Minimum Spanning Tree, Union-Find e Ordinamento Topologico

Alice Cortinovis e Edoardo Morassutto

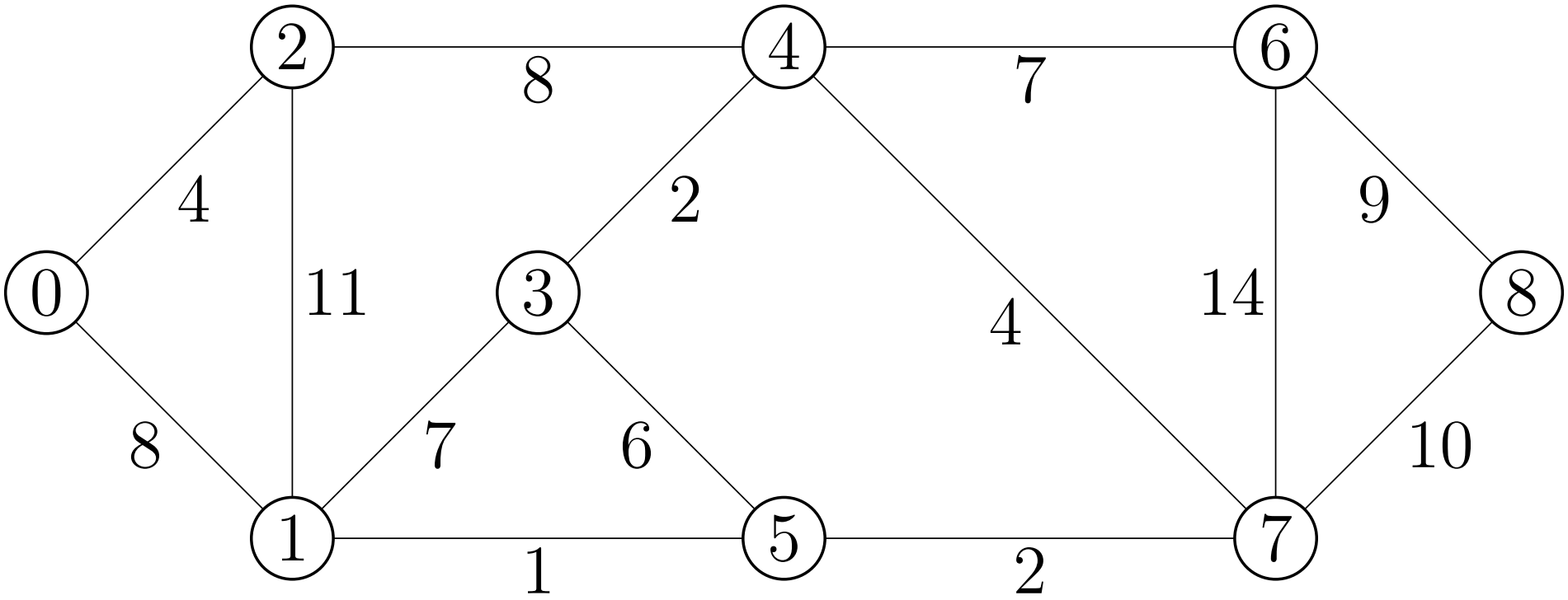
Online, 10 aprile 2021

Minimum Spanning Tree

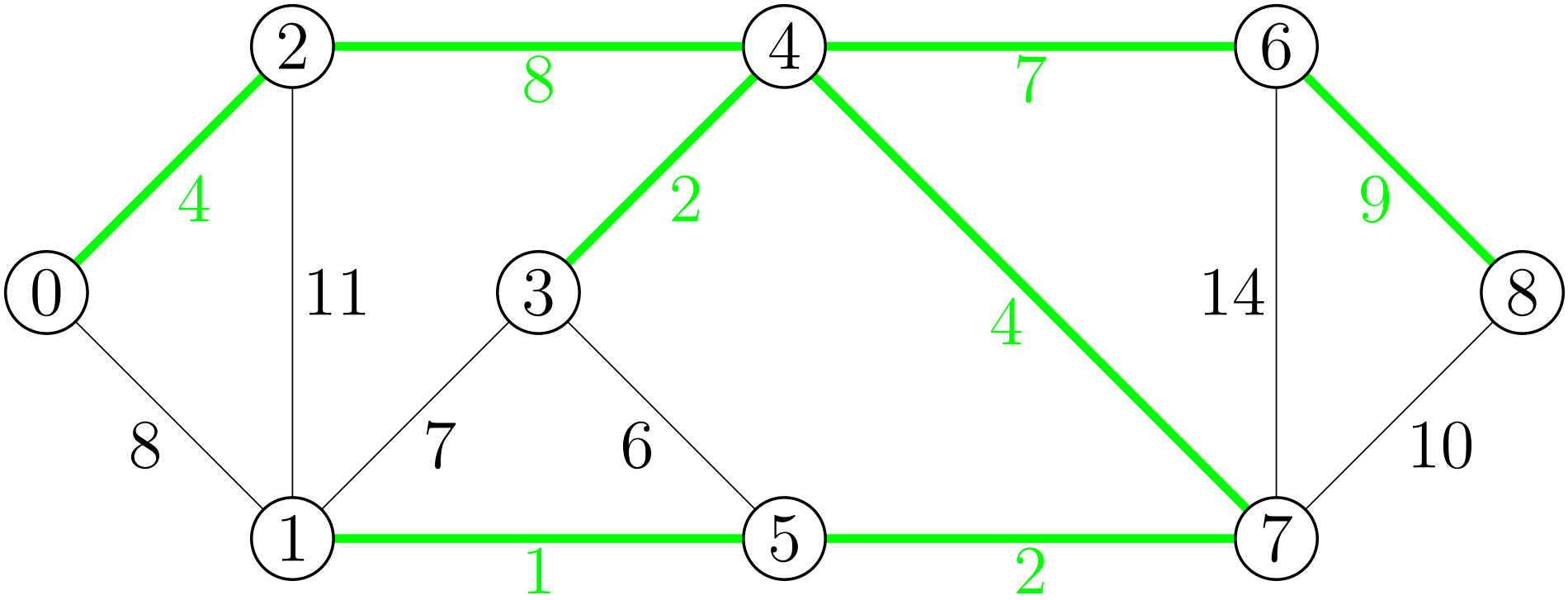
Problema: Dato un grafo non diretto $G = (V, E)$ con N vertici e M archi, trovare l'albero ricoprente di minimo costo totale.

Minimum Spanning Tree

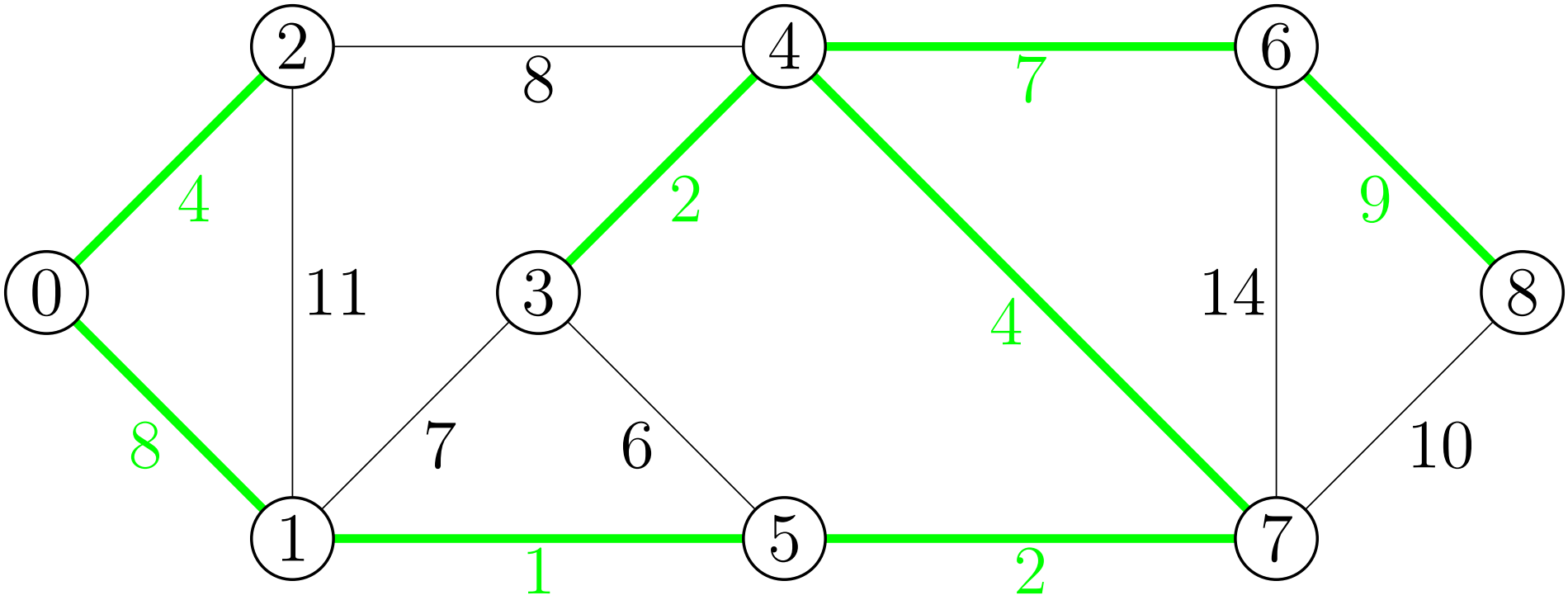
Problema: Dato un grafo non diretto $G = (V, E)$ con N vertici e M archi, trovare l'albero ricoprente di minimo costo totale.



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

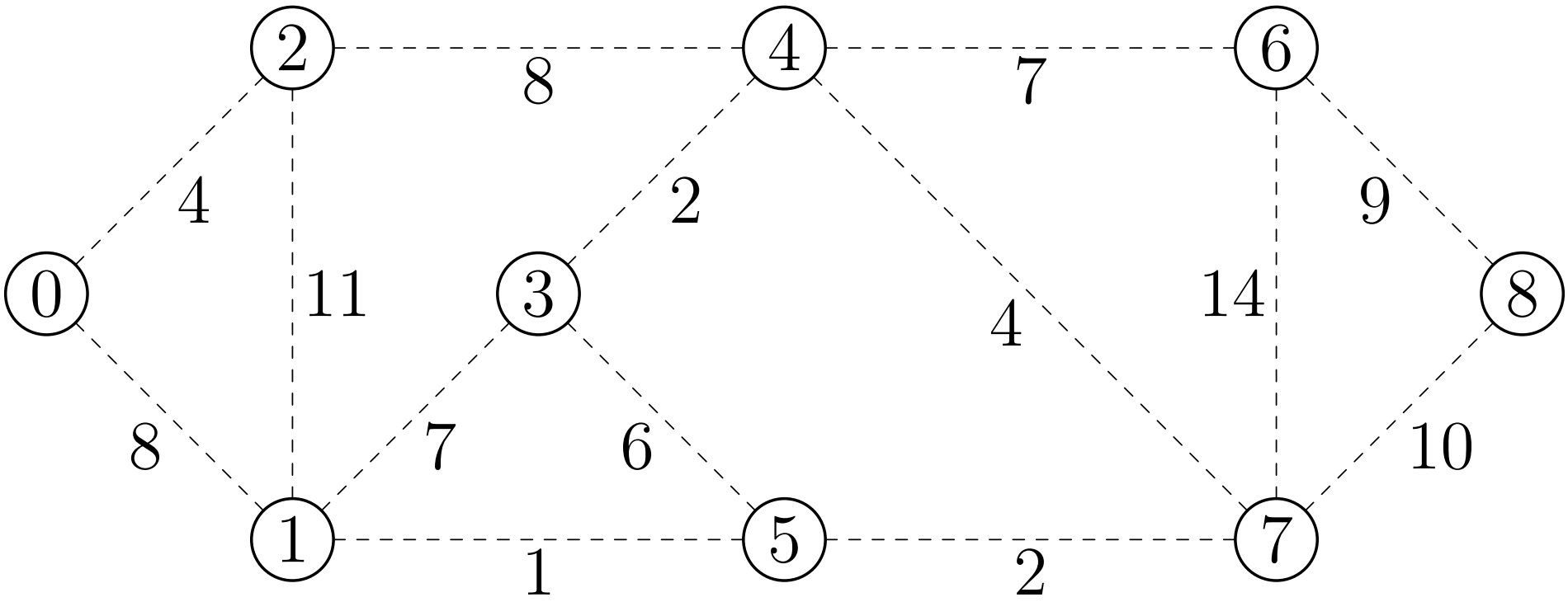
- $S := \emptyset$
- Ordina gli archi per peso crescente
- Ora guarda ogni arco in tale ordine: se è *utile* lo aggiungi a S, altrimenti ti dimentichi di lui per sempre

Algoritmo di Kruskal

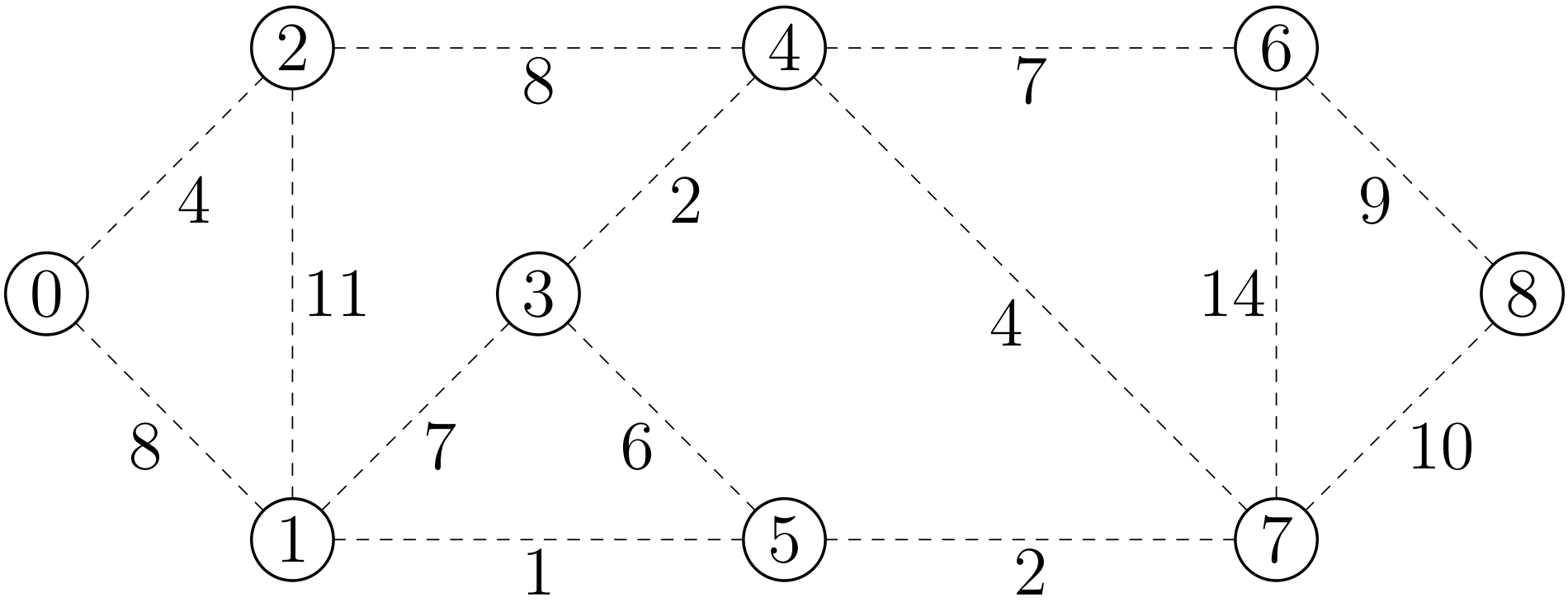
- $S := \emptyset$
- Ordina gli archi per peso crescente
- Ora guarda ogni arco in tale ordine: se è *utile* lo aggiungi a S, altrimenti ti dimentichi di lui per sempre

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

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

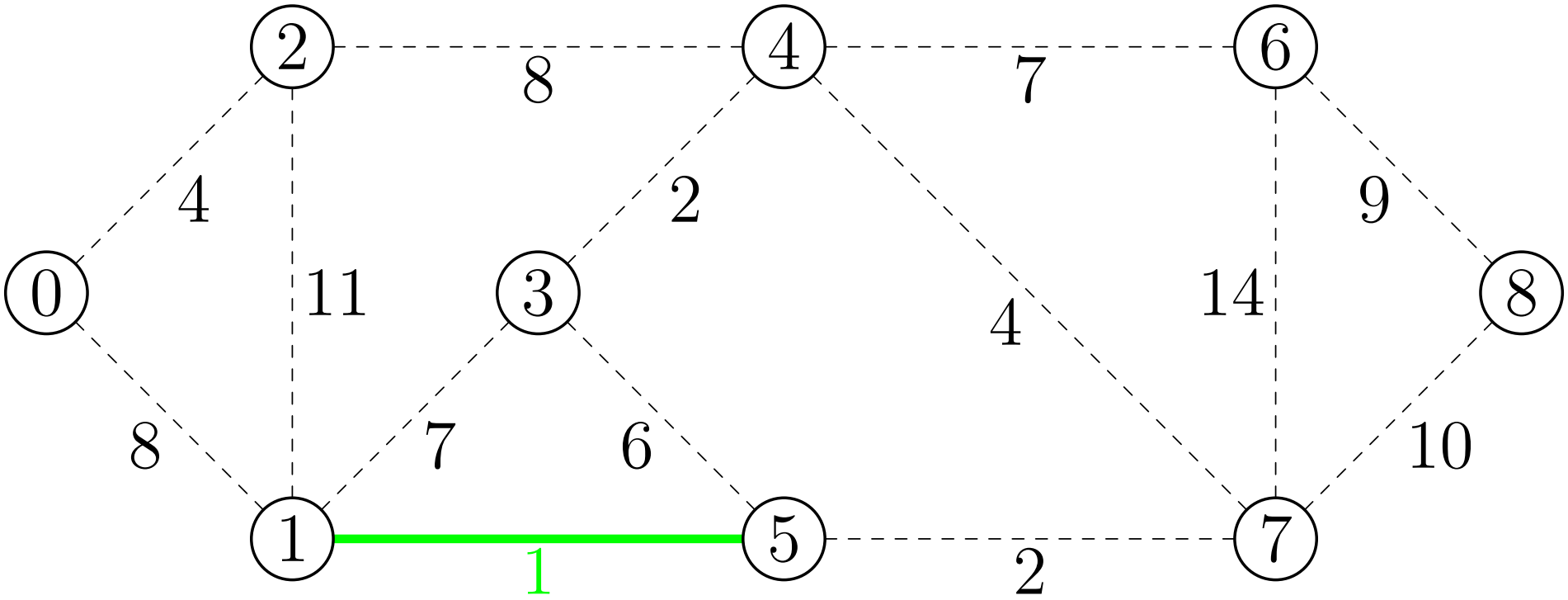


Vediamo cosa succede sul grafo di esempio visto all'inizio.

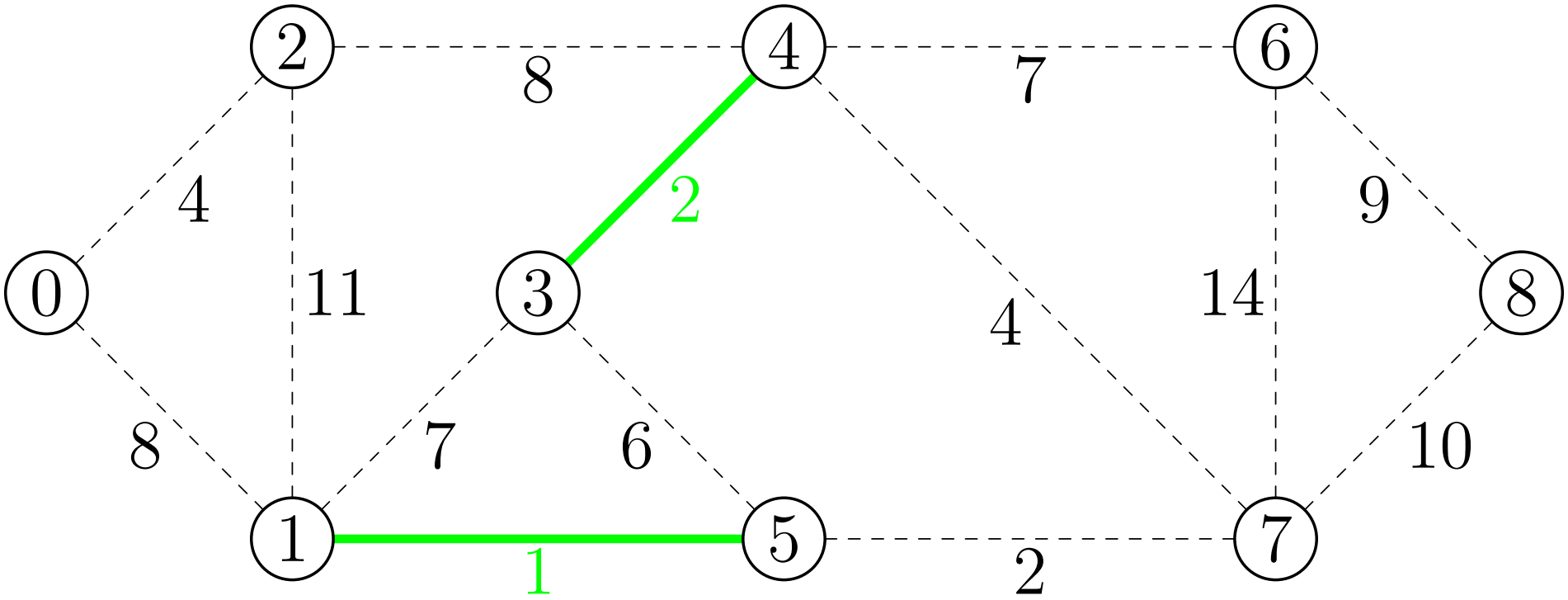


Gli archi ordinati sono:
$(1,5),(3,4),(5,7),(4,7),(0,2),(5,3),(1,3),$
$(4,6),(2,4),(0,1),(6,8),(7,8),(1,2),(6,7)$

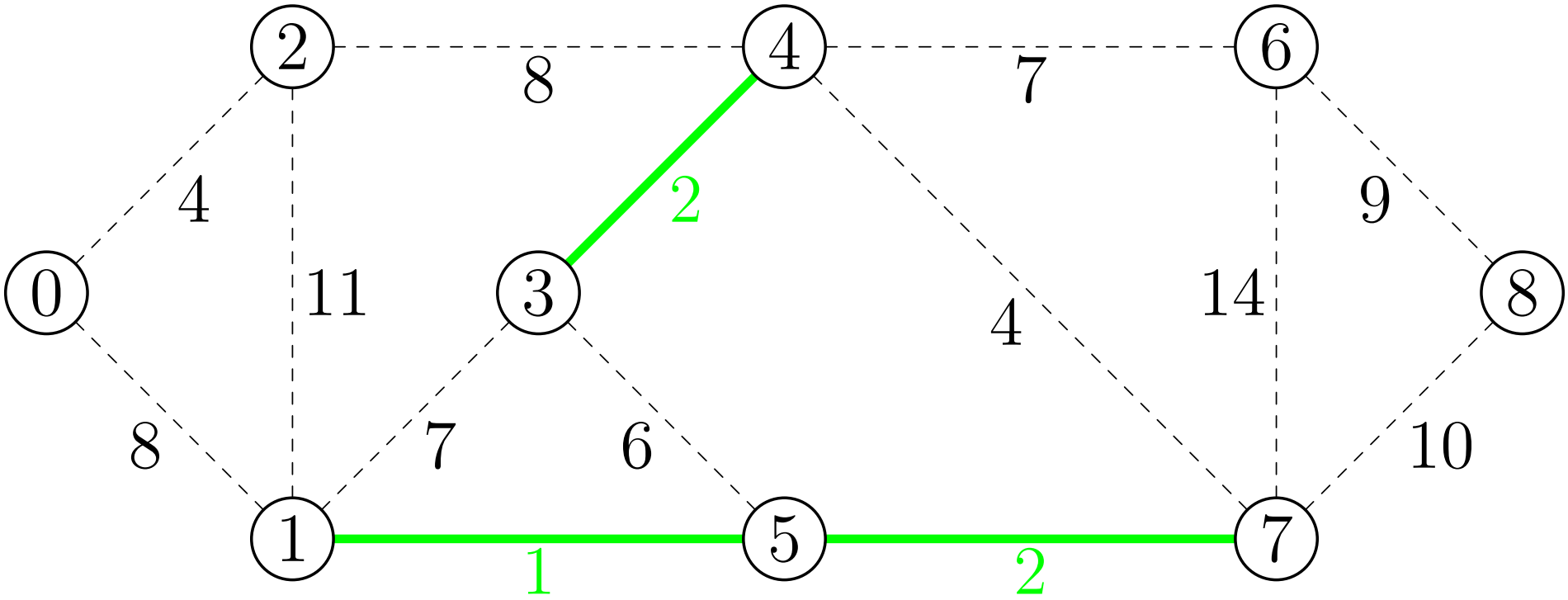
Arco $(1, 5)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



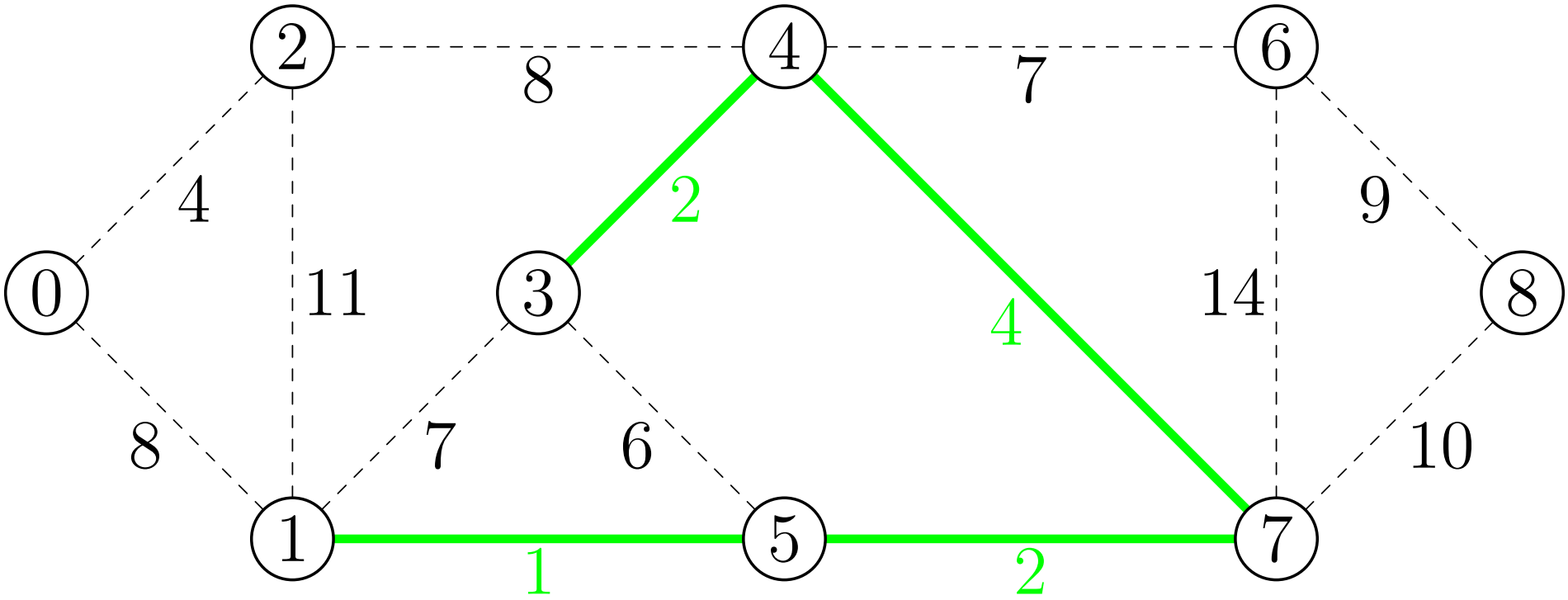
Arco $(3, 4)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



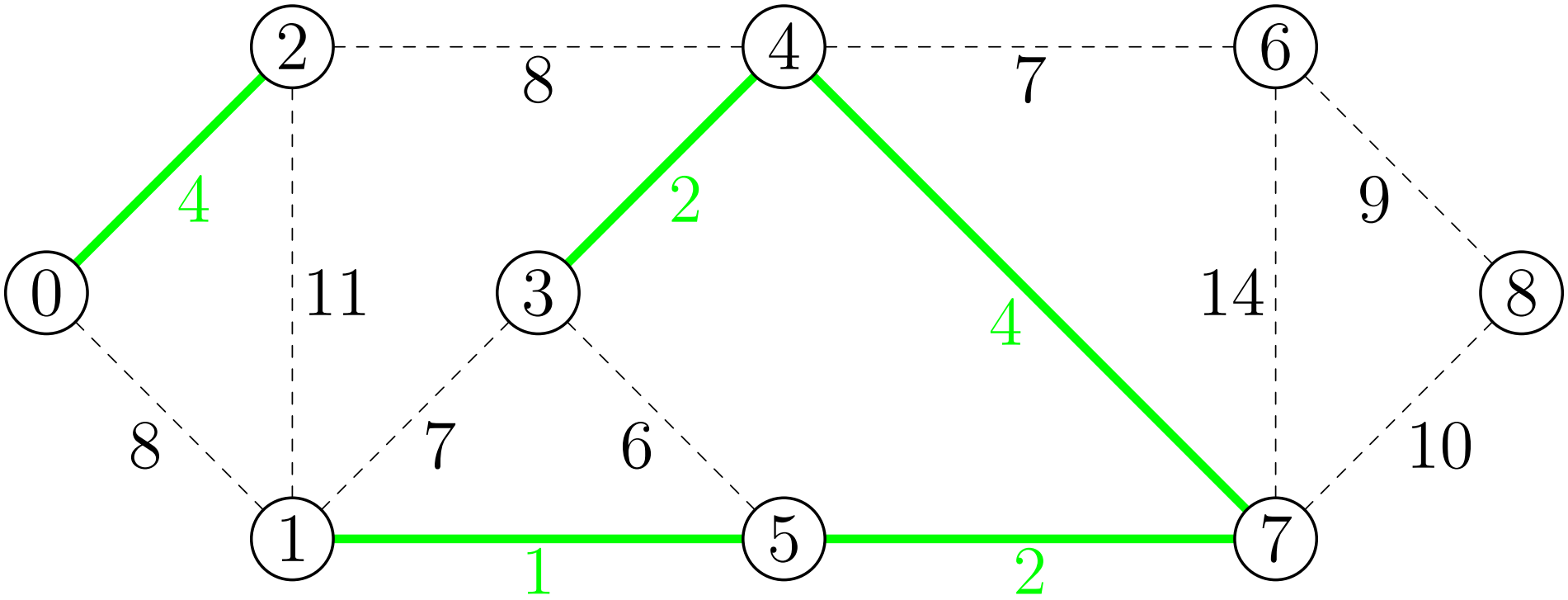
Arco $(5, 7)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



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

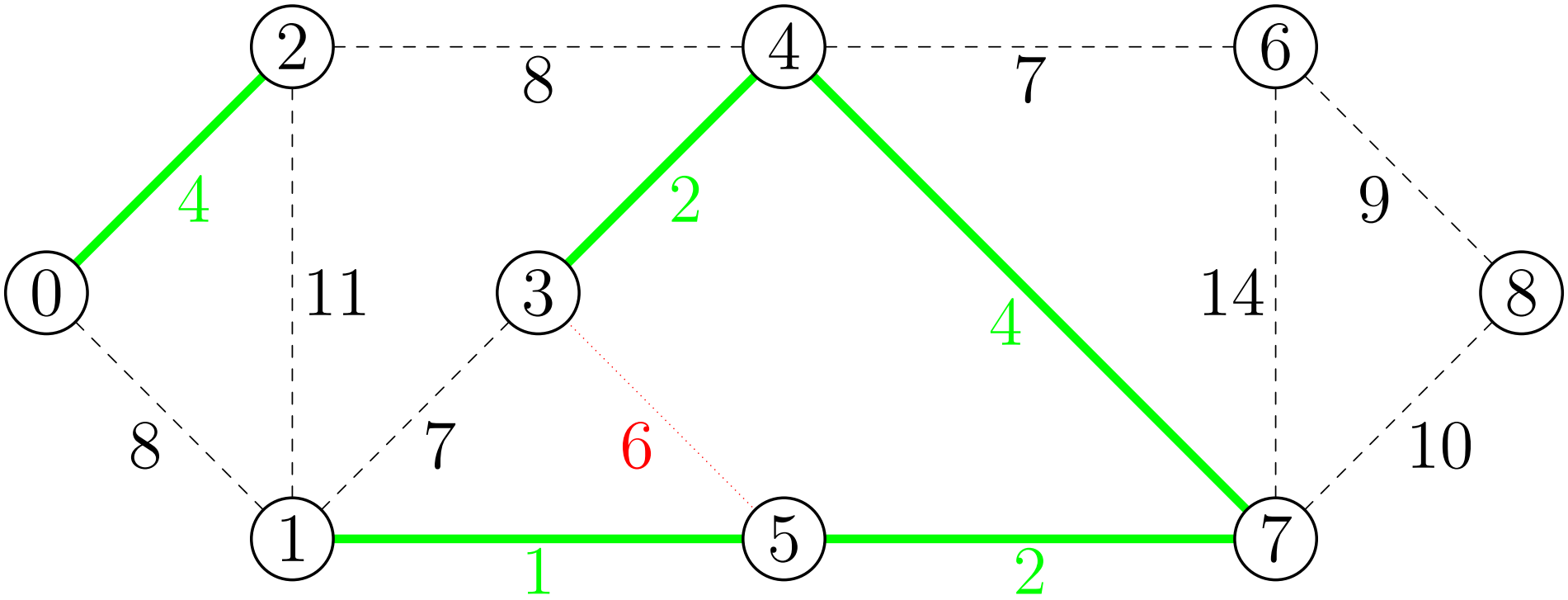


Arco $(0, 2)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



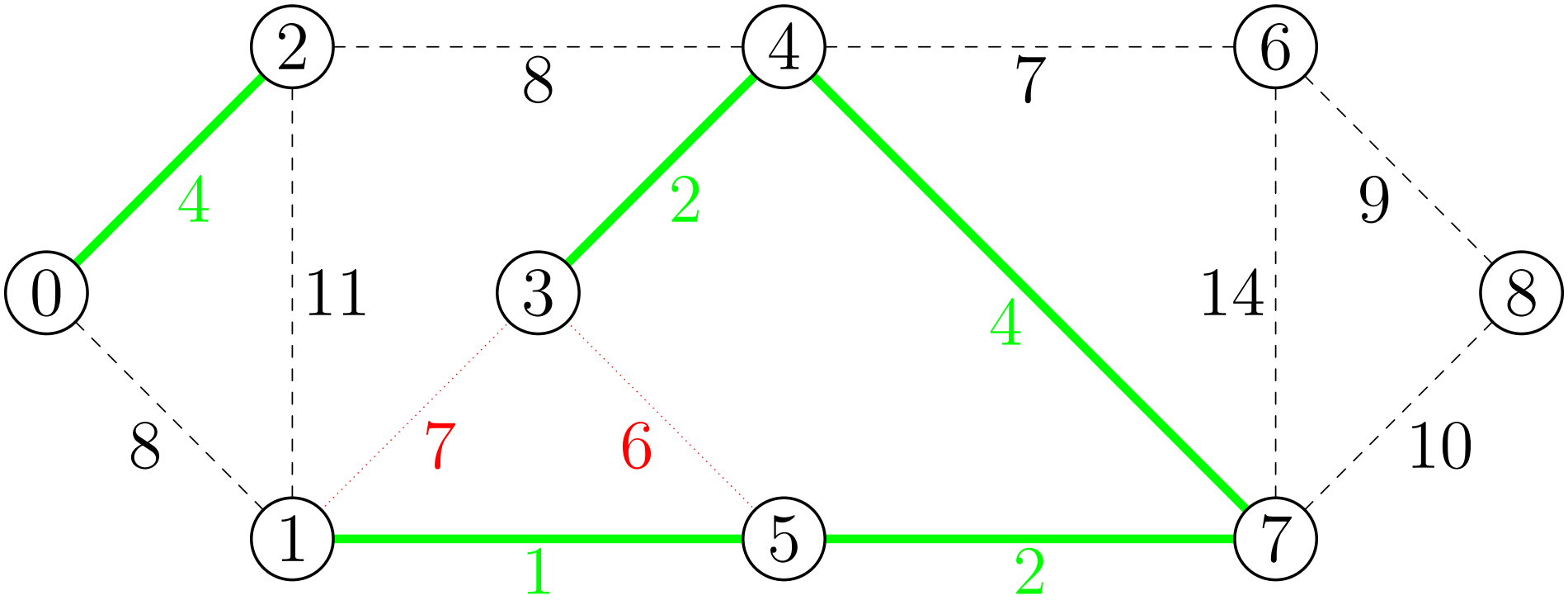
Arco $(5, 3)$: è inutile!

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

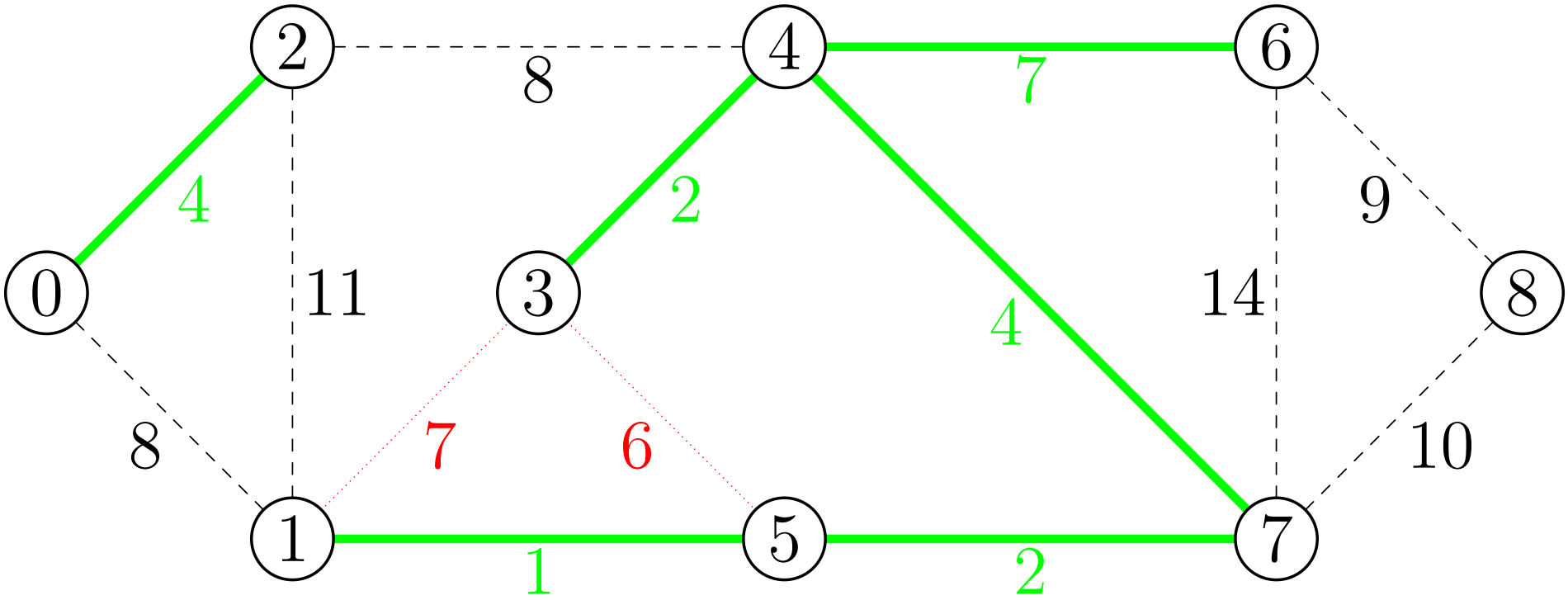


Arco $(1, 3)$: è inutile!

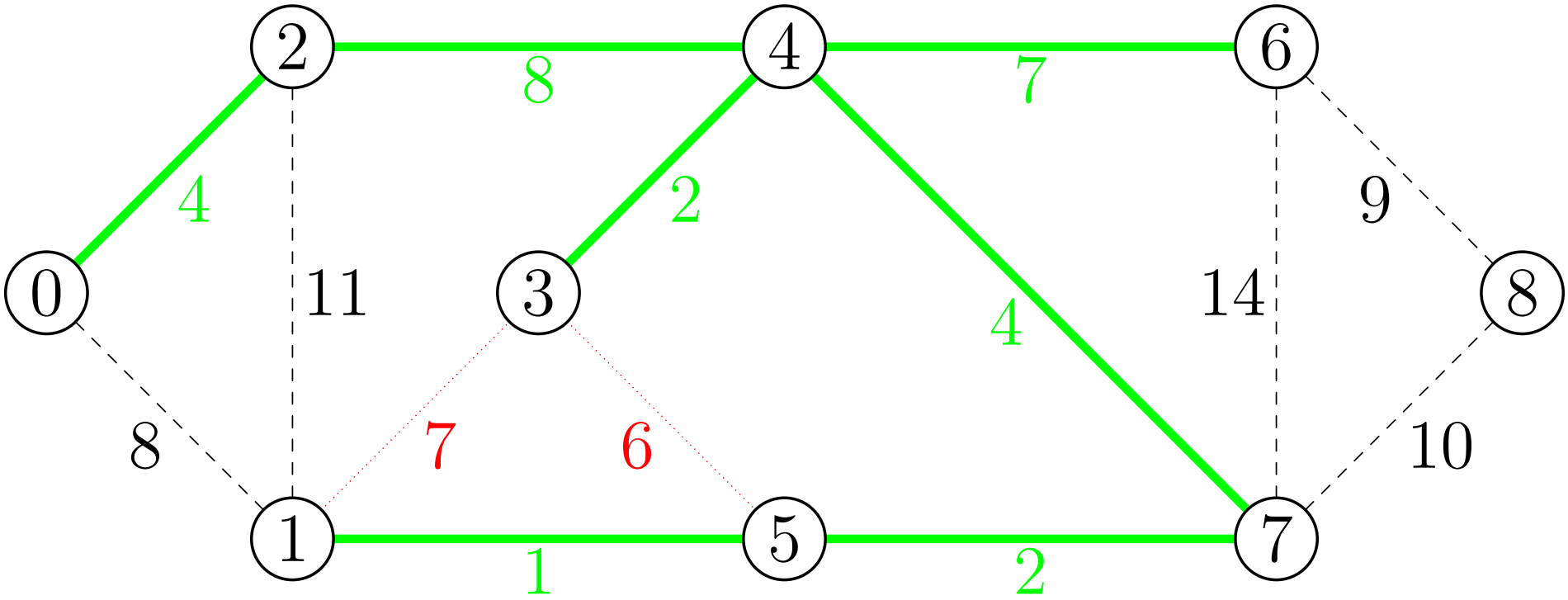
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



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

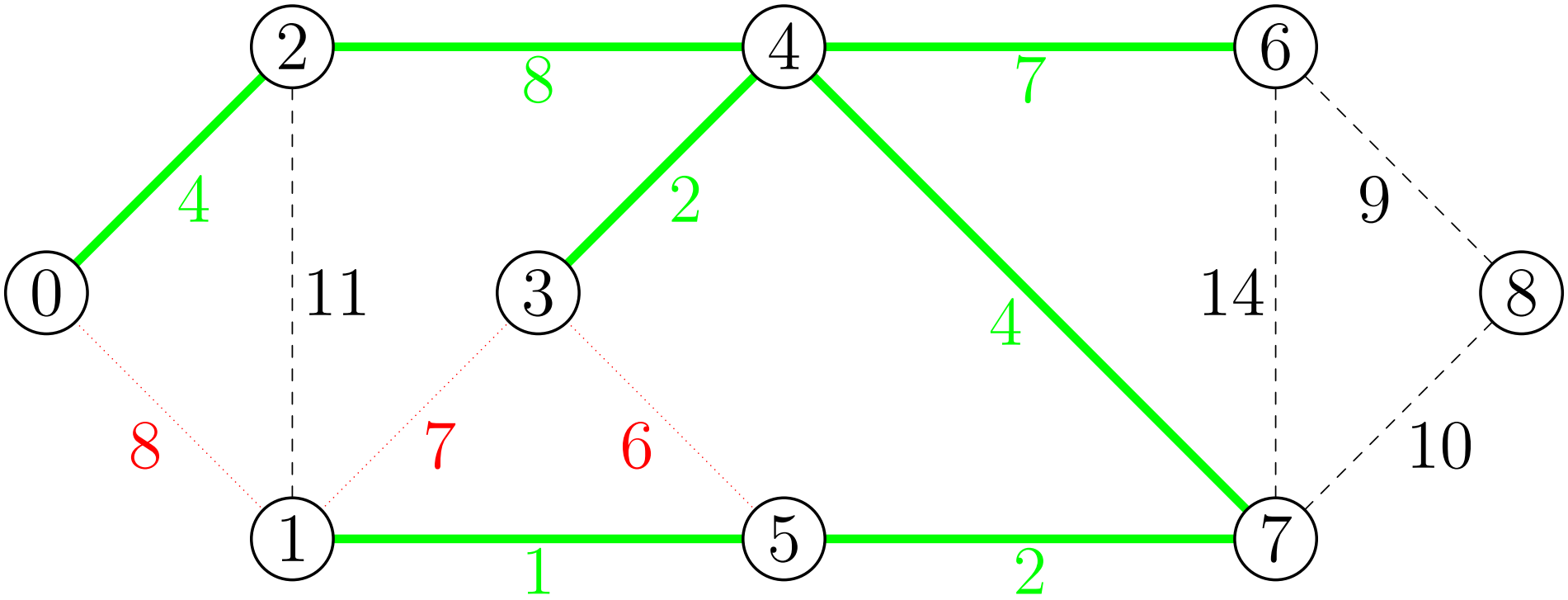


Arco $(2, 4)$: è utile, lo prendo.
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$

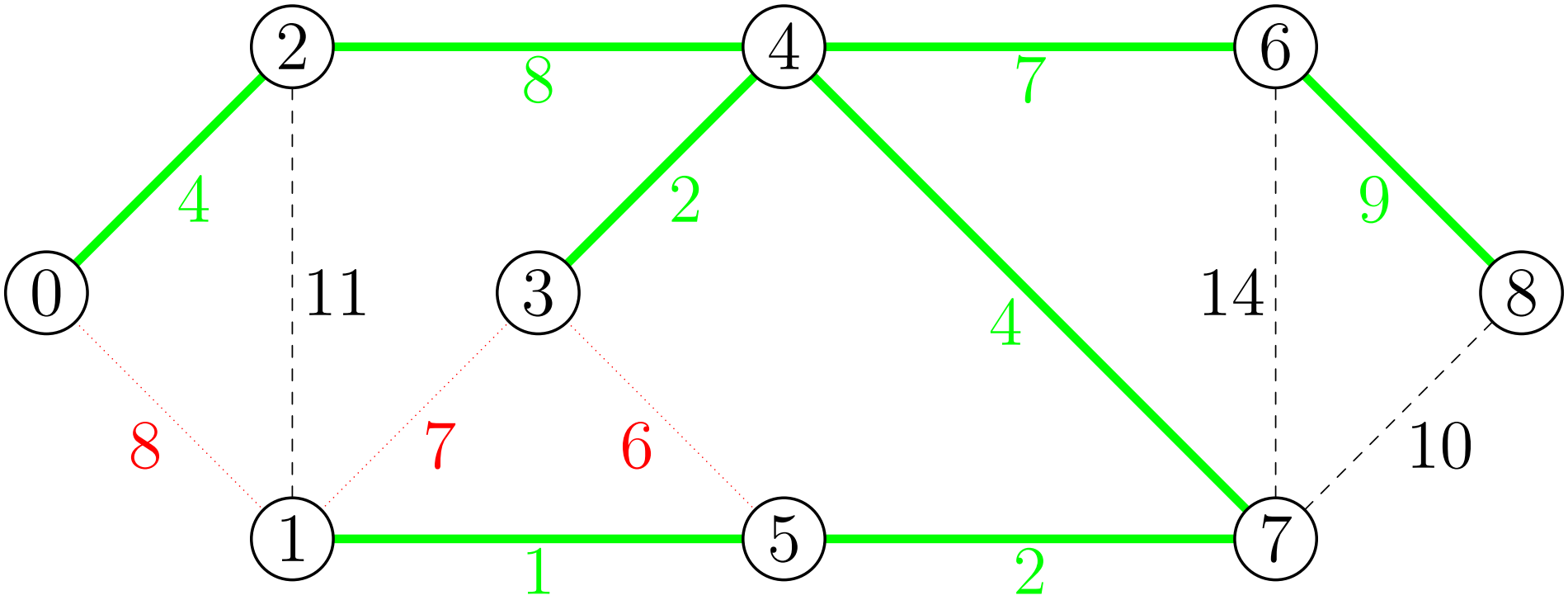


Arco $(0, 1)$: è inutile!

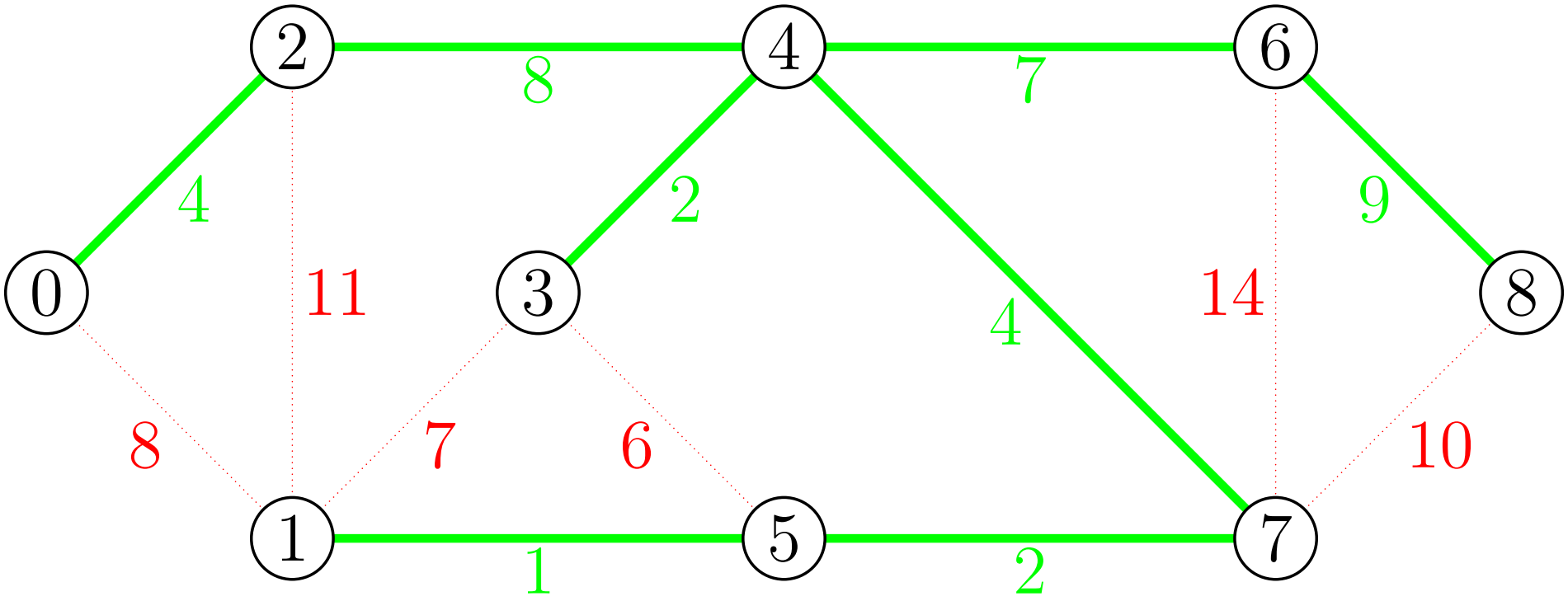
$(1, 5), (3, 4), (5, 7), (4, 7), (0, 2), (5, 3), (1, 3),$
$(4, 6), (2, 4), (0, 1), (6, 8), (7, 8), (1, 2), (6, 7)$



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

Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

Perché funziona?

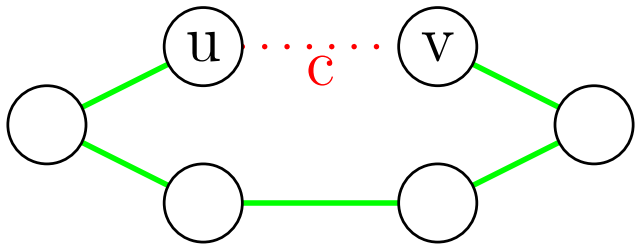
Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

Sia $a = (u, v)$ di costo c il primo arco che l'algoritmo prende ma non è in nessuna soluzione di MST che contiene gli archi precedentemente presi. Sia S una soluzione di MST (cioè un insieme di archi) che contiene gli archi presi precedentemente; se aggiungo a, ottengo un grafo con esattamente un ciclo.

Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

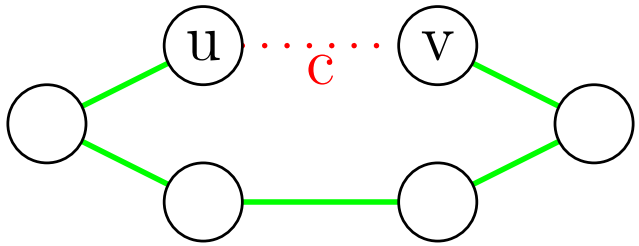
Sia $a = (u, v)$ di costo c il primo arco che l'algoritmo prende ma non è in nessuna soluzione di MST che contiene gli archi precedentemente presi. Sia S una soluzione di MST (cioè un insieme di archi) che contiene gli archi presi precedentemente; se aggiungo a, ottengo un grafo con esattamente un ciclo.



Perché funziona?

Supponiamo per assurdo che l'algoritmo di Kruskal sbagli.

Sia $a = (u, v)$ di costo c il primo arco che l'algoritmo prende ma non è in nessuna soluzione di MST che contiene gli archi precedentemente presi. Sia S una soluzione di MST (cioè un insieme di archi) che contiene gli archi presi precedentemente; se aggiungo a, ottengo un grafo con esattamente un ciclo.



Ottengo uno spanning tree togliendo qualsiasi arco di quel ciclo. Poiché l'algoritmo di Kruskal ha aggiunto l'arco a, vuol dire che non era inutile, cioè non c'erano già tutti gli altri archi del ciclo; quindi almeno uno degli archi del ciclo ha peso maggiore o uguale a c; ma allora prendendo la soluzione S e sostituendo l'arco di costo massimo con l'arco a, si ottiene uno spanning tree di costo minore o uguale a S, dunque un'altra soluzione; questo è 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       for (int i = 1; i <= N; i++)
6          if (p[i] == p[y])
7             p[i] = p[x];
8 }
9 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       for (int i = 1; i <= N; i++)
6          if (p[i] == p[y])
7             p[i] = p[x];
8 }
9 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?

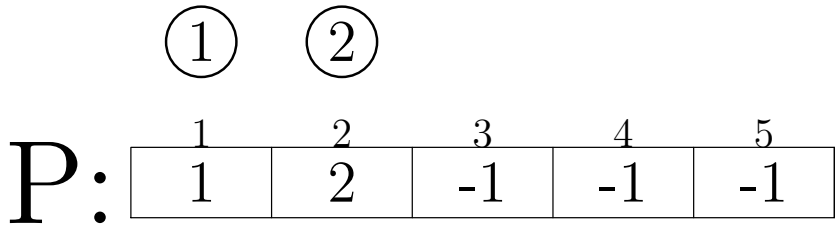
- FIND $\rightarrow \mathcal{O}(N)$
- MAKE_SET $\rightarrow \mathcal{O}(1)$
- UNION $\rightarrow \mathcal{O}(N)$

Esempio

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

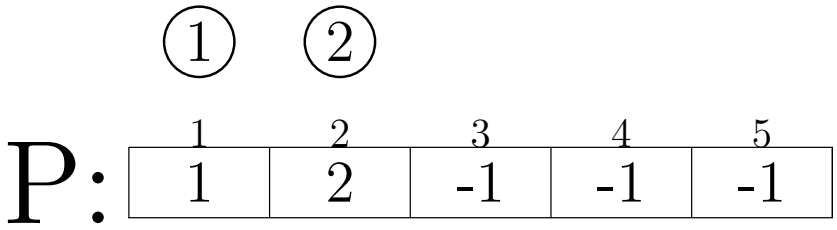
Esempio

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



Esempio

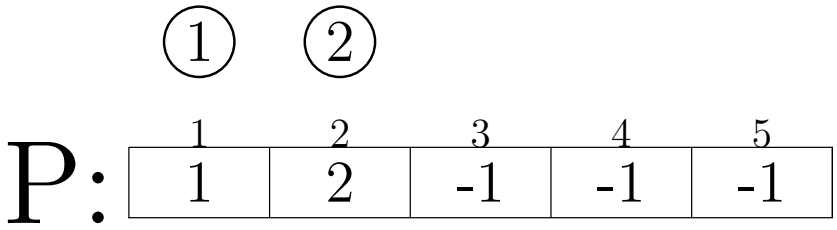
```
MAKE_SET(1);
MAKE_SET(2);
```



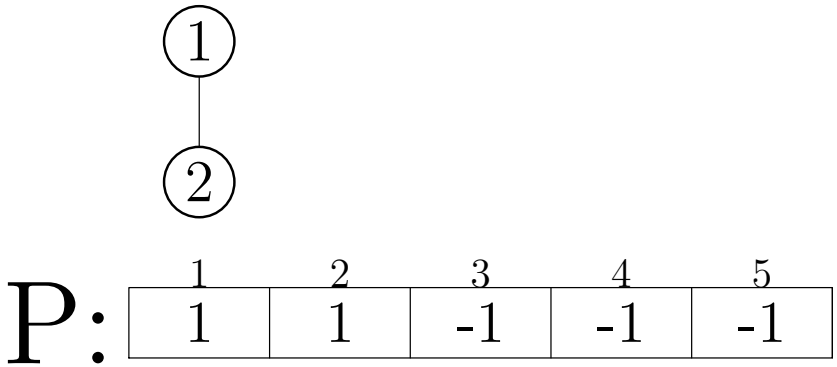

```
UNION(1, 2);
```

Esempio

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

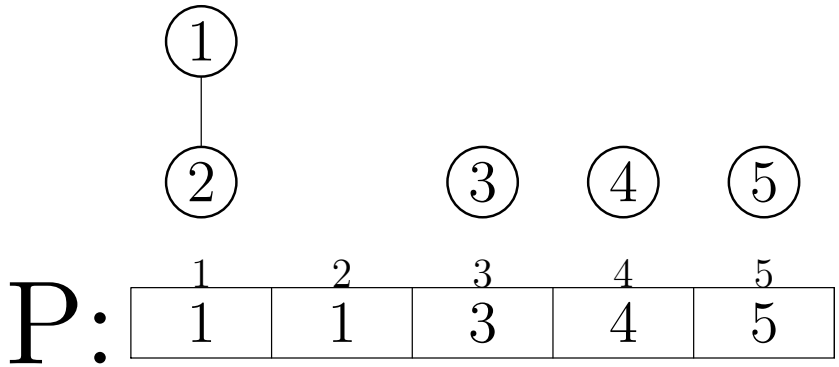



```
UNION(1, 2);
```

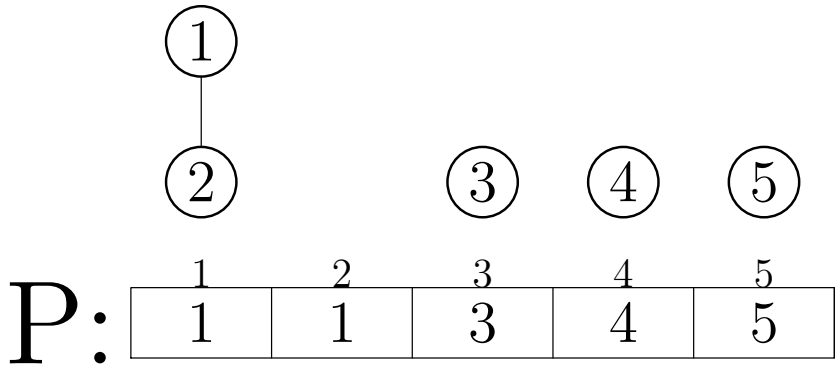


```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

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

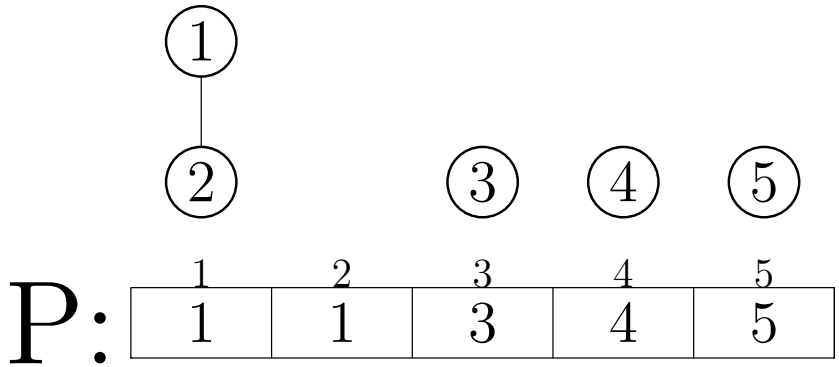


```
MAKE_SET(3); MAKE_SET(4); MAKE_SET(5);
```

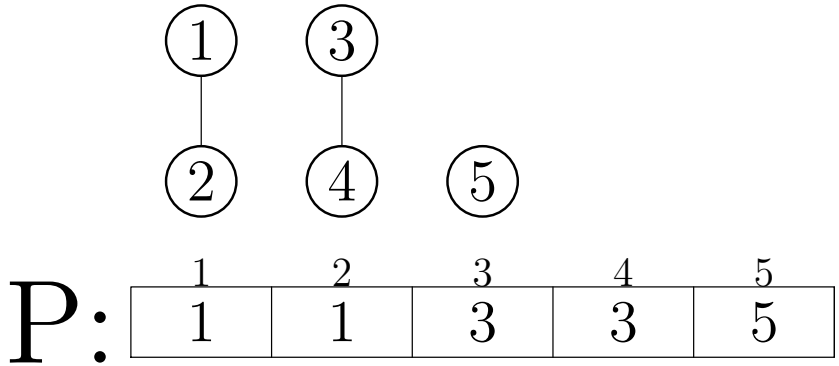



```
UNION(3, 4);
```

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

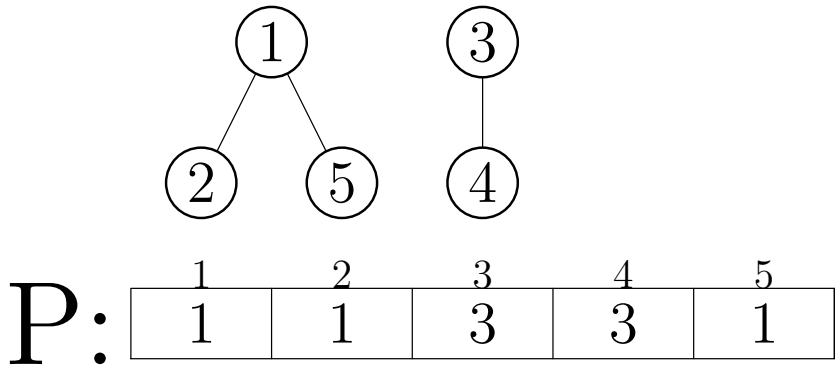



```
UNION(3, 4);
```

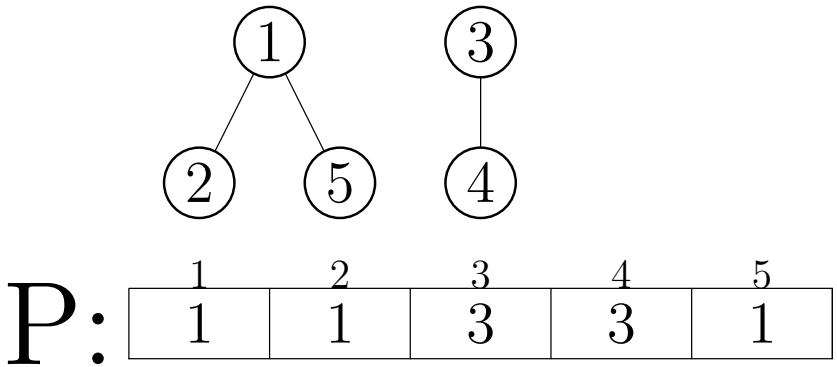


```
UNION(2, 5);
```

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

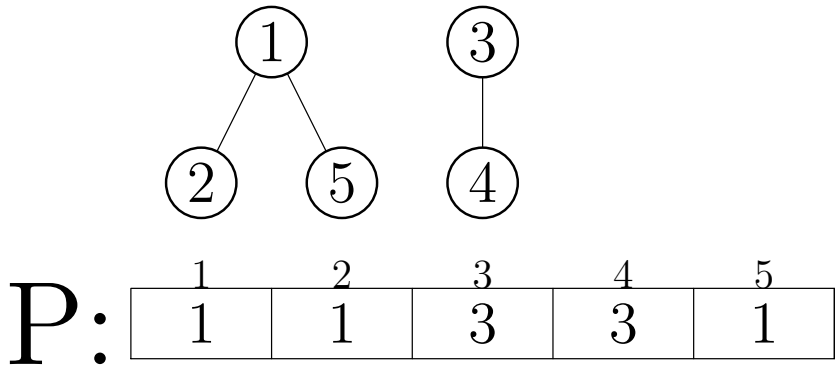


```
UNION(2, 5);
```

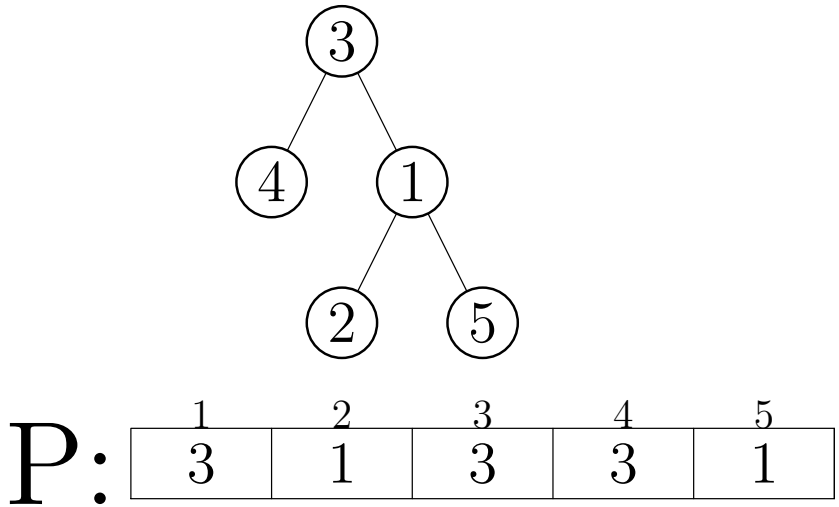



```
UNION(1, 4);
```

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




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



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

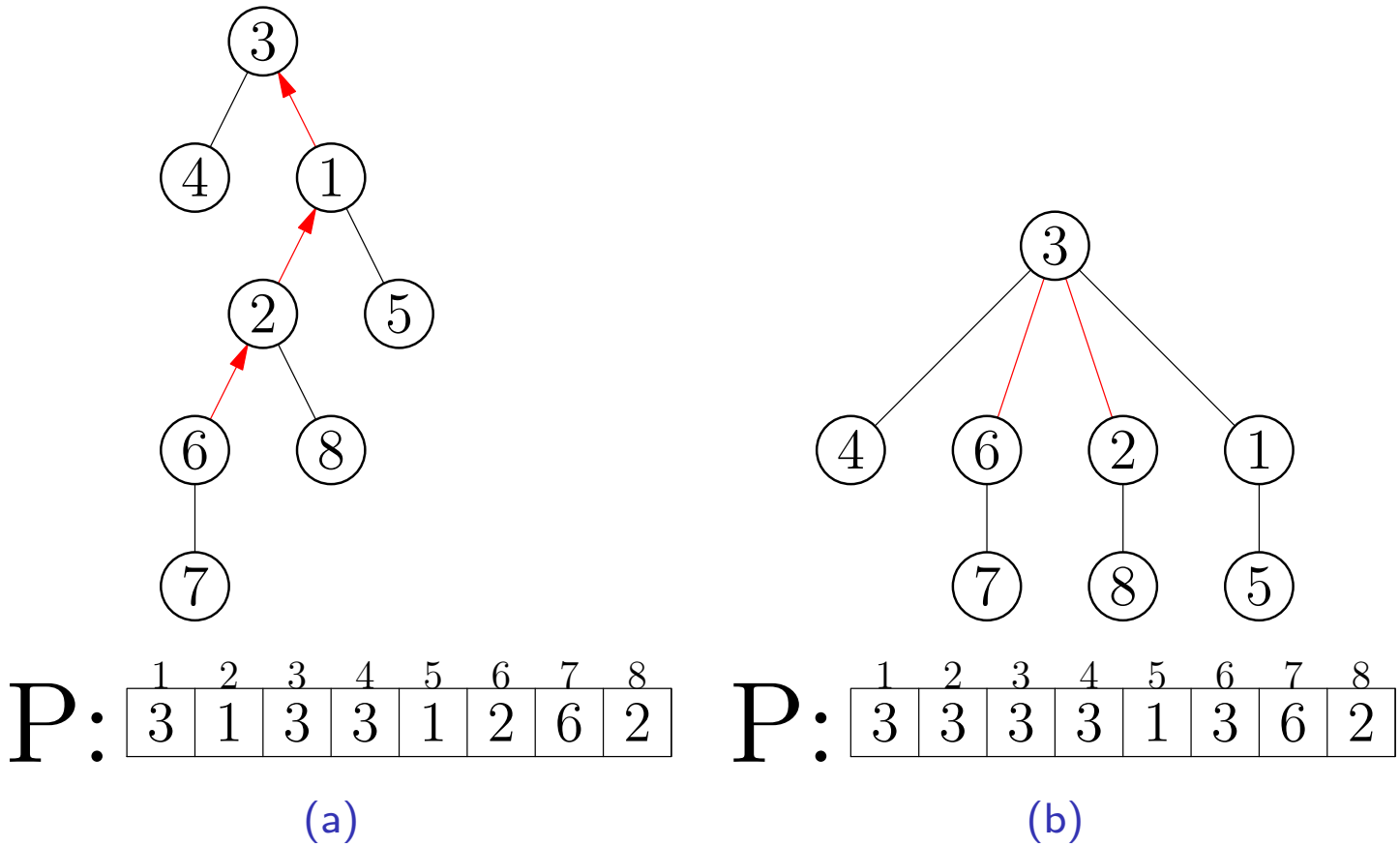


Figura: Ecco cosa succede quando viene chiamata `FIND(6)`

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

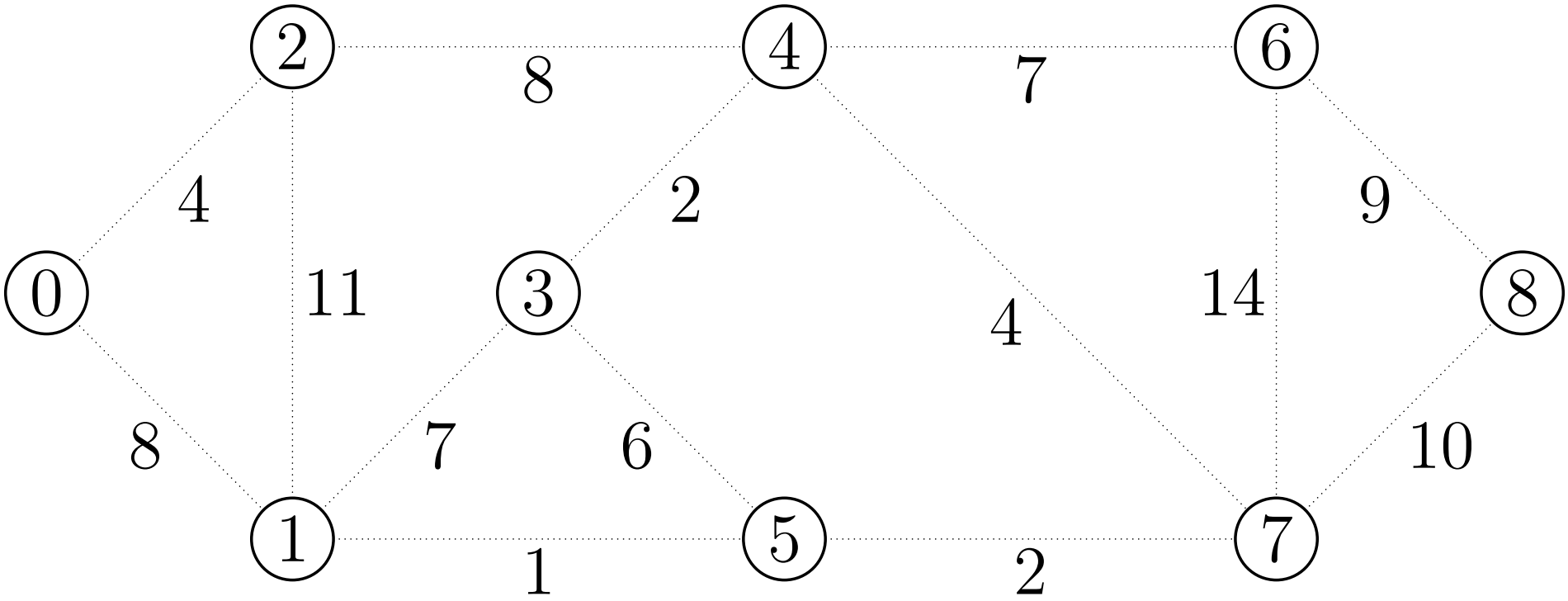
Algoritmo di Prim

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

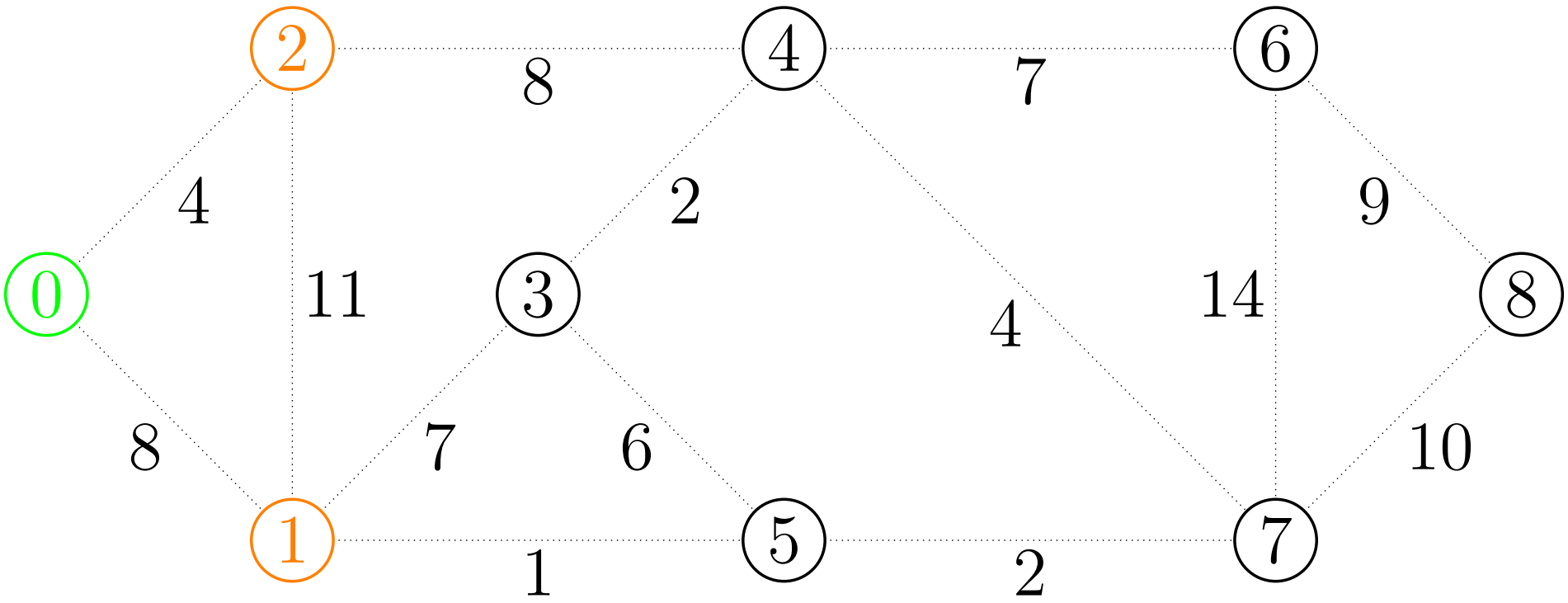
Algoritmo di Prim

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

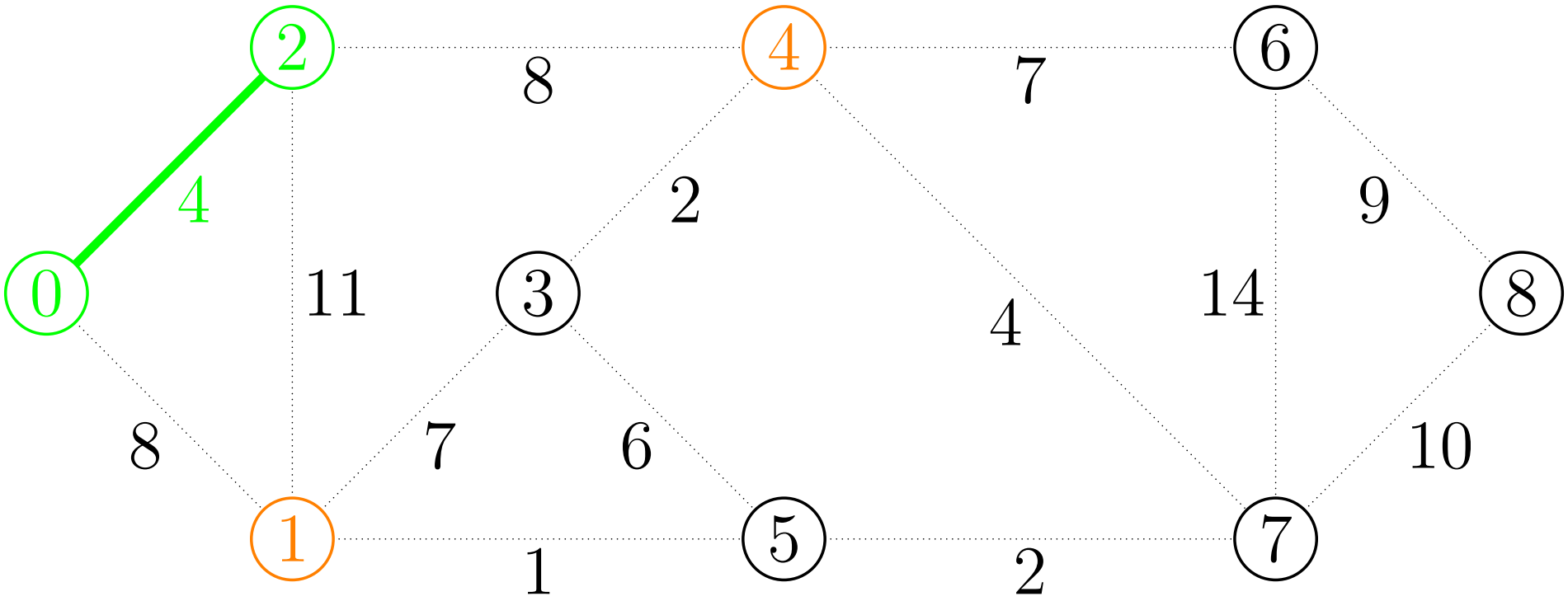
Vediamo come funziona l'algoritmo di Prim sul grafo di esempio.



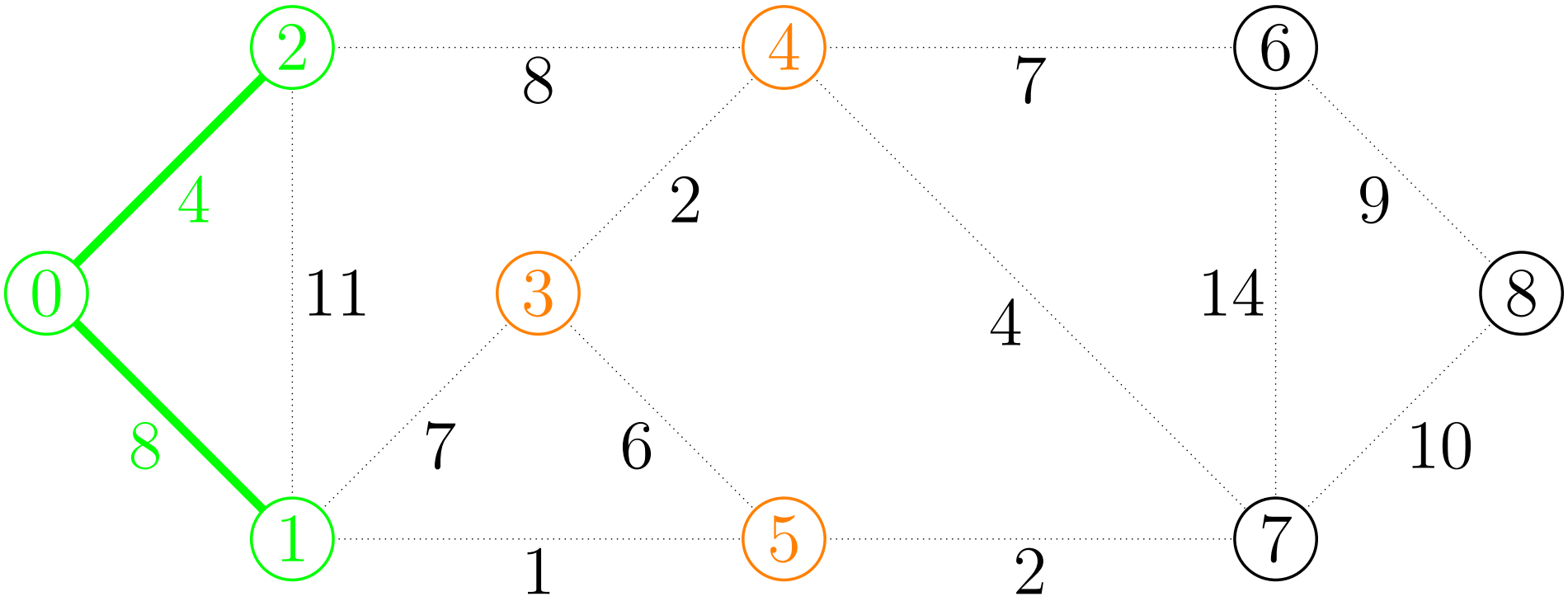
Coloriamo di verde i nodi già raggiunti e di arancione i candidati ad essere il nuovo nodo inserito; coloriamo di verde gli archi presi.


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

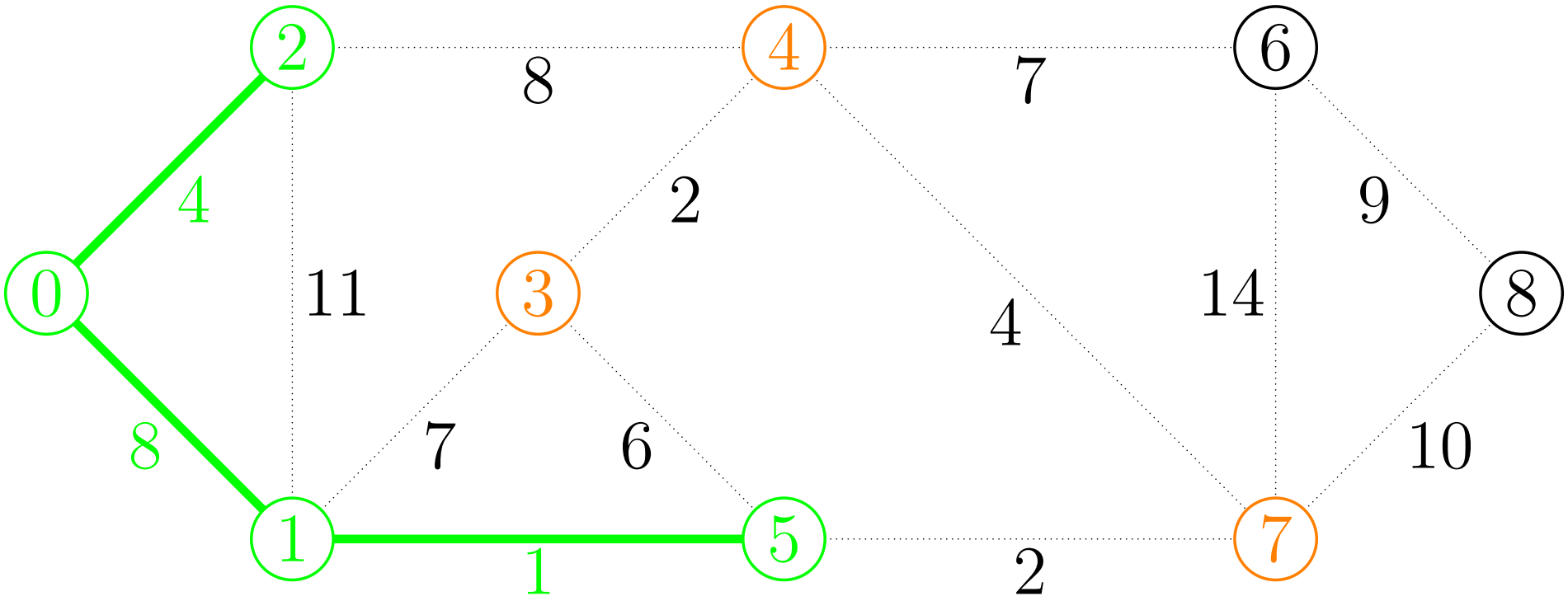
Aggiungo il vertice 2 che è a distanza 4 dai nodi presi, e l'arco $(0, 2)$.



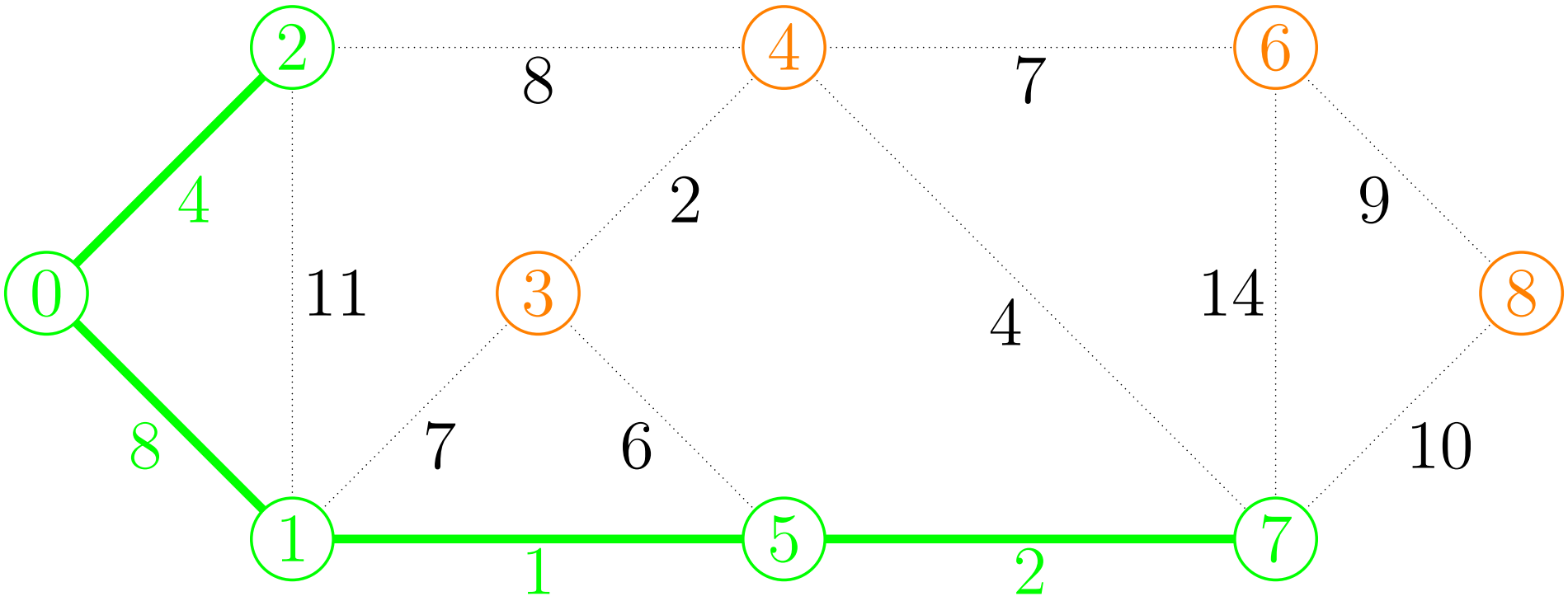
Aggiungo il vertice 1 che è a distanza 8 dai nodi presi, e l'arco $(0, 1)$.



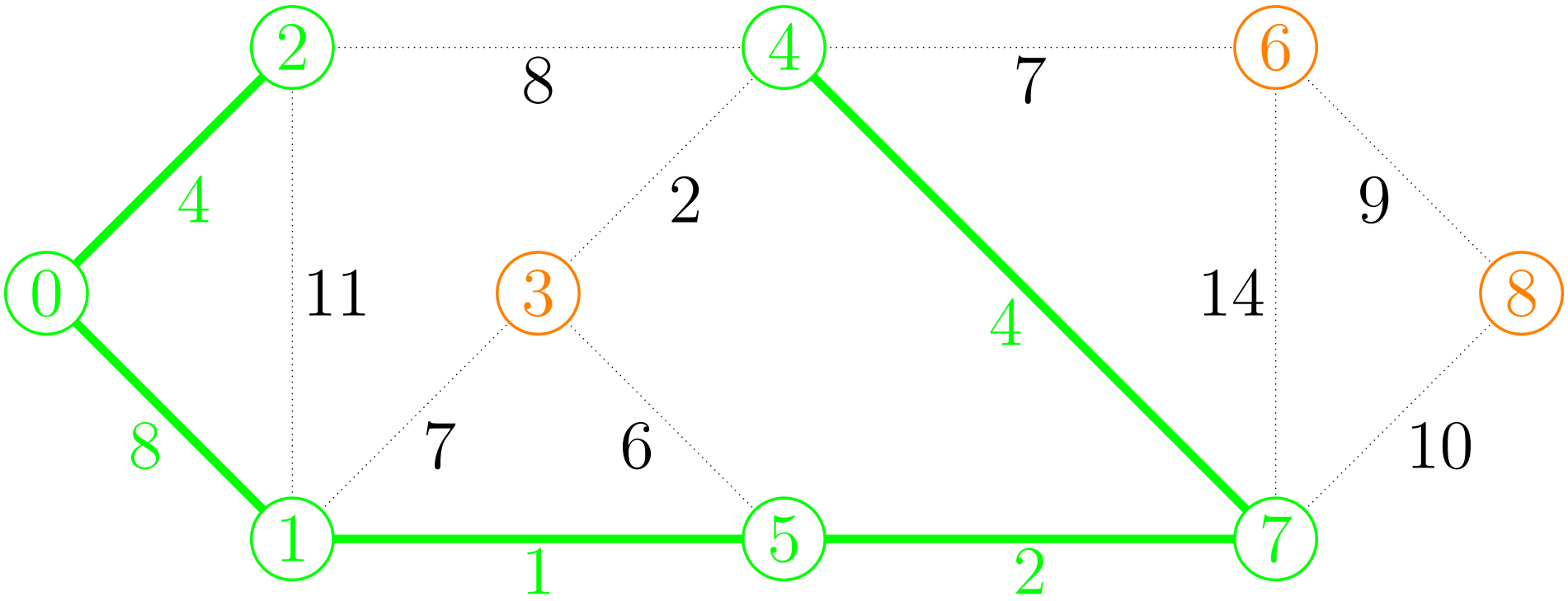
Aggiungo il vertice 5 che è a distanza 1 dai nodi presi, e l'arco $(1, 5)$.



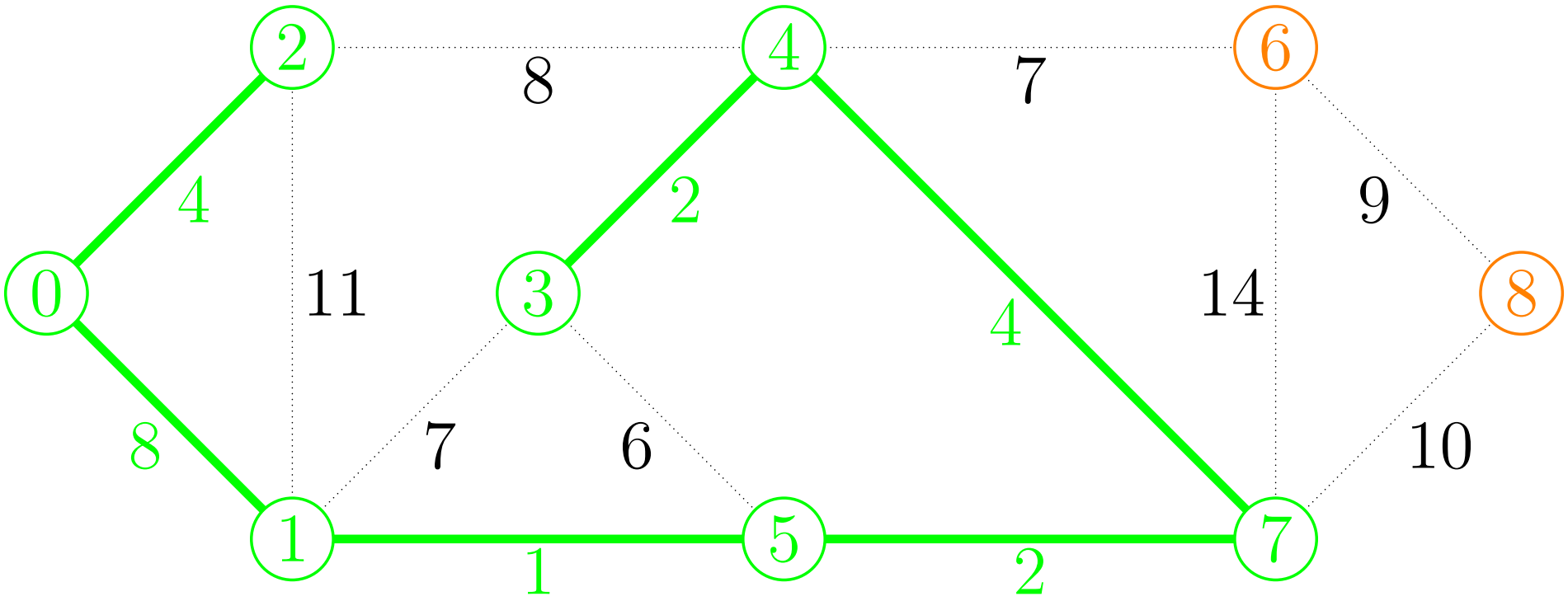
Aggiungo il vertice 7 che è a distanza 2 dai nodi presi, e l'arco $(5, 7)$.



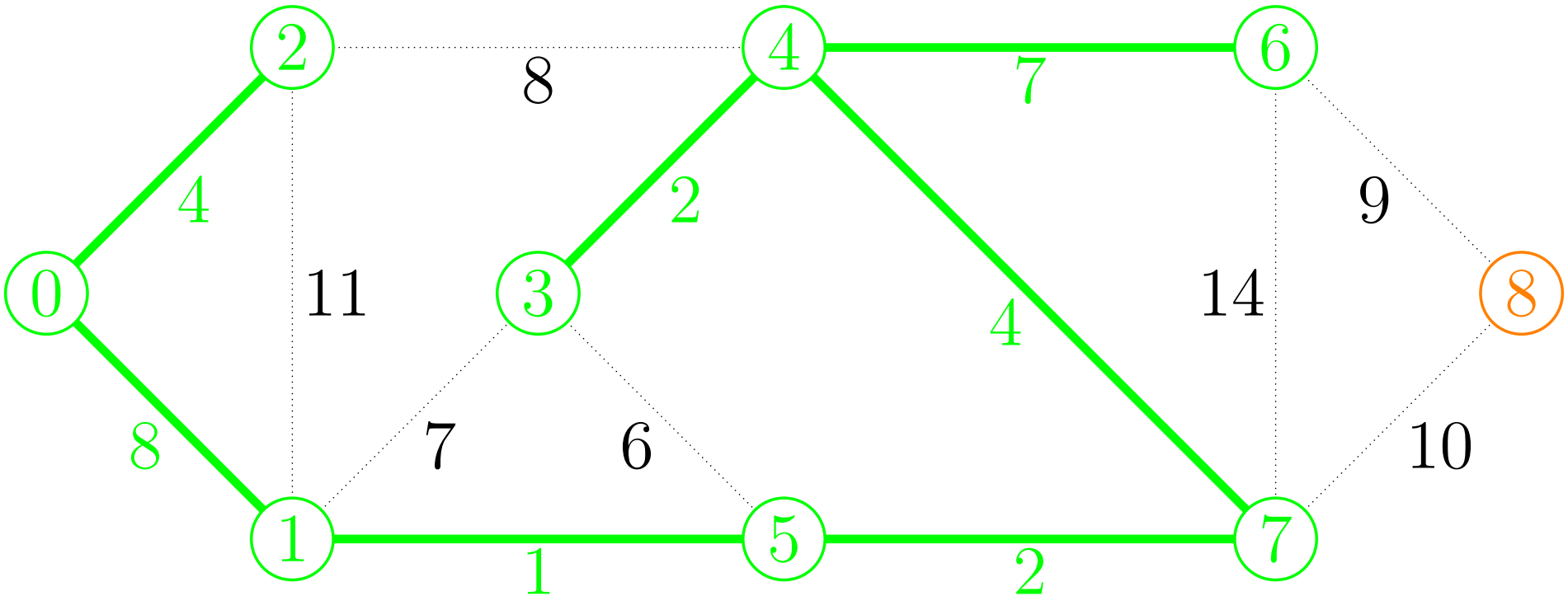
Aggiungo il vertice 4 che è a distanza 4 dai nodi presi, e l'arco $(4, 7)$.



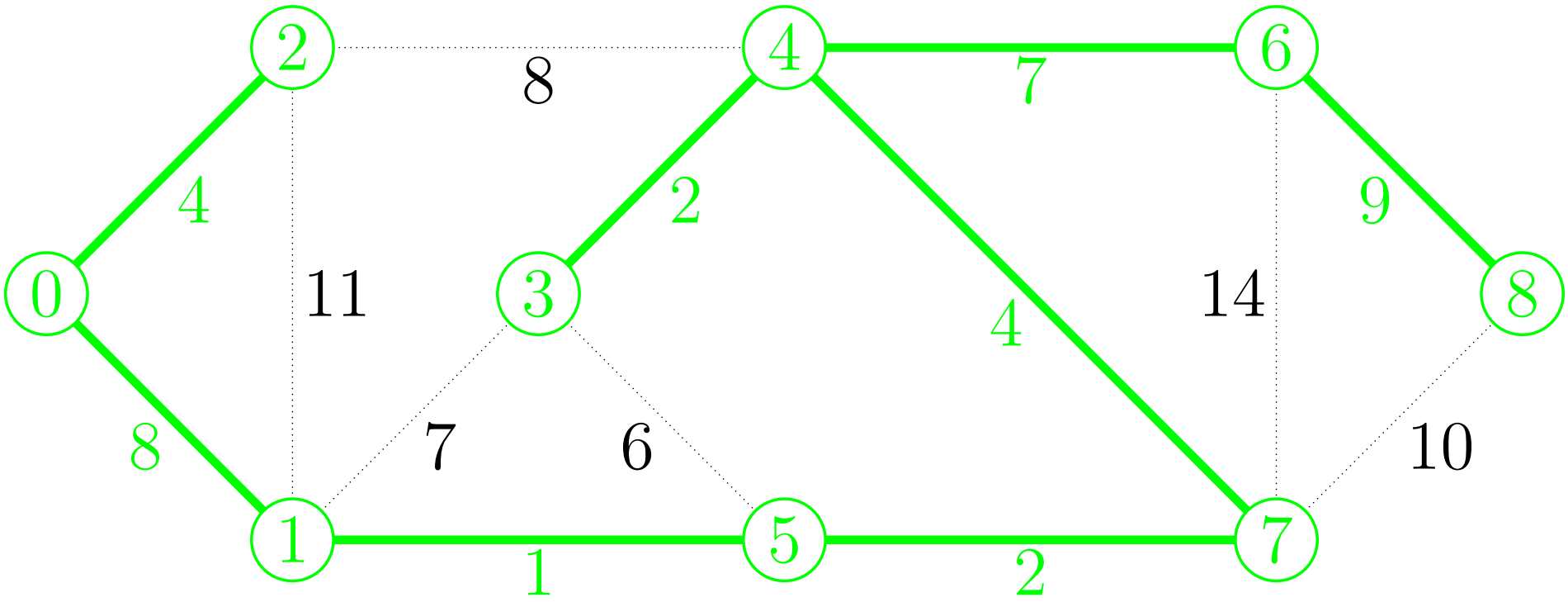
Aggiungo il vertice 3 che è a distanza 2 dai nodi presi, e l'arco $(3, 4)$.



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



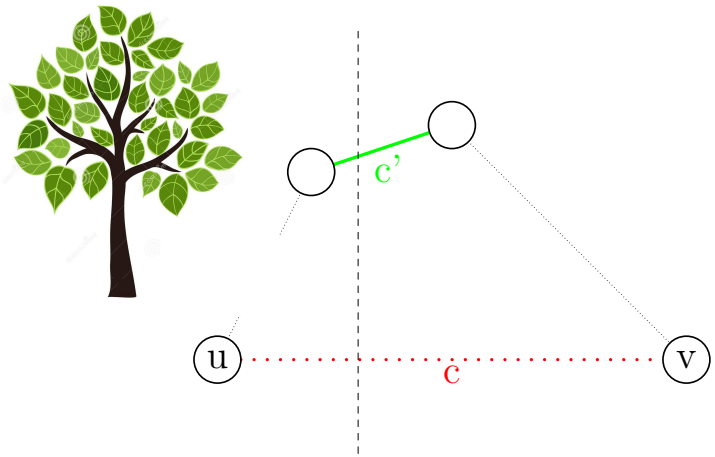
Infine aggiungo il vertice 8 che è a distanza 9 dai nodi presi, e l'arco $(6, 8)$.



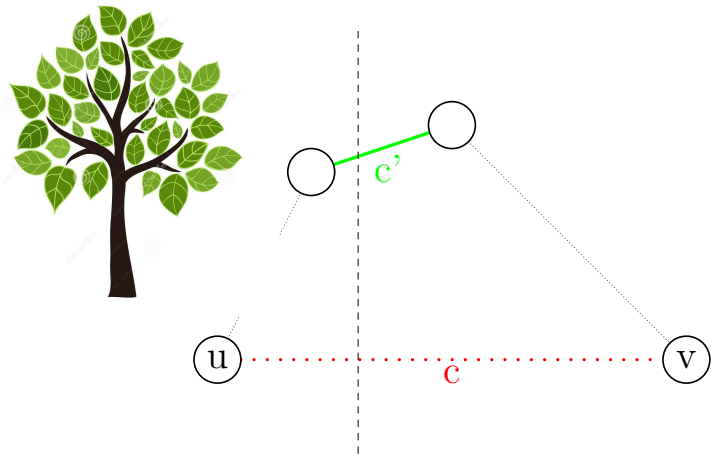
Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.

Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.
Consideriamo la prima volta che inserisce in S un arco $a = (u, v)$ di costo c che non appartiene a una soluzione coerente con gli archi presi precedentemente, in corrispondenza del nuovo vertice v. Prendo una soluzione di MST che abbia come archi quelli già presi tranne a, e aggiungo l'arco a all'albero, ottenendo così un grafo con esattamente un ciclo (di cui a fa parte). C'è almeno un altro arco b, che passa dall'insieme $R - \{u\}$ al suo complementare; sia c' il suo costo.

Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.
Consideriamo la prima volta che inserisce in S un arco $a = (u, v)$ di costo c che non appartiene a una soluzione coerente con gli archi presi precedentemente, in corrispondenza del nuovo vertice v. Prendo una soluzione di MST che abbia come archi quelli già presi tranne a, e aggiungo l'arco a all'albero, ottenendo così un grafo con esattamente un ciclo (di cui a fa parte). C'è almeno un altro arco b, che passa dall'insieme $R - \{u\}$ al suo complementare; sia c' il suo costo.



Perché funziona? Supponiamo per assurdo che l'algoritmo non sia corretto.
Consideriamo la prima volta che inserisce in S un arco $a = (u, v)$ di costo c che non appartiene a una soluzione coerente con gli archi presi precedentemente, in corrispondenza del nuovo vertice v. Prendo una soluzione di MST che abbia come archi quelli già presi tranne a, e aggiungo l'arco a all'albero, ottenendo così un grafo con esattamente un ciclo (di cui a fa parte). C'è almeno un altro arco b, che passa dall'insieme $R - \{u\}$ al suo complementare; sia c' il suo costo.



Allora poiché l'algoritmo ha dovuto considerare l'arco b prima di scegliere a, sicuramente $c \leq c'$. Poiché togliendo l'arco b (e lasciando l'arco a) si ottiene uno spanning tree e per ipotesi *non* è un MST, allora $c > c'$, il che è 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, vertice 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, vertice 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, vertice 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 $\mathcal{O}(M \log M)$.

Esercizi su MST

Esercizio

- *Minimum Spanning Tree training – mst*
- *UVa 10600 – ACM Contest and Blackout*

Ordinamento topologico

Ordinamento topologico

Definizione

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

Ordinamento topologico

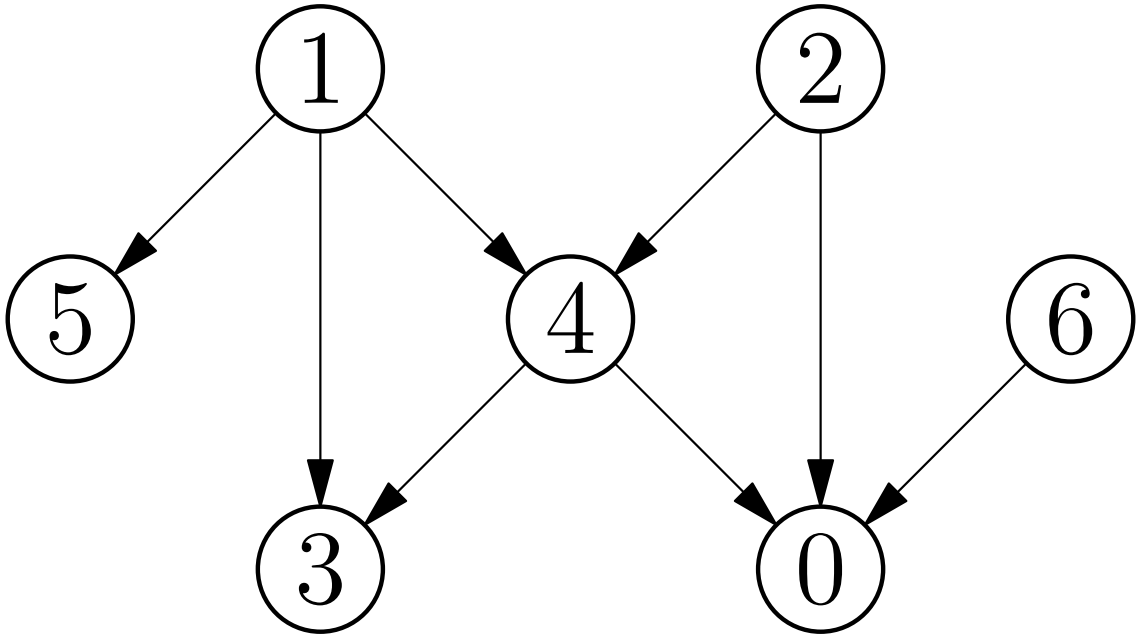
Definizione

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

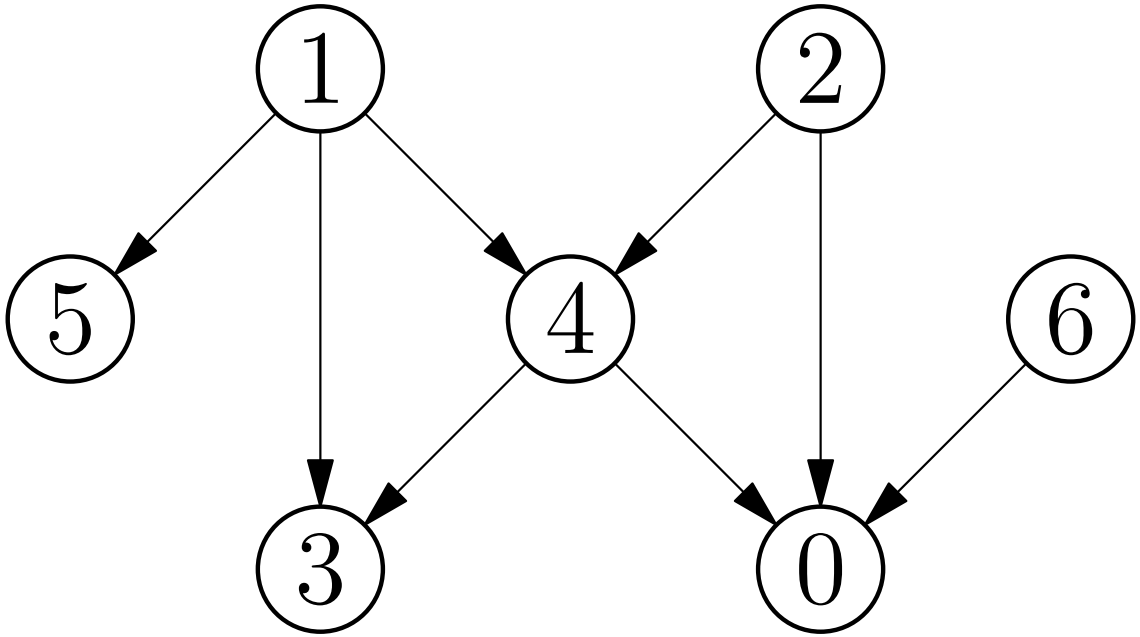
Definizione

Un ordinamento topologico è un ordinamento dei vertici tale che ogni vertice stia prima di tutti i vertici collegati ai suoi archi uscenti.

Esempio di ordinamento topologico

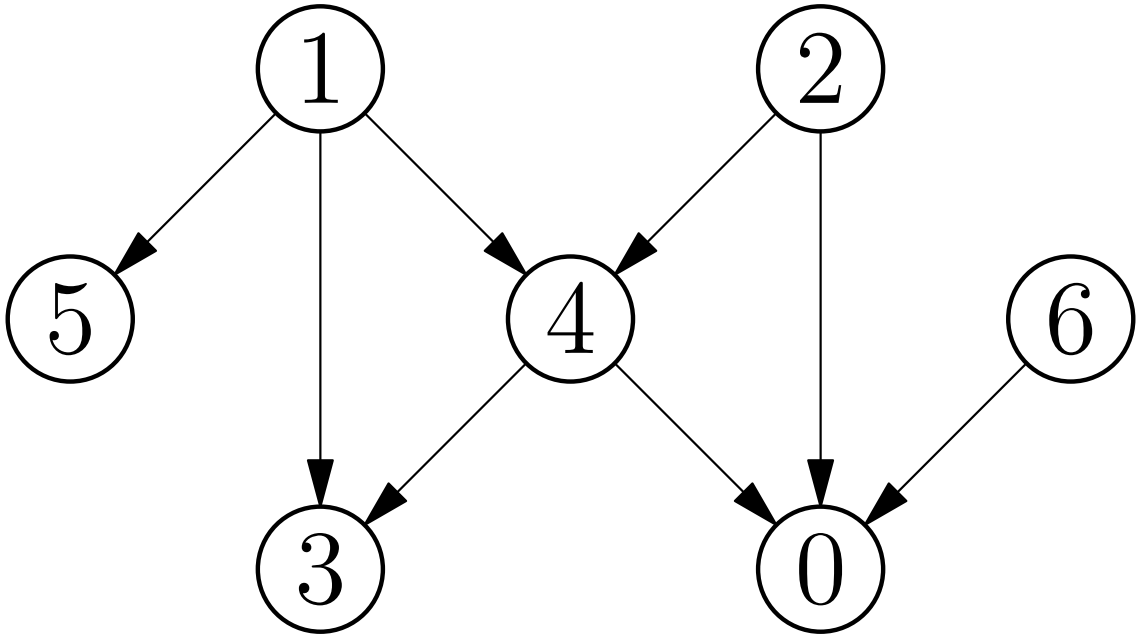


Esempio di ordinamento topologico



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

Esempio di ordinamento topologico



Un ordinamento dei vertici 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 (un nodo a viene prima di b se $t_a > t_b$).

Algoritmo

Facciamo una DFS e ordiniamo in base al tempo di chiusura decrescente dei nodi (un nodo a viene prima di b se $t_a > t_b$).

```
1  int t = 0; // Variabile globale tempo
2  int tempo_chiusura[N]; // Inizializzato a INF
3
4  for (int i = 0; i < N; i++) {
5      if (tempo_chiusura[i] == INF)
6          dfs(i);
7  }
8
9  void dfs(int x) {
10     for (auto y : grafo[x]) {
11         if (tempo_chiusura[y] == INF) {
12             dfs(y);
13         }
14     }
15     t++;
16     tempo_chiusura[x] = t;
17 }
```

Perché funziona?

Perché funziona?

Supponiamo per assurdo che l'algoritmo sbagli. Se sbaglia, *necessariamente* mette un nodo a prima di un nodo b quando esiste un arco da b ad a. Il nodo a è stato chiuso DOPO il nodo b; ma poiché c'è un arco da b ad a, nella funzione `dfs(b)` devo aver analizzato l'arco (b, a) e quindi devo aver chiamato `dfs(a)` prima di aver chiuso il nodo b alla fine della `dfs(b)`. Questo è assurdo.

Esercizi su Toposort

Esercizio

- *UVa 10305 – Ordering Tasks*
- *SPOJ – Toposort*
- *training – picarats*
- *CodeForces 510C - Fox And Names*