Lezione 30

Sommario della Lezione

- Esercizi

1. Larghezza di un albero

La "larghezza" di un albero $T = (V, E)$ è definita come il

$$\max\{N(d) : d = 1, \ldots, |V|\},$$

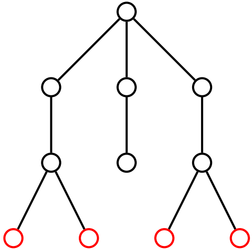
dove $N(d)$ =numero di nodi a distanza d dalla radice.

1. Larghezza di un albero

La "larghezza" di un albero $T = (V, E)$ è definita come il

$$\max\{N(d) : d = 1, \ldots, |V|\},$$

dove $N(d)$ =numero di nodi a distanza d dalla radice.



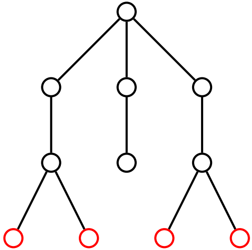
larghezza$= 4$

1. Larghezza di un albero

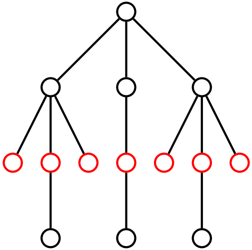
La "larghezza" di un albero $T = (V, E)$ è definita come il

$$\max\{N(d) : d = 1, \ldots, |V|\},$$

dove $N(d)$ =numero di nodi a distanza d dalla radice.



larghezza$= 4$



larghezza$= 7$

Calcolo della larghezza di un albero

Modifichiamo la visita in ampiezza BFS in modo da calcolarci un vettore count, dove per ogni $d = 1, \ldots, |V|$, count$[d]$ contiene il numero di nodi che hanno distanza d dalla radice s dell'albero.

Calcolo della larghezza di un albero

Modifichiamo la visita in ampiezza BFS in modo da calcolarci un vettore `count`, dove per ogni $d = 1, \ldots, |V|$, $\texttt{count}[d]$ contiene il numero di nodi che hanno distanza d dalla radice s dell'albero.

Cercheremo poi il massimo di questo vettore, che ovviamente corrisponderà alla larghezza dell'albero.

1. Calcolo della larghezza di un albero

```
BFS(T, s)
1.   Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
```

1. Calcolo della larghezza di un albero

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$

1. Calcolo della larghezza di un albero

```
BFS(T, s)
1.  Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.  Inizializza L[0] in modo che contenga s; d[s] ← 0
3.  Poni i a 0; S = ∅, count[d] = 0 ∀d = 1, ..., |V|
```

1. Calcolo della larghezza di un albero

BFS(T, s)

1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$

1. Calcolo della larghezza di un albero

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$
4. While $L[i]$ non è vuota

1. Calcolo della larghezza di un albero

```
BFS(T, s)
1.  Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.  Inizializza L[0] in modo che contenga s; d[s] ← 0
3.  Poni i a 0; S = ∅, count[d] = 0 ∀d = 1, . . . , |V|
4.  While L[i] non è vuota
5.    Inizializza una lista vuota L[i + 1]
```

1. Calcolo della larghezza di un albero

BFS(T, s)

1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$
4. While $L[i]$ non è vuota
5. Inizializza una lista vuota $L[i+1]$
6. For ogni nodo $u \in L[i]$

1. Calcolo della larghezza di un albero

BFS(T, s)

Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$

Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$

Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$

While $L[i]$ non è vuota

  Inizializza una lista vuota $L[i+1]$

  For ogni nodo $u \in L[i]$

    Considera ciascun arco (u, v) incidente su u

1. Calcolo della larghezza di un albero

BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then

1. Calcolo della larghezza di un albero

BFS(T, s)

Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$

Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$

Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$

While $L[i]$ non è vuota

  Inizializza una lista vuota $L[i+1]$

  For ogni nodo $u \in L[i]$

    Considera ciascun arco (u, v) incidente su u

    If Scoperto[v]=false then

      Poni Scoperto[v]=true

1. Calcolo della larghezza di un albero

```
BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
Inizializza L[0] in modo che contenga s; d[s] ← 0
Poni i a 0; S = ∅, count[d] = 0 ∀d = 1, ..., |V|
While L[i] non è vuota
  Inizializza una lista vuota L[i + 1]
  For ogni nodo u ∈ L[i]
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
        Aggiungi l'arco (u, v) all'albero S
```

1. Calcolo della larghezza di un albero

BFS(T, s)

Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
      Aggiungi l'arco (u, v) all'albero S
      Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
        count[$d[v]$] $\leftarrow$count[$d[v]$] $+ 1$

1. Calcolo della larghezza di un albero

BFS(T, s)
1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count$[d] = 0$ $\forall d = 1, \ldots, |V|$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Considera ciascun arco (u, v) incidente su u
8.   If Scoperto$[v]$=false then
9.   Poni Scoperto$[v]$=true
10.   Aggiungi l'arco (u, v) all'albero S
11.   Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
        count$[d[v]] \leftarrow$count$[d[v]] + 1$
12.   Incrementa il contatore di livelli i di uno

1. Calcolo della larghezza di un albero

BFS(T, s)

1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count$[d] = 0$ $\forall d = 1, \ldots, |V|$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Considera ciascun arco (u, v) incidente su u
8.   If Scoperto$[v]$=false then
9.   Poni Scoperto$[v]$=true
10.   Aggiungi l'arco (u, v) all'albero S
11.   Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i + 1$,
          count$[d[v]] \leftarrow$count$[d[v]] + 1$
12.   Incrementa il contatore di livelli i di uno
13. return(max(count$[1]$, ..., count$[|V|]$)

1. Calcolo della larghezza di un albero

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $S = \emptyset$, count[d] $= 0$ $\forall d = 1, \ldots, |V|$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Considera ciascun arco (u, v) incidente su u
8.   If Scoperto[v]=false then
9.   Poni Scoperto[v]=true
10.   Aggiungi l'arco (u, v) all'albero S
11.   Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
          count[$d[v]$] $\leftarrow$count[$d[v]$] $+ 1$
12.   Incrementa il contatore di livelli i di uno
13. return(max(count[1], ..., count[$|V|$])

Complessità: $O(|V| + |E|) = O(|V|)$

2. Distanza media in un grafo

Sia dato un grafo $G = (V; E)$ non orientato e connesso, con $n = |V|$. Sia $s \in V$.

2. Distanza media in un grafo

Sia dato un grafo $G = (V; E)$ non orientato e connesso, con $n = |V|$. Sia $s \in V$. La distanza da s ad un nodo $v \in V$ è uguale alla lunghezza (misurata in numero di archi) del più breve cammino da s a v.

2. Distanza media in un grafo

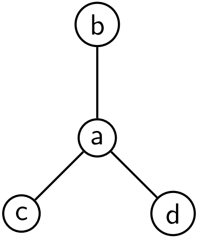
Sia dato un grafo $G = (V; E)$ non orientato e connesso, con $n = |V|$. Sia $s \in V$. La distanza da s ad un nodo $v \in V$ è uguale alla lunghezza (misurata in numero di archi) del più breve cammino da s a v. Vogliamo un algoritmo che ritorni la distanza media di s da tutti gli altri nodi del grafo (escluso s).

2. Distanza media in un grafo

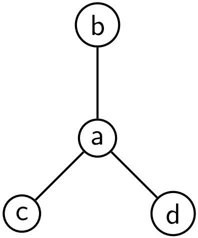
Sia dato un grafo $G = (V; E)$ non orientato e connesso, con $n = |V|$. Sia $s \in V$. La distanza da s ad un nodo $v \in V$ è uguale alla lunghezza (misurata in numero di archi) del più breve cammino da s a v. Vogliamo un algoritmo che ritorni la distanza media di s da tutti gli altri nodi del grafo (escluso s).

Useremo la BFS per calcolarci le distanze dei nodi $v \in V$ da s, sommeremo le distanze e divideremo per $n - 1$ per ottenere la distanza media.

Esempio

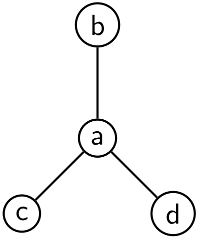


Esempio



La distanza media di a è $(1 + 1 + 1)/3 = 1$.

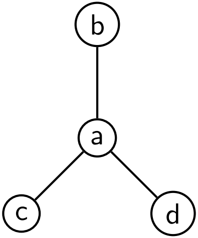
Esempio



La distanza media di a è $(1 + 1 + 1)/3 = 1$.

La distanza media di b è $(1 + 2 + 2)/3 = 5/3$.

Esempio

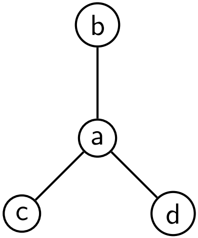


La distanza media di a è $(1+1+1)/3 = 1$.

La distanza media di b è $(1+2+2)/3 = 5/3$.

La distanza media di c è $(1+2+2)/3 = 5/3$.

Esempio



La distanza media di a è $(1 + 1 + 1)/3 = 1$.

La distanza media di b è $(1 + 2 + 2)/3 = 5/3$.

La distanza media di c è $(1 + 2 + 2)/3 = 5/3$.

La distanza media di d è $(1 + 2 + 2)/3 = 5/3$.

Calcolo della distanza media in un grafo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$

Calcolo della distanza media in un grafo

BFS(T, s)
1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $T = \emptyset$, tot$= 0$

Calcolo della distanza media in un grafo

BFS(T, s)

Poni Scoperto[s] $=$true e Scoperto[v] $=$false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
Poni i a 0; $T = \emptyset$, tot$= 0$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
       Aggiungi l'arco (u, v) all'albero T
       Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
          tot=tot$+d[v]$

Calcolo della distanza media in un grafo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $T = \emptyset$, tot$= 0$
4. While $L[i]$ non è vuota
5. Inizializza una lista vuota $L[i+1]$
6. For ogni nodo $u \in L[i]$
7. Considera ciascun arco (u, v) incidente su u
8. If Scoperto[v]=false then
9. Poni Scoperto[v]=true
10. Aggiungi l'arco (u, v) all'albero T
11. Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
tot=tot$+d[v]$ 12. Incrementa il contatore
di livelli i di uno

Calcolo della distanza media in un grafo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $T = \emptyset$, tot$= 0$
4. While $L[i]$ non è vuota
5. Inizializza una lista vuota $L[i+1]$
6. For ogni nodo $u \in L[i]$
7. Considera ciascun arco (u, v) incidente su u
8. If Scoperto[v]=false then
9. Poni Scoperto[v]=true
10. Aggiungi l'arco (u, v) all'albero T
11. Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
tot=tot+$d[v]$ 12. Incrementa il contatore
di livelli i di uno
13. return(tot$/(n-1)$)

Calcolo della distanza media in un grafo

BFS(T, s)

1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $T = \emptyset$, tot$= 0$
4. While $L[i]$ non è vuota
5. Inizializza una lista vuota $L[i+1]$
6. For ogni nodo $u \in L[i]$
7. Considera ciascun arco (u, v) incidente su u
8. If Scoperto$[v]$=false then
9. Poni Scoperto$[v]$=true
10. Aggiungi l'arco (u, v) all'albero T
11. Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$, tot=tot$+d[v]$ 12. Incrementa il contatore di livelli i di uno
13. return(tot$/(n-1)$)

Complessità: $O(|V| + |E|)$

3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.

3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$,

3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

3. Cicli in grafi

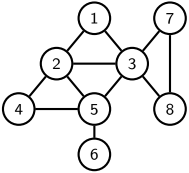
Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```

3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

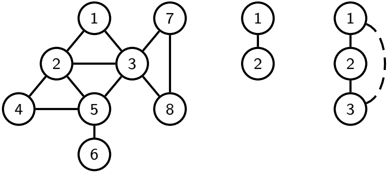
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "Sì" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "No", altrimenti.

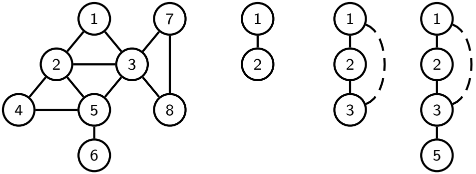
```
DFS(u)
Marca il nodo u ``Esplorato'' ed inseriscilo in T
For ogni arco (u,v) incidente su u
  If v non è marcato ``Esplorato''
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

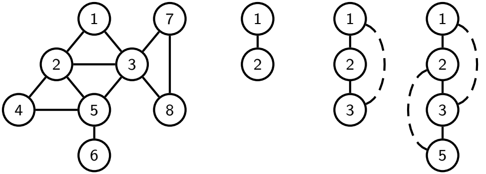
```
DFS(u)
Marca il nodo u ``Esplorato'' ed inseriscilo in T
For ogni arco (u,v) incidente su u
  If v non è marcato ``Esplorato''
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

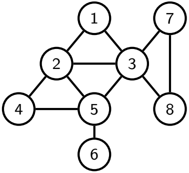
```
DFS(u)
Marca il nodo u ``Esplorato'' ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ``Esplorato''
    ricorsivamente chiama DFS(v)
```

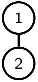


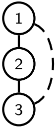
3. Cicli in grafi

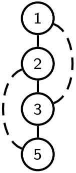
Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

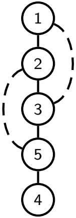
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```







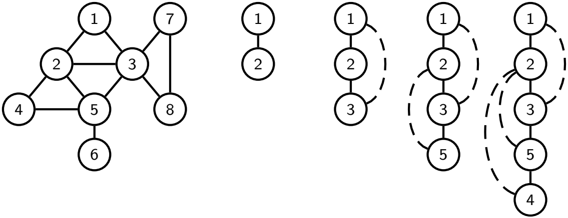




3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

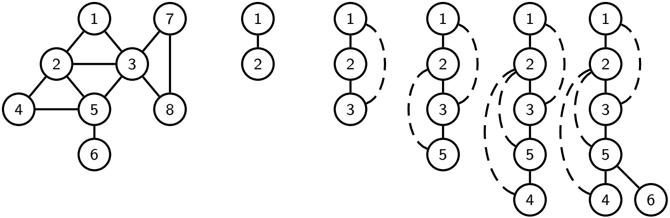
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

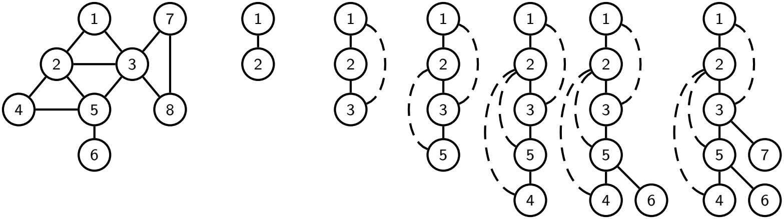
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u,v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

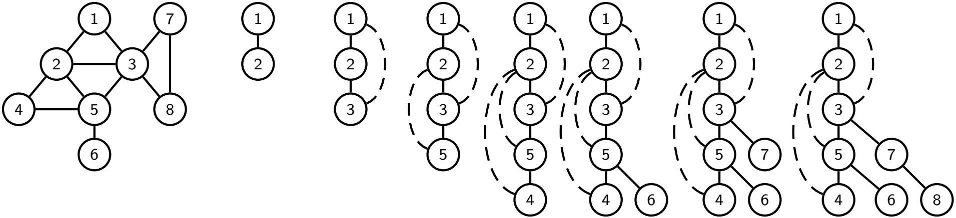
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u,v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \dots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \dots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

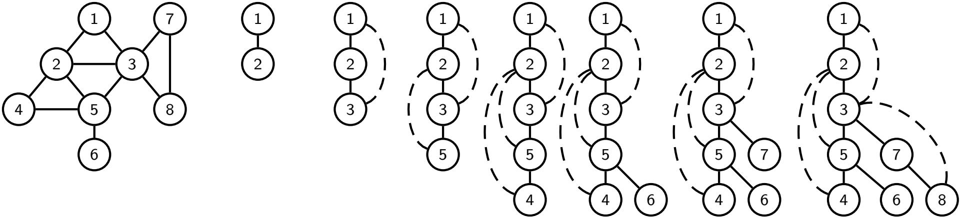
```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u,v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```



3. Cicli in grafi

Input: Un grafo non diretto $G = (V, E)$.
Output: "SI" se esiste un ciclo in G, ovvero una sequenza di vertici $v_0, v_1, \ldots, v_n$ tale che $(v_i, v_{i+1}) \in E$, per $i = 0, \ldots, n-1$ e $v_0 = v_n$, "NO", altrimenti.

```
DFS(u)
Marca il nodo u ‘‘Esplorato’’ ed inseriscilo in T
For ogni arco (u, v) incidente su u
  If v non è marcato ‘‘Esplorato’’
    ricorsivamente chiama DFS(v)
```

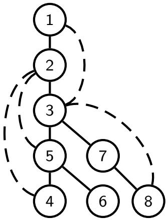


Ricordiamo la seguente proprietà della visita DFS

> Sia T un albero DFS e siano x e y due nodi in T. Sia (x, y) un arco del grafo G che non è un arco di T. Allora o x è un ancestore di y o y è un ancestore di x.

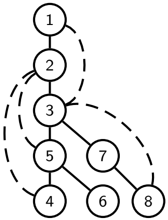
Ricordiamo la seguente proprietà della visita DFS

> Sia T un albero DFS e siano x e y due nodi in T. Sia (x, y) un arco del grafo G che non è un arco di T. Allora o x è un ancestore di y o y è un ancestore di x.



Ricordiamo la seguente proprietà della visita `DFS`

> Sia T un albero `DFS` e siano x e y due nodi in T. Sia (x, y) un arco del grafo G che non è un arco di T. Allora o x è un ancestore di y o y è un ancestore di x.



Pertanto per scoprire se ci sono cicli nel grafo G basta solo verificare che esistano archi nel grafo G che **non vengono inseriti** dall'algoritmo `DFS` nell'albero T, essi rappresentano "scorciatoie" nell'albero DFS, ovvero creano cicli nel grafo G.

Più precisamente, modifichiamo l'algoritmo DFS

DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] $\leftarrow$ Falso;
  predecessore[u] $\leftarrow$ NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)

Più precisamente, modifichiamo l'algoritmo `DFS`

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)
```

Più precisamente, modifichiamo l'algoritmo DFS

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
```

Più precisamente, modifichiamo l'algoritmo DFS

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
```

Più precisamente, modifichiamo l'algoritmo `DFS`

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
  do if esplorato[v]=Falso
```

Più precisamente, modifichiamo l'algoritmo `DFS`

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
  do if esplorato[v]=Falso
    do predecessore[v] ← u
```

Più precisamente, modifichiamo l'algoritmo `DFS`

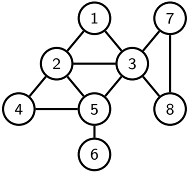
```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
```

Più precisamente, modifichiamo l'algoritmo DFS

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
      else if Predecessore[u] ≠ v
           return("SI", ciclo esiste)
```

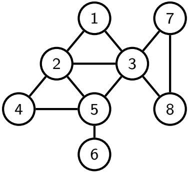
Più precisamente, modifichiamo l'algoritmo DFS

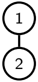
```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
      else if  Predecessore[u] ≠ v
            return("SI", ciclo esiste)
```

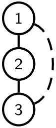


Più precisamente, modifichiamo l'algoritmo DFS

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
      else if  Predecessore[u] ≠ v
           return("SI", ciclo esiste)
```







Complessità dell'algoritmo

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
```

Complessità dell'algoritmo

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
  do esplorato[u] ← Falso;
  predecessore[u] ← NIL
for ogni vertice u in G
  do if esplorato[u]=Falso
    DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)
```

Complessità dell'algoritmo

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
```

Complessità dell'algoritmo

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return( "NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
```

Complessità dell'algoritmo

```
DFS-Segnala_cicli(G)
for ogni vertice u in G
   do esplorato[u] ← Falso;
   predecessore[u] ← NIL
for ogni vertice u in G
   do if esplorato[u]=Falso
      DFS-Segnala_cicli(u)
return("NO", ciclo non esiste)

DFS-Segnala_cicli(u)
esplorato[u] ← Vero
for ogni vertice v adiacente a v
   do if esplorato[v]=Falso
      do predecessore[v] ← u
        DFS-Segnala_cicli(v)
      else if Predecessore[u] ≠ v
           return("SI", ciclo esiste)
```

La stessa di DFS(G), ovvero $O(|V| + |E|)$.

Possiamo far meglio? Sembrerebbe di no, in quanto solo per leggere il grafo spendiamo tempo $O(|V|+|E|)$....

Possiamo far meglio? Sembrerebbe di no, in quanto solo per leggere il grafo spendiamo tempo $O(|V| + |E|)$....
Però per scoprire cicli in grafi *non diretti* non è sempre necessario leggere tutto il grafo.

Ricordiamo il seguente fatto:

Ricordiamo il seguente fatto:
Sia G un grafo con n nodi. Ciascuna coppia delle seguenti affermazioni implica la terza (e quindi ogni coppia rappresenta una *equivalente* ed alternativa definizione di albero).

- G è connesso.

Ricordiamo il seguente fatto:
Sia G un grafo con n nodi. Ciascuna coppia delle seguenti affermazioni implica la terza (e quindi ogni coppia rappresenta una *equivalente* ed alternativa definizione di albero).

- G è connesso.
- G non contiene cicli.

Ricordiamo il seguente fatto:
Sia G un grafo con n nodi. Ciascuna coppia delle seguenti affermazioni implica la terza (e quindi ogni coppia rappresenta una *equivalente* ed alternativa definizione di albero).

- G è connesso.
- G non contiene cicli.
- G ha esattamente $n - 1$ archi.

Ricordiamo il seguente fatto:
Sia G un grafo con n nodi. Ciascuna coppia delle seguenti affermazioni implica la terza (e quindi ogni coppia rappresenta una *equivalente* ed alternativa definizione di albero).

- G è connesso.
- G non contiene cicli.
- G ha esattamente $n - 1$ archi.

Da ciò segue che un grafo non diretto con più di $n - 1$ archi dovrà *necessariamente* contenere dei cicli!

Ricordiamo il seguente fatto:
Sia G un grafo con n nodi. Ciascuna coppia delle seguenti affermazioni implica la terza (e quindi ogni coppia rappresenta una *equivalente* ed alternativa definizione di albero).

- G è connesso.
- G non contiene cicli.
- G ha esattamente $n-1$ archi.

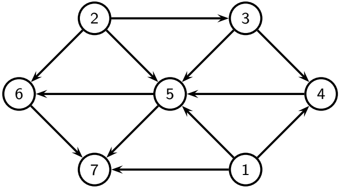
Da ciò segue che un grafo non diretto con più di $n-1$ archi dovrà *necessariamente* contenere dei cicli!

Da qui si può partire per arrivare ad un algoritmo che in tempo $O(|V|)$ decide se il grafo è aciclico o meno.

La questione è diversa su grafi diretti.

La questione è diversa su grafi diretti.

Es.

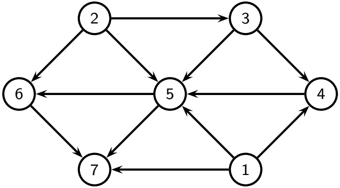


G

non è nè un albero nè ha cicli.

La questione è diversa su grafi diretti.

Es.

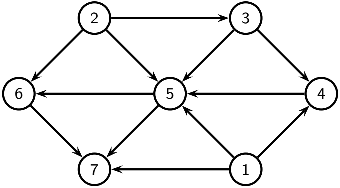


G

non è nè un albero nè ha cicli.

Pertanto in tali grafi non si può usare l'osservazione precedente ed occorre usare l'algoritmo `DFS-Segnala_cicli`(G), di complessità $O(|V| + |E|)$.

La questione è diversa su grafi diretti.
Es.



G

non è nè un albero nè ha cicli.
Pertanto in tali grafi non si può usare l'osservazione precedente ed occorre usare l'algoritmo `DFS-Segnala_cicli`(G), di complessità $O(|V| + |E|)$.

Esercizio: Modificare la visita BFS in modo tale che essa segnali l'eventuale esistenza di cicli, sempre in grafi *non diretti*.

Cicli che coinvolgono archi

Cicli che coinvolgono archi
Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

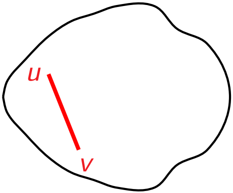
Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$

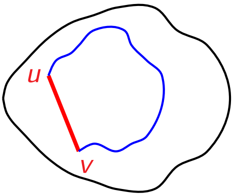


Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$

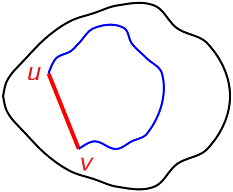


Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



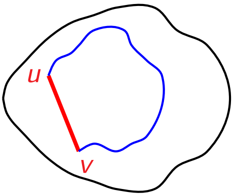
Deve esistere un'altro modo per andare da u a v,

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



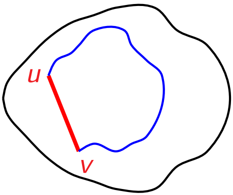
Deve esistere un'altro modo per andare da u a v, ovvero, se rimovessimo l'arco $e = (u, v)$ deve ancora esistere un cammino da u a v.

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



Deve esistere un'altro modo per andare da u a v, ovvero, se rimovessimo l'arco $e = (u, v)$ deve ancora esistere un cammino da u a v.

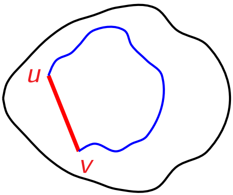
Per scoprire questo fatto, potremmo togliere l'arco $e = (u, v)$ da G,

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



Deve esistere un'altro modo per andare da u a v, ovvero, se rimovessimo l'arco $e = (u, v)$ deve ancora esistere un cammino da u a v.

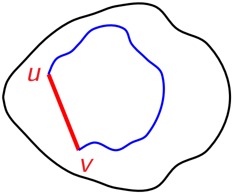
Per scoprire questo fatto, potremmo togliere l'arco $e = (u, v)$ da G, poi eseguire $\texttt{DFS}(u)$,

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



Deve esistere un'altro modo per andare da u a v, ovvero, se rimovessimo l'arco $e = (u, v)$ deve ancora esistere un cammino da u a v.

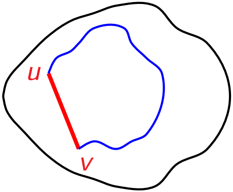
Per scoprire questo fatto, potremmo togliere l'arco $e = (u, v)$ da G, poi eseguire DFS(u), e se alla fine risulta che Esplorato(v)=True, ritornare "SI",

Cicli che coinvolgono archi

Input: Un grafo connesso, non diretto $G = (V, E)$, arco $(u, v) \in E$.

Output: "SI" se esiste un ciclo in G contenente l'arco (u, v), "NO", altrimenti.

Chiedamoci in che situazione esiste un ciclo che contiene un arco $e = (u, v) \in E$



Deve esistere un'altro modo per andare da u a v, ovvero, se rimovessimo l'arco $e = (u, v)$ deve ancora esistere un cammino da u a v.

Per scoprire questo fatto, potremmo togliere l'arco $e = (u, v)$ da G, poi eseguire $\texttt{DFS}(u)$, e se alla fine risulta che $\texttt{Esplorato}(v) = \texttt{True}$, ritornare "SI", ritornare "NO", altrimenti.

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di “antipatia” tra persone,

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di "antipatia" tra persone, è possibile assegnare ogni invitato ad uno dei due tavoli disponibili, in modo tale che nessuna persona sta allo stesso tavolo con una persona che ritiene antipatica?

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di "antipatia" tra persone, è possibile assegnare ogni invitato ad uno dei due tavoli disponibili, in modo tale che nessuna persona sta allo stesso tavolo con una persona che ritiene antipatica?

Rappresentiamo le persone e le mutue antipatie con un grafo $G = (V, E)$, in cui $V =$insieme delle persone

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di “antipatia” tra persone, è possibile assegnare ogni invitato ad uno dei due tavoli disponibili, in modo tale che nessuna persona sta allo stesso tavolo con una persona che ritiene antipatica?

Rappresentiamo le persone e le mutue antipatie con un grafo $G = (V, E)$, in cui $V =$insieme delle persone e $(a, b) \in E$ se e solo se le persone a e b sono in relazione di mutua antipatia.

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di "antipatia" tra persone, è possibile assegnare ogni invitato ad uno dei due tavoli disponibili, in modo tale che nessuna persona sta allo stesso tavolo con una persona che ritiene antipatica?

Rappresentiamo le persone e le mutue antipatie con un grafo $G = (V, E)$, in cui $V =$insieme delle persone e $(a, b) \in E$ se e solo se le persone a e b sono in relazione di mutua antipatia.

Il problema in questione è equivalente a trovare una partizione di V in due insiemi A, B tali che

- $A \cup B = V$, $A \cap B = \emptyset$

Problema del Matrimonio...

Dati un insieme di invitati ad un matrimonio ed una relazione (simmetrica) di "antipatia" tra persone, è possibile assegnare ogni invitato ad uno dei due tavoli disponibili, in modo tale che nessuna persona sta allo stesso tavolo con una persona che ritiene antipatica?

Rappresentiamo le persone e le mutue antipatie con un grafo $G = (V, E)$, in cui $V =$insieme delle persone e $(a, b) \in E$ se e solo se le persone a e b sono in relazione di mutua antipatia.

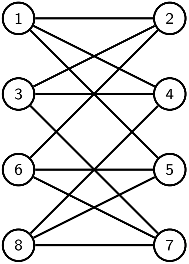
Il problema in questione è equivalente a trovare una partizione di V in due insiemi A, B tali che

- $A \cup B = V$, $A \cap B = \emptyset$ e
- $\forall \{a, b\} \in E$ vale che $a \in A$ e $b \in B$

Un grafo non diretto $G = (V, E)$ è detto bipartito se e solo se $V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Un grafo non diretto $G = (V, E)$ è detto bipartito se e solo se $V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

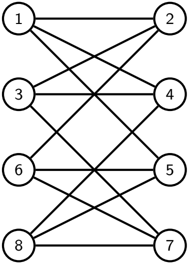
Esempio:



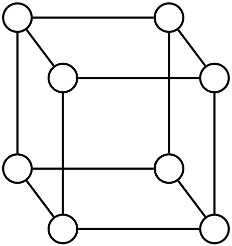
Il grafo è chiaramemte bipartito

Un grafo non diretto $G = (V, E)$ è detto bipartito se e solo se $V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Esempio:



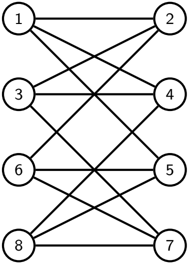
Il grafo è chiaramemte bipartito



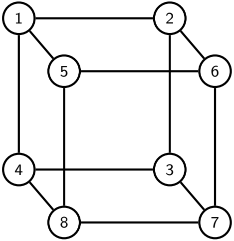
E questo lo è?

Un grafo non diretto $G = (V, E)$ è detto bipartito se e solo se $V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Esempio:



Il grafo è chiaramemte bipartito



Ovviamente sì...

Cerchiamo di capire come è fatto un grafo bipartito
$G = (V, E)$ è bipartito $\Leftrightarrow V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Cerchiamo di capire come è fatto un grafo bipartito
$G = (V, E)$ è bipartito $\Leftrightarrow$ $V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Equivalentemente, possiamo dire che $G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso (ad es. quelli in A) e Nero (quelli in B)

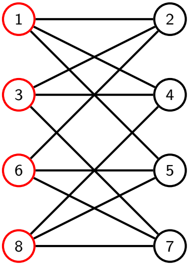
Cerchiamo di capire come è fatto un grafo bipartito
$G = (V, E)$ è bipartito $\Leftrightarrow V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Equivalentemente, possiamo dire che $G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso (ad es. quelli in A) e Nero (quelli in B) in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cerchiamo di capire come è fatto un grafo bipartito
$G = (V, E)$ è bipartito $\Leftrightarrow V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Equivalentemente, possiamo dire che $G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso (ad es. quelli in A) e Nero (quelli in B) in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

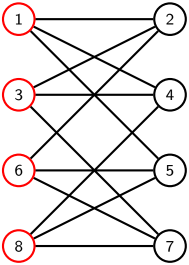
Esempio:

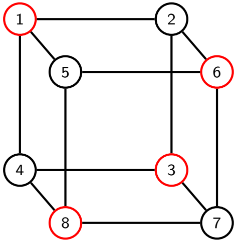


Cerchiamo di capire come è fatto un grafo bipartito
$G = (V, E)$ è bipartito $\Leftrightarrow V = A \cup B$, $A \neq \emptyset \neq B$, $A \cap B = \emptyset$, e $\forall \{u, v\} \in E$ vale che l'arco $\{u, v\}$ va da vertici in A a vertici in B.

Equivalentemente, possiamo dire che $G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso (ad es. quelli in A) e Nero (quelli in B) in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Esempio:





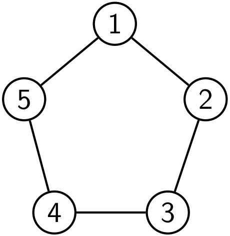
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)

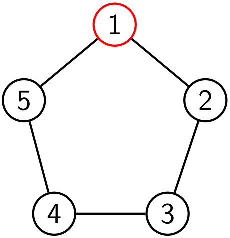
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



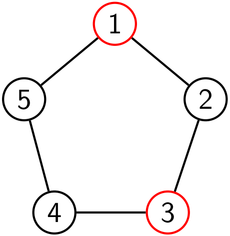
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



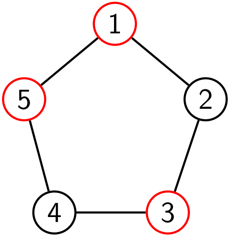
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



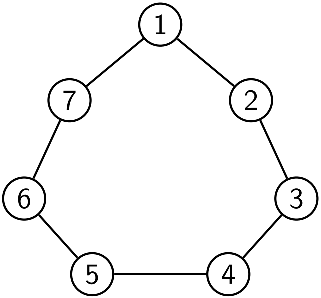
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



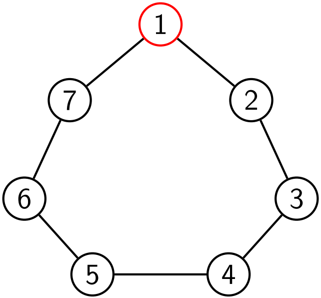
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



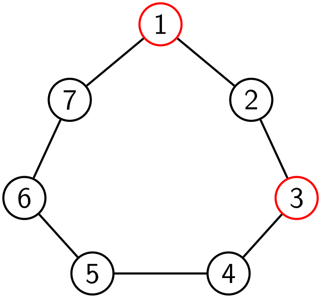
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



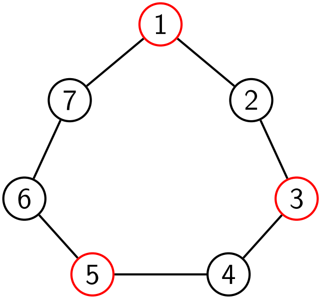
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



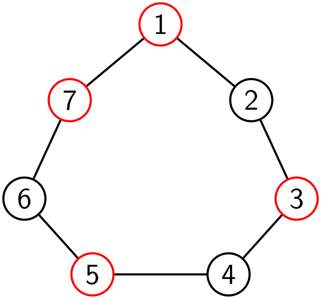
$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



$G = (V, E)$ è bipartito $\Leftrightarrow$ è possibile colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso.

Cicli di lunghezza dispari **non** possono essere colorati in modo siffatto (e quindi non sono bipartiti)
Esempio:



Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari
Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso.

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari
Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero.

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari
Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso,

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari.

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari
Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari. Vi ricorda qualcosa?

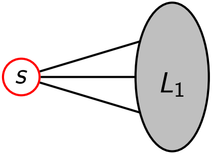
Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari. Vi ricorda qualcosa?

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari
Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari. Vi ricorda qualcosa?

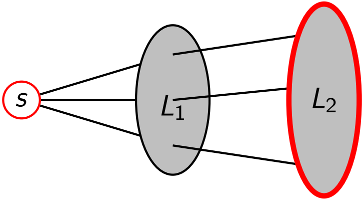


nodi a distanza 1 da s

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari. Vi ricorda qualcosa?



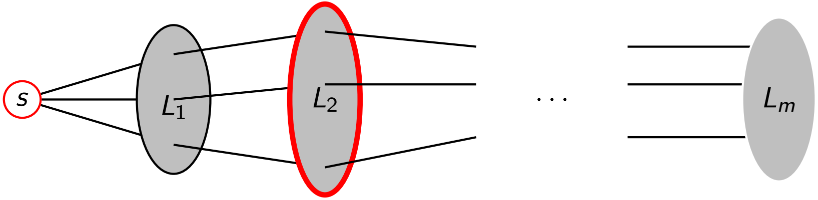
nodi a distanza 1 da s

nodi a distanza 2 da s

Quindi, se un grafo è bipartito, non può contenere cicli di lunghezza dispari

Un semplice algoritmo per *tentare* di colorare i vertici di G con i colori Rosso e Nero in modo che gli archi vadano esclusivamente tra vertici di colore diverso potrebbe essere il seguente:

Parti da un nodo arbitrario s e coloralo di Rosso. I suoi vicini devono essere necessariamente essere colorati Nero. I vicini di tali nodi devono essere colorati necessriamente Rosso, e così via, fino all'ultimo livello, che sarà colorato Rosso se è pari, Nero se è dispari. Vi ricorda qualcosa?



nodi a distanza 1 da s

nodi a distanza 2 da s

nodi a distanza massima m da s

Modifichiamo BFS(G, s) per colorare i vertici di G

```
BFS(G, s)
1.   Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.   Inizializza L[0] in modo che contenga s;
       color[s] ← Rosso
3.   Poni il contatore dei livelli i a 0
```

Modifichiamo BFS(G, s) per colorare i vertici di G

```
BFS(G, s)
1.  Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.  Inizializza L[0] in modo che contenga s;
      color[s] ← Rosso
3.  Poni il contatore dei livelli i a 0
4.  While L[i] non è vuota
5.    Inizializza una lista vuota L[i + 1]
```

Modifichiamo BFS(G, s) per colorare i vertici di G

BFS(G, s)
1. Poni Scoperto[s] $=$true e Scoperto[v] $=$false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s;
   color[s] $\leftarrow$ Rosso
3. Poni il contatore dei livelli i a 0
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Estrai u da $L[i]$
8.   For ogni arco (u, v) incidente su u
9.   If Scoperto[v]=false then
10.   Poni Scoperto[v]=true

Modifichiamo BFS(G, s) per colorare i vertici di G

BFS(G, s)

1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s;
   color$[s] \leftarrow$ Rosso
3. Poni il contatore dei livelli i a 0
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Estrai u da $L[i]$
8.   For ogni arco (u, v) incidente su u
9.   If Scoperto$[v]$=false then
10.   Poni Scoperto$[v]$=true
11.   Aggiungi v alla lista $L[i+1]$; color$[v] \leftarrow$Rosso
           se $i+1$ è pari, color$[v] \leftarrow$Nero altrimenti

Modifichiamo BFS(G, s) per colorare i vertici di G

BFS(G, s)
1. Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s;
   color$[s] \leftarrow$ Rosso
3. Poni il contatore dei livelli i a 0
4. While $L[i]$ non è vuota
5. Inizializza una lista vuota $L[i+1]$
6. For ogni nodo $u \in L[i]$
7. Estrai u da $L[i]$
8. For ogni arco (u, v) incidente su u
9. If Scoperto$[v]$=false then
10. Poni Scoperto$[v]$=true
11. Aggiungi v alla lista $L[i+1]$; color$[v] \leftarrow$Rosso
    se $i+1$ è pari, color$[v] \leftarrow$Nero altrimenti
12. Incrementa il contatore di livelli i di uno

Modifichiamo BFS(G, s) per colorare i vertici di G

```
BFS(G, s)
1.   Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.   Inizializza L[0] in modo che contenga s;
       color[s] ← Rosso
3.   Poni il contatore dei livelli i a 0
4.   While L[i] non è vuota
5.     Inizializza una lista vuota L[i + 1]
6.     For ogni nodo u ∈ L[i]
7.       Estrai u da L[i]
8.       For ogni arco (u, v) incidente su u
9.       If Scoperto[v]=false then
10.         Poni Scoperto[v]=true
11.         Aggiungi v alla lista L[i + 1]; color[v] ←Rosso
             se i + 1 è pari, color[v] ←Nero altrimenti
12.    Incrementa il contatore di livelli i di uno
```

Tempo: $\Theta(|V| + |E|)$

Quando funziona l'algoritmo?

Quando funziona l'algoritmo?

Sia G un grafo e siano $L[1], L[2], \ldots$ i livelli prodotti da BFS partendo dal nodo s.

Quando funziona l'algoritmo?

Sia G un grafo e siano $L[1], L[2], \ldots$ i livelli prodotti da BFS partendo dal nodo s. Vale:

1. Se non esiste alcun arco di G che unisce nodi di uno stesso livello, allora G è bipartito e l'algoritmo ritorna una colorazione di vertici tale che gli archi vanno solo da vertici Rossi a vertici Neri.

Quando funziona l'algoritmo?

Sia G un grafo e siano $L[1], L[2], \ldots$ i livelli prodotti da BFS partendo dal nodo s. Vale:

1. Se non esiste alcun arco di G che unisce nodi di uno stesso livello, allora G è bipartito e l'algoritmo ritorna una colorazione di vertici tale che gli archi vanno solo da vertici Rossi a vertici Neri.
2. Se esiste un arco che collega due vertici dello stesso livello (e quindi cui l'algoritmo ha assegnato lo stesso colore) allora esiste sicuramente un ciclo dispari in G ed il grafo non è bipartito.

Quando funziona l'algoritmo?

Sia G un grafo e siano $L[1], L[2], \ldots$ i livelli prodotti da BFS partendo dal nodo s. Vale:

1. Se non esiste alcun arco di G che unisce nodi di uno stesso livello, allora G è bipartito e l'algoritmo ritorna una colorazione di vertici tale che gli archi vanno solo da vertici Rossi a vertici Neri.
2. Se esiste un arco che collega due vertici dello stesso livello (e quindi cui l'algoritmo ha assegnato lo stesso colore) allora esiste sicuramente un ciclo dispari in G ed il grafo non è bipartito.

Per provare le affermazioni di sopra, ricordiamo il seguente:

Fatto: Sia x un nodo che appare nel livello $L[i]$ e y un nodo del livello $L[j]$. Se x e y sono uniti da un arco del grafo G allora o $L[i]$ e $L[j]$ sono lo *stesso* livello o sono due livelli *consecutivi*.

Prova delle affermazioni.

Prova delle affermazioni.
Poichè l'ipotesi in 1. è che non esiste alcun arco di G che unisce nodi di uno stesso livello, dal Fatto otteniamo che se (x, y) è un qualsiasi arco di G, allora i suoi estremi x e y appartengono a livelli consecutivi.

Prova delle affermazioni.
Poichè l'ipotesi in 1. è che non esiste alcun arco di G che unisce nodi di uno stesso livello, dal Fatto otteniamo che se (x, y) è un qualsiasi arco di G, allora i suoi estremi x e y appartengono a livelli consecutivi.

Ma l'algoritmo assegna colori diversi a nodi in livelli consecutivi, pertanto ogni arco di G ha estremi di colore diverso, e quindi G è bipartito.

Prova delle affermazioni.
Poichè l'ipotesi in 1. è che non esiste alcun arco di G che unisce nodi di uno stesso livello, dal Fatto otteniamo che se (x, y) è un qualsiasi arco di G, allora i suoi estremi x e y appartengono a livelli consecutivi.

Ma l'algoritmo assegna colori diversi a nodi in livelli consecutivi, pertanto ogni arco di G ha estremi di colore diverso, e quindi G è bipartito.

Se siamo invece sotto le ipotesi di 2. (ovvero che esiste un arco tra due nodi x e y appartenenti ad uno stesso livello $L[i]$) ciò vuol dire innanzitutto che nell'albero prodotto dalla BFS esiste un cammino $p(x)$ da s a x di lunghezza i ed un cammino $p(y)$ da s a y della stessa lunghezza i.

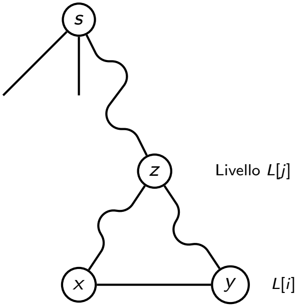
Prova delle affermazioni.
Poichè l'ipotesi in 1. è che non esiste alcun arco di G che unisce nodi di uno stesso livello, dal Fatto otteniamo che se (x, y) è un qualsiasi arco di G, allora i suoi estremi x e y appartengono a livelli consecutivi.

Ma l'algoritmo assegna colori diversi a nodi in livelli consecutivi, pertanto ogni arco di G ha estremi di colore diverso, e quindi G è bipartito.

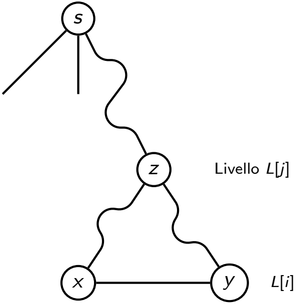
Se siamo invece sotto le ipotesi di 2. (ovvero che esiste un arco tra due nodi x e y appartenenti ad uno stesso livello $L[i]$) ciò vuol dire innanzitutto che nell'albero prodotto dalla BFS esiste un cammino $p(x)$ da s a x di lunghezza i ed un cammino $p(y)$ da s a y della stessa lunghezza i.

Questi due cammini potrebbero avere una prima parte in comune (diciamo di lunghezza $j < i$ e poi le restanti parti di lunghezza $(i - j)$ differenti.

Ovvero avremmo una situazione di questo tipo



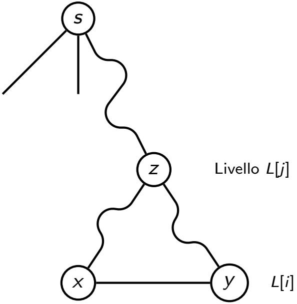
Ovvero avremmo una situazione di questo tipo



In altri termini, avremmo un ciclo che coinvolge x, y e z di lunghezza $(i-j)+(i-j)+1$ dispari!

In conclusione, se l'algoritmo ritorna una colorazione dei vertici con i colori Rosso e Nero in modo che tutti gli archi hanno estremi di colore diverso, allora il grafo è chiaramente bipartito.

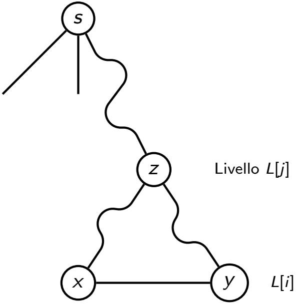
Ovvero avremmo una situazione di questo tipo



In altri termini, avremmo un ciclo che coinvolge x, y e z di lunghezza $(i-j)+(i-j)+1$ dispari!

In conclusione, se l'algoritmo ritorna una colorazione dei vertici con i colori Rosso e Nero in modo che tutti gli archi hanno estremi di colore diverso, allora il grafo è chiaramente bipartito. Se l'algoritmo "sbaglia", ovvero ritorna una colorazione dei vertici con i colori Rosso e Nero in modo che almeno un arco ha estremi dello stesso colore,

Ovvero avremmo una situazione di questo tipo



In altri termini, avremmo un ciclo che coinvolge x, y e z di lunghezza $(i-j)+(i-j)+1$ dispari!

In conclusione, se l'algoritmo ritorna una colorazione dei vertici con i colori Rosso e Nero in modo che tutti gli archi hanno estremi di colore diverso, allora il grafo è chiaramente bipartito. Se l'algoritmo "sbaglia", ovvero ritorna una colorazione dei vertici con i colori Rosso e Nero in modo che almeno un arco ha estremi dello stesso colore, allora vuol dire che nel grafo esiste un ciclo di lunghezza dispari e di conseguenza il grafo non è bipartito.

Calcolo dei vicini dei vicini di un nodo

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi vicini (adiacenti) di s?

Calcolo dei vicini dei vicini di un nodo

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi vicini (adiacenti) di s?

Risposta: basta contare il numero dei nodi che sono nella lista di adiacenza di s.

Calcolo dei vicini dei vicini di un nodo

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi vicini (adiacenti) di s?

Risposta: basta contare il numero dei nodi che sono nella lista di adiacenza di s.

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi adiacenti ai vicini di s?

Calcolo dei vicini dei vicini di un nodo

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi vicini (adiacenti) di s?

Risposta: basta contare il numero dei nodi che sono nella lista di adiacenza di s.

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi adiacenti ai vicini di s?

Risposta: basta sommare il numero di nodi che sono nella lista di adiacenza di ciascun nodo vicino di s.

Calcolo dei vicini dei vicini di un nodo

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi vicini (adiacenti) di s?

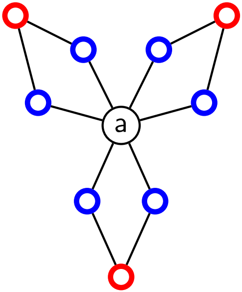
Risposta: basta contare il numero dei nodi che sono nella lista di adiacenza di s.

Domanda: dato un grafo $G = (V, E)$ ed un nodo $s \in V$, quanti sono i nodi adiacenti ai vicini di s?

Risposta: basta sommare il numero di nodi che sono nella lista di adiacenza di ciascun nodo vicino di s.

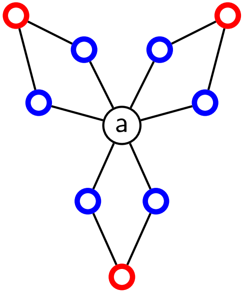
Sicuro?

Vediamo un esempio



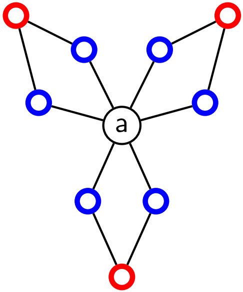
Il nodo **a** ha 6 vicini e 3 nodi vicini ai suoi vicini (escludendo **a** stesso)

Vediamo un esempio



Il nodo **a** ha 6 vicini e 3 nodi vicini ai suoi vicini (escludendo **a** stesso) mentre la somma del numero dei nodi vicini al nodo **a** più i nodi nelle liste di adiacenze dei vicini di **a** sarebbe pari a 6 (in quanto i nodi rossi sarebbero contati due volte).

Vediamo un esempio



Il nodo **a** ha 6 vicini e 3 nodi vicini ai suoi vicini (escludendo **a** stesso) mentre la somma del numero dei nodi vicini al nodo **a** più i nodi nelle liste di adiacenze dei vicini di **a** sarebbe pari a 6 (in quanto i nodi rossi sarebbero contati due volte).

Possiamo modificare BFS, in modo tale che esplori solo i primi due livelli del grafo, e conti i nodi incontrati nell'esplorazione.

Calcolo dei vicini dei vicini di un nodo

BFS(T, s)

1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] \leftarrow 0$
3. Poni i a 0; $T = \emptyset$, tot$= 0$
4. While $L[i]$ non è vuota AND $i \leq 1$
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$
7.   Considera ciascun arco (u, v) incidente su u
8.   If Scoperto[v]=false then
9.   Poni Scoperto[v]=true
10.   Aggiungi l'arco (u, v) all'albero T
11.   Aggiungi v alla lista $L[i+1]$; $d[v] \leftarrow i+1$,
        tot=tot$+1$
12.   Incrementa il contatore di livelli i di uno
13. return(tot)

Calcolo dei vicini dei vicini di un nodo

```
BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
Inizializza L[0] in modo che contenga s; d[s] ← 0
Poni i a 0; T = ∅, tot= 0
While L[i] non è vuota AND i ≤ 1
  Inizializza una lista vuota L[i + 1]
  For ogni nodo u ∈ L[i]
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
       Aggiungi l'arco (u, v) all'albero T
       Aggiungi v alla lista L[i + 1]; d[v] ← i + 1,
            tot=tot+1
  Incrementa il contatore di livelli i di uno
return(tot)
```

Complessità: $O(|V| + |E|)$

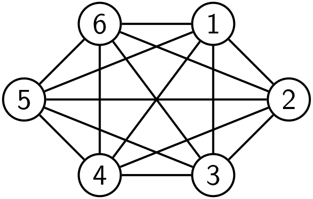
Orientazione aciclica di grafi completi

Un grafo non diretto $G = (V, E)$ è detto *completo* se esiste un arco tra ogni coppia di vertici.

Orientazione aciclica di grafi completi

Un grafo non diretto $G = (V, E)$ è detto *completo* se esiste un arco tra ogni coppia di vertici.

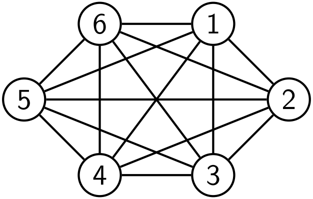
Esempio:



Orientazione aciclica di grafi completi

Un grafo non diretto $G = (V, E)$ è detto *completo* se esiste un arco tra ogni coppia di vertici.

Esempio:

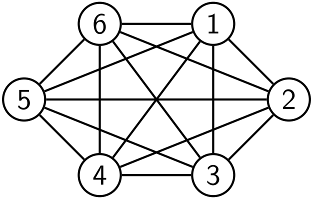


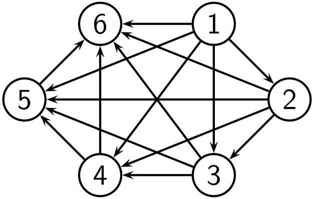
Problema: orientare gli archi del grafo in modo che non vi siano cicli.

Orientazione aciclica di grafi completi

Un grafo non diretto $G = (V, E)$ è detto *completo* se esiste un arco tra ogni coppia di vertici.

Esempio:





Problema: orientare gli archi del grafo in modo che non vi siano cicli.

Possiamo usare ciò che apprendemmo al tempo dei DAG, ove osservammo che un grafo diretto senza cicli ha sicuramente un nodo in cui non entrano archi.

Possiamo usare ciò che apprendemmo al tempo dei DAG, ove osservammo che un grafo diretto senza cicli ha sicuramente un nodo in cui non entrano archi.

Preso quindi un nodo arbitrario u del grafo completo G, potremmo soddisfare questa condizione orientando tutti gli archi di u “all’infuori”, cosicchè sicuramente nessun ciclo passerà mai per u.

Possiamo usare ciò che apprendemmo al tempo dei DAG, ove osservammo che un grafo diretto senza cicli ha sicuramente un nodo in cui non entrano archi.

Preso quindi un nodo arbitrario u del grafo completo G, potremmo soddisfare questa condizione orientando tutti gli archi di u "all'infuori", cosicchè sicuramente nessun ciclo passerà mai per u.

Avendo sistemato u, prendiamo un'altro arbitrario vertice v che ha ancora qualche arco non diretto incidente su di esso, ed orientiamo tutti gli archi non diretti di v "all'infuori", cosicchè sicuramente nessun ciclo passerà mai neanche per v.

Possiamo usare ciò che apprendemmo al tempo dei DAG, ove osservammo che un grafo diretto senza cicli ha sicuramente un nodo in cui non entrano archi.

Preso quindi un nodo arbitrario u del grafo completo G, potremmo soddisfare questa condizione orientando tutti gli archi di u "all'infuori", cosicchè sicuramente nessun ciclo passerà mai per u.

Avendo sistemato u, prendiamo un'altro arbitrario vertice v che ha ancora qualche arco non diretto incidente su di esso, ed orientiamo tutti gli archi non diretti di v "all'infuori", cosicchè sicuramente nessun ciclo passerà mai neanche per v.

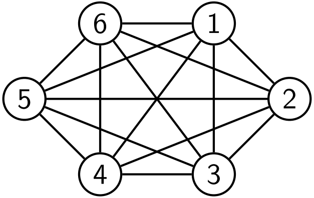
Itereremo poi sul resto dei nodi...

1. `Numera i vertici da 1 a` n

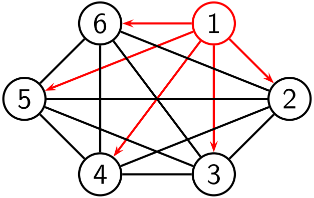
1. Numera i vertici da 1 a n
2. For $i = 1$ to $n - 1$ do

1. `Numera i vertici da 1 a` n
2. `For` $i = 1$ `to` $n - 1$ `do`
3. `orienta gli archi non diretti incidenti su` i `‘‘all’infuori’’`

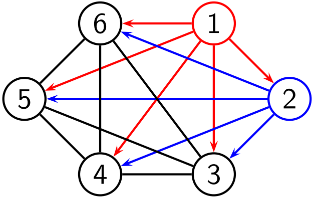
1. Numera i vertici da 1 a n
2. For $i = 1$ to $n - 1$ do
3. orienta gli archi non diretti incidenti su i ‘‘all’infuori’’



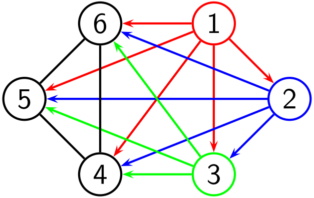
1. `Numera i vertici da 1 a` n
2. `For` $i = 1$ `to` $n - 1$ `do`
3. `orienta gli archi non diretti incidenti su` i `‘‘all’infuori’’`



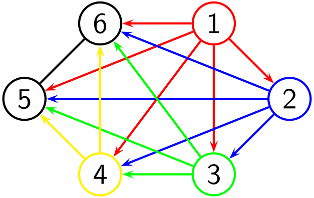
1. Numera i vertici da 1 a n
2. For $i = 1$ to $n - 1$ do
3. orienta gli archi non diretti incidenti su i
   ‘‘all’infuori’’



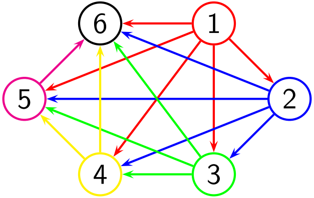
1. `Numera i vertici da 1 a` n
2. `For` $i = 1$ `to` $n - 1$ `do`
3. `orienta gli archi non diretti incidenti su` i `‘‘all’infuori’’`



1. Numera i vertici da 1 a n
2. For $i = 1$ to $n - 1$ do
3. orienta gli archi non diretti incidenti su i ‘‘all’infuori’’



1. Numera i vertici da 1 a n
2. For $i = 1$ to $n - 1$ do
3. orienta gli archi non diretti incidenti su i ``all'infuori''



Numero di cammini minimi tra due nodi

Dato un grafo non orientato e connesso $G = (V, E)$ vogliamo calcolare il numero di cammini minimi distinti che vanno da un nodo sorgente s a ciascun altro nodo u

Numero di cammini minimi tra due nodi

Dato un grafo non orientato e connesso $G = (V, E)$ vogliamo calcolare il numero di cammini minimi distinti che vanno da un nodo sorgente s a ciascun altro nodo u

Due cammini si considerano diversi se differiscono per almeno per un arco

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s;

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente,

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo s (che ha distanza 0 da se stesso),

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo s (che ha distanza 0 da se stesso), poi i nodi adiacenti a distanza 1,

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo s (che ha distanza 0 da se stesso), poi i nodi adiacenti a distanza 1, poi quelli a distanza 2 e così via.

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo s (che ha distanza 0 da se stesso), poi i nodi adiacenti a distanza 1, poi quelli a distanza 2 e così via.

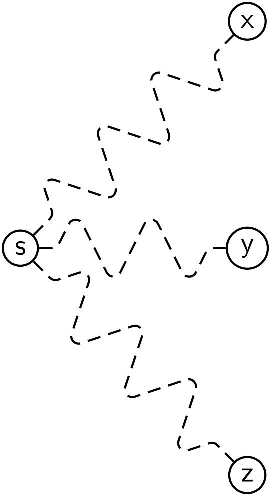
Supponiamo di aver già calcolato il numero di cammini minimi $c[u]$ per ogni nodo u che si trovi a distanza $d[u] = k$ dalla sorgente s.

L'algoritmo di visita in ampiezza (BFS) può essere utilizzato per individuare un cammino minimo tra un nodo sorgente s e ciascun nodo raggiungibile da s; possiamo però estenderlo per calcolare il numero di cammini minimi distinti tra s e i nodi da esso raggiungibili.

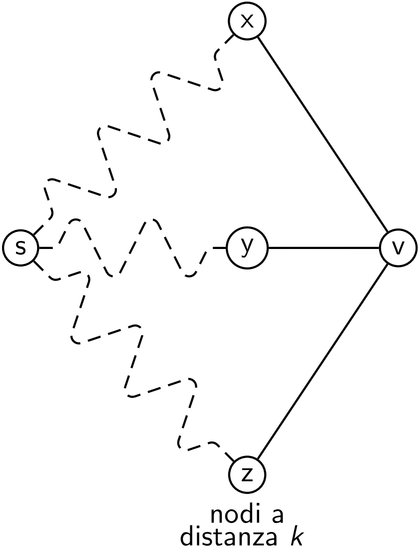
Ricordiamo che l'algoritmo BFS visita i nodi in ordine di distanza non decrescente dalla sorgente, ossia viene prima visitato il nodo s (che ha distanza 0 da se stesso), poi i nodi adiacenti a distanza 1, poi quelli a distanza 2 e così via.

Supponiamo di aver già calcolato il numero di cammini minimi $c[u]$ per ogni nodo u che si trovi a distanza $d[u] = k$ dalla sorgente s. Il numero di cammini minimi che portano ad un generico nodo v che si trova a distanza $k + 1$ può essere espresso come:

$$c[v] = \sum_{\{u,v\} \in E : d[u] = k} c[u].$$



nodi a
distanza k


x
s
y
v
z
nodi a
distanza k

Possiamo calcolare il valore $c[v]$ per ogni nodo v mano a mano che il grafo viene visitato.

Possiamo calcolare il valore $c[v]$ per ogni nodo v mano a mano che il grafo viene visitato.

Poniamo inizialmente $c[v] = 0$ per ogni v, ad eccezione della sorgente s per cui poniamo $c[s] = 1$.

Possiamo calcolare il valore $c[v]$ per ogni nodo v mano a mano che il grafo viene visitato.

Poniamo inizialmente $c[v] = 0$ per ogni v, ad eccezione della sorgente s per cui poniamo $c[s] = 1$.

Ogni volta che l'algoritmo BFS attraversa l'arco non orientato $\{u, v\}$ che conduce dal nodo u ad un nodo v a distanza $d[v] = d[u] + 1$, poniamo $c[v] = c[v] + c[u]$.

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
3. Poni i a 0; $S = \emptyset$, $c[v] = 0$ $\forall v \neq s$

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
3. Poni i a 0; $S = \emptyset$, $c[v] = 0$ $\forall v \neq s$
4. While $L[i]$ non è vuota

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
3. Poni i a 0; $S = \emptyset$, $c[v] = 0 \ \forall v \neq s$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i + 1]$

L'algoritmo

BFS(T, s)
1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
3. Poni i a 0; $S = \emptyset$, $c[v] = 0 \ \forall v \neq s$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i+1]$
6.   For ogni nodo $u \in L[i]$

L'algoritmo

```
BFS(T, s)
1.  Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
2.  Inizializza L[0] in modo che contenga s; d[s] = 0, c[s] = 1
3.  Poni i a 0; S = ∅, c[v] = 0 ∀v ≠ s
4.  While L[i] non è vuota
5.    Inizializza una lista vuota L[i + 1]
6.    For ogni nodo u ∈ L[i]
7.      Considera ciascun arco (u, v) incidente su u
```

L'algoritmo

BFS(T, s)

Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$

Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$

Poni i a 0; $S = \emptyset$, $c[v] = 0$ $\forall v \neq s$

While $L[i]$ non è vuota

  Inizializza una lista vuota $L[i+1]$

  For ogni nodo $u \in L[i]$

    Considera ciascun arco (u, v) incidente su u

    If Scoperto$[v]$=false then

L'algoritmo

BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
Poni i a 0; $S = \emptyset$, $c[v] = 0 \ \forall v \neq s$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true

L'algoritmo

```
BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
Inizializza L[0] in modo che contenga s; d[s] = 0, c[s] = 1
Poni i a 0; S = ∅, c[v] = 0 ∀v ≠ s
While L[i] non è vuota
  Inizializza una lista vuota L[i + 1]
  For ogni nodo u ∈ L[i]
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
       Aggiungi l'arco (u, v) all'albero S
```

L'algoritmo

```
BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false ∀v ≠ s
Inizializza L[0] in modo che contenga s; d[s] = 0, c[s] = 1
Poni i a 0; S = ∅, c[v] = 0 ∀v ≠ s
While L[i] non è vuota
  Inizializza una lista vuota L[i + 1]
  For ogni nodo u ∈ L[i]
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
       Aggiungi l'arco (u, v) all'albero S
       Aggiungi v alla lista L[i + 1]; d[v] = i + 1,
```

L'algoritmo

BFS(T, s)

1. Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
2. Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
3. Poni i a 0; $S = \emptyset$, $c[v] = 0$ $\forall v \neq s$
4. While $L[i]$ non è vuota
5.   Inizializza una lista vuota $L[i + 1]$
6.   For ogni nodo $u \in L[i]$
7.   Considera ciascun arco (u, v) incidente su u
8.   If Scoperto[v]=false then
9.   Poni Scoperto[v]=true
10.   Aggiungi l'arco (u, v) all'albero S
11.   Aggiungi v alla lista $L[i + 1]$; $d[v] = i + 1$,
12.   If $d[v] = d[u] + 1$ then $c[v] = c[v] + c[u]$

L'algoritmo

BFS(T, s)
Poni Scoperto[s] =true e Scoperto[v] =false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
Poni i a 0; $S = \emptyset$, $c[v] = 0 \ \forall v \neq s$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto[v]=false then
      Poni Scoperto[v]=true
      Aggiungi l'arco (u, v) all'albero S
      Aggiungi v alla lista $L[i+1]$; $d[v] = i + 1$,
     If $d[v] = d[u] + 1$ then $c[v] = c[v] + c[u]$
  Incrementa il contatore di livelli i di uno

L'algoritmo

BFS(T, s)
Poni Scoperto$[s]$ =true e Scoperto$[v]$ =false $\forall v \neq s$
Inizializza $L[0]$ in modo che contenga s; $d[s] = 0, c[s] = 1$
Poni i a 0; $S = \emptyset$, $c[v] = 0 \ \forall v \neq s$
While $L[i]$ non è vuota
  Inizializza una lista vuota $L[i+1]$
  For ogni nodo $u \in L[i]$
    Considera ciascun arco (u, v) incidente su u
    If Scoperto$[v]$=false then
      Poni Scoperto$[v]$=true
       Aggiungi l'arco (u, v) all'albero S
       Aggiungi v alla lista $L[i+1]$; $d[v] = i + 1$,
     If $d[v] = d[u] + 1$ then $c[v] = c[v] + c[u]$
  Incrementa il contatore di livelli i di uno
return$(c[v], \forall v \in V)$