Minimum Spanning Tree, Union-Find e Ordinamento Topologico

Alice Cortinovis e Edoardo Morassutto

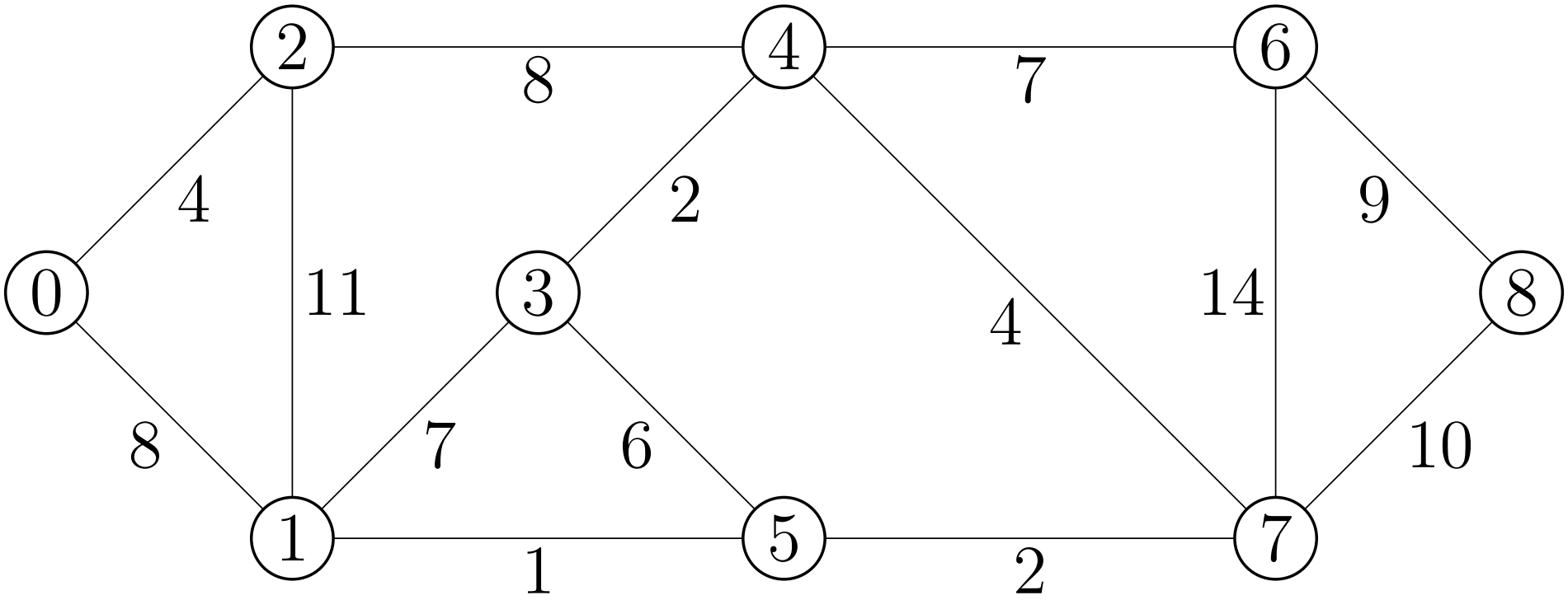
Online, 5 dicembre 2021

Minimum Spanning Tree

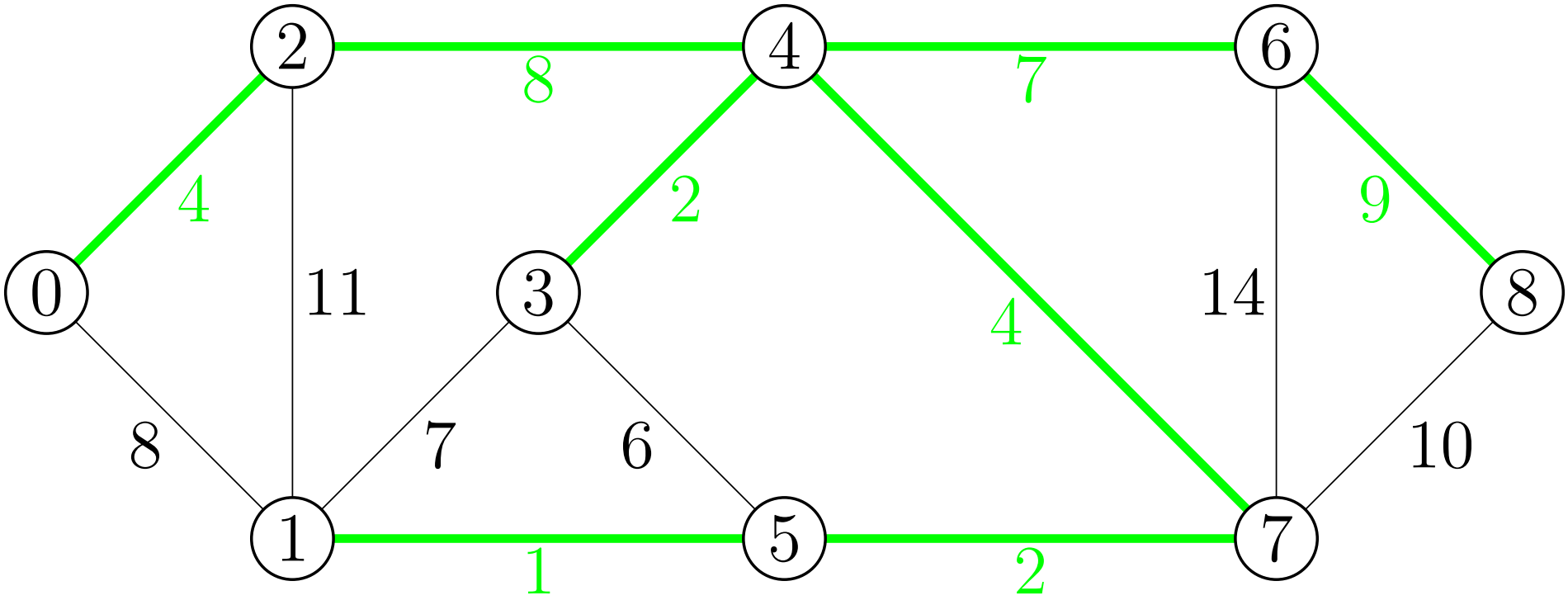
Problema: Dato un grafo non diretto $G = (V, E)$ con N nodi e M archi, trovare l'albero ricoprente di minimo costo totale.

Minimum Spanning Tree

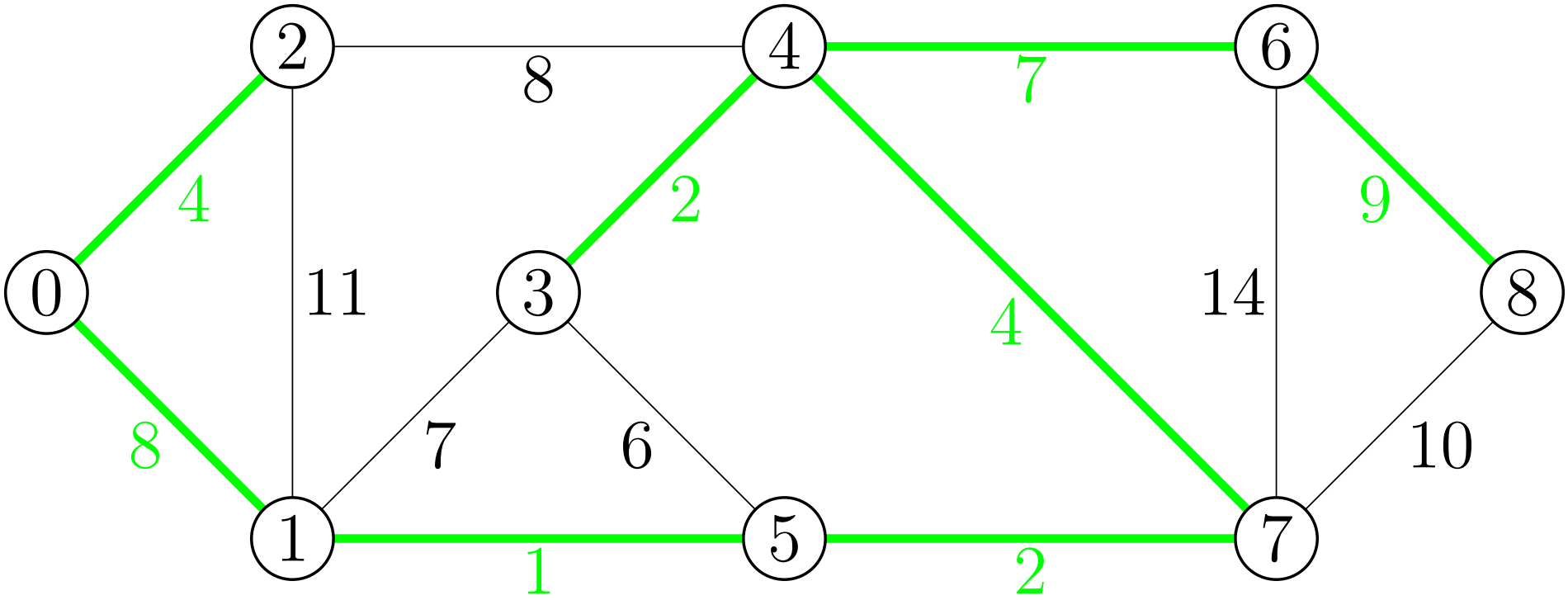
Problema: Dato un grafo non diretto $G = (V, E)$ con N nodi e M archi, trovare l'albero ricoprente di minimo costo totale.



Ecco *una* soluzione:



Eccone un'altra:



Due algoritmi per il calcolo del MST:

- Kruskal
- Prim

Due algoritmi per il calcolo del MST:

- Kruskal
- Prim

Idea: Teniamo un insieme S degli archi scelti, che all'inizio è vuoto. Ad ogni passo aggiungiamo a S un nuovo arco che faccia parte di una soluzione che contiene gli elementi di S.

Due algoritmi per il calcolo del MST:

- Kruskal
- Prim

Idea: Teniamo un insieme S degli archi scelti, che all'inizio è vuoto. Ad ogni passo aggiungiamo a S un nuovo arco che faccia parte di una soluzione che contiene gli elementi di S.

Come si fa ad essere sicuri che un arco sia parte di una soluzione?

Algoritmo di Kruskal

- $S := \emptyset$

Algoritmo di Kruskal

- $S := \emptyset$
- Ordina gli archi per peso crescente

Algoritmo di Kruskal

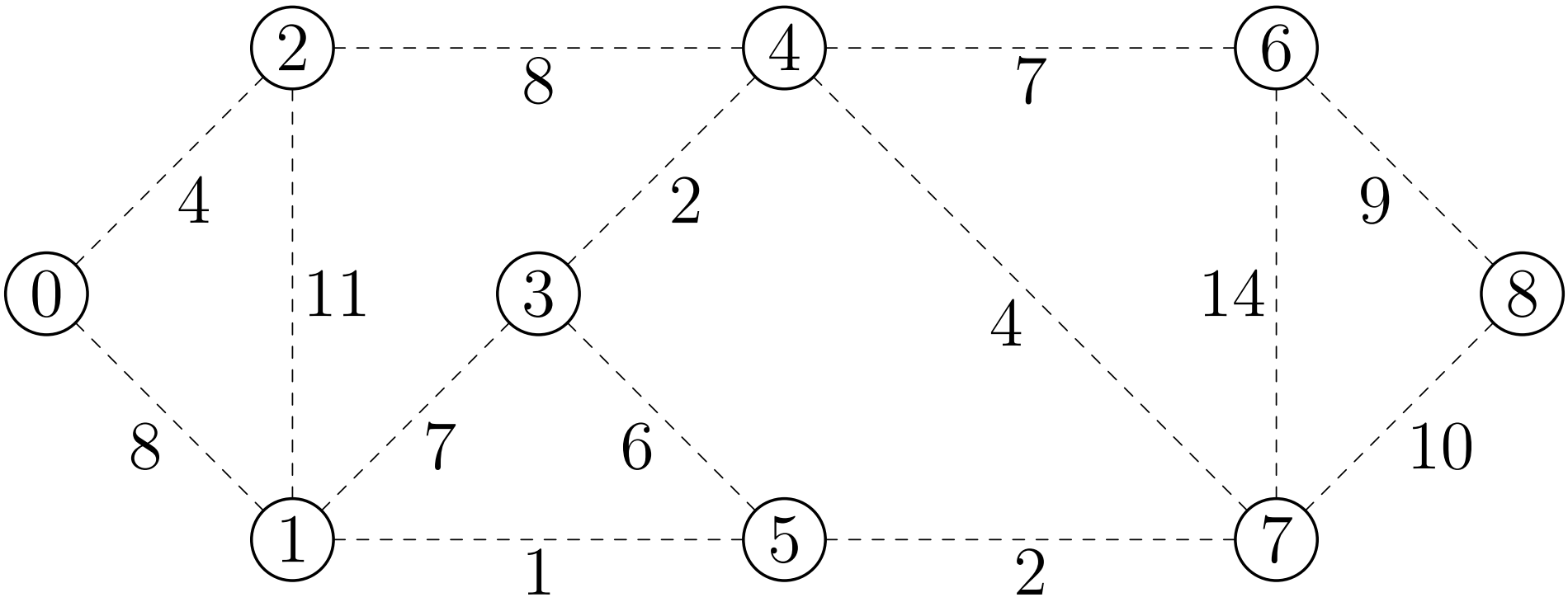
- $S := \emptyset$
- Ordina gli archi per peso crescente
- Ora guarda ogni arco in tale ordine: se è *utile* lo aggiungi a S, altrimenti ti dimentichi di lui per sempre

Algoritmo di Kruskal

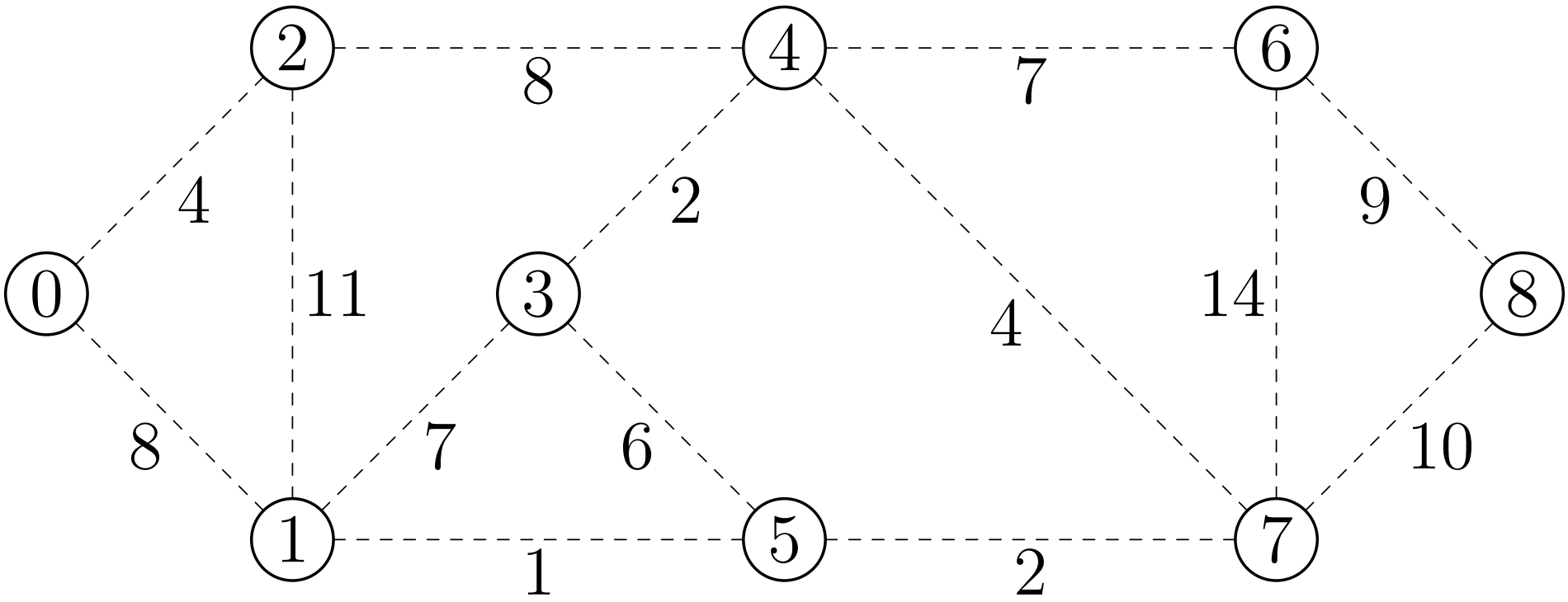
- $S := \emptyset$
- Ordina gli archi per peso crescente
- Ora guarda ogni arco in tale ordine: se è *utile* lo aggiungi a S, altrimenti ti dimentichi di lui per sempre

Arco utile vuol dire che connette due nodi che stanno in due componenti connesse diverse di (V, S).

Vediamo cosa succede sul grafo di esempio visto all'inizio.

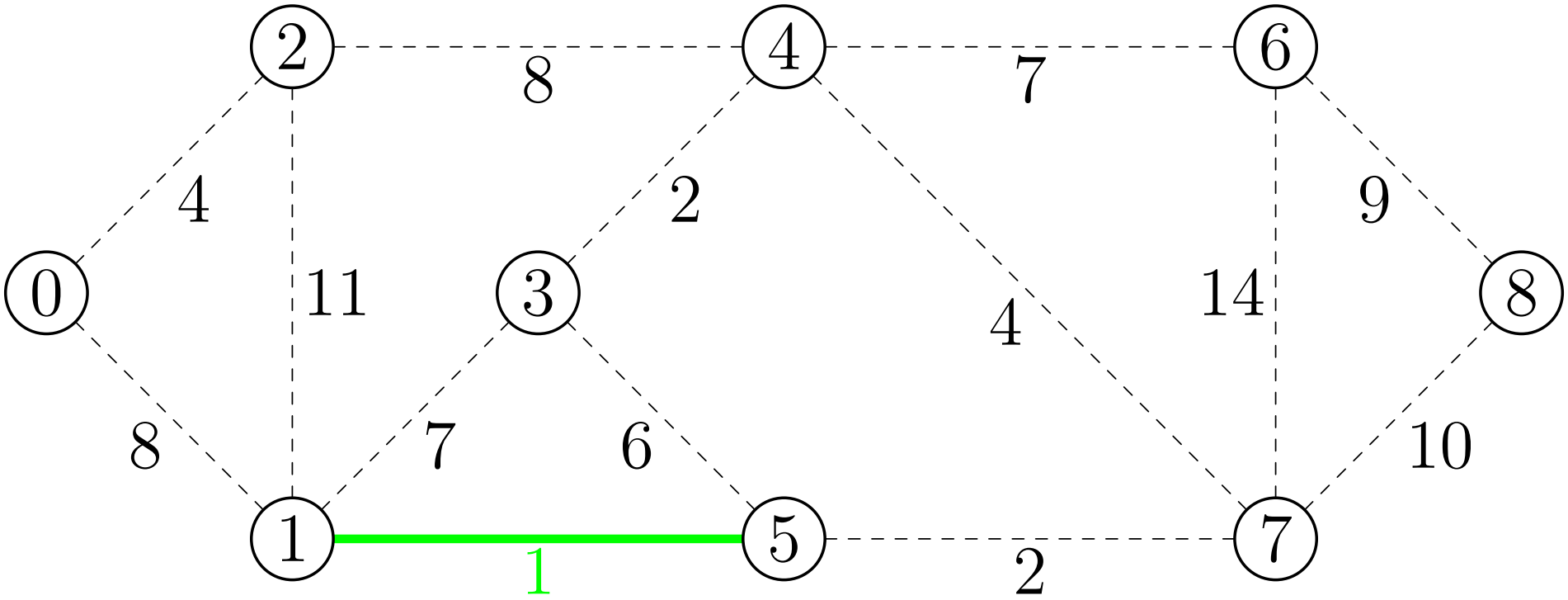


Vediamo cosa succede sul grafo di esempio visto all'inizio.

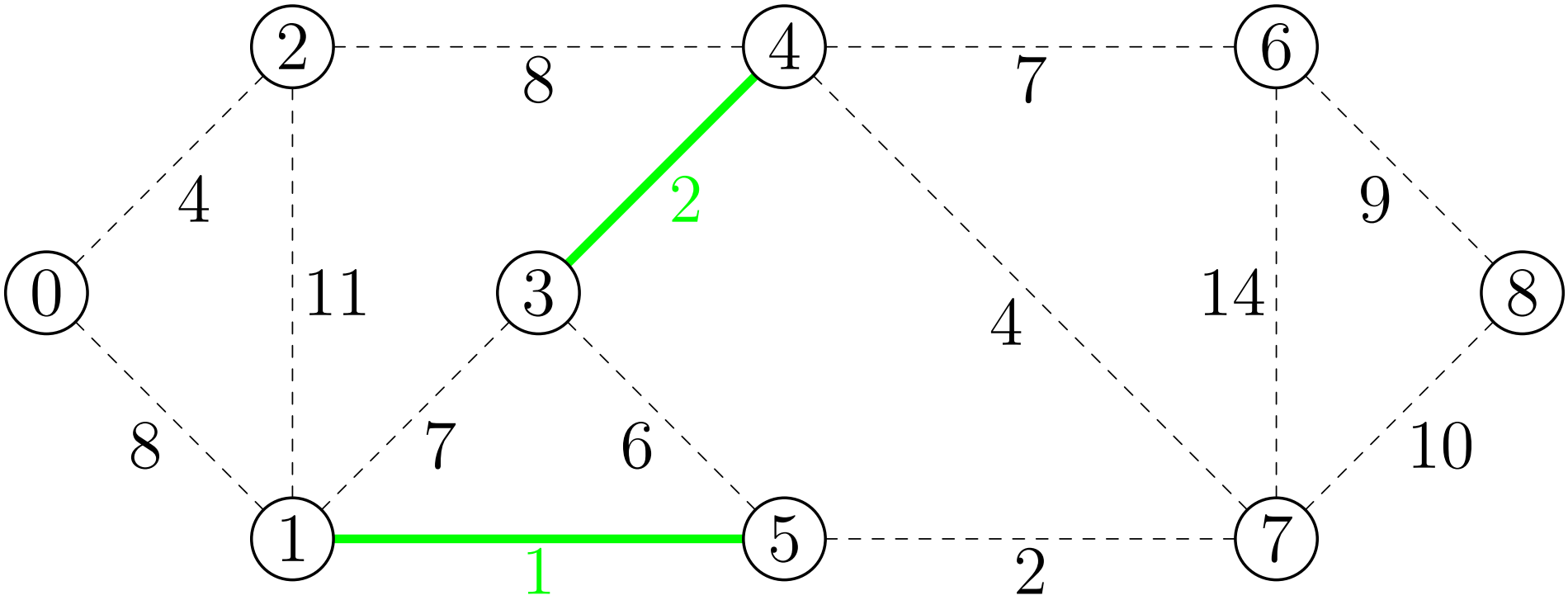


Gli archi ordinati sono:
$(1,5),(3,4),(5,7),(4,7),(0,2),(5,3),(1,3),$
$(4,6),(2,4),(0,1),(6,8),(7,8),(1,2),(6,7)$

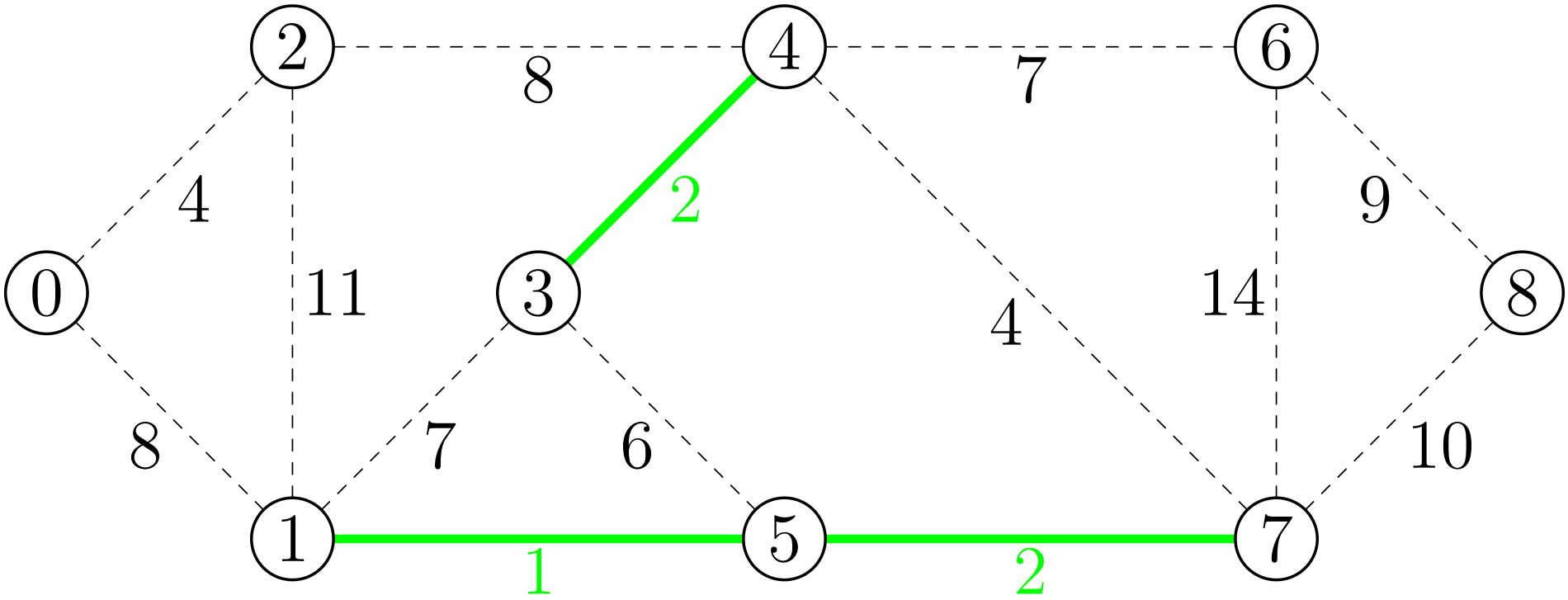
Arco $(1, 5)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



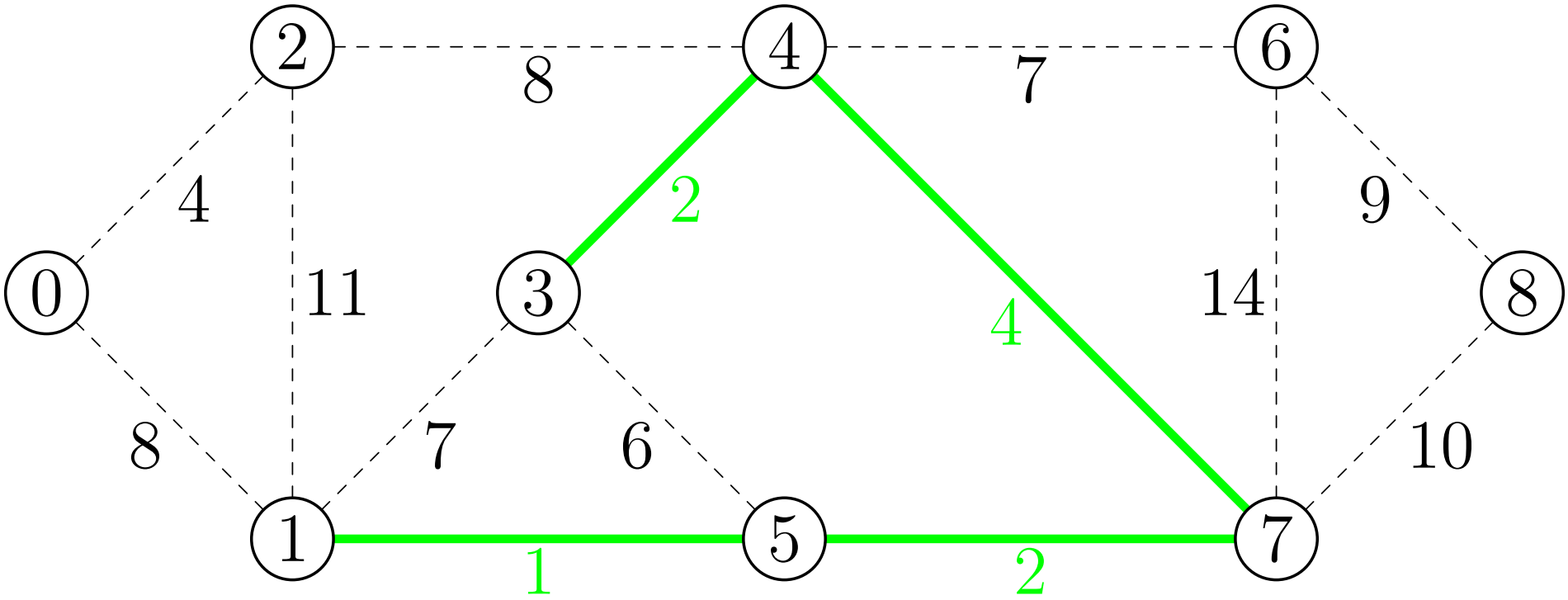
Arco $(3, 4)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



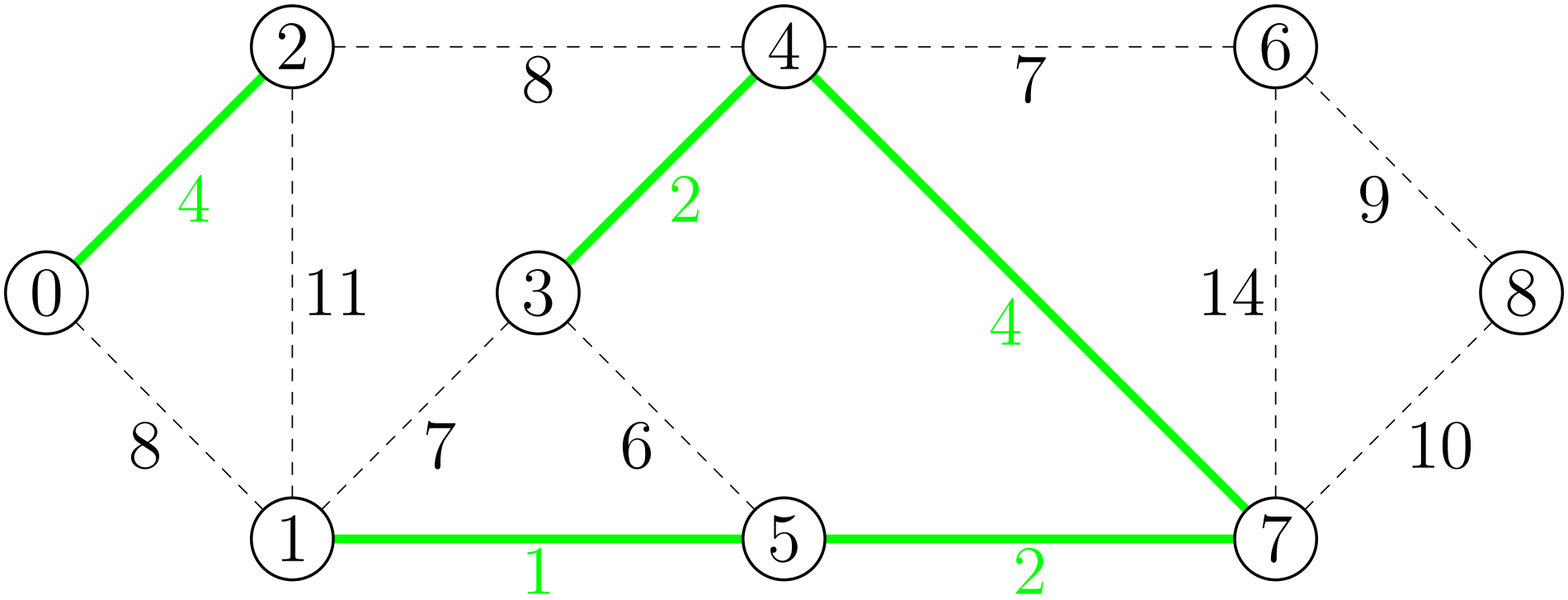
Arco $(5, 7)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



Arco $(4, 7)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$

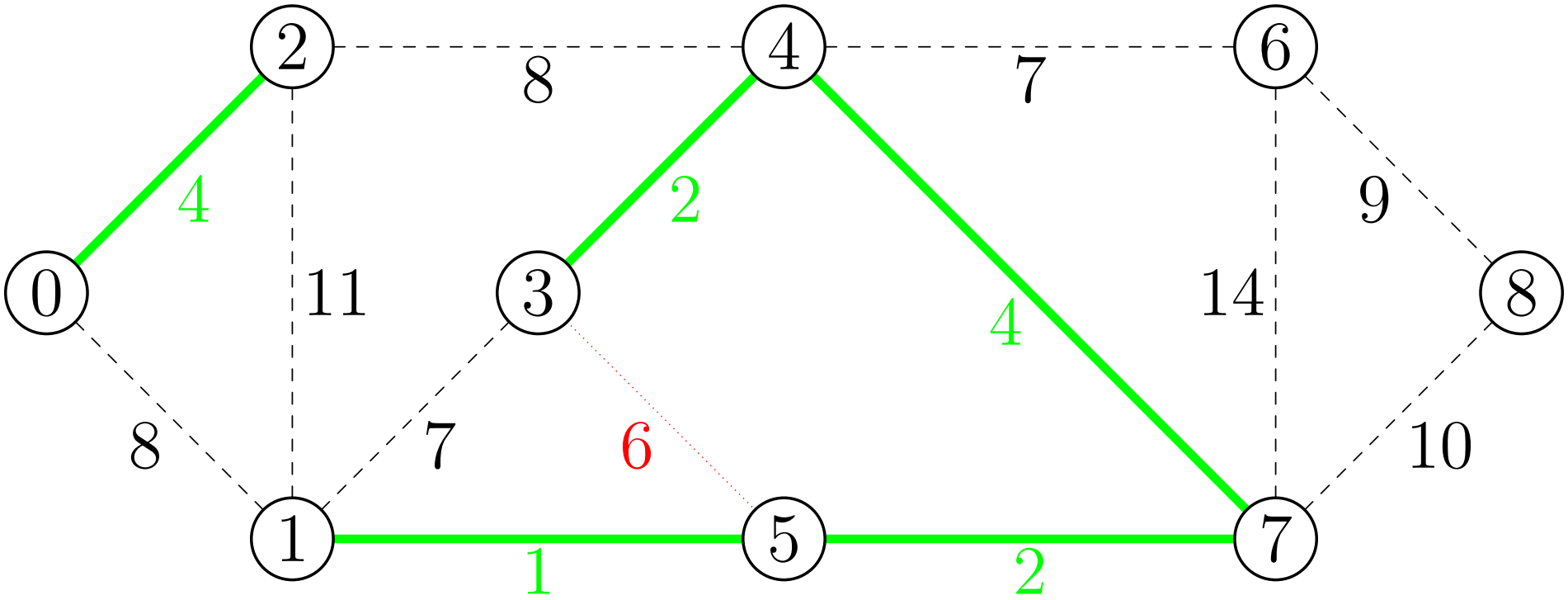


Arco $(0, 2)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



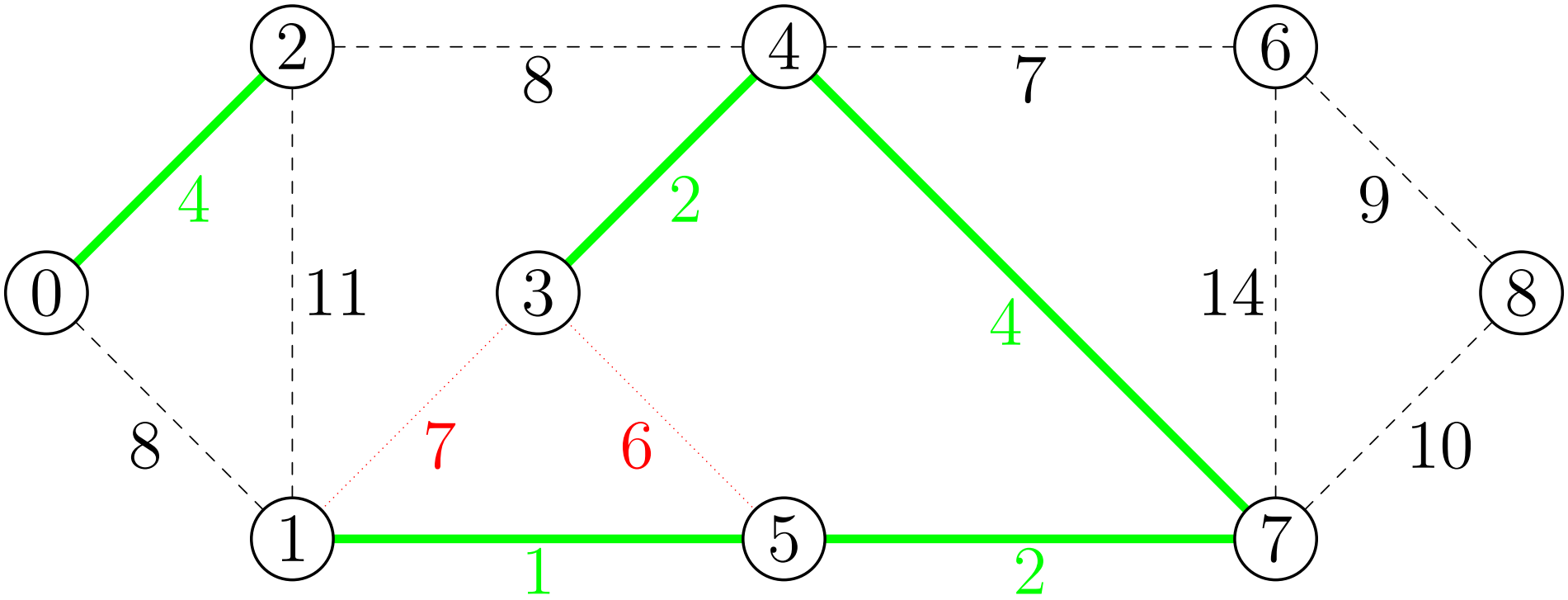
Arco $(5, 3)$: è inutile!

$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$

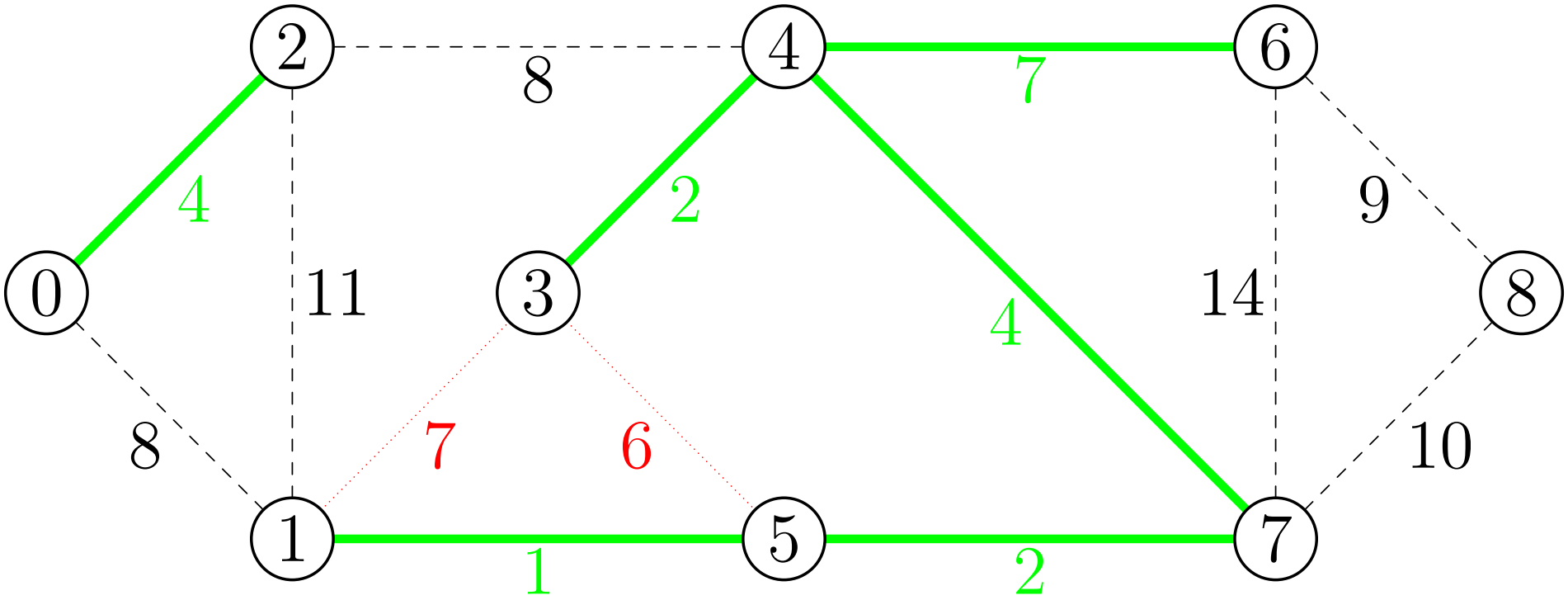


Arco $(1, 3)$: è inutile!

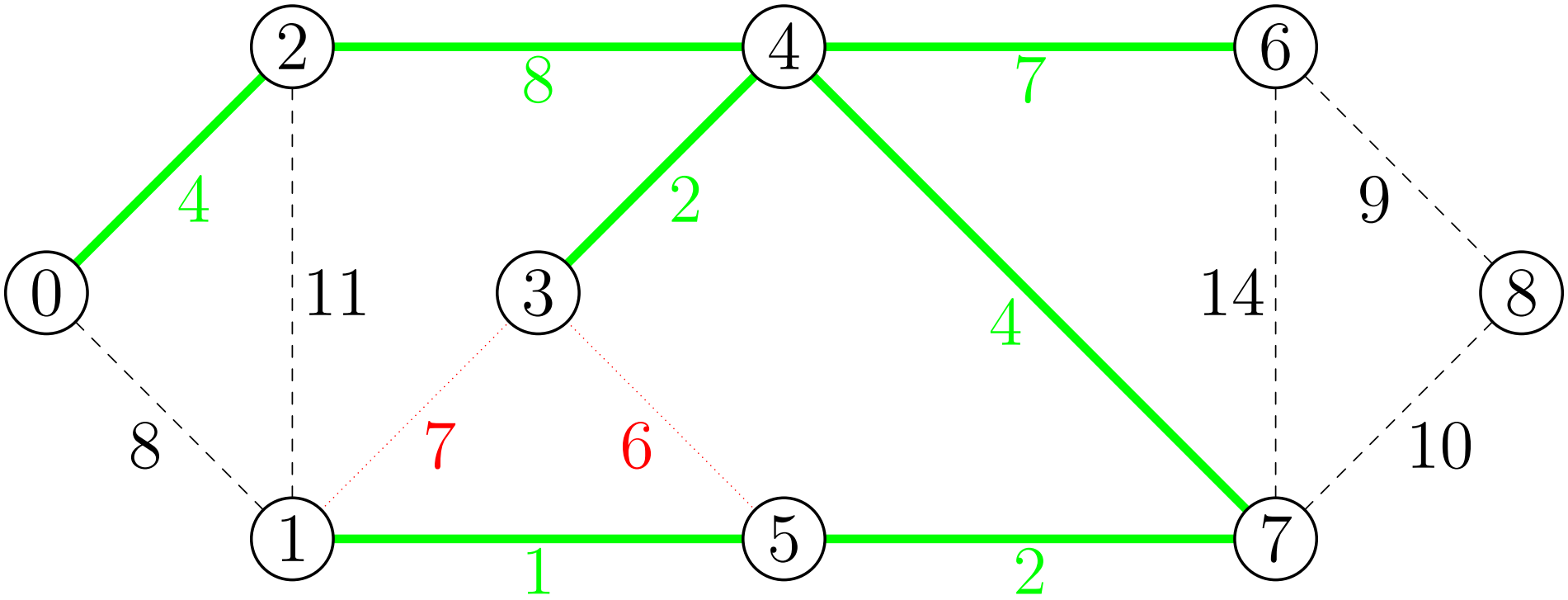
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



Arco $(4, 6)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$

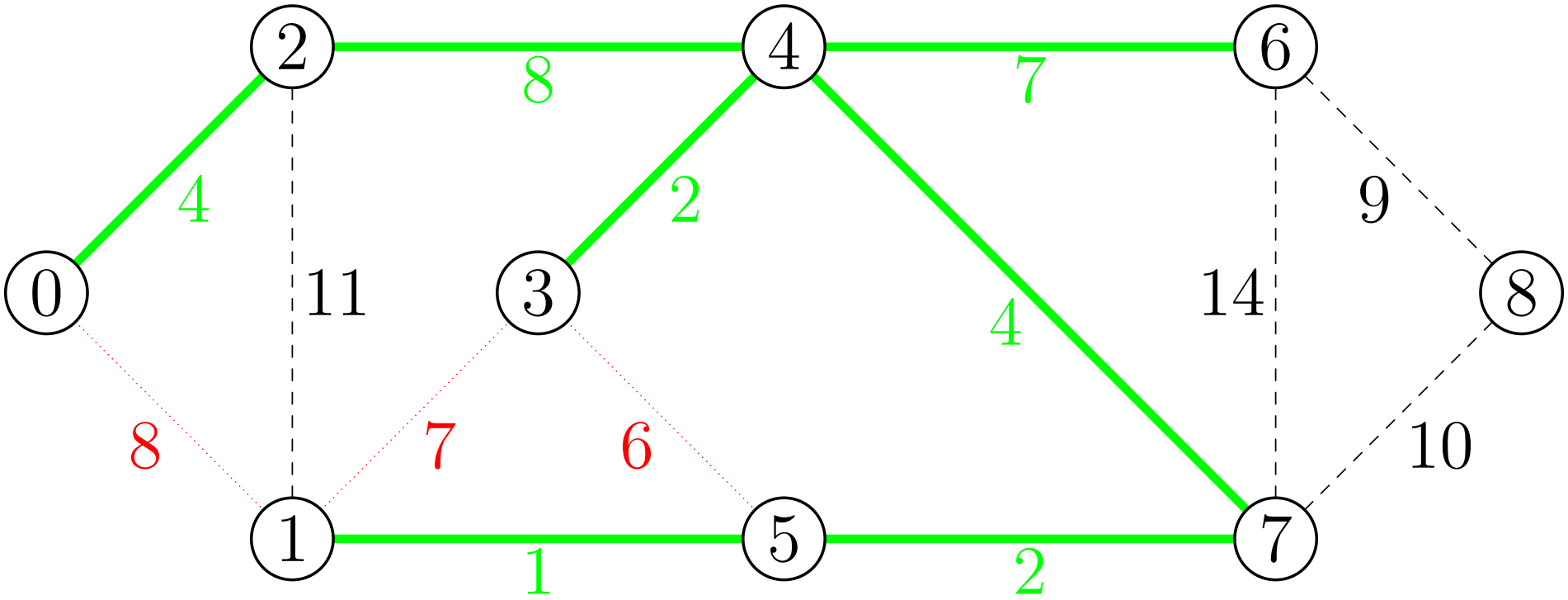


Arco $(2, 4)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$

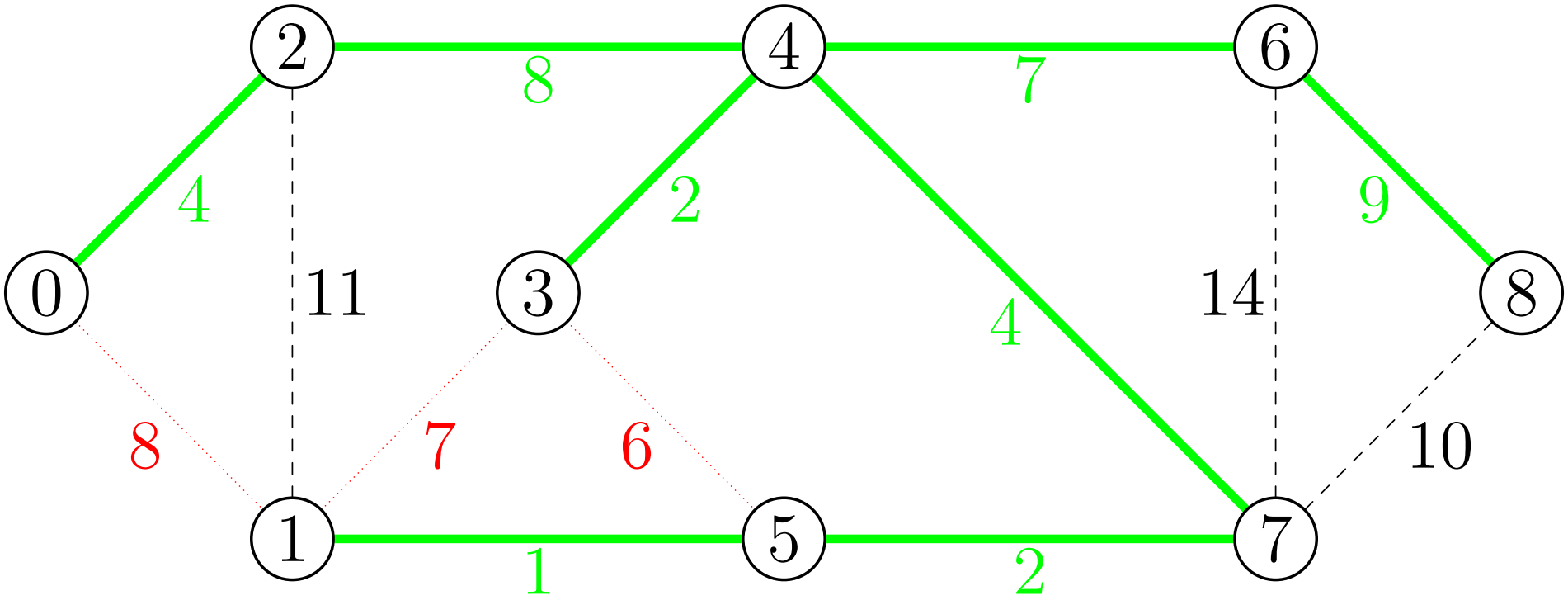


Arco $(0, 1)$: è inutile!

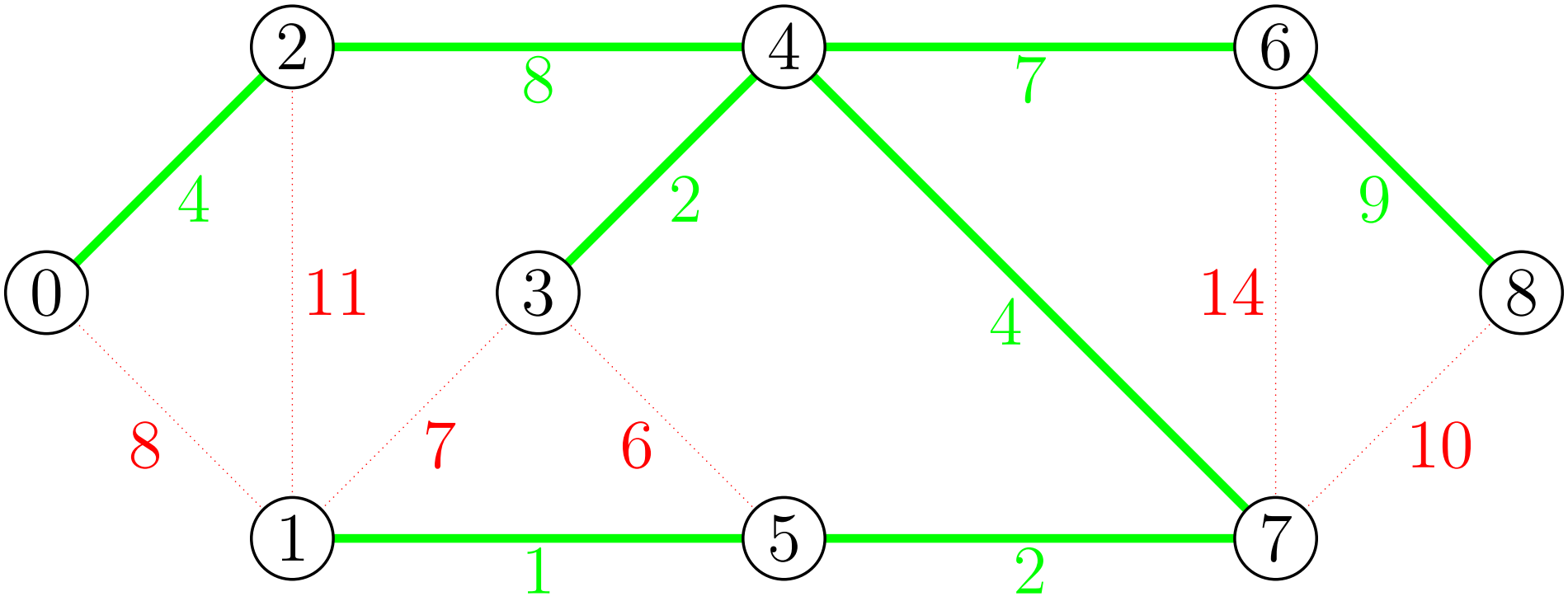
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



$(1,5), (3,4), (5,7), (4,7), (0,2), (5,3), (1,3),$
$(4,6), (2,4), (0,1), (6,8), (7,8), (1,2), (6,7)$
Arco $(6,8)$: è utile, lo prendo.



$(1,5), (3,4), (5,7), (4,7), (0,2), (5,3), (1,3),$
$(4,6), (2,4), (0,1), (6,8), (7,8), (1,2), (6,7)$
Poiché ho già ottenuto un albero, tutti gli archi successivi sono inutili.



Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.

Perché funziona?

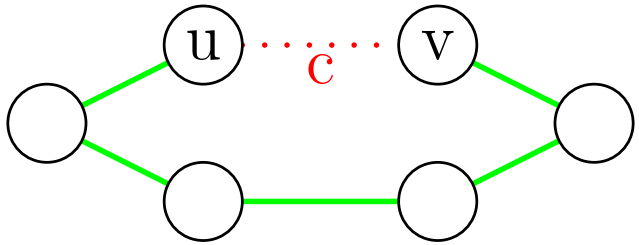
Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).

Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

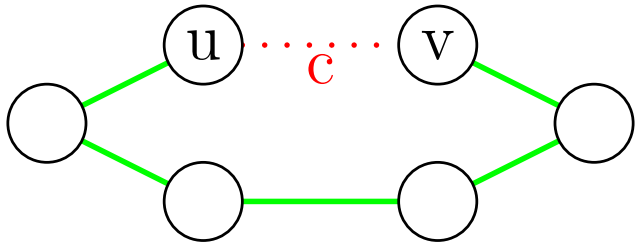
- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v.



Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v.

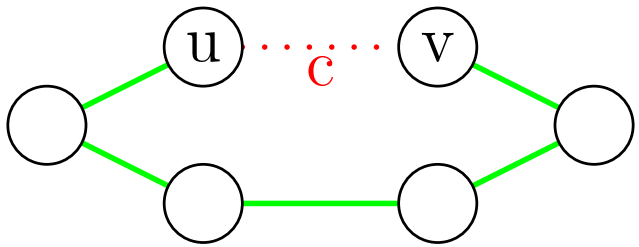


- Dato che Kruskal ha scelto (u, v) almeno uno degli altri archi deve avere peso maggiore o uguale a c.

Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v.



- Dato che Kruskal ha scelto (u, v) almeno uno degli altri archi deve avere peso maggiore o uguale a c.
- Se scambi (u, v) con quell'arco otteniamo un MST di valore minore o uguale di S, contenente anche (u, v). Assurdo.

Come faccio a capire in fretta se un arco è utile?

Come faccio a capire in fretta se un arco è utile?

Problema: voglio tenere un certo numero di insiemi disgiunti, ciascuno con un'etichetta e, e voglio supportare le seguenti operazioni.

- `MAKE_SET`(x): crea un nuovo insieme con dentro solo l'elemento x
- `FIND`(x): restituisce l'etichetta dell'insieme contenente x
- `UNION`(x, y): unisce gli insiemi contenenti x e y

Nel seguito supporremo che gli elementi degli insiemi siano i numeri da 1 a N.

Come faccio a capire in fretta se un arco è utile?

Problema: voglio tenere un certo numero di insiemi disgiunti, ciascuno con un'etichetta e, e voglio supportare le seguenti operazioni.

- `MAKE_SET`(x): crea un nuovo insieme con dentro solo l'elemento x
- `FIND`(x): restituisce l'etichetta dell'insieme contenente x
- `UNION`(x, y): unisce gli insiemi contenenti x e y

Nel seguito supporremo che gli elementi degli insiemi siano i numeri da 1 a N.

```
MAKE_SET(1); MAKE_SET(2); MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

$$1 : \{1\}$$
$$2 : \{2\}$$
$$3 : \{3\}$$
$$4 : \{4\}$$
$$5 : \{5\}$$

Come faccio a capire in fretta se un arco è utile?

Problema: voglio tenere un certo numero di insiemi disgiunti, ciascuno con un'etichetta e, e voglio supportare le seguenti operazioni.

- `MAKE_SET`(x): crea un nuovo insieme con dentro solo l'elemento x
- `FIND`(x): restituisce l'etichetta dell'insieme contenente x
- `UNION`(x, y): unisce gli insiemi contenenti x e y

Nel seguito supporremo che gli elementi degli insiemi siano i numeri da 1 a N.

```
UNION(2, 3); UNION(1, 5)
```

$$1 : \{1, 5\}$$
$$3 : \{2, 3\}$$
$$4 : \{4\}$$

Come faccio a capire in fretta se un arco è utile?

Problema: voglio tenere un certo numero di insiemi disgiunti, ciascuno con un'etichetta e, e voglio supportare le seguenti operazioni.

- `MAKE_SET`(x): crea un nuovo insieme con dentro solo l'elemento x
- `FIND`(x): restituisce l'etichetta dell'insieme contenente x
- `UNION`(x, y): unisce gli insiemi contenenti x e y

Nel seguito supporremo che gli elementi degli insiemi siano i numeri da 1 a N.

`FIND(1) = 1` `FIND(5) = 1` `FIND(2) = 3`

$$1 : \{1, 5\}$$
$$3 : \{2, 3\}$$
$$4 : \{4\}$$

Algoritmo naive

L'etichetta di un singoletto è lui stesso; ogni volta che unisco due insiemi, assegno agli elementi del secondo l'etichetta del primo.

```
1  int p[N]; // inizializzato a -1
2  void MAKE_SET(int x) { p[x] = x; }
3  void UNION(int x, int y) {
4     if (p[x] != p[y]) {
5        int py = p[y];
6        for (int i = 1; i <= N; i++)
7           if (p[i] == py)
8              p[i] = p[x];
9     }
10 }
11 int FIND(int x) { return p[x]; }
```

Complessità?

Algoritmo naive

L'etichetta di un singoletto è lui stesso; ogni volta che unisco due insiemi, assegno agli elementi del secondo l'etichetta del primo.

```
1  int p[N]; // inizializzato a -1
2  void MAKE_SET(int x) { p[x] = x; }
3  void UNION(int x, int y) {
4     if (p[x] != p[y]) {
5        int py = p[y];
6        for (int i = 1; i <= N; i++)
7           if (p[i] == py)
8              p[i] = p[x];
9     }
10 }
11 int FIND(int x) { return p[x]; }
```

Complessità?

- `FIND` $\rightarrow \mathcal{O}(1)$
- `MAKE_SET` $\rightarrow \mathcal{O}(1)$
- `UNION` $\rightarrow \mathcal{O}(N)$

Insiemi come alberi

Insiemi come alberi

```
1  int p[N]; // inizializzato a -1
2
3  void MAKE_SET(int x) { p[x] = x; }
4
5  void UNION(int x, int y) {
6     int a = FIND(x);
7     int b = FIND(y);
8     if (a != b)
9        p[b] = a;
10 }
11
12 int FIND(int x) {
13    if (x == p[x]) return x;
14    else return FIND(p[x]);
15 }
```

Complessità?

Insiemi come alberi

```
1  int p[N]; // inizializzato a -1
2
3  void MAKE_SET(int x) { p[x] = x; }
4
5  void UNION(int x, int y) {
6      int a = FIND(x);
7      int b = FIND(y);
8      if (a != b)
9          p[b] = a;
10 }
11
12 int FIND(int x) {
13     if (x == p[x]) return x;
14     else return FIND(p[x]);
15 }
```

Complessità?

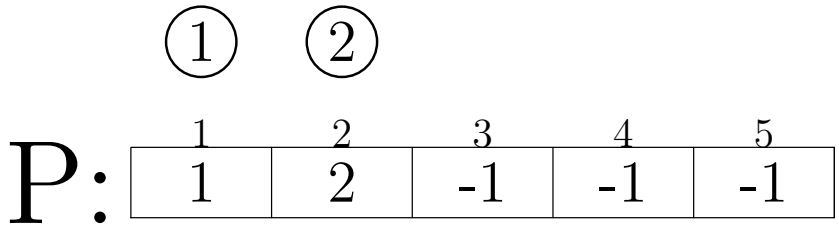
- FIND $\rightarrow \mathcal{O}(N)$
- MAKE_SET $\rightarrow \mathcal{O}(1)$
- UNION $\rightarrow \mathcal{O}(N)$

Esempio

```
MAKE_SET(1);
MAKE_SET(2);
```

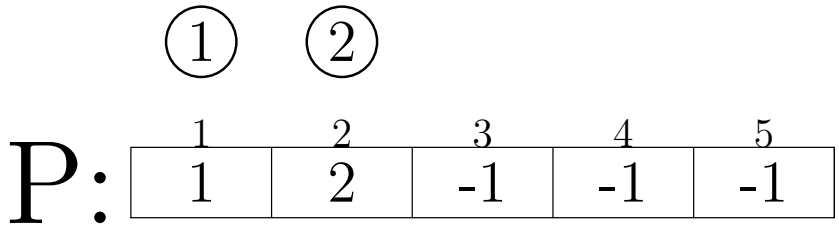
Esempio

```
MAKE_SET(1);
MAKE_SET(2);
```



Esempio

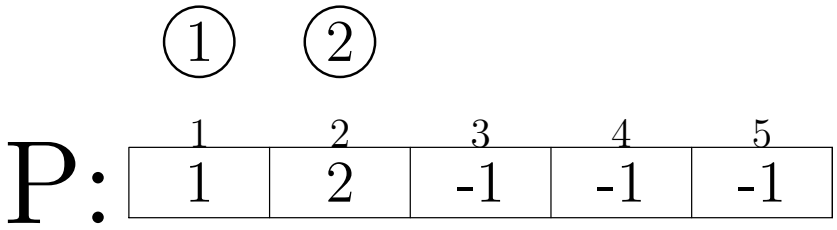
```
MAKE_SET(1);
MAKE_SET(2);
```



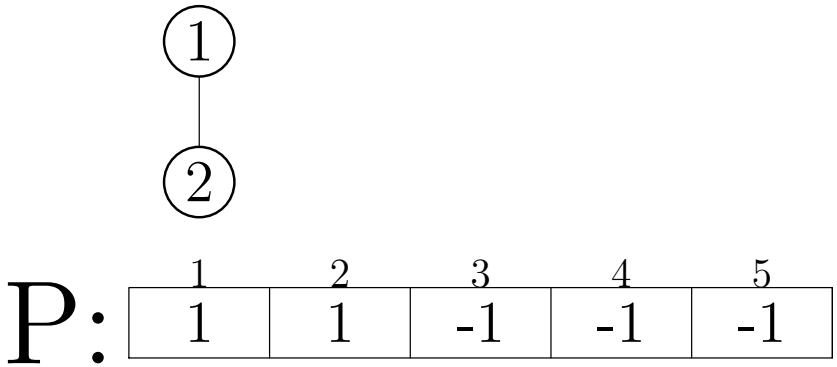

```
UNION(1, 2);
```

Esempio

```
MAKE_SET(1);
MAKE_SET(2);
```

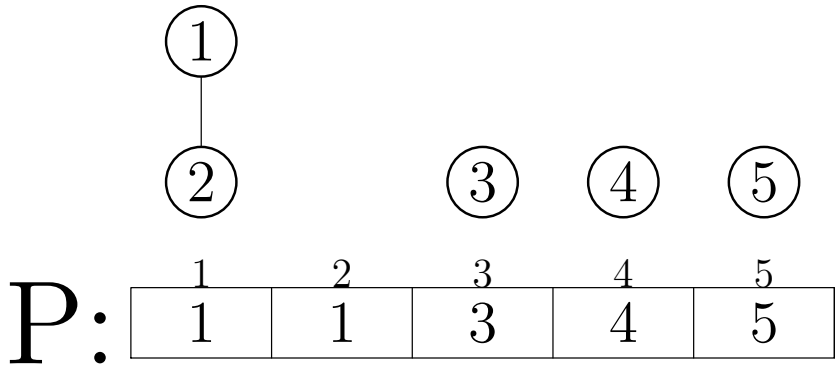



```
UNION(1, 2);
```

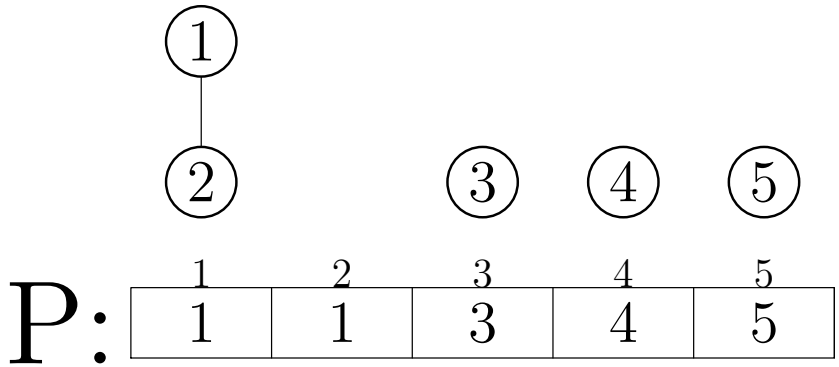


```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

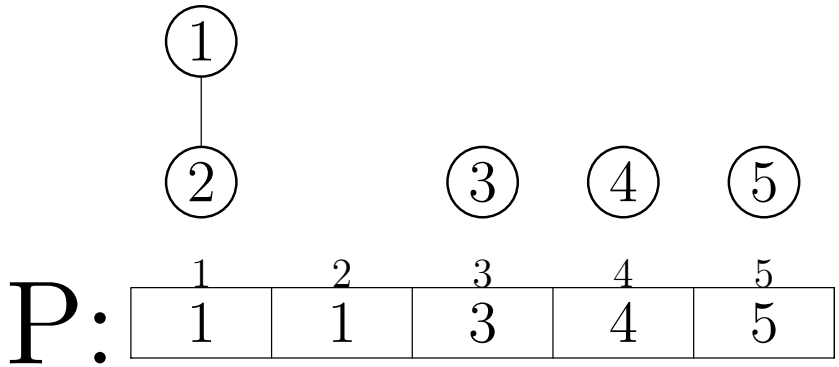


```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

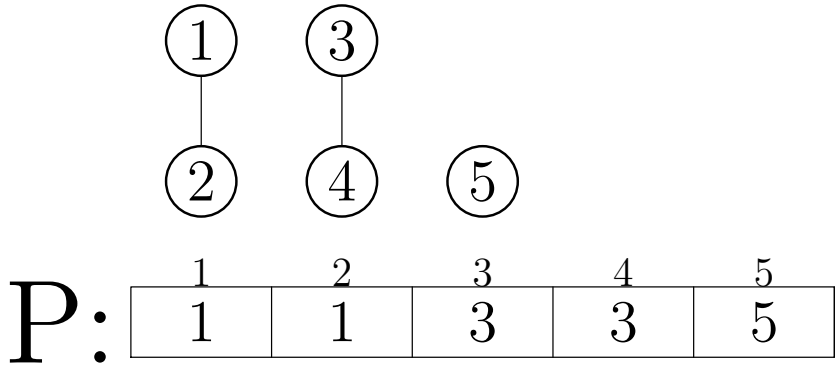



```
UNION(3, 4);
```

```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

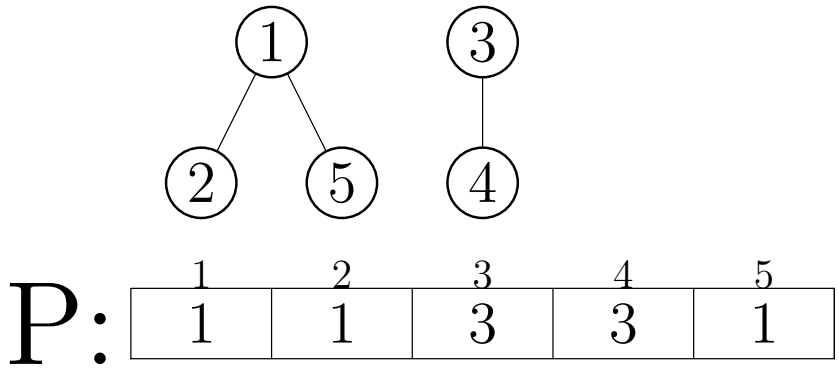



```
UNION(3, 4);
```

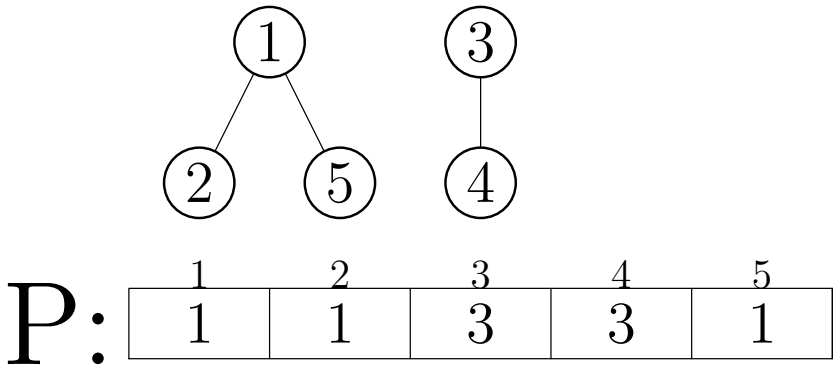


```
UNION(2, 5);
```

```
UNION(2, 5);
```

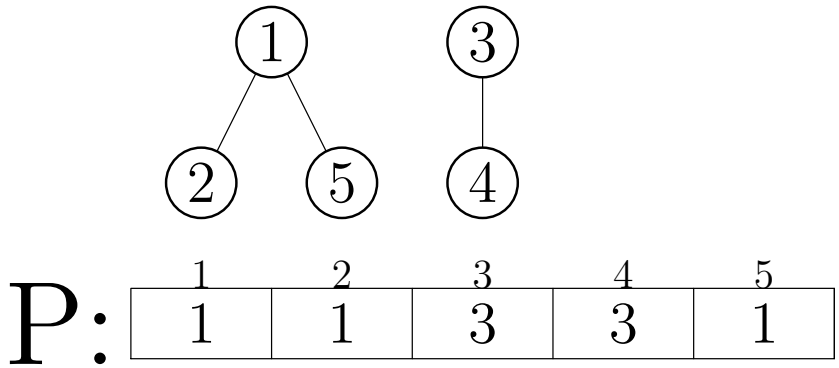


```
UNION(2, 5);
```

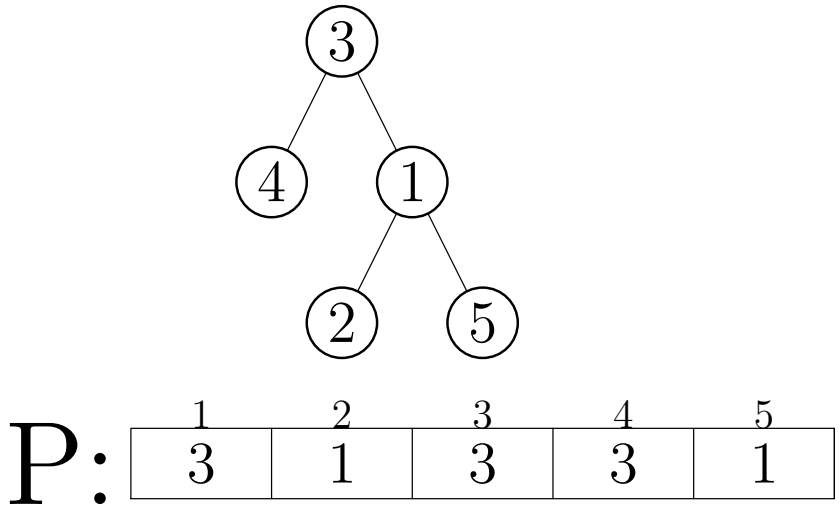



```
UNION(4, 1);
```

```
UNION(2, 5);
```




```
UNION(4, 1);
```



Path compression

Basta modificare il `FIND` nel seguente modo, ricordando sostanzialmente i `FIND` già eseguiti.

```
1 int FIND(x) {
2     if (P[x] == x) return x;
3     else return P[x] = FIND(P[x]);
4 }
```

Esempio

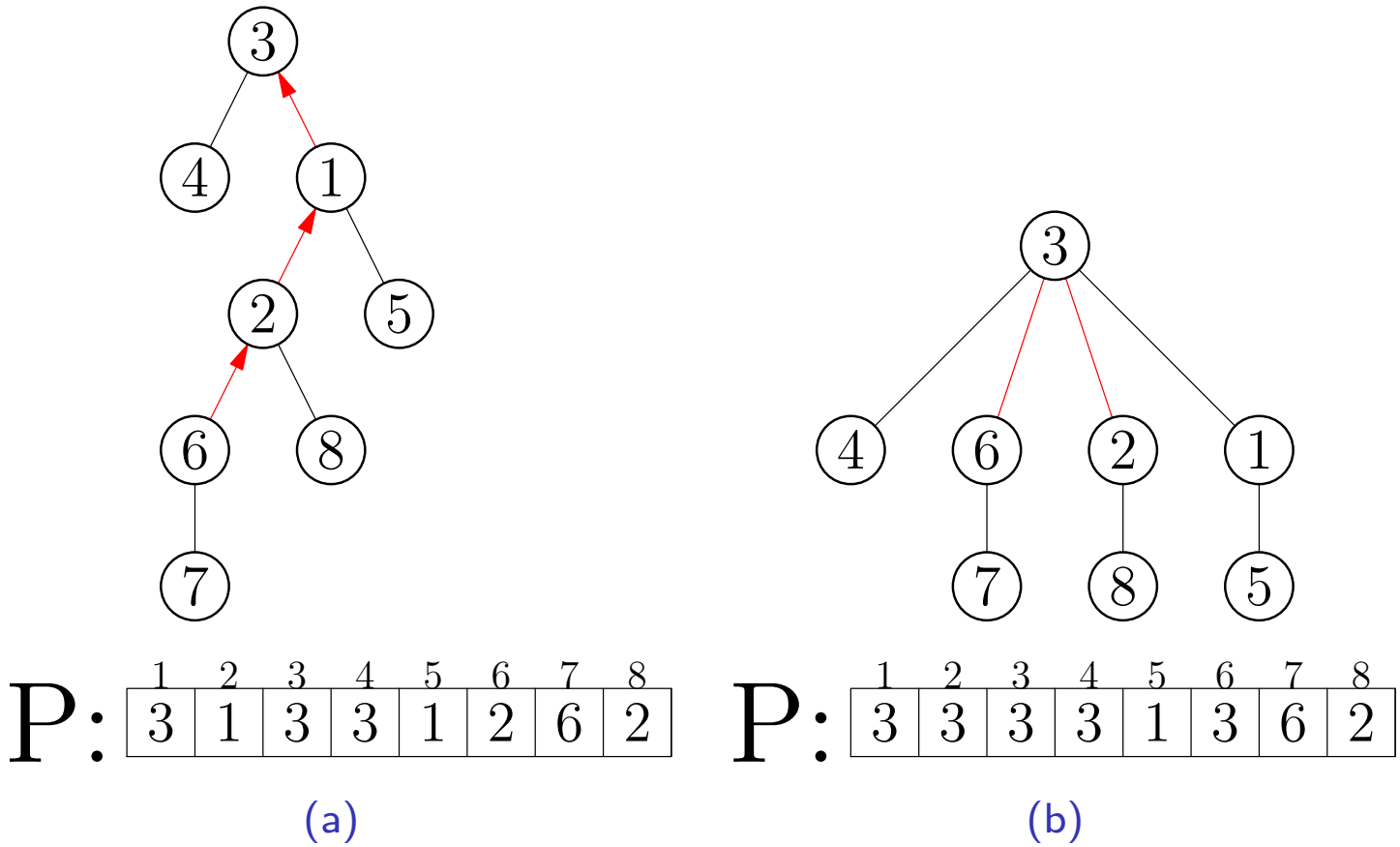


Figura: Ecco cosa succede quando viene chiamata `FIND`(6)

Complessità dell'algoritmo di Union-Find con path compression

Complessità dell'algoritmo di Union-Find con path compression

Teorema

Iniziando con una struttura dati vuota, l'algoritmo esegue qualsiasi sequenza di $M \geq N$ `FIND` *e* $N - 1$ `UNION` *in tempo* $\mathcal{O}(M \log N)$.

Complessità dell'algoritmo di Union-Find con path compression

Teorema

Iniziando con una struttura dati vuota, l'algoritmo esegue qualsiasi sequenza di $M \geq N$ `FIND` *e* $N - 1$ `UNION` *in tempo* $\mathcal{O}(M \log N)$.

Quindi ci va benissimo per l'algoritmo di Kruskal!

Link by rank

Idea: dati due alberi, è meglio appendere il più piccolo sotto il più grosso piuttosto che il contrario

Link by rank

Idea: dati due alberi, è meglio appendere il più piccolo sotto il più grosso piuttosto che il contrario

Come si decide quale dei due alberi è il più *piccolo*?
Due strategie possibili: numero di nodi oppure upper-bound sull'altezza.
A lato pratico sono pressoché equivalenti, vediamo l'implementazione con upper-bound sull'altezza.

```
1  void MAKE_SET(x) {
2      P[x] = x;
3      R[x] = 0;
4  }
5
6  int FIND(x) {
7      if (P[x] == x) return x;
8      else return P[x] = FIND(P[x]);
9  }
10
11 void UNION(x, y) {
12     a = FIND(x);
13     b = FIND(y);
14     if (a != b) {
15         if (R[a] < R[b]) {
16             P[a] = b;  // connetti il piccolo sotto al grande
17         } else {
18             P[b] = a;
19             if (R[a] == R[b])
20                 R[a]++; // l'altezza "cresce"
21         }
22     }
23 }
```

La funzione di Ackermann è definita dalla seguente ricorrenza:

$$
\begin{aligned}
A(1, j) &= 2^j && j \geq 1 \\
A(i, 1) &= A(i-1, 2) && i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) && i, j \geq 2
\end{aligned}
$$

La funzione di Ackermann è definita dalla seguente ricorrenza:

$$
\begin{aligned}
A(1, j) &= 2^j && j \geq 1 \\
A(i, 1) &= A(i-1, 2) && i \geq 2 \\
A(i, j) &= A(i-1, A(i, j-1)) && i, j \geq 2
\end{aligned}
$$

Definiamo $\alpha(m, n) = \min\{i \geq 1 | A(i, \lfloor m/n \rfloor) > \log n\}$.

La funzione di Ackermann è definita dalla seguente ricorrenza:

$$\begin{aligned} A(1,j) &= 2^j && j \geq 1 \\ A(i,1) &= A(i-1,2) && i \geq 2 \\ A(i,j) &= A(i-1,A(i,j-1)) && i,j \geq 2 \end{aligned}$$

Definiamo $\alpha(m,n) = \min\{i \geq 1 | A(i, \lfloor m/n \rfloor) > \log n\}$.

Teorema

Allora l'algoritmo con M operazioni `FIND` e $N-1$ operazioni `UNION` impiega tempo $\mathcal{O}(N + M\alpha(M+N, N))$ ed è asintoticamente ottimo.

Osservazione

Vale $\alpha(m,n) \leq 4$ per ogni valore ragionevole in input.

Implementazione dell'algoritmo di Kruskal

```
1  struct arco_t { int a, b; long long w; };
2  vector<arco_t> A;  // lista degli archi
3
4  long long sol;
5  vector<arco_t> archi_sol;
6
7  sort(A.begin(), A.end(), comp); // ordina gli archi per costo crescente
8  for (int i = 0; i < N; i++) MAKE_SET(i);
9  for (int i = 0; i < M; i++) {
10    if (FIND(A[i].a) != FIND(A[i].b)) {
11       UNION(A[i].a, A[i].b);
12       sol = sol + A[i].w;
13       archi_sol.push_back(A[i]);
14    }
15 }
```

Complessità

Complessità

- Ordinamento degli archi: $\mathcal{O}(M \log M)$

Complessità

- Ordinamento degli archi: $\mathcal{O}(M \log M)$
- $\mathcal{O}(M)$ operazioni `FIND`

Complessità

- Ordinamento degli archi: $\mathcal{O}(M \log M)$
- $\mathcal{O}(M)$ operazioni `FIND`
- $\mathcal{O}(N)$ operazioni `UNION`

Complessità

- Ordinamento degli archi: $\mathcal{O}(M \log M)$
- $\mathcal{O}(M)$ operazioni `FIND`
- $\mathcal{O}(N)$ operazioni `UNION`

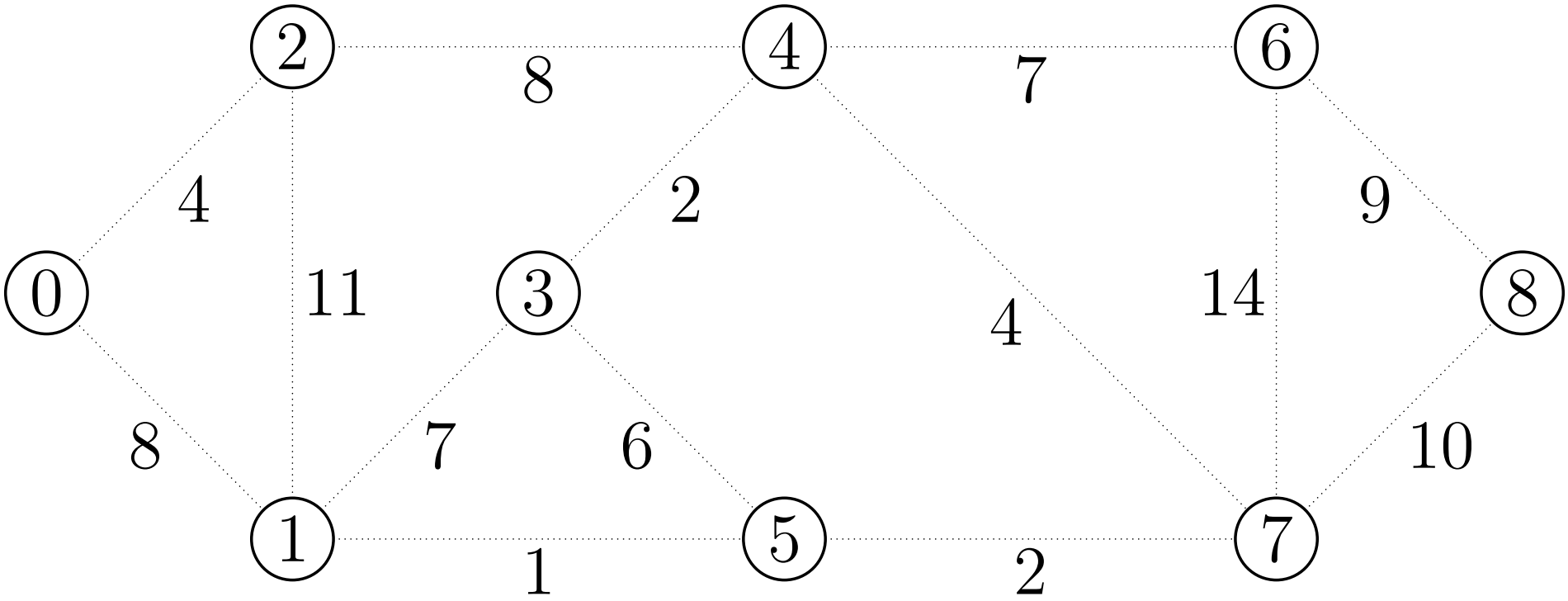
Totale: $\mathcal{O}(M \log M)$

Algoritmo di Prim

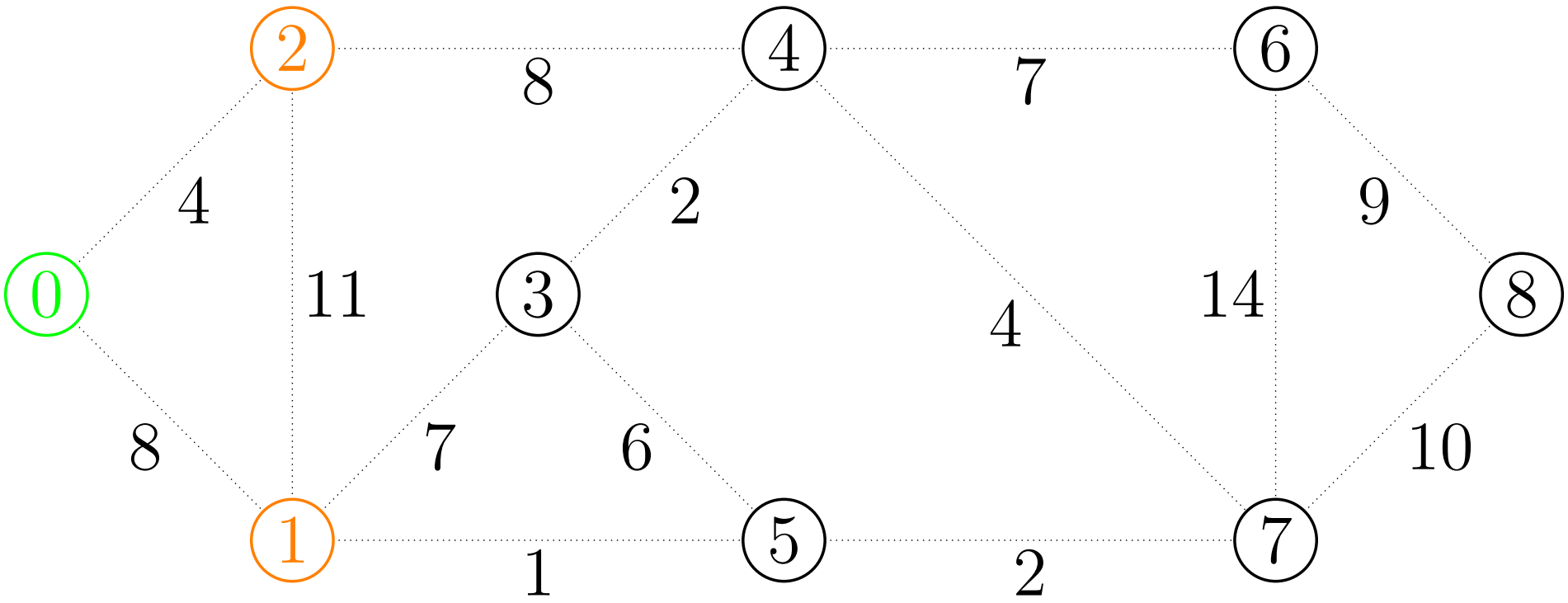
Algoritmo di Prim

- Tengo un insieme S degli archi scelti, inizialmente vuoto, e tengo un insieme R dei nodi raggiunti, che inizialmente contiene un nodo arbitrario, per esempio 0.
- A ogni passo aggiungo a R il nodo non ancora raggiunto che sta più vicino agli elementi di R, e continuo così fino a quando R contiene tutti i nodi del grafo.

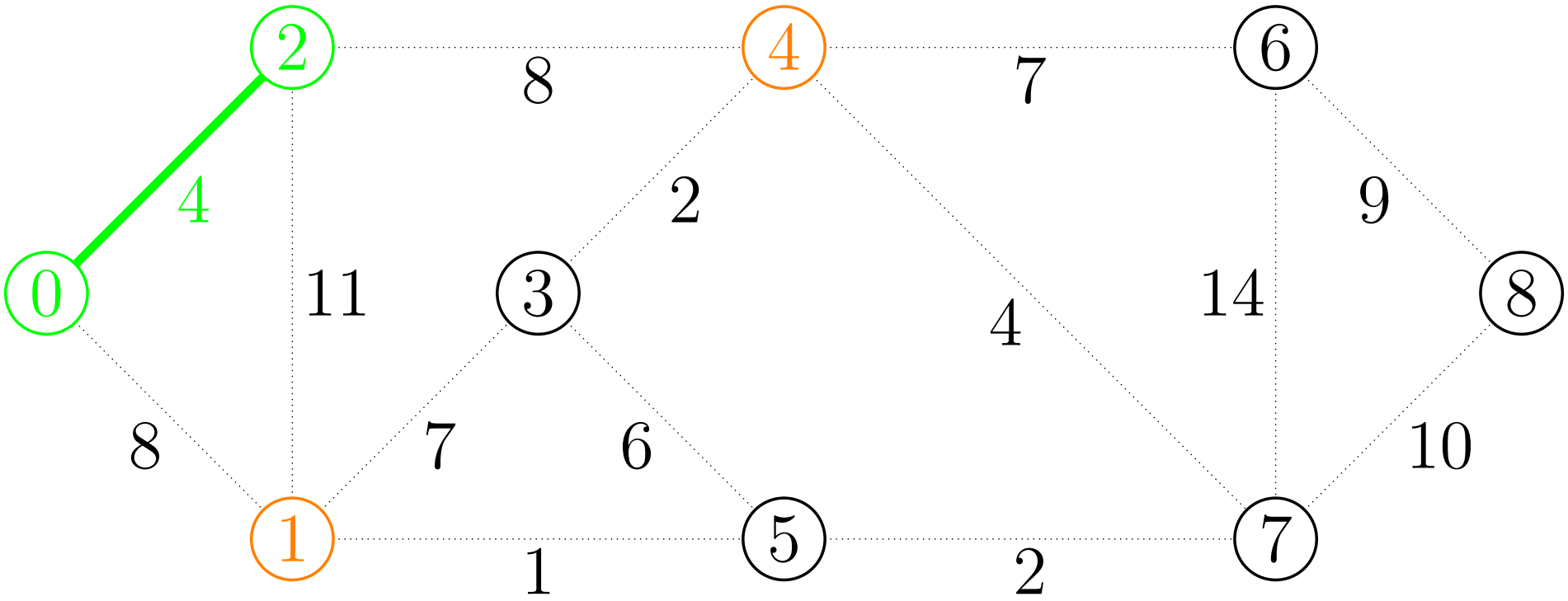
Vediamo come funziona l'algoritmo di Prim sul grafo di esempio.



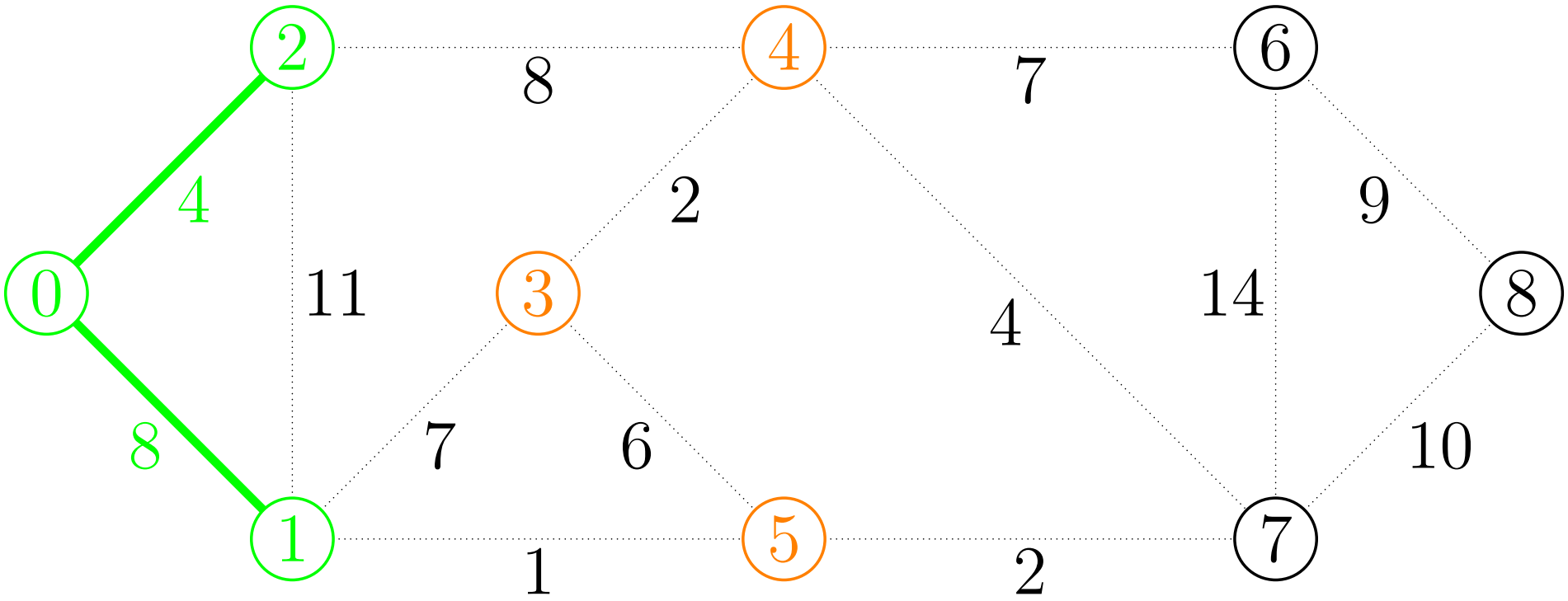
Coloriamo di verde i nodi già raggiunti e di arancione i candidati ad essere il nuovo nodo inserito; coloriamo di verde gli archi presi.


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

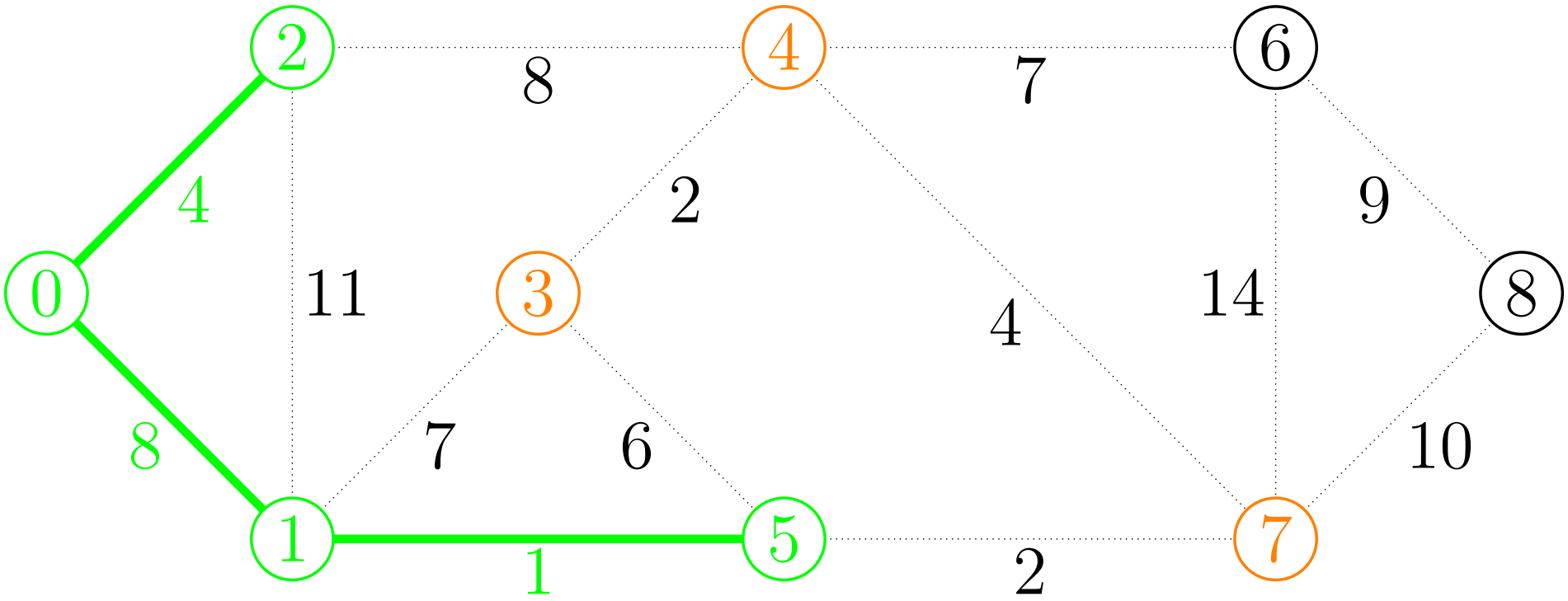
Aggiungo il nodo 2 che è a distanza 4 dai nodi presi, e l'arco $(0, 2)$.



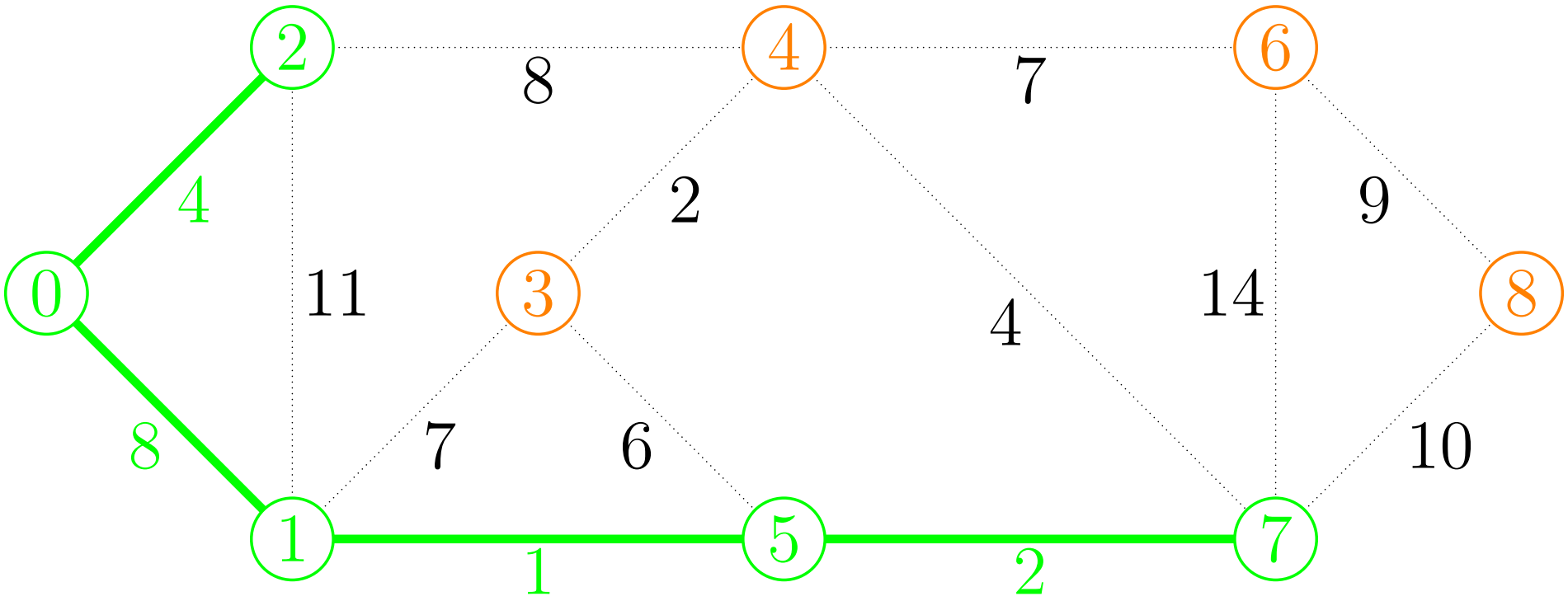
Aggiungo il nodo 1 che è a distanza 8 dai nodi presi, e l'arco $(0, 1)$.



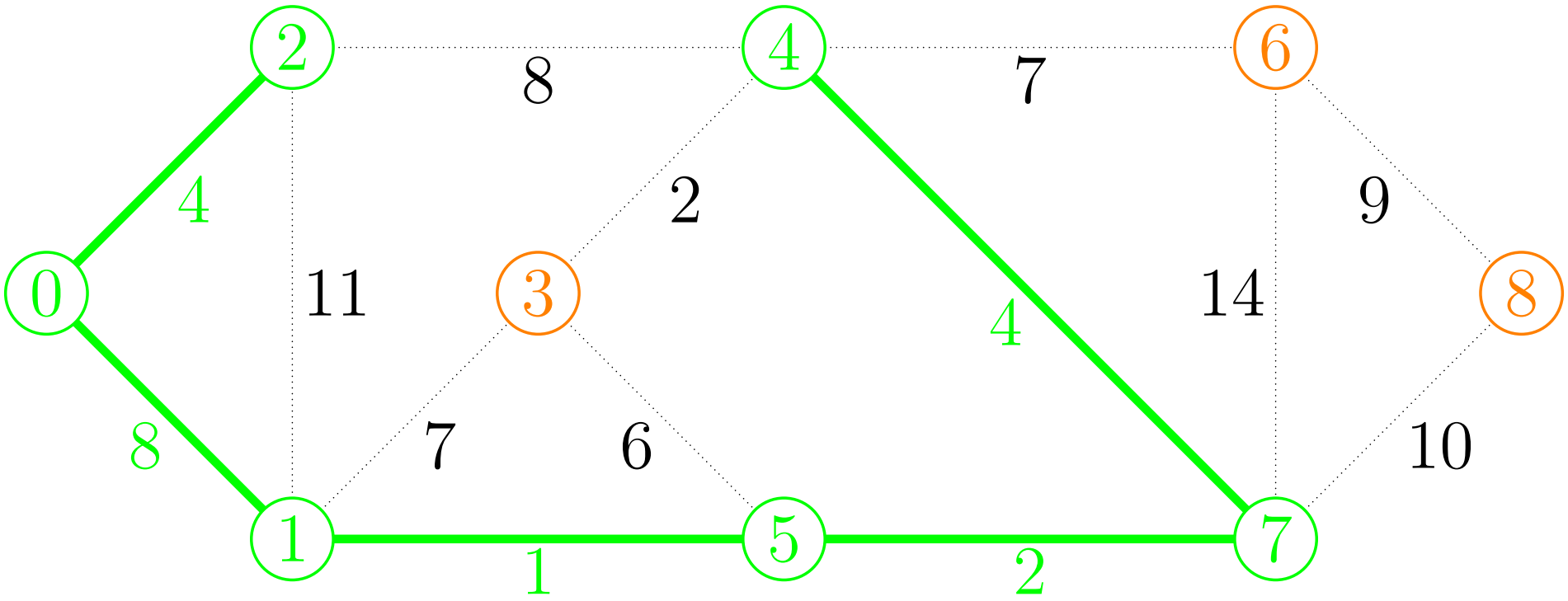
Aggiungo il nodo 5 che è a distanza 1 dai nodi presi, e l'arco $(1, 5)$.



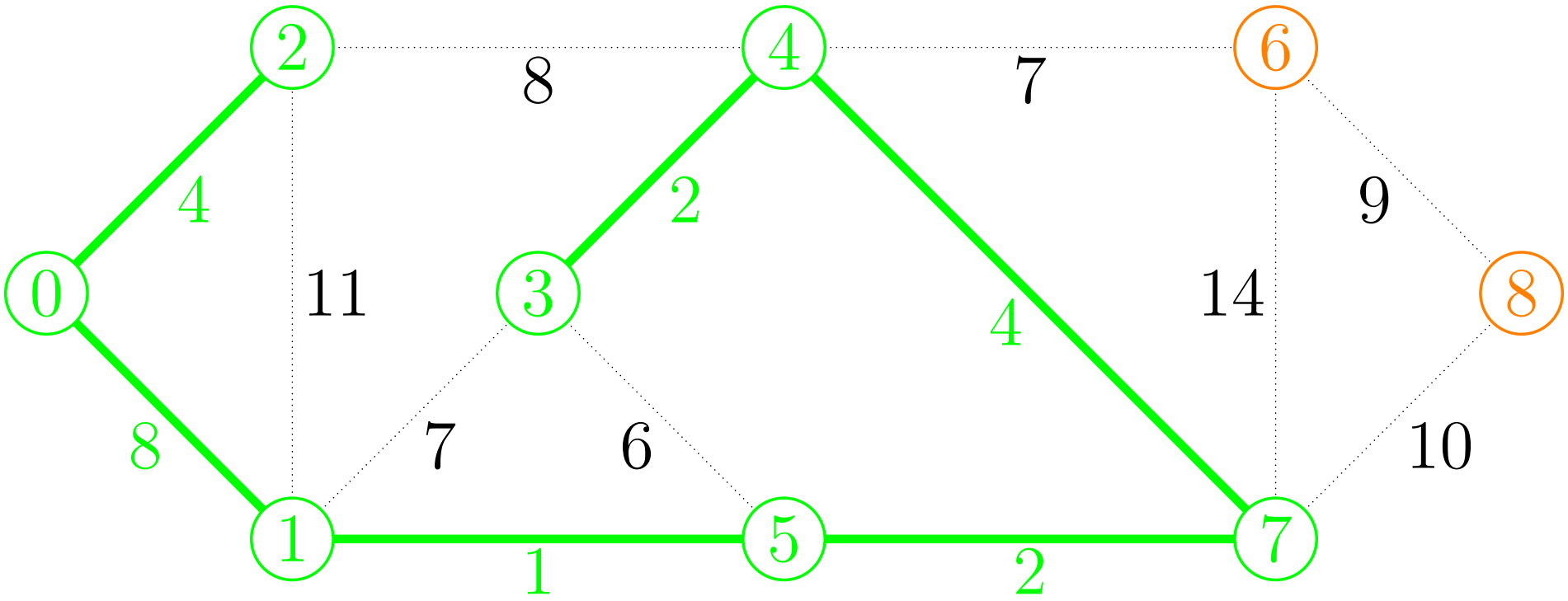
Aggiungo il nodo 7 che è a distanza 2 dai nodi presi, e l'arco $(5, 7)$.



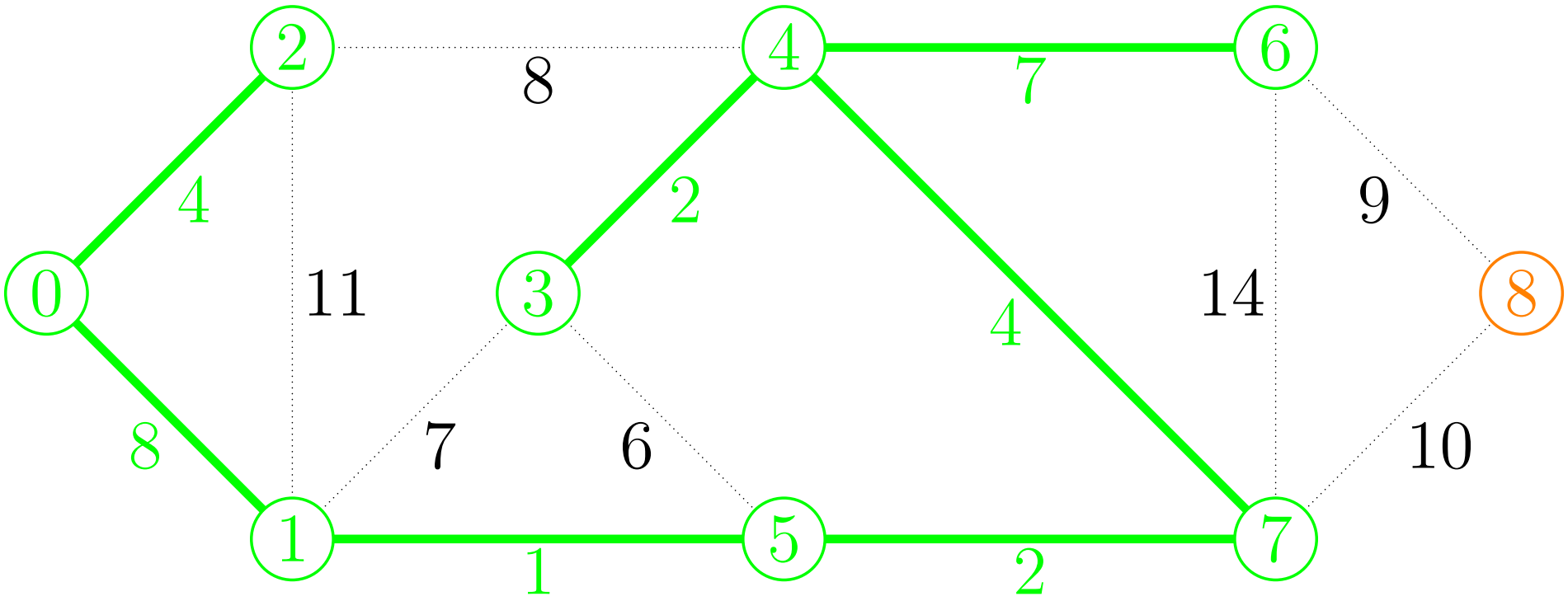
Aggiungo il nodo 4 che è a distanza 4 dai nodi presi, e l'arco $(4, 7)$.



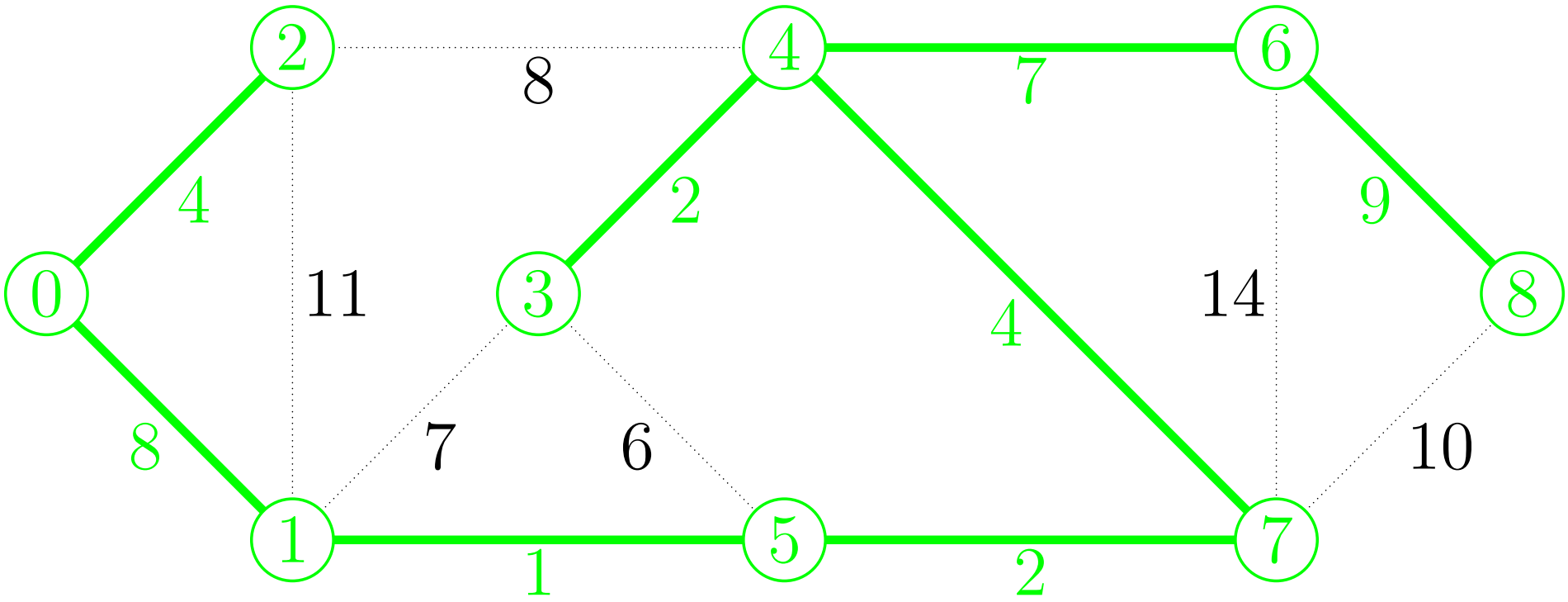
Aggiungo il nodo 3 che è a distanza 2 dai nodi presi, e l'arco $(3, 4)$.



Aggiungo il nodo 6 che è a distanza 7 dai nodi presi, e l'arco $(4, 6)$.



Infine aggiungo il nodo 8 che è a distanza 9 dai nodi presi, e l'arco $(6, 8)$.



Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

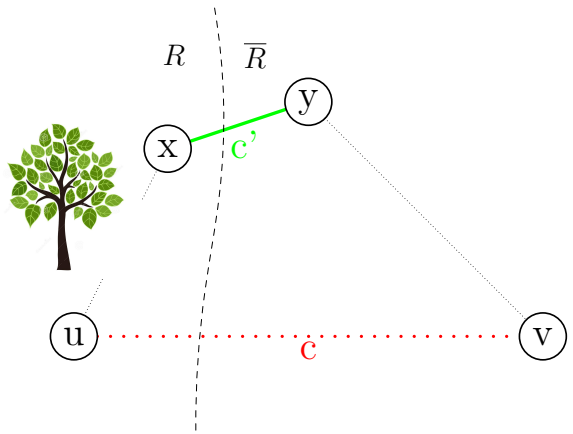
- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.

Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).

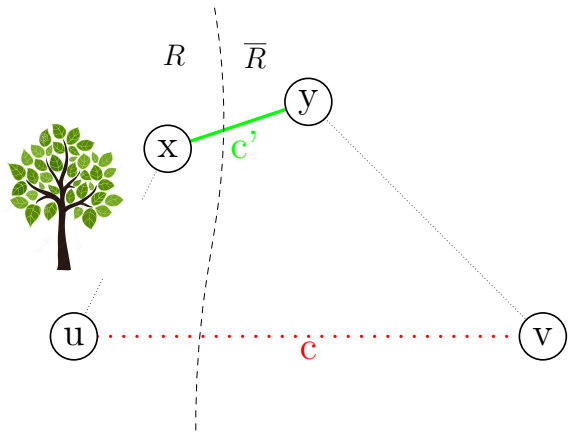
Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v, il quale include un arco (x, y) (di costo c') con $x \in R$ e $y \in \overline{R}$.



Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

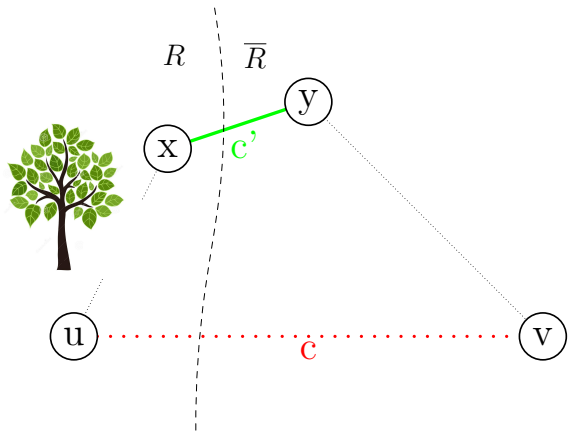
- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v, il quale include un arco (x, y) (di costo c') con $x \in R$ e $y \in \overline{R}$.



- Visto che l'algoritmo ha scelto (u, v) prima di (x, y), allora $c \leq c'$.

Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

- L'algoritmo sceglie bene alcuni archi, ma quando sceglie (u, v) (di peso c) non esiste più alcuna soluzione valida.
- Quindi esiste una soluzione S che contiene tutti gli archi scelti fino a (u, v), ma non (u, v).
- Visto che S non contiene (u, v), deve esistere un altro percorso che collega u e v, il quale include un arco (x, y) (di costo c') con $x \in R$ e $y \in \overline{R}$.



- Visto che l'algoritmo ha scelto (u, v) prima di (x, y), allora $c \leq c'$.
- Però scambiando in S (x, y) con (u, v) otteniamo un MST di valore minore o uguale di S. Assurdo.

Dettagli implementativi

- I nodi arancioni (quelli che devo considerare di inserire) li mettiamo in una priority queue di coppie del tipo (distanza dall'insieme R, nodo nuovo)

Dettagli implementativi

- I nodi arancioni (quelli che devo considerare di inserire) li mettiamo in una priority queue di coppie del tipo (distanza dall'insieme R, nodo nuovo)
- Ad ogni passaggio, prendiamo il primo elemento della priority queue *controllando di non averlo già messo in R precedentemente*

Dettagli implementativi

- I nodi arancioni (quelli che devo considerare di inserire) li mettiamo in una priority queue di coppie del tipo (distanza dall'insieme R, nodo nuovo)
- Ad ogni passaggio, prendiamo il primo elemento della priority queue *controllando di non averlo già messo in R precedentemente*
- Aggiungiamo tale elemento v ad R e, per ogni arco (v, w) di costo c uscente da v, aggiungiamo la coppia (c, w) alla priority queue.

```
int new_nodo = 0;
int n_raggiunti = 1;

using pq_item_t = pair<long long int, pair<int, int>>;
priority_queue<pq_item_t, vector<pq_item_t>, greater<pq_item_t>> pq;

while (n_raggiunti < N) {
   preso[new_nodo] = true;
   for (size_t i = 0; i < G[new_nodo].size(); i++) {
      int arrivo = G[new_nodo][i].first;
      int peso = G[new_nodo][i].second;
      if (!preso[arrivo]) {
         pq.push(make_pair(peso, make_pair(new_nodo, arrivo)));
      }
   }

   // Scelta del nuovo nodo:
   pair<long long int, pair<int, int>> x;
   do {
      x = pq.top();
      pq.pop();
   } while (preso[x.second.second]);

   archi_sol.push_back(x.second);
   sol += x.first;
   n_raggiunti++;
   new_nodo = x.second.second;
}
```

Complessità

Complessità

Per N volte devo estrarre il più vicino (tempo $\mathcal{O}(\log |\text{CODA}|)$ usando una priority queue), e devo inserire al massimo M elementi nella priority queue. Quindi la complessità è $\mathcal{O}(M \log M)$.

Ordinamento topologico

Ordinamento topologico

Definizione

Un grafo diretto si dice *DAG* (Directed Acyclic Graph) se non contiene cicli.

Ordinamento topologico

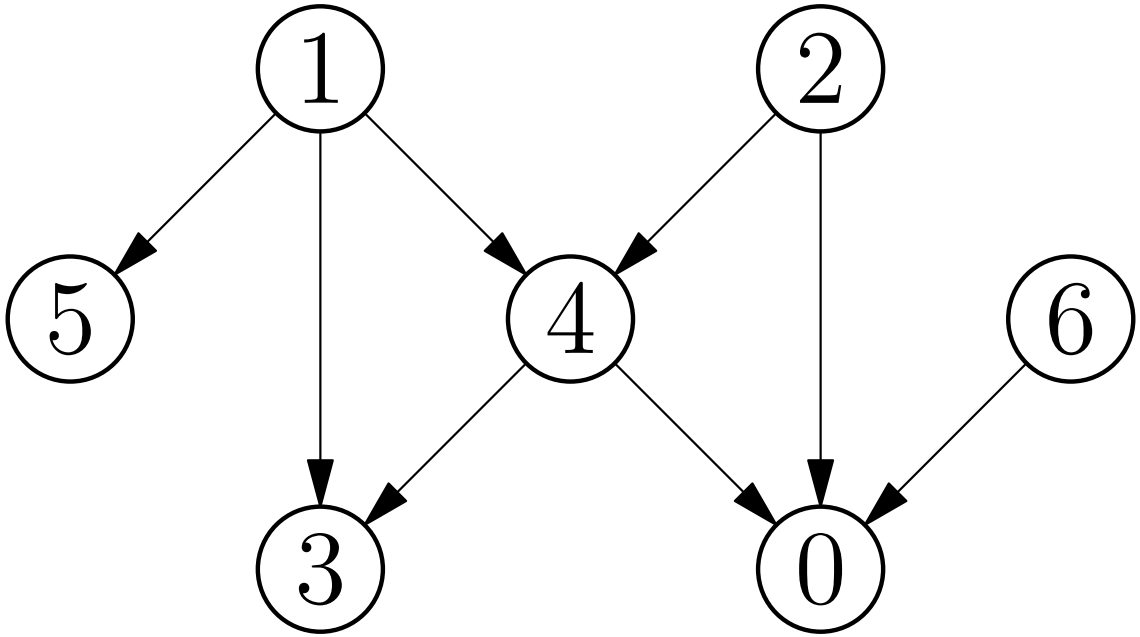
Definizione

Un grafo diretto si dice *DAG* (Directed Acyclic Graph) se non contiene cicli.

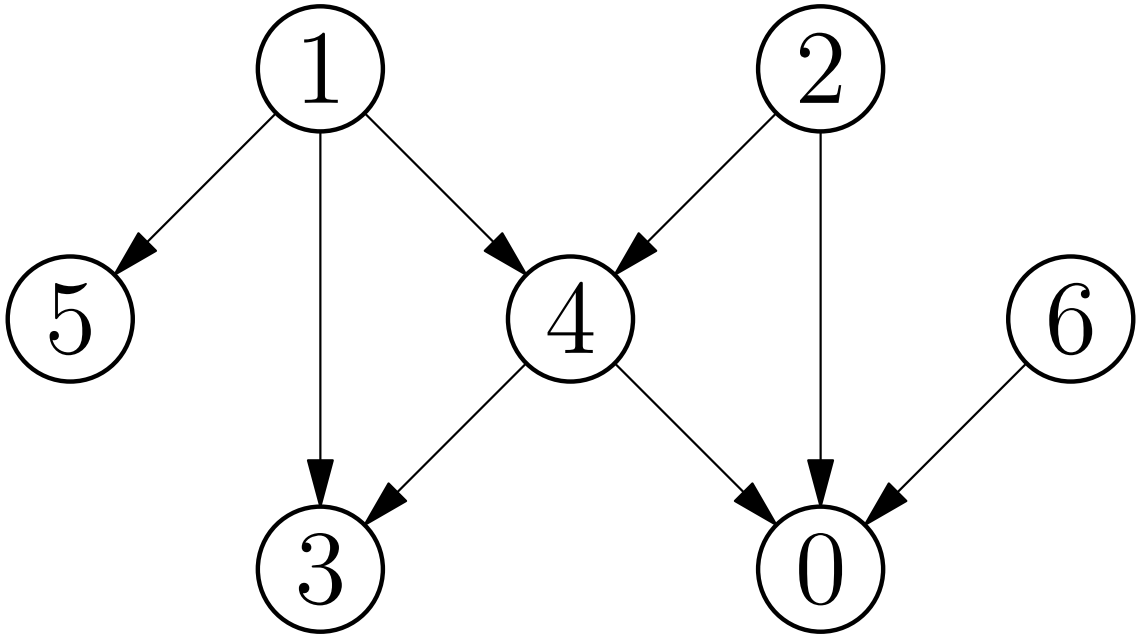
Definizione

Un ordinamento topologico è un ordinamento dei nodi, tale che, per ogni arco $u \to v$, u precede v nell'ordine.

Esempio di ordinamento topologico

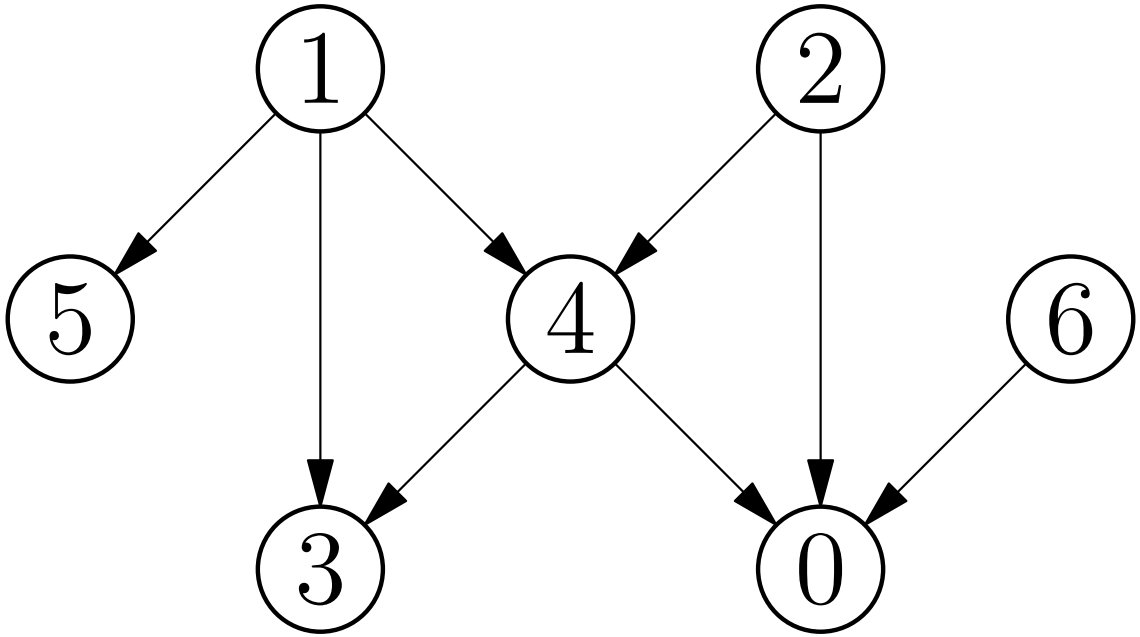


Esempio di ordinamento topologico



Un ordinamento dei nodi è il seguente: $\{1, 2, 4, 3, 5, 6, 0\}$.

Esempio di ordinamento topologico



Un ordinamento dei nodi è il seguente: $\{1, 2, 4, 3, 5, 6, 0\}$.
Non è l'unico! Ne trovate un altro?

Algoritmo

Facciamo una DFS e ordiniamo in base al *tempo di chiusura* decrescente dei nodi.
Ogni volta che finisco di visitare i vicini di un nodo, il tempo avanza di 1.

Algoritmo

Facciamo una DFS e ordiniamo in base al *tempo di chiusura* decrescente dei nodi.
Ogni volta che finisco di visitare i vicini di un nodo, il tempo avanza di 1.

```
1  int t = 0;                          // variabile globale tempo
2  vector<int> tempo_chiusura;         // inizializzato a -1
3  vector<int> ordinamento;            // ordinamento topologico INVERSO!
4
5  for (int i = 0; i < N; i++) {
6      if (tempo_chiusura[i] == -1)
7          dfs(i);
8  }
9
10 void dfs(int x) {
11     for (auto y : grafo[x]) {
12         if (tempo_chiusura[y] == -1) {
13             dfs(y);
14         }
15     }
16     // ho visitato tutti i vicini di x: inserisco x nell'ordinamento
17     t++;
18     tempo_chiusura[x] = t;
19     ordinamento.push_back(x);
20 }
```

Perché funziona?

Supponiamo per assurdo che l'algoritmo non sia corretto.

- Se sbaglia, allora mette a prima di b nell'ordinamento, quando però è presente un arco $b \to a$.

Perché funziona?

Supponiamo per assurdo che l'algoritmo non sia corretto.

- Se sbaglia, allora mette a prima di b nell'ordinamento, quando però è presente un arco $b \to a$.
- Se mette a prima di b, allora a viene chiuso **dopo** di b.

Perché funziona?

Supponiamo per assurdo che l'algoritmo non sia corretto.

- Se sbaglia, allora mette a prima di b nell'ordinamento, quando però è presente un arco $b \to a$.
- Se mette a prima di b, allora a viene chiuso **dopo** di b.
- Durante `dfs`(b) viene visitato l'arco $b \to a$, chiamando `dfs`(a). Quindi `dfs`(a) deve ritornare *prima* di `dfs`(b), chiudendo a **prima** di b. Assurdo.

Quando è utile?

Avere un ordinamento topologico dei nodi rende possibile fare programmazioni dinamiche su DAG! Per esempio:

- Cammini minimi in tempo lineare!
- Cammino di lunghezza massima
- Numero di cammini

In più ci sono anche molti altri utilizzi più *ingegneristici*.

Esercizi

Esercizio

Minimum Spanning Tree

- *training – mst*

Esercizio

Ordinamento topologico

- *cses – Course Schedule*
- *cses – Longest Flight Route*
- *cses – Game Routes*
- *training – picarats*