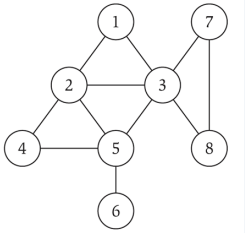
Grafi non orientati

Notazione. $G = (V, E)$

- V = nodi (o vertici).
- E = archi (o lati) tra coppie di nodi.
- Modella relazioni definite tra coppie di oggetti.
- Taglia di un grafo: numero di nodi e archi,: $n = |V|, m = |E|$.
- due nodi si dicono vicini se esiste un arco tra di essi
- il grado di un nodo v, indicato con $d(v)$, è il numero dei suoi vicini



$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$

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

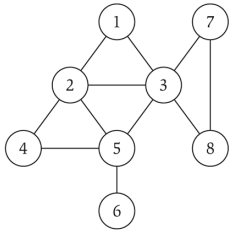
$m = 11, n = 8$

Grafi (non orientati)

Una prima proprietà fondamentale: Per ogni grafo $G = (V, E)$ la somma dei gradi dei vertici di G è uguale al doppio del numero di archi di G

- $\Sigma_{u \in V} d(u) = 2m = 2|E|$

ogni arco (u, v) viene contato due volte nella somma: una volta in *d(u)* e una volta in *d(v)*



$d(1) = 2, \quad d(2) = 4, \quad d(3) = 5, \quad d(4) = 2,$

$d(5) = 4, \quad d(6) = 1, \quad d(7) = 2, \quad d(8) = 2$

$m = |E| = 11$

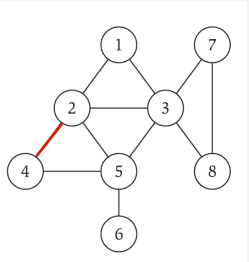
Tipiche applicazioni di modelli basati su grafi

grafo	nodi	archi
comunicazioni	telefoni, computer	cavi di fibra ottica
circuiti	porte, registri, processori	cavi
dispositivi meccanici	articolazioni	assi, raggi, molle
finanza	azioni, valuta	transazioni
trasporti	incroci, aeroporti	autostrade, rotte aeree
internet	ISP o AS	interconnessioni
giochi	posizioni	mosse possibili
relazioni sociali	individui, attori	amicizie, co-partecipazione
reti neuronali	neuroni	sinapsi
reti di proteine	proteine	interazione proteina-proteina
molecole	atomi	legami

Rappresentazione di Grafi: matrice delle adiacenze

Matrice delle Adiacenze. matrice n-per-n con $A_{uv} = 1$ se (u, v) è un arco.

- Due elementi per ogni arco.
- Spazio proporzionale a n^2.
- Testare se esiste un arco (u, v) richiede tempo $\Theta(1)$.
- Identificare tutti gli archi richiede tempo $\Theta(n^2)$.



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

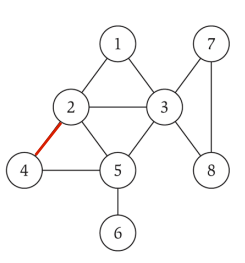
Rappresentazione di Grafi: matrice delle adiacenze

Grafo-linea. n vertici e n-1 archi.

- matrice delle adiacenze contiene quasi tutti 0.

Rete sociale (es. facebook).

- ~1 miliardo di vertici
- assunzione: ogni persona ha circa 130 amici
- -> 10^{18} byte per memorizzare 65x10^9 archi



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

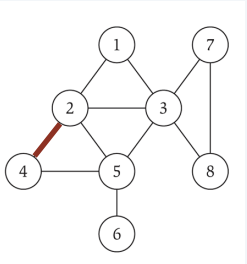
Rappresentazione di Grafi: liste di adiacenza

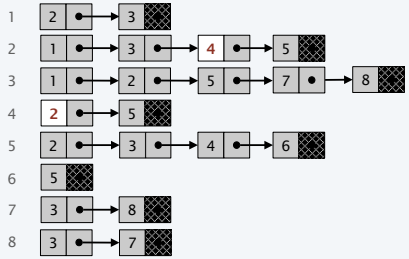
Liste di adiacenza. Un array di liste indicizzate per vertice.

- Adj[v]: lista di adiacenza di v contiene tutti i vicini di v

- Due elementi per ogni arco.
- Spazio necessario $\Theta(m + n)$.
- Testare se esiste un arco (u, v) richiede $O(d(u))$.
- Identificare tutti gli archi richiede $\Theta(m + n)$.

grado = numero di vicini di u





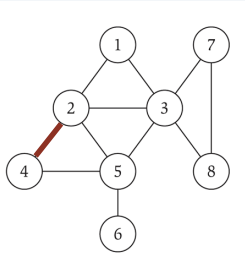
Rappresentazione di Grafi: liste di adiacenza

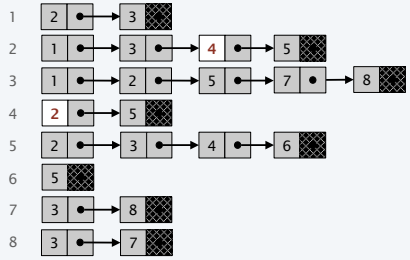
Grafo-linea. n vertici e n-1 archi.

- un vettore di taglia n e n liste di taglia 1 ~ 2n elementi .

Rete sociale (es. facebook).

- ~1 miliardo di vertici
- assunzione: ogni persona ha circa 130 amici
- -> 1.5x10^{11} byte per memorizzare circa 65x10^9 archi



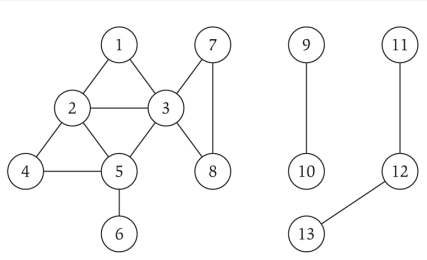


Cammini e connettività

Def. Un cammino in un grafo (non orientato) $G = (V, E)$ è una sequenza di nodi $v_1, v_2, \ldots, v_k$ con la proprietà che ogni coppia di nodi consecutivi v_{i-1}, v_i risulta connessa da un arco in E.

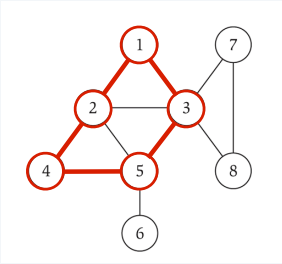
Def. Un cammino è semplice se tutti i nodi sono distinti.

Def. Un grafo non orientato si dice connesso se esiste un cammino tra ogni coppia di nodi u e v.



Cicli

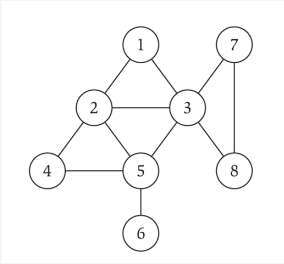
Def. Un ciclo è un cammino $v_1, v_2, \ldots, v_k$ in cui $v_1 = v_k$, $k > 2$, e i primi $k-1$ nodi sono tutti distinti.



ciclo C = 1-2-4-5-3-1

Distanza tra nodi

Def. La distanza tra due vertici s, t in un grafo G è il numero di archi nel cammino più corto tra s e t in G.



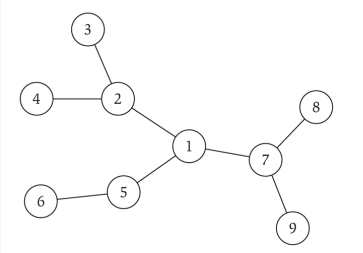
Distanza tra 1 e 4 = 2
Distanza tra 7 e 4 = 3
Distanza tra 7 e 8 = 1

Alberi

Def. Un grafo (non orientato) è un albero se è connesso e non contiene alcun ciclo.

Teorema. Sia G un grafo connesso su n nodi. Ogni coppia delle seguenti affermazioni implica la terza.

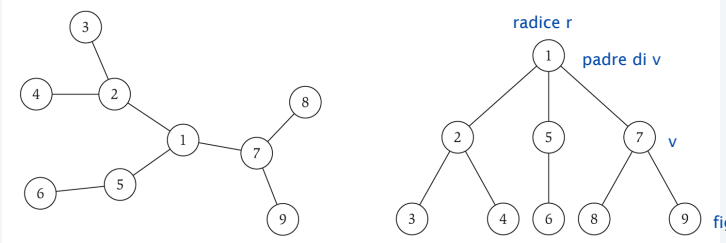
- G è connesso.
- G non presenta cicli.
- G ha $n-1$ archi.



Alberi radicati (rooted trees)

Dato un albero T, scegliamo una radice r e orientiamo ogni arco in direzione opposta a quella dove si trova r.

Importanza. Modelli di strutture gerarchiche.

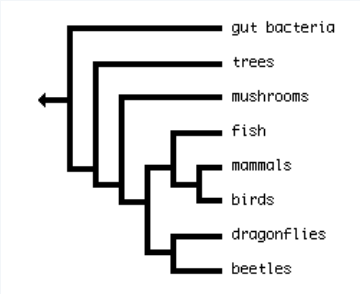


un albero

lo stesso albero, radicato in 1

Alberi filogenetici

Descrizione di processi evolutivi.



CONNETTIVITÀ E VISITE BFS E DFS

- Problemi di connettività
- visita in ampiezza (BFS)
- la struttura dati coda
- visita in profondità (DFS)
- la struttura dati pila

Connettività

Problema della connettività tra due nodi. Dati due nodi s e t, esiste un cammino **tra** s **e** t?

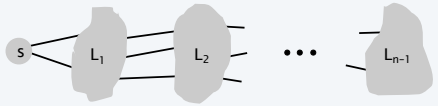
Problema del cammino minimo tra due nodi. Dati due nodi s e t, qual è la lunghezza del più piccolo cammino **tra** s **e** t?

Esempi.

- Analisi di relazioni definite da reti sociali.
- Attraversamento di labirinti.
- Kevin Bacon number.
- Minimo numero di “hop” in una rete di comunicazioni.

Visita per ampiezza: Breadth-first search (BFS)

BFS - l’idea. Esplora partendo da s in tutte le possibili direzioni, suddividendo/visitando i nodi per "livello".



Algoritmo BFS.

- $L_0 = \{ s \}$.
- L_1 = tutti i vicini di L_0.
- L_2 = tutti i nodi non in L_0 o L_1, e che sono vicini di un nodo in L_1.
- L_{i+1} = tutti i nodi che non appartengono ad un livello precedente, e che sono vicini di (sono collegati da un arco ad) un nodo di L_i.

Teorema. Per ogni i, L_i consiste di tutti i nodi a distanza esattamente i da s. Esiste un cammino da s a t se e solo se t appare in qualche livello.

Visita per ampiezza: Albero BFS

Def. Un albero BFS per un grafo G = (V,E) è l'albero "determinato" da una visita BFS su G

- La radice è il punto di partenza della visita
- un nodo u è il padre di un nodo v se v viene raggiunto per la prima volta mentre l'algoritmo esplora i vicini di u

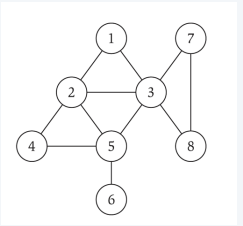
Nota. Per uno stesso grafo esistono diversi alberi BFS.

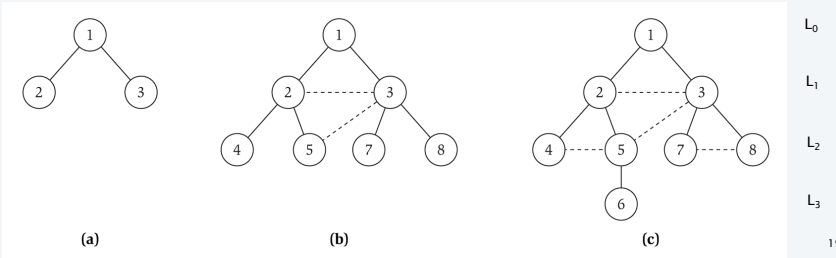
- dipende dal primo nodo visitato e dall'ordine scelto per visitare i vicini di un nodo

Visita in ampiezza (Breadth-first search)

Un albero BFS per un grafo G = (V,E) è l'albero "determinato" da una visita BFS su G

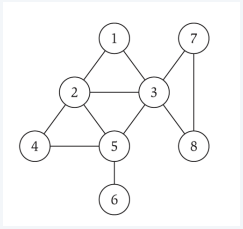
- La radice è il punto di partenza della visita (nell' esempio, il nodo 1)
- un nodo u è il padre di un nodo v
  - se v viene raggiunto per la prima volta mentre l'algoritmo esplora i vicini di u (nell'esempio, 1 e padre di 2 e 3)

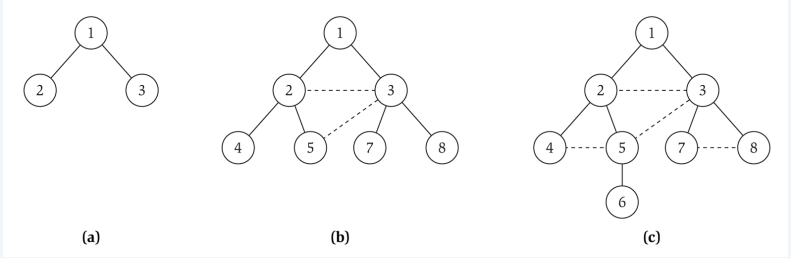




Visita in ampiezza (Breadth-first search)

Proprietà. Sia T un albero BFS di $G = (V, E)$, e sia (x, y) un arco di G. Allora, i livelli di x e y in T differiscono di al più 1.





Visita in ampiezza (Breadth-first search): implementazione

Struttura dati: Coda

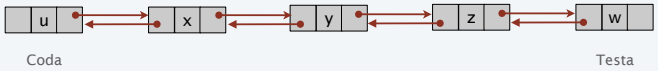
Una coda è una lista ordinata di oggetti, dinamica, in cui ogni nuovo oggetto viene aggiunto alla fine (in coda) della lista ed ogni oggetto che lascia la coda viene preso dalla testa della lista.

- politica FIFO (first in, first out)
- implementata da una lista a puntatori, doppiamente puntata
- le operazioni di aggiunta (Enqueue(S,u)) e di estrazione (Dequeue(S)) costano O(1)

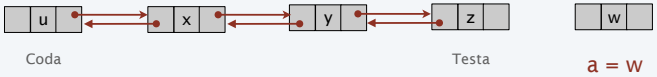
Strutture dati: Coda

La coda S contiene gli elementi: x, y, z, w.
L'elemento w è in testa e l'elemento x è in coda

Aggiungere un elemento: Enqueue(S, u)



Estrarre/Rimuovere un elemento: a = Dequeue(S)



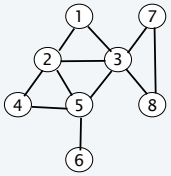
BFS - implementazione mediante una coda

Notazione

- Adj[v] : lista di adiacenza del nodo v
  - il grafo è rappresentato mediante liste di adiacenza
- Enqueue(S, x): aggiunge x alla coda S
- Dequeue(S): ritorna l'elemento in testa alla coda S e lo rimuove da S

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)    // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```


BFS mediante una coda - Esempio



S:

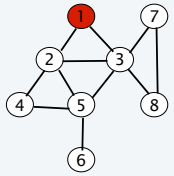
BFS tree T

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)    // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 1

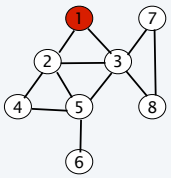
BFS tree T

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)    // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S:

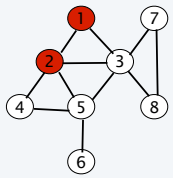
BFS tree T

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 2

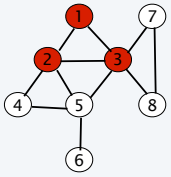
BFS tree T


```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)                    v = 2
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

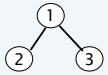
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 3 2

BFS tree T

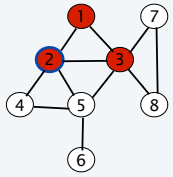



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)                    v = 3
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

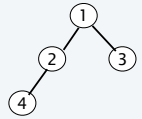
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 3

BFS tree T

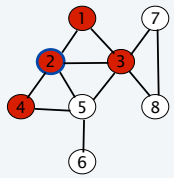



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                       u = 2
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

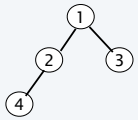
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 4 3

BFS tree T

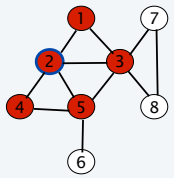



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)                    v = 4
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

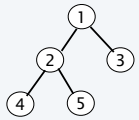
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 5 4 3

BFS tree T

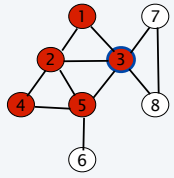



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)                    v = 5
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

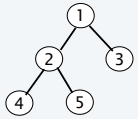
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 5 4

BFS tree T

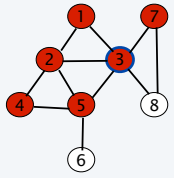



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 3
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

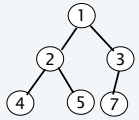
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 7 5 4

BFS tree T

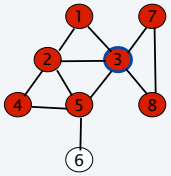



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)                    v = 7
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

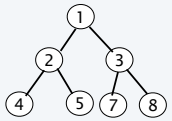
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 8 7 5 4

BFS tree T

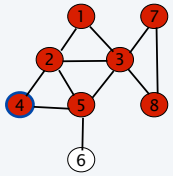



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)                    v = 8
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

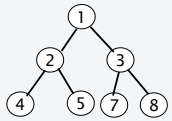
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 8 7 5

BFS tree T

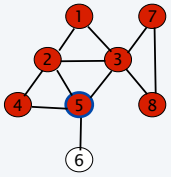



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 4
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

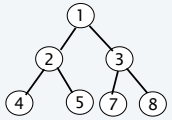
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 8 7

BFS tree T

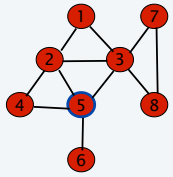



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 5
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

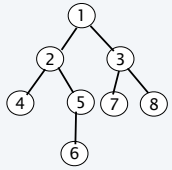
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 6 8 7

BFS tree T

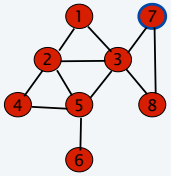



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)                    v = 6
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Liste di adiacenza

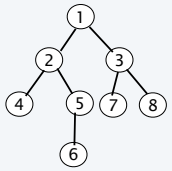
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 6 8

BFS tree T

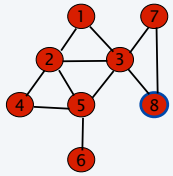



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 7
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

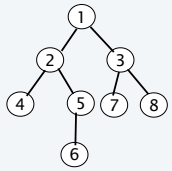
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S: 6

BFS tree T

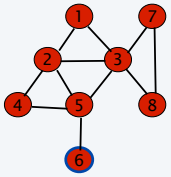



```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 8
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

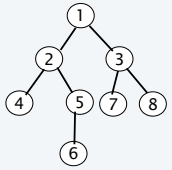
Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

BFS mediante una coda - Esempio



S:

BFS tree T




```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {                          u = 6
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Liste di adiacenza

Adj[1] : 2, 3
Adj[2] : 1, 3, 4, 5
Adj[3] : 1, 2, 5, 7, 8
Adj[4] : 2, 5
Adj[5] : 2, 3, 4, 6
Adj[6] : 5
Adj[7] : 3, 8
Adj[8] : 3, 7

Visita in ampiezza (Breadth-first search): analysis

Teorema. La BFS può essere implementata in tempo $O(m + n)$ se il grafo è rappresentato dalle liste delle adiacenze.

Dim.

- non è difficile mostrare $O(n^2)$:
  - ogni nodo entra nella lista S una sola volta (poi rimane sempre visited)
  - il ciclo while viene eseguito al più una volta per ogni nodo, $\leq n$ volte
  - ogni ciclo for viene eseguito $\leq n$ volte (liste di adiacenza sono $\leq n$-1)
  - spendiamo tempo $O(1)$ in ogni iterazione del for

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
  u ← Dequeue(S)     // visited[u] rimane true
  for each v in Adj[u]
    if (visited[v] = false)
      add edge (u,v) to the BFS tree T
      visited[v] = true
      Enqueue(S,v)
```

Visita in ampiezza (Breadth-first search): analysis

Teorema. La BFS può essere implementata in tempo $O(m + n)$ se il grafo è rappresentato dalle liste delle adiacenze.

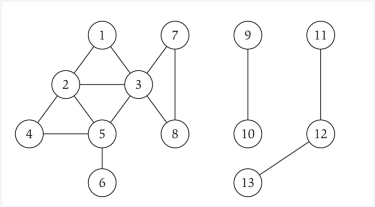
Dim.

- Mostriamo ora che l'algoritmo visto per la BFS richiede tempo $O(n+m)$:
  - ogni nodo entra nella lista S una sola volta (poi rimane sempre visited)
  - il ciclo while viene eseguito al più una volta per ogni nodo
  - per un nodo u il for viene eseguito $d(u)$ volte (il grado del nodo u)
  - spendiamo in tutto tempo $\Sigma_{u \in V} d(u) = 2m = O(m)$ per tutte le iterazioni del while
  - all'inizio spendiamo $O(n)$ per settare `visited[]= false`
  - quindi $O(n+m)$

```
BFS(s)
Set visited[v] = false  for each vertex v
Enqueue(S, s), visited[s] = true
while (S is not empty) {
   u ← Dequeue(S)     // visited[u] rimane true
   for each v in Adj[u]
      if (visited[v] = false)
         add edge (u,v) to the BFS tree T
         visited[v] = true
         Enqueue(S,v)
```

Componenti connesse

Componente connessa. Trova tutti i nodi raggiungibili da s.

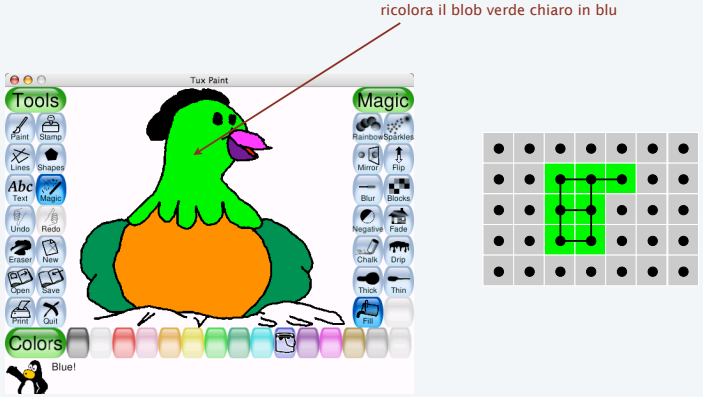


La componente connessa del nodo $1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

Flood fill

Flood fill. Dato un pixel di colore verde chiaro, cambia in blu l'intero blob di pixel verde chiaro vicini al pixel dato.

- Nodi: pixel.
- Archi: due pixel vicini dello stesso colore.
- Blob: componente connessa dei pixel di colore verde chiaro.



Flood fill

Flood fill. Dato un pixel di colore verde chiaro, cambia in blu l'intero blob di pixel verde chiaro vicini al pixel dato.

- Nodi: pixel.
- Archi: due pixel vicini dello stesso colore.
- Blob: componente connessa dei pixel di colore verde chiaro.