Grafi (II parte)

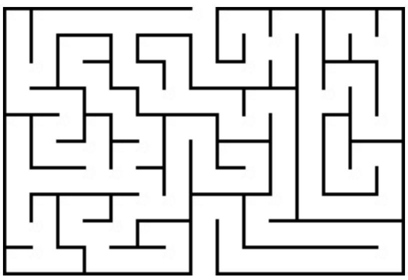
Progettazione di Algoritmi a.a. 2021-22

Matricole congrue a 1

Docente: Annalisa De Bonis

Depth first search (visita in profondità)

- La visita in profondità riproduce il comportamento di una persona che esplora un labirinto di camere interconnesse
  - La persona parte dalla prima camera (nodo s) e si sposta in una delle camere accessibili dalla prima (nodo adiacente ad s), di lì si sposta in una delle camere accessibili dalla seconda camera visitata e così via fino a quando raggiunge una camera da cui non è possibile accedere a nessuna altra camera non ancora visitata. A questo punto torna nella camera precedentemente visitata e di lì prova a raggiungere nuove camere.



Depth first search (visita in profondità)

- La visita DFS parte dalla sorgente s e si spinge in profondità fino a che non è più possibile raggiungere nuovi nodi.
  - La visita parte da s, segue uno degli archi uscenti da s ed esplora il vertice v a cui porta l'arco.
  - Una volta in v, se c'è un arco uscente da v che porta in un vertice w non ancora esplorato allora l'algoritmo esplora w
  - Uno volta in w segue uno degli archi uscenti da w e così via fino a che non arriva in un nodo del quale sono già stati esplorati tutti i vicini.
  - A questo punto l'algoritmo fa backtrack (torna indietro) fino a che torna in un vertice a partire dal quale può visitare un vertice non ancora esplorato in precedenza.

Depth first search: pseudocodice

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u,v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

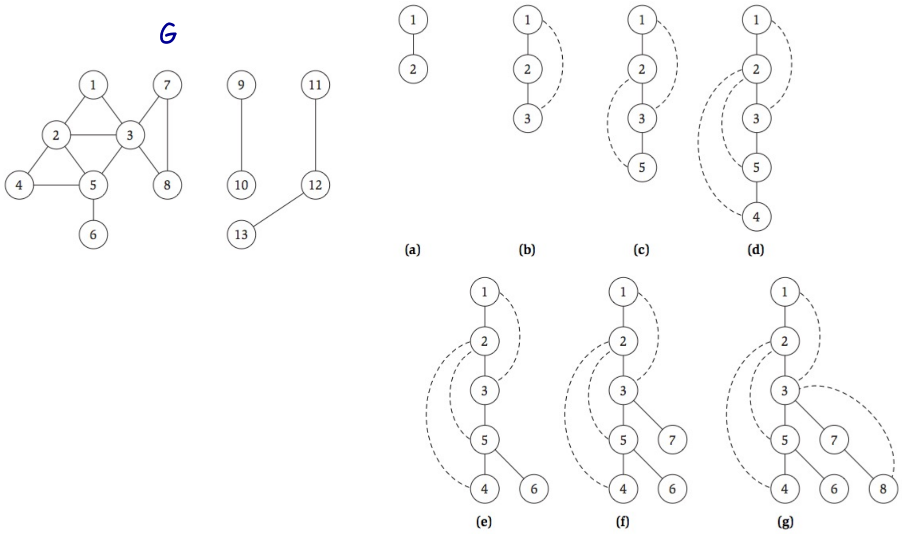
R = insieme dei vertici raggiunti

Analisi: (assumendo G rappresentato con liste di adiacenza)

- Se ignoriamo il tempo delle chiamate ricorsive al suo interno, ciascuna visita ricorsiva richiede tempo O(1+deg(u)) : O(1) per marcare u e aggiungerlo ad R e O(deg(u)) per eseguire il for.
- Se inizialmente invochiamo DFS su un nodo s, allora DFS viene invocata ricorsivamente su tutti i nodi raggiungibili a partire da s. Il costo totale è quindi al più

$$\sum_{u\in V} O(1+deg(u)) = O(\sum_{u\in V} 1 + \sum_{u\in V} deg(u))$$
$$= O(n+m)$$

Esempio di esecuzione di DFS



5

5

Depth First Search Tree (Albero DFS)

- Proprietà. L'algoritmo DFS produce un albero che ha come radice la sorgente s e come nodi tutti i nodi del grafo raggiungibili da s.

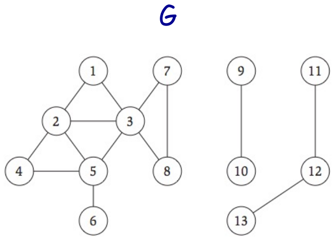
- L'albero si ottiene in questo modo:
  - Consideriamo il momento in cui viene invocata DFS(v)
    - Ciò avviene durante l'esecuzione di DFS(u) per un certo nodo u. In particolare durante l'esame dell'arco (u,v) nella chiamata DFS(u).
    - In questo momento, aggiungiamo l'arco (u,v) e il nodo v all'albero

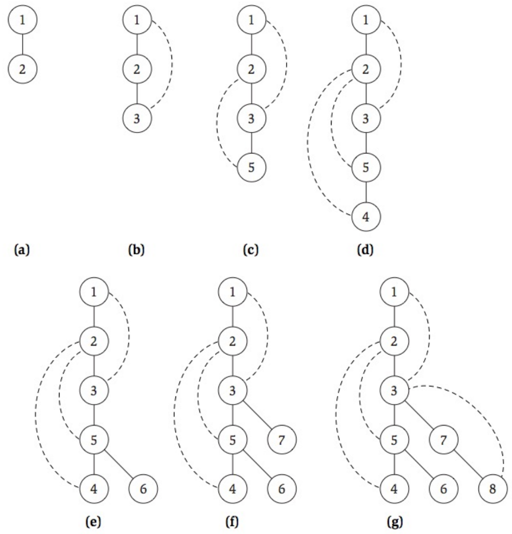
PROGETTAZIONE DI ALGORITMI a.a. 2021-22
A. DE BONIS

6

6

Esempio di albero DFS





Gli archi non tratteggiati fanno parte del DFS tree.

Albero DFS

- Proprietà 1. Per una data chiamata ricorsiva DFS(u), tutti i nodi che vengono etichettati come "Esplorati" tra l'inizio e la fine della chiamata DFS(u), sono discendenti di u nell'albero DFS.

Dim. Proprietà 1.

- Sia x un nodo esplorato tra l'inizio e la fine della chiamata DFS(u)
- La dimostrazione è per induzione sul numero m di chiamate ricorsive iniziate dopo l'inizio di DFS(u) e non ancora terminate **all'inizio** di DFS(x) (esclusa DFS(u) e inclusa DFS(x)).
- **Base. m=1** → Una sola chiamata cominciata dopo l'inizio di DFS(u) e che si deve ancora concludere nel momento in cui esploriamo x → questa chiamata è proprio DFS(x) → DFS(x) è invocata nel foreach di DFS(u) e in questo caso x diventa figlio di u →la proprieta` è soddisfatta (es. u=5, x=6 in slide precedente)
- **Passo induttivo.** Supponiamo vera la proprieta` fino ad m-1>=1 e dimostriamo che è vera per m. Siccome m>=2 → ci deve essere almeno una chiamata a DFS che non è ancora terminata prima che venga invocata DFS(x) → DFS(x) non è invocata nel foreach di DFS(u). Infatti, se DFS(x) venisse invocata nel foreach di DFS(u) allora in quel momento sarebbero gia` terminate tutte le chiamate sui nodi adiacenti ad u esaminati prima di x e tutte le chiamate da esse innescate e sarebbe m=1.
- Cio` vuol dire che x sara` marcato come esplorato da una DFS innescata da una delle DFS invocate nel foreach di DFS(u). Sia z il nodo adiacente ad u per cui si ha che DFS(z) innesca DFS(x**). In altre parole x≠z e x è marcato come esplorato tra l'inizio e la fine di DFS(z).** Il numero di chiamate iniziate dopo l'inizio di DFS(z) e non ancora terminate quando inizia DFS(x) è pari a m-1 per cui possiamo applicare l'ipotesi induttiva. Per ipotesi induttiva x è discendente di z. Siccome z è figlio di u allora x è anch'esso discendente di u.

Albero DFS

Questa proprieta` vale solo se il grafo è non direzionato.

- Proprietà 2. Sia T un albero DFS e siano x e y due nodi di T collegati dall'arco (x,y) in G. Si ha che x e y sono l'uno antenato dell'altro in T.

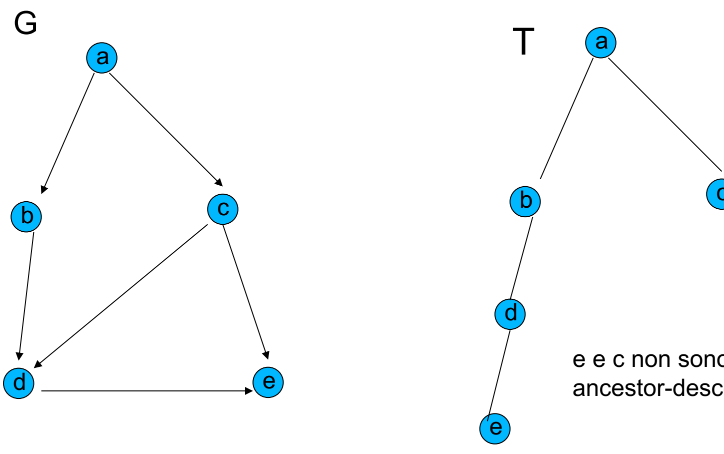
Dim. Proprietà 2

- **Caso (x,y) e` in T.** In questo caso la proprietà e` ovviamente soddisfatta.
- **Caso (x,y) non e` in T.** Supponiamo senza perdere di generalità che DFS(x) venga invocata prima di DFS(y). Ciò vuol dire che quando viene invocata DFS(x), y non è ancora etichettato come "Esplorato".
- La chiamata DFS(x) esamina l'arco (x,y) e per ipotesi non inserisce (x,y) in T. Ciò si verifica solo se y è già stato etichettato come "Esplorato". Siccome y non era etichettato come "Esplorato" all'inizio di DFS(x) vuol dire è stato esplorato tra l'inizio e la fine della chiamata DFS(x). La proprietà 1 implica che y è discendente di x.

Albero DFS

Facciamo vedere che la proprieta` 2 non vale in generale per i grafi direzionati.

Usiamo un controesempio:



e e c non sono in relazione ancestor-descendant

Implementazione di DFS mediante uno stack

```
DFS(s):
1.  Poni Explored[s] = true ed Explored[v] = false per tutti gli altri nodi
2.  Inizializza  S con uno stack contenente s
3.  While S non è vuoto
4.      Metti in u  il nodo  al top di S
5.      If  c'e` un arco (u, v) incidente su u non ancora esaminato then
6.         If Explored[v] = false then
7.            Poni Explored[v] = true
8.            Inserisci v al top di S
9.         Endif
10.     Else  // tutti gli archi incidenti su u sono stati esaminati
11.        Rimuovi il top di S
12.     Endif
13. Endwhile
```

- Per implementare la linea 6 in modo efficiente possiamo mantenere per ogni vertice u un puntatore al nodo della lista di adiacenza di u corrispondente al prossimo arco (u,v) da scandire.
- Si noti che un nodo u rimane nello stack fino a che non vengono scanditi tutti gli archi incidenti su u.

11

11

Analisi di DFS implementata mediante uno stack

Assumiamo G rappresentato con liste di adiacenza

```
DFS(s):
1.  Poni Explored[s] = true ed Explored[v] = false per tutti gli altri nodi   O(m)
2.  Inizializza  S con uno stack contenente s   O(1)
3.  While S non è vuoto
4.      Metti in u  il nodo  al top di S
5.      If  c'e` un arco (u, v) incidente su u non ancora esaminato then
6.         If Explored[v] = false then
7.            Poni Explored[v] = true
8.            Inserisci v al top di S
9.         Endif
10.     Else  // tutti gli archi incidenti suu sono stati esaminati
11.        Rimuovi il top di S
12.     Endif                                   O(n+m)
13. Endwhile
```

- Analisi linee 3-13: Il while viene iterato deg(v)+1 volte per ogni nodo v inserito in S: ogni volta che v viene a trovarsi al top dello stack viene esaminato uno dei suoi archi non ancora esaminati oppure se non esiste un tale arco, v viene rimosso dallo stack. Vengono quindi effettuate deg(v) iterazioni del while prima di quella in cui v viene rimosso dallo stack → in totale il while e` iterato un numero di volte pari al piu` a $\sum_{v\in V}(deg(v)+1) = \sum_{v\in V} deg(v) + \sum_{v\in V} 1 \leq 2m+n$
- Se manteniamo traccia del prossimo arco da scandire (vedi slide precedente), la linea 6 richiede tempo O(1). Di conseguenza il corpo del while richiede O(1) per ogni iterazione →tempo totale per tutte le iterazioni O(2m+n)=O(m+n).

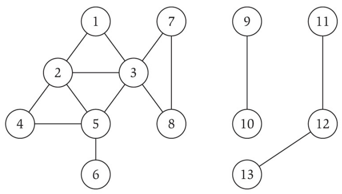
12

12

Componente connessa

- Componente connessa. Sottoinsieme di vertici tale per ciascuna coppia di vertici u e v esiste un percorso tra u e v

- Componente connessa contenente s. Formata da tutti i nodi raggiungibili da s

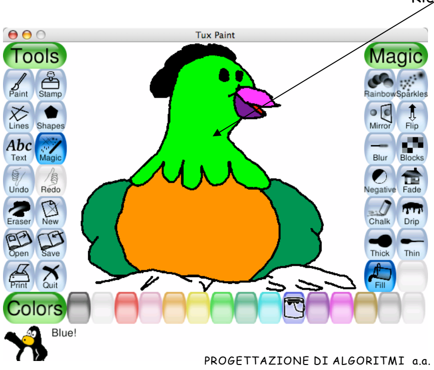


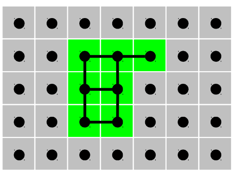
- Componente connessa contenente il nodo 1 è { 1, 2, 3, 4, 5, 6, 7, 8 }.

Flood Fill

- Flood fill. Data un'immagine, cambia il colore dell'area di pixel vicini di colore verde lime in blu.
  - Nodo: pixel.
  - Arco: due pixel vicini di colore verde lime.
  - Area di pixel vicini di colore verde lime: componente connessa di nodi associati a pixel verde lime.

Ricolora l'area verde lime di blue

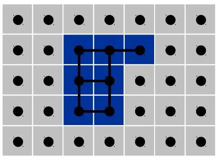




Flood Fill

- Flood fill. Data un'immagine, cambia il colore dell'area di pixel vicini di colore verde lime in blu.
  - Nodo: pixel.
  - Arco: due pixel vicini di colore verde lime.
  - Area di pixel vicini: componente connessa di pixel di colore verde lime.

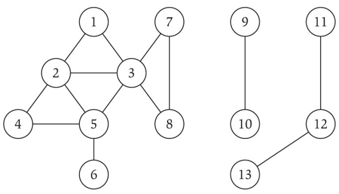
Ricolora l'area verde lime di blue



15

Componente connessa

- Componente connessa contenente s. Trova tutti i nodi raggiungibili da s
  - Come trovarla. Esegui BFS o DFS utilizzando s come sorgente
- Insieme di tutte le componenti connesse. Trova tutte le componenti connesse
  - Come trovarlo. Fino a quando ci sono nodi che non sono stati scoperti (esplorati), scegli uno di questi nodi ed esegui BFS (o DFS) su u utilizzando questo nodo come sorgente



Esempio: il grafo sottostante ha tre componenti connesse

16

Insieme di tutte componenti connesse

- Teorema. Per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o disgiunte
- Dim.
- Caso 1. Esiste un percorso tra s e t. In questo caso ogni nodo u raggiungibile da s è anche raggiungibile da t (basta andare da t ad s e da s ad u) e ogni ogni nodo u raggiungibile da t è anche raggiungibile da s (basta andare da s ad t e da t ad u). Ne consegue che un nodo u è nella componente connessa di s **se e solo** se è anche in quella di t e quindi le componenti connesse di s e t sono uguali.
- Caso 2. Non esiste un percorso tra s e t. In questo caso non può esserci un nodo che appartiene sia alla componente connessa di s che a quella di t. Se esistesse un tale nodo v questo sarebbe raggiungibile sia da s che da t e quindi potremmo andare da s a v e poi da v ad t. Ciò contraddice l'ipotesi che non c'è un percorso tra s e t.

Insieme di tutte componenti connesse

- Il teorema precedente implica che le componenti connesse di un grafo sono a due a due disgiunte.
- Algoritmo per trovare l'insieme di tutte le componenti connesse

```
AllComponents(G)
 Per ogni nodo u di G setta discovered[u]=false
 For each node u of G
    If Discovered[u] = false
      BFS(u)
  Endif
 Endfor
```

- BFS modificata in modo tale che nella fase di inizializzazione non vengano settati a False le entrate dell'array Discovered
- Al posto della BFS possiamo usare la DFS e al posto dell'array Discovered l'array Explored

Insieme di tutte componenti connesse: analisi

- Indichiamo con k il numero di componenti connesse
- Indichiamo con n_i e con m_i rispettivamente il numero di nodi e di archi della componente i-esima
- L'esecuzione della visita BFS o DFS sulla componente i-esima richiede tempo $O(n_i+m_i)$
- Il tempo totale richiesto da tutte le visite BFS o DFS e`

$$\sum_{i=1}^{k} O(n_i+m_i)=O(\sum_{i=1}^{k}(n_i+m_i))$$

- Poiche' le componenti sono a due a due disgiunte, si ha che

$$\sum_{i=1}^{k}(n_i+m_i)=n+m$$

- e il tempo totale di esecuzione dell'algoritmo che scopre le componenti connesse e` $O(n)+O(n+m)=O(n+m)$

19

Insieme di tutte componenti connesse: alcune considerazioni

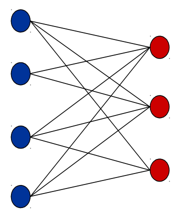
- Se l'algoritmo utilizza BFS allora BFS deve essere modificata in modo che non resetti a false ogni volta i campi discovered.
- E` possibile modificare AllComponents in modo che assegni a ciascun nodo la componente di cui fa parte. A questo scopo usiamo:
  - contatore delle componenti.
  - array Component t.c. Component[u] = j se u appartiene alla componente j-esima.

  - Esercizio: modificare lo pseudocodice dell'algoritmo AllComponents in modo che assegni a ciascun nodo la componente di cui fa parte. Ricordatevi che occorre modificare anche l'algoritmo di visita invocato da AllComponents.

20

Grafi bipartiti

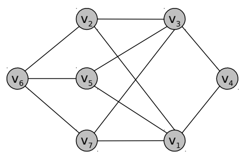
- Def. Un grafo non direzionato è **bipartito** se l'insieme di nodi può essere partizionato in due sottoinsiemi X e Y tali che ciascun arco del grafo ha una delle due estremità in X e l'altra in Y
  - Possiamo colorare i nodi con due colori (ad esempio, rosso e blu) in modo tale che ogni arco ha un'estremita rossa e l'altra blu.
- Applicazioni.
  - Scheduling: macchine = rosso, job = blu.



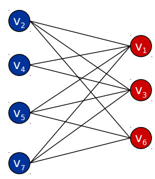
Un grafo bipartito

Testare se un grafo è bipartito

- Testare se un grafo è bipartito. Dato un grafo G, vogliamo scoprire se è bipartito.
  - Molti problemi su grafi diventano:
    - Più facili se il grafo sottostante è bipartito (matching: sottoinsieme di archi tali che non hanno estremità in comune)
    - Trattabili se il grafo è bipartito (max insieme indipendente)



Un grafo bipartito G



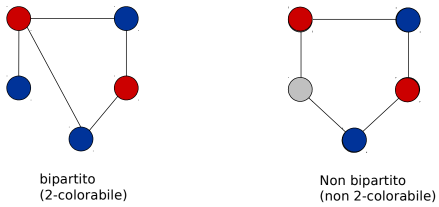
Modo alternativo di disegnare G

Se volessimo considerare tutti i possibili modi di colorare i nodi con due colori dovremmo considerare 2^{n-1} possibilita`: per ogni sottoinsieme U di V possiamo colorare i nodi di U di rosso e i nodi di V-U di blu → $2^n$ coppie (U,V-U)→ $2^n$ modi di colorare i vertici con due colori. Siccome non occorre considerare entrambe le coppie (U,V-U) e (V-U,U) in quanto cio` corrisponderebbe a scambiare i colori dei due insiemi → numero colorazioni da esaminare $=2^n/2= 2^{n-1}$

Grafi bipartiti

- Lemma. Se un grafo G è bipartito, non può contenere un ciclo dispari (formato da un numero dispari di archi)
- In pratica vale l'implicazione: G bipartito→ nessun ciclo dispari in G

- Dim. Non è possibile colorare di rosso e blu i nodi su un ciclo dispari in modo che ogni arco abbia le estremità di diverso colore.

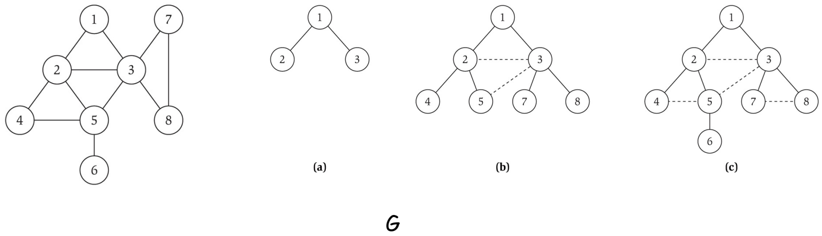


Breadth First Search Tree

- Vi ricordate questa proprieta?

- Proprietà. Si consideri un'esecuzione di BFS su $G = (V, E)$, e sia (x, y) un arco di G. I livelli di x e y differiscono di al più di 1.

- Sfutteremo questa proprieta` per provare che la BFS, nella versione con i layer, puo` essere utilizzata per determinare se un grafo è bipartito

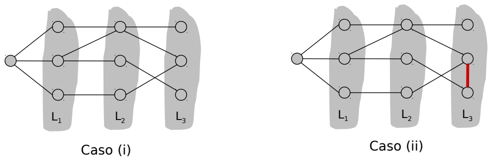


Grafi bipartiti

- Osservazione. Sia G un grafo connesso e siano L_0, …, L_k i livelli prodotti da un'esecuzione di BFS a partire dal nodo s. Può avvenire o che si verifichi la (i) o la (ii)

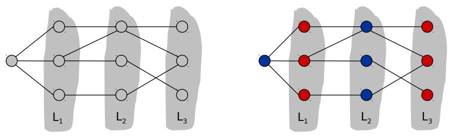
  (i) Nessun arco di G collega due nodi sullo stesso livello

  (ii) Un arco di G collega due nodi sullo stesso livello



Grafi Bipartiti

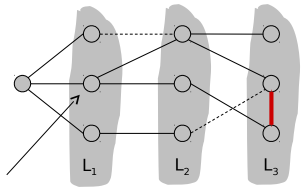
- Nel caso (i) il grafo è bipartito.
- Dim.
- Per la proprietà sulla distanza tra livelli contenenti nodi adiacenti, si ha che due nodi adiacenti o si trovano nello stesso livello o in livelli consecutivi.
- Poiché nel caso (i) non ci sono archi tra nodi di uno stesso livello allora tutti gli archi del grafo collegano nodi in livelli consecutivi.
- Quindi se coloro i livelli di indice dispari di rosso e quelli di indice pari di blu, ho che le estremità di ogni arco sono di colore diverso



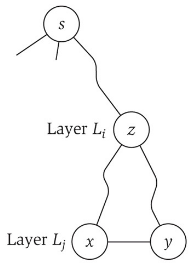
Caso (i)

Grafi Bipartiti

Nel caso (ii) il grafo non è bipartito.
Dim. Dimostriamo che il grafo contiene un ciclo dispari: supponiamo che esista l'arco (x,y) tra due vertici x e y di L_j. Indichiamo con z l'antenato comune a x e y nell'albero BFS che si trova più vicino a x e y. Sia L_i il livello in cui si trova z. Possiamo ottenere un ciclo dispari del grafo prendendo il percorso seguito dalla BFS da z a x (j-i archi), quello da z a y (j-i archi) e l'arco (x,y). In totale il ciclo contiene 2(j-i)+1 archi.



Antenato comune più vicino (lowest common ancestor)



Caso (ii)

Algoritmo che usa BFS per determinare se un grafo è bipartito

Modifichiamo BFS come segue:

- Usiamo un array Color per assegnare i colori ai nodi
- Ogni volta che aggiungiamo un nodo v alla lista L[i+1] poniamo Color[v] uguale a rosso se i+1 è pari e uguale a blu altrimenti
- Alla fine esaminiamo tutti gli archi per vedere se c'è ne è uno con le estremità dello stesso colore . Se c'è concludiamo che G non è bipartito (perche'?); altrimenti concludiamo che G è bipartito (perche'?).

- Tempo: O(n+m)