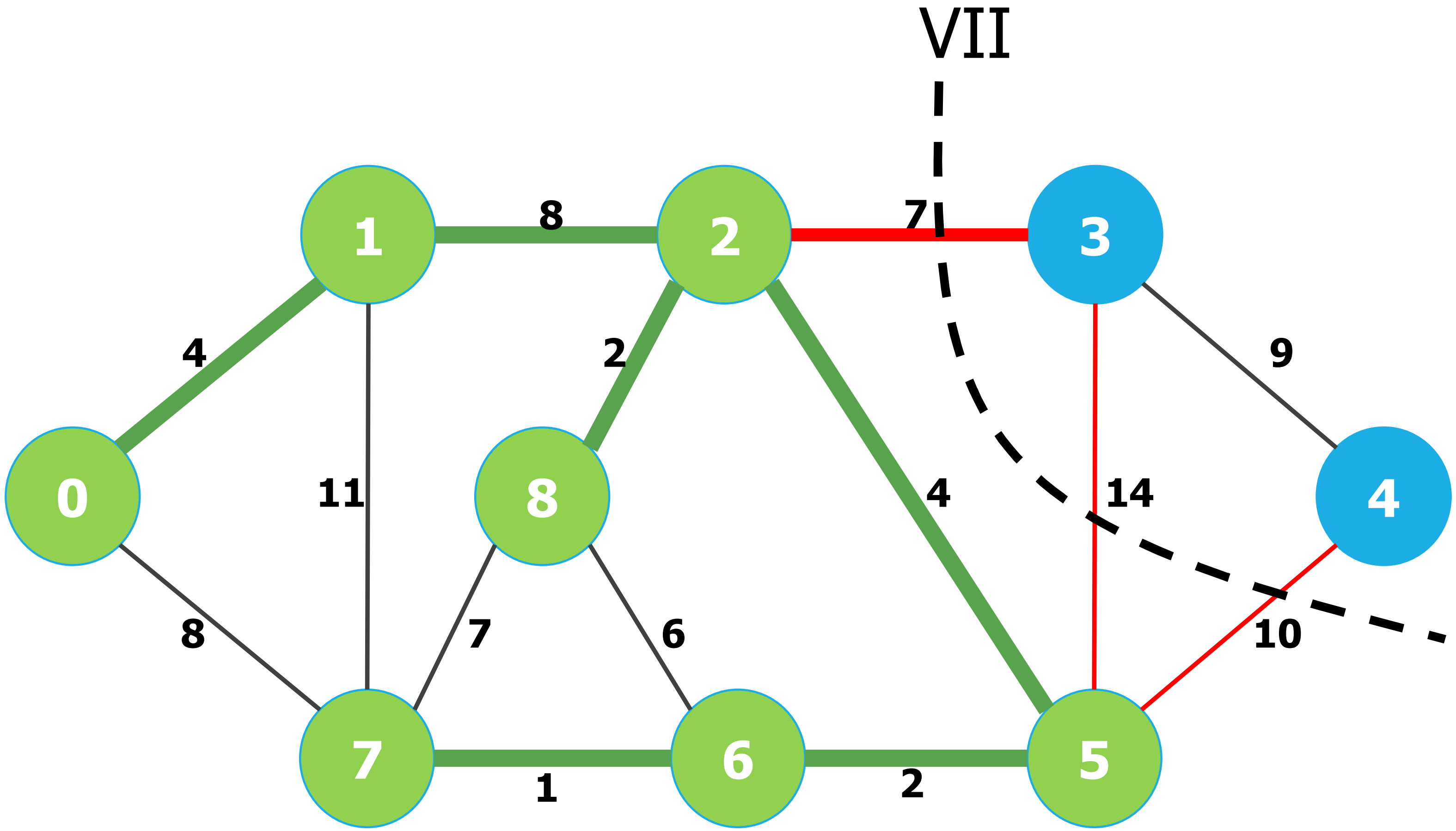
V
1
2
3
0
8
4
7
6
5
8
7
4
2
9
11
4
14
8
7
6
10
1
2
S = {0,1,2,5,8}
V-S = {3,4,6,7}

VI



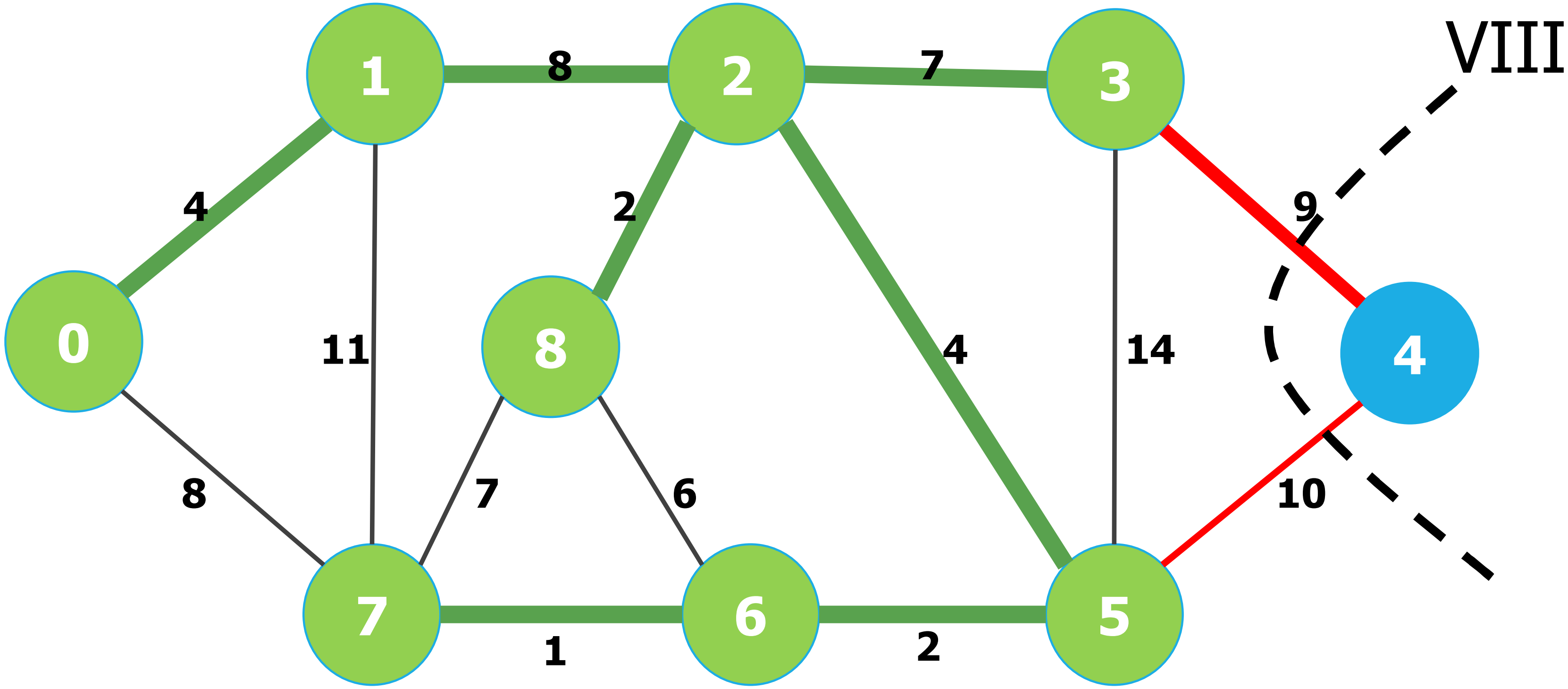
S = {0,1,2,5,6,8}

V-S = {3,4,7}



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

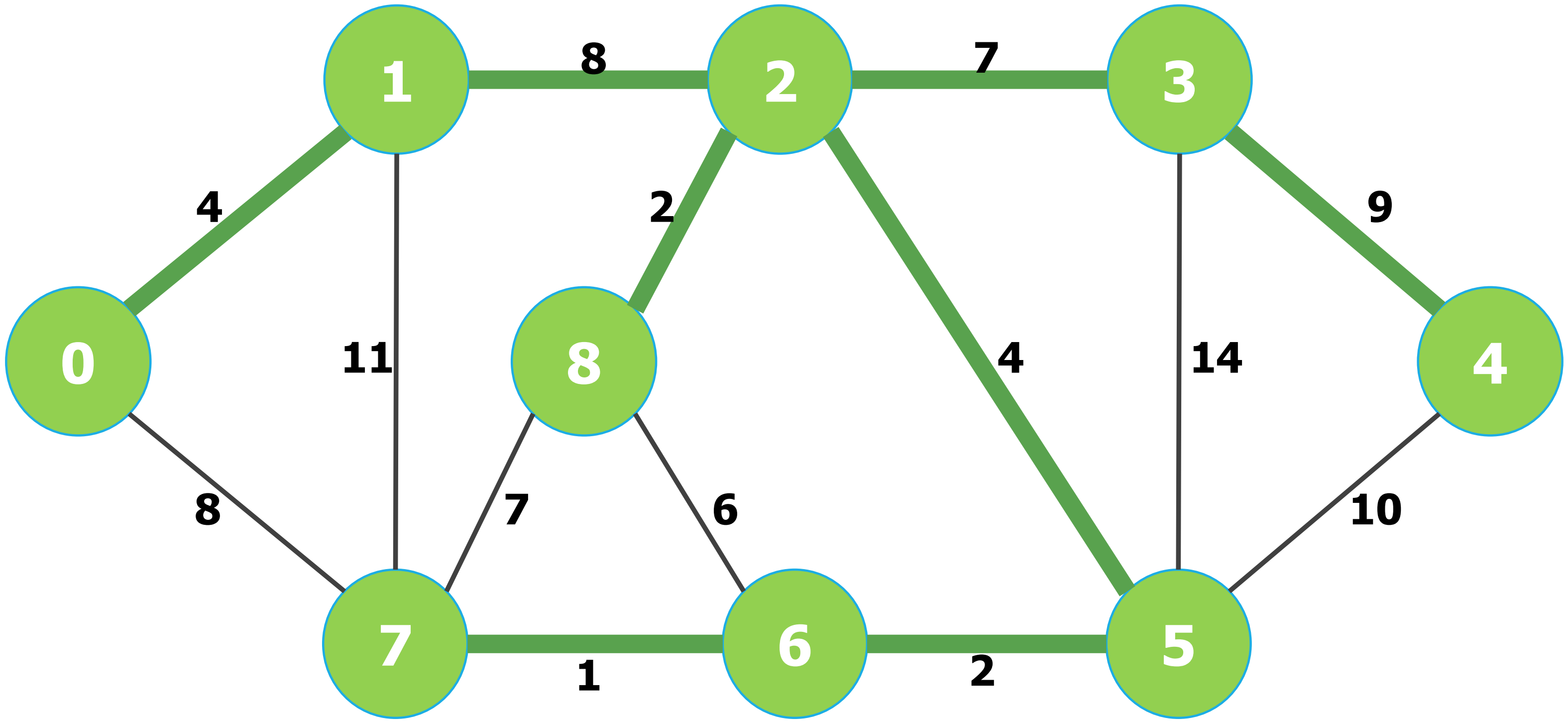
V-S = {3,4}



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

V-S = {4}

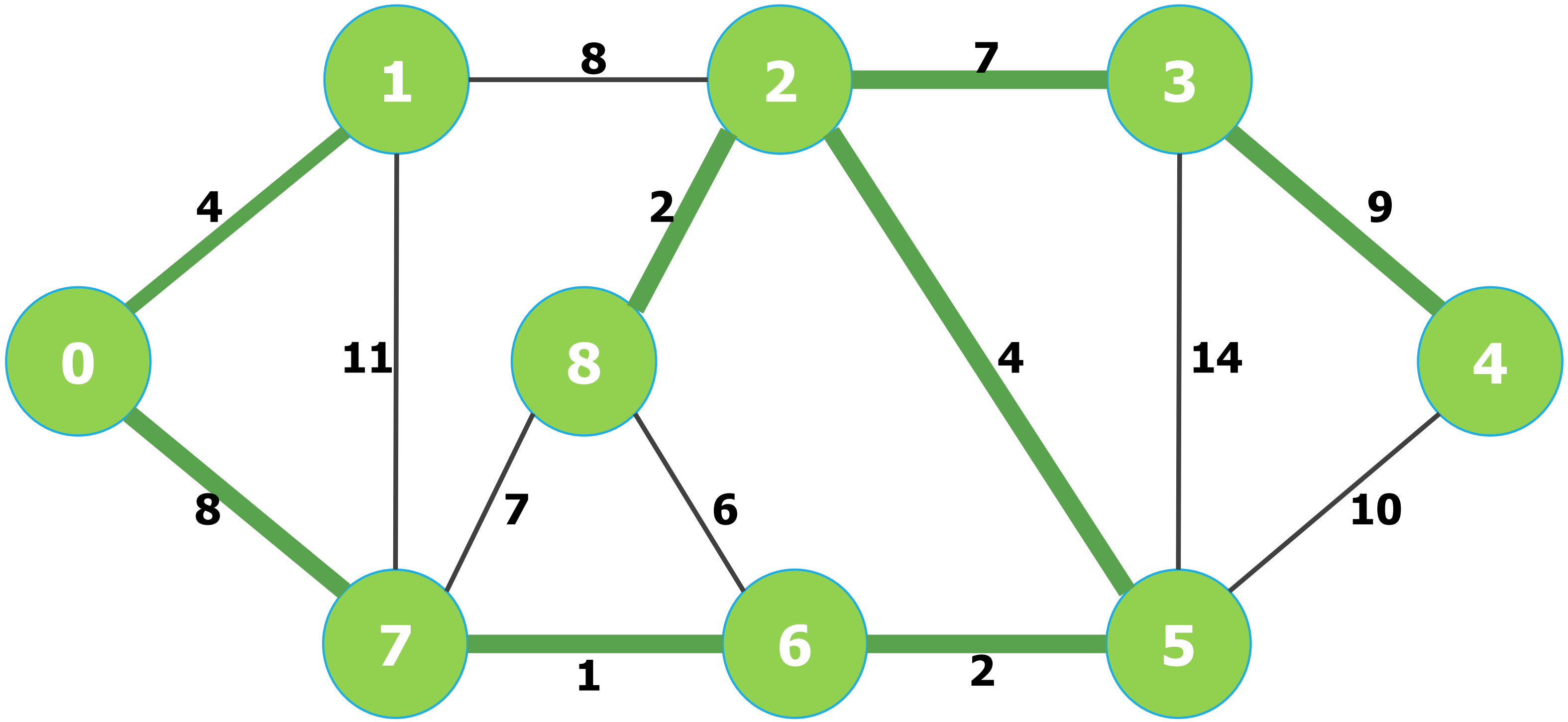
Somma minima dei pesi degli archi della soluzione: 37



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

V-S = ∅

Soluzione equivalente



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

V-S = ∅

Algoritmo

- L’approccio incrementale consiste nell’aggiungere ad ogni passo un vertice v a S
- ciò che interessa è la distanza minima da ogni vertice ancora in V-S ai vertici già in S
- quando si aggiunge il vertice v a S, ogni vertice w in V-S può avvicinarsi ai vertici già in S
- non serve memorizzare la distanza tra w e tutti i vertici in S, basta quella minima e verificare se l’aggiunta di v a S la diminuisce, nel qual caso la si aggiorna.

Strutture dati

- Grafo rappresentato come matrice delle adiacenze dove l’assenza di un arco si indica con `maxWT` anziché 0
- vettore `st` di `G->V` elementi per registrare per ogni vertice che appartiene ad S il padre
- vettore fringe (frangia) `fr` di `G->V` elementi per registrare per ogni vertice di V-S quale è il vertice di S più vicino. È dichiarato static in `Graph.c`

- vettore `wt` di `G->V+1` elementi 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
  - si considera un elemento fittizio con arco di peso `maxWT`
  - il vettore è inizializzato con `maxWT`
- variabile `min` per il vertice in V-S più vicino a vertici di S.

Azioni:

- ciclo esterno sui vertici prendendo di volta in volta quello a minima distanza (identificato da `min`) e aggiungendolo a S. Inizialmente `min` è il vertice 0

```
 for (min=0; min!=G->V; ) {
   v=min;  st[min]=fr[min];
```

  Notare che nel ciclo `for` non si incrementa `min`, `min` viene assegnato opportunamente nel corpo del ciclo

- ciclo interno sui vertici `w` non ancora in S (`st[w]==-1`):
  - se l'arco (v,w) migliora la stima (`if (G->madj[v][w]<wt[w])`)
    - la si aggiorna (`wt[w] = G->madj[v][w]`)
    - e si indica che il vertice in S più vicino a `w` è `v` (`fr[w] = v`)
  - se `w` è diventato il vertice più vicino a S (`if (wt[w]<wt[min])`), si aggiorna `min` (`min=w`).

wrapper

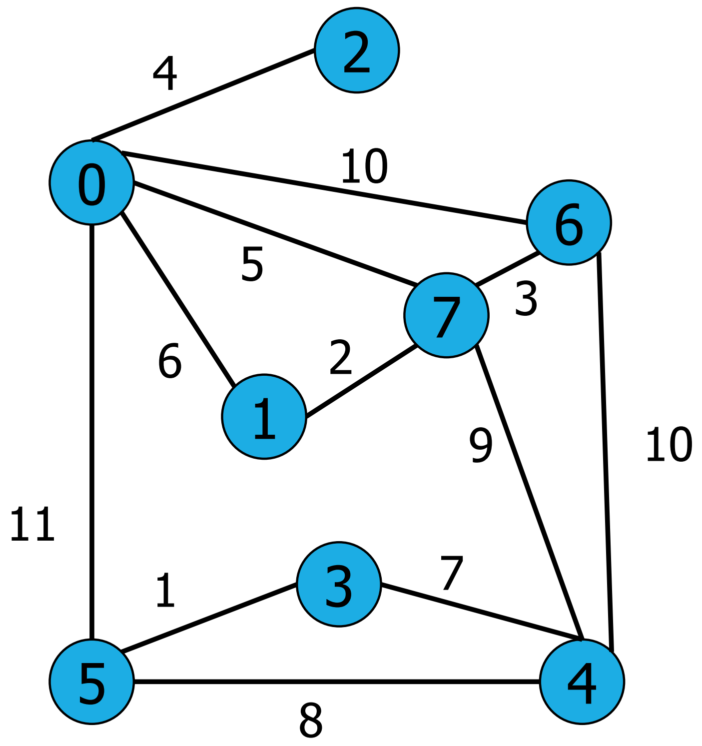
```
void GRAPHmstP(Graph G) {
  int v, *st, *wt, weight = 0;
  st = malloc(G->V*sizeof(int));
  wt = malloc((G->V+1)*sizeof(int));

  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",STsearchByIndex(G->tab,st[v]),
                         STsearchByIndex(G->tab,v));
      weight += wt[v];
    }
  }
  printf("\nminimum weight: %d\n", weight);
}
```

```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->madj[v][w] < wt[w]) {
          wt[w] = G->madj[v][w]; fr[w] = v;
        }
        if (wt[w] < wt[min]) min = w;
      }
  }
}
```

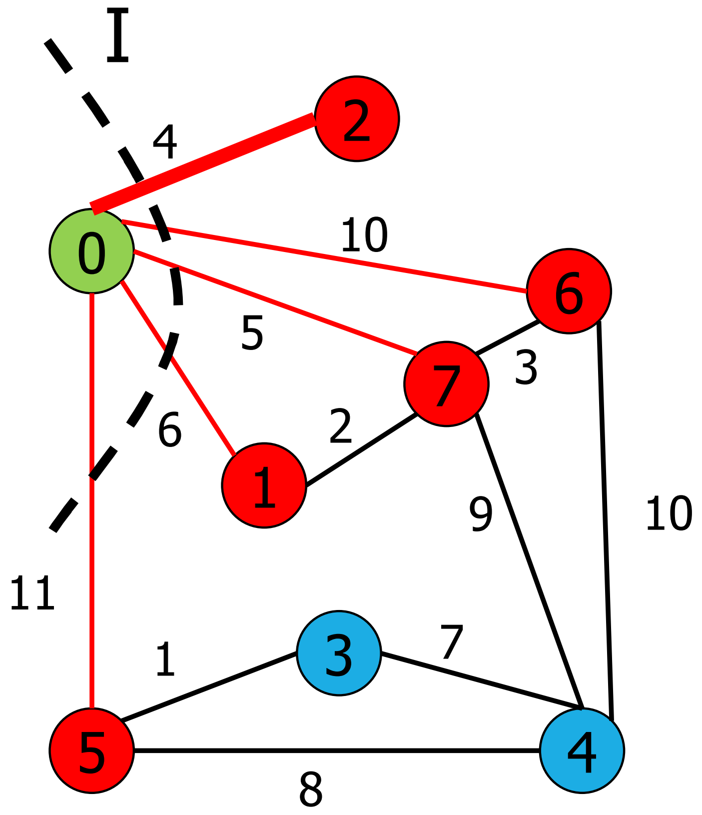
Esempio



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

$S = \varnothing$

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

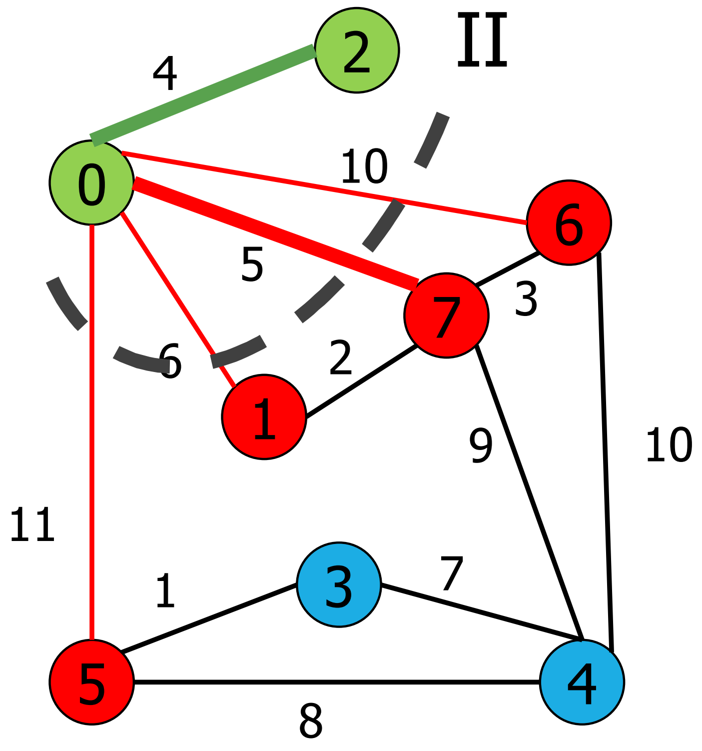


	0	1	2	3	4	5	6	7	8
st	0	-1	-1	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- min = 0, aggiorno st[0] e wt[0]
- fringe contiene 1, 2, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 2 è quello più vicino a S, in quanto 0-2 è l’arco a peso minimo che attraversa il taglio I
- min = 2

S = {0}

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

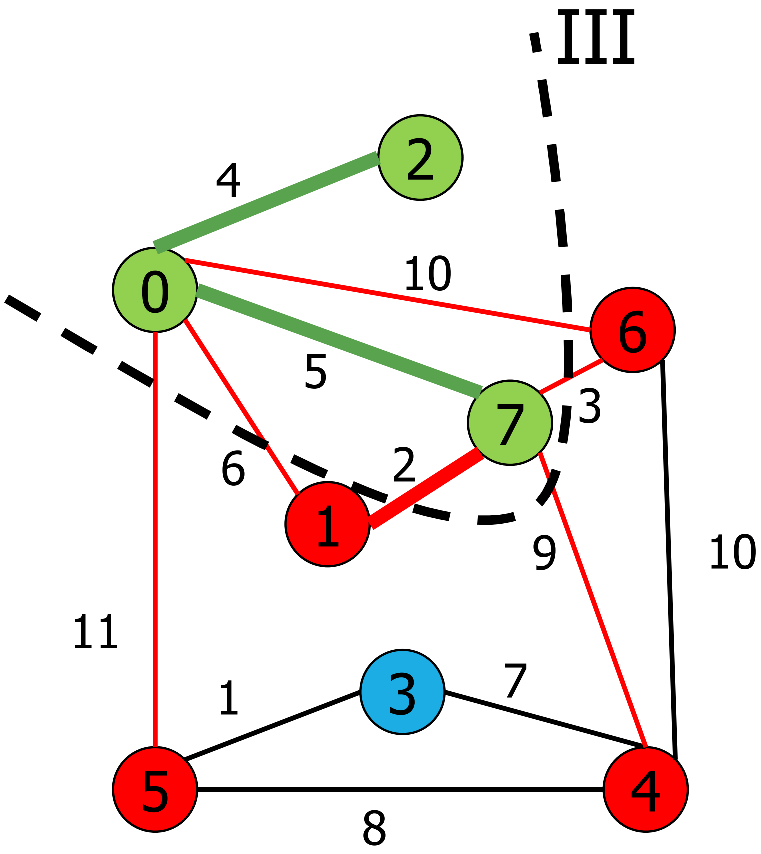


	0	1	2	3	4	5	6	7	8
st	0	-1	0	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- aggiungo 2 alla soluzione e aggiorno st[2]
- fringe contiene 1, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 7 è quello più vicino a S, in quanto 0-7 è l'arco a peso minimo che attraversa il taglio II
- min = 7

S = {0,2}

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

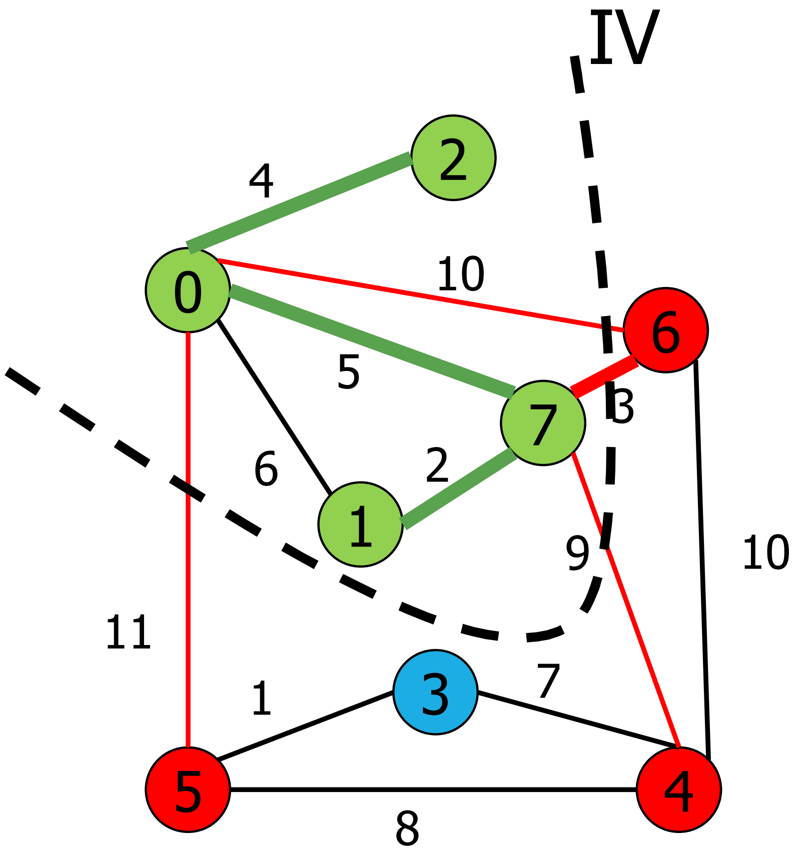


	0	1	2	3	4	5	6	7	8
st	0	-1	0	-1	-1	-1	-1	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 7 alla soluzione e aggiorno st[7]
- fringe contiene 1, 4, 5, 6
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 1 è quello più vicino a S, in quanto 1-7 è l’arco a peso minimo che attraversa il taglio III
- min = 1

S = {0,2,7}

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

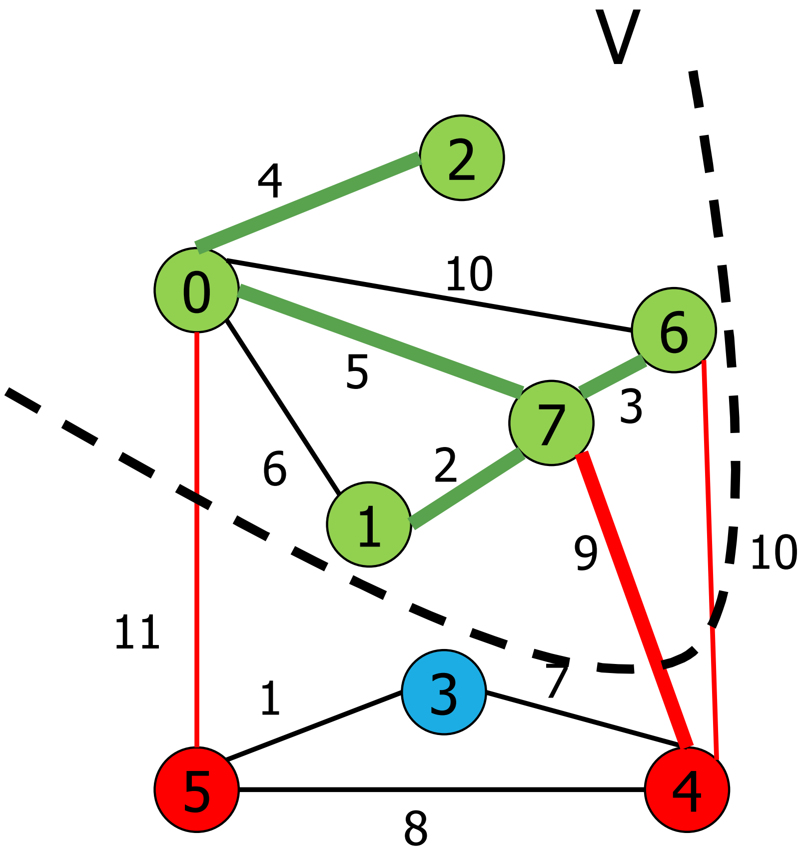


	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	-1	-1	-1	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 1 alla soluzione e aggiorno st[1]
- fringe contiene 4, 5, 6
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 6 è quello più vicino a S, in quanto 6-7 è l'arco a peso minimo che attraversa il taglio IV
- min = 6

S = {0,2,7,1}

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

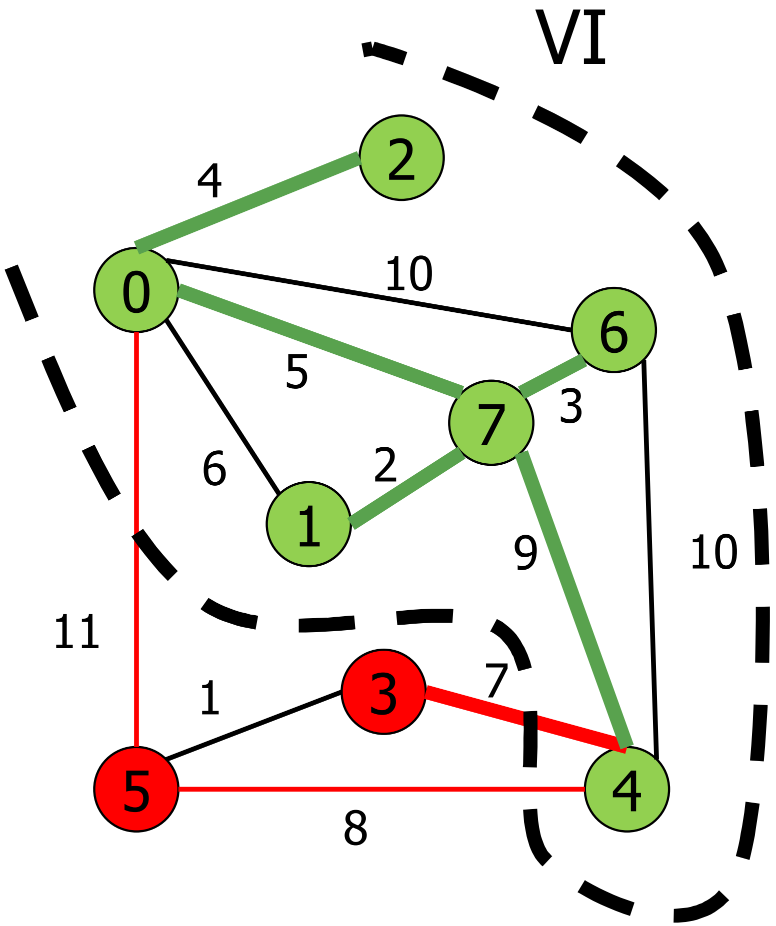


	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	-1	-1	7	0	
wt	0	2	4	∞	9	11	3	5	∞
fr	0	7	0	3	7	0	7	0	

- aggiungo 6 alla soluzione e aggiorno st[6]
- fringe contiene 4, 5
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 4 è quello più vicino a S, in quanto 4-7 è l'arco a peso minimo che attraversa il taglio V
- min = 4

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

V-S = {3,4,5}

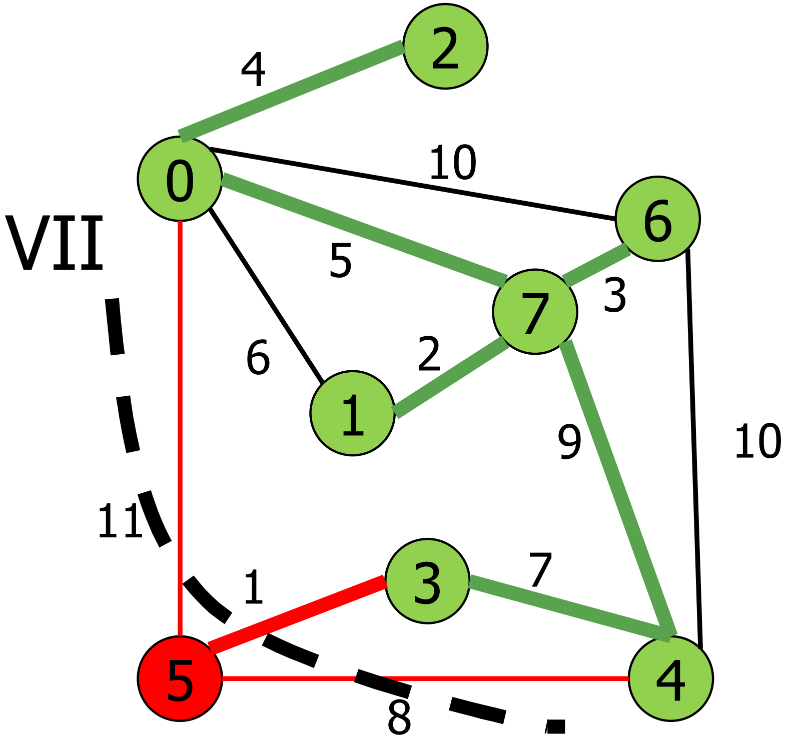


	0	1	2	3	4	5	6	7	8
st	0	7	0	-1	7	-1	7	0	
wt	0	2	4	7	9	8	3	5	∞
fr	0	7	0	4	7	4	7	0	

- aggiungo 4 alla soluzione e aggiorno st[4]
- fringe contiene 3, 5
- aggiorno wt in base agli archi che attraversano il taglio
- il vertice 3 è quello più vicino a S, in quanto 3-4 è l'arco a peso minimo che attraversa il taglio VI
- min = 3

S = {0,2,7,1,6,4}

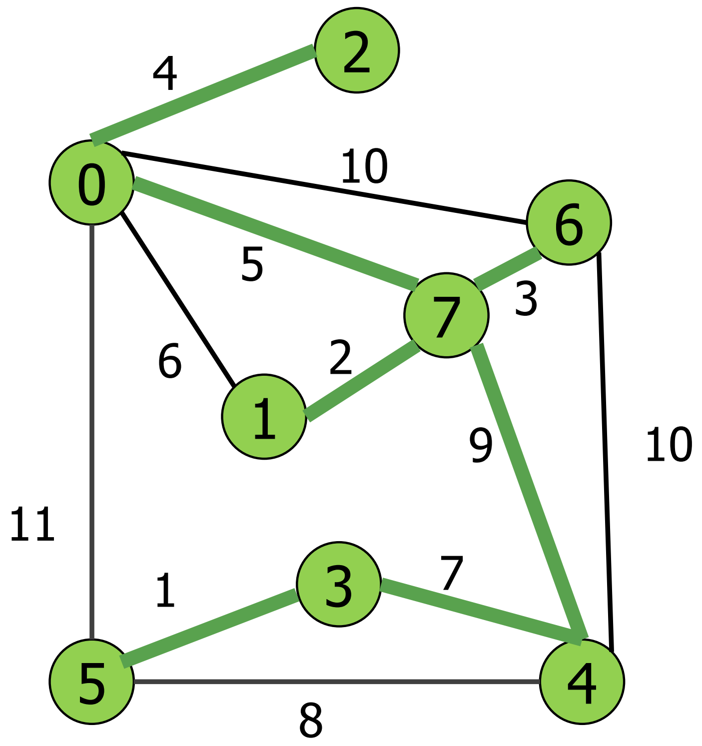
V-S = {3,5}



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

- aggiungo 3 alla soluzione e aggiorno st[3]
- fringe contiene 5
- aggiorno wt in base agli archi che attraversano il taglio
- il vertice 5 è quello più vicino a S, in quanto 3-5 è l’arco a peso minimo che attraversa il taglio VII
- min = 5

S = {0,2,7,1,6,4,3} V-S = {5}



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

- aggiungo 5 alla soluzione e aggiorno st[5]
- terminazione
- somma minima dei pesi 31.

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

V-S = ∅

Complessità

Per grafi densi: $T(n) = O(|V|^2)$

Possibili miglioramenti per grafi sparsi: usare una coda a priorità per gestire la fringe.
Con coda a priorità implementata con heap $T(n) = O(|E| \log |V|)$.

Riferimenti

- Rappresentazione:
  - Sedgewick Part 5 20.1
- Principi:
  - Sedgewick Part 5 20.2
  - Cormen 23.1
- Algoritmo di Kruskal
  - Sedgewick Part 5 20.4
  - Cormen 23.2
- Algoritmo di Prim
  - Sedgewick Part 5 20.3
  - Cormen 23.2

Esercizi di teoria

- 11. Alberi ricoprenti minimi
  - 11.2 Algoritmi di Kruskal e Prim