BREADTH FIRST SEARCH

FAGADAU DANIEL
845279

BREADTH FIRST SEARCH

ISTANZA: G = (V, E) *(orientato o meno)*, s ∈ V *(vertice sorgente)*

COSA FA

1. Scopre i vertici ***raggiungibili*** dal vertice sorgente *s*.
2. Calcola ∀ v ∈ V la ***distanza*** di v dal vertice sorgente *s*.
3. *(Principio di funzionamento)*
   Amplia la frontiera tra vertici già scoperti e vertici non ancora scoperti in modo tale che:
   - prima scopre i vertici a distanza 0 da *s (ovvero s stesso)*,
   - poi scopre i vertici a distanza 1 da *s*,
   - poi scopre i vertici a distanza 2 da *s* e così via fino a raggiungere la profondità massima del sotto grafo di *s*.
4. Genera un ***albero radicato*** *(Albero BFS)* con radice *s* contenente tutti i vertici raggiungibili da *s (tramite le informazioni contenute in* π*)*.
   ALBERO BFS
   Denotiamo l'albero BFS con $\mathbf{G_\pi = (V_\pi, E_\pi)}$ tale che:
   V_π contiene tutti i vertici raggiungibili dalla sorgente *s*.
   Possiamo quindi definire V_π in 3 modi equivalenti, ovvero:
   - $V_\pi = \{ v \in V \mid v.col \neq WHITE \}$ *(oppure v.col = BLACK)*
   - $V_\pi = \{ v \in V \mid v.d \neq \infty \}$
   - $V_\pi = \{ v \in V \mid v.\pi \neq NIL \} \cup \{s\}$

   E_π sono gli archi dell'albero.
   Definiamo E_π come
   - $E_\pi = \{ (v.\pi, v) \mid v \in V_\pi - \{s\} \}$
   - $E_\pi = \{ (u, v) \mid u, v \in V_\pi - \{s\} : v.\pi = u \} \cup \{ (s, w) \mid w \in Adj[s] \}$ *(non sono sicuro della correttezza di questa)*

ALGORITMO

Uso dei colori per dire se un certo nodo è stato scoperto, completamente esplorato e non scoperto;

- WHITE - Non scoperto.
- GREY - Scoperto ma non completamente esplorato, non ho ancora scoperto tutta la sua lista di adiacenza.
- BLACK - Scoperto e completamente esplorato, ho scoperto la sua lista di adiacenza.

Inoltre salvo un'informazione relativa a chi ha scoperto un certo vertice *v* in un suo parametro π *(parent)*. Ovviamente la sorgente non ha alcun parent.

Se mi serve sapere solo i nodi raggiungibili dalla sorgente, ovviamente non ho bisogno dei colori o dei predecessori. Queste sono informazioni che ci servono per ulteriori implementazioni dell'algoritmo.

```
BFS(G, s) {
INIZIALIZZAZIONE   Tempo Θ(|V|) visto che inizializziamo tutti i v ∈ V.
for all u ∈ V - {s}
   u.col = WHITE
   u.d = ∞
   u.π = NIL
s.col = GREY
s.d = 0
s.π = NIL
enqueue(Q, s)

ESECUZIONE  Tempo O(|E|) visto che percorro solo gli archi raggiungibili da s (potrebbero essere tutti)
while Q ≠ ∅
   u = dequeue(Q)
   for all v ∈ Adj[u]
      if v.col == WHITE
         v.col = GREY
         v.d = u.d + 1
         v.π = u
         enqueue(Q, v)

   u.col = BLACK
}
```

Nella coda spesso ci saranno dei nodi a distanza *k* insieme a dei nodi a distanza *k + 1*, una volta levati quelli a distanza *k* avrò solo quelli a distanza *k + 1* se esistono, altrimenti avrò la coda vuota fermando il ciclo while.

ESERCIZI

CONTARE VERTICI RAGGIUNGIBILI

Dato un grafo G = (V, E) $x \in V$, contare i vertici raggiungibili da x.

```
BFS(G, x)
n = 0
for all v ∈ V
   if v.col != WHITE (oppure v.d != ∞)
      n++
Return n
```

Ovviamente la BFS esplora solo i nodi raggiungibili dalla sorgente, lasciando bianchi quelli non raggiungibili da essa *(e quindi con la distanza pari ad ∞)*, quindi mi basta contare quanti nodi esplorati ottengo alla fine della BFS.

STABILIRE SE UN GRAFO N OR È UN ALBERO

Dato un grafo non orientato scrivere un algoritmo che decide se il grafo è un albero.
Sappiamo che un albero è un grafo non orientato, connesso e aciclico; ci dicono già che è non orientato, dobbiamo stabilire se è connesso e aciclico.

DECIDERE SE È CONNESSO O NO
Basta estrarre un vertice a caso e lanciare BFS su quel vertice; se alla fine dell'esecuzione ci sono dei vertici bianchi o con distanza = ∞ il grafo sicuramente non è connesso.

```
isConnected(G) {
s = Random(V)
BFS(G, s)
for all v ∈ V
   if v.d == ∞
      return FALSE
return TRUE
}
```

DECIDERE SE È ACICLICO

```
BFS(G, s) {
for all u ∈ V - {s}
   u.col = WHITE
s.col = GREY
enqueue(Q, s)

while Q ≠ ∅
   u = dequeue(Q)
   for all v ∈ Adj[u]
      if v.col == WHITE
         v.col = GREY
         enqueue(Q, v)
      else if v.col == GREY
         Return FALSE
Return TRUE
}
```

DECIDERE SE È UN ALBERO
Per decidere quindi se un grafo è un albero potremmo unire le due informazioni ottenute fino ad ora. Tuttavia, possiamo anche sfruttare un'altra definizione di albero e, dopo aver controllato che il grafo si connesso, confrontare il numero di vertici e di lati.

```
BFS(G, s) {
for all u ∈ V - {s}
   u.d = ∞
s.d = 0
enqueue(Q, s)
e = 0

while Q ≠ ∅
   u = dequeue(Q)
   for all v ∈ Adj[u]
      e++ (e contiene dunque il doppio del numero di archi all'interno della CC di cui fa parte s)
      if v.d == ∞
         v.d = u.d + 1
         enqueue(Q, v)
}
for all v ∈ V
   if v.d == ∞
      Return FALSE

n = Adj.length (n = |V| !!!)
if e/2 == n - 1
   Return TRUE
else
   Return FALSE
```

ATTENZIONE: *e* alla fine conterrà il doppio del numero degli archi contenuti nella CC di cui fa parte *s* visto che stiamo trattando dei grafi non orientati e quindi all'interno della lista di adiacenza di ogni nodo ci sarà anche il nodo parent.

STABILIRE SE UNA CC È UN ALBERO

Dato un grafo G = (V, E) non orientato e dato un vertice $x \in V$, stabilire se la CC contenente *x* è un albero.

Esercizio più facile di quello di prima, infatti se consideriamo una sola componente connessa, è ovviamente ***connessa***; ci basta confrontare il numero di archi con il numero di vertici per dire se sia un albero o meno.

```
BFS(G, s) {  (s sarebbe il nostro x in questo caso)
for all u ∈ V - {s}
   u.d = ∞
s.d = 0
enqueue(Q, s)
e = 0, nv = 0

while Q ≠ ∅
   u = dequeue(Q)
   nv++  (ogni volta che tiro fuori un vertice dalla coda lo conto)
   for all v ∈ Adj[u]
      e++
      if v.d == ∞
         v*.d = u.d + 1
         enqueue(Q, v)

if e/2 == nv - 1
   Return TRUE
else
   Return FALSE
}
```

NODI A DISTANZA $\leq$ K

Dato un grafo G = (V, E), un vertice $s \in V$ ed un valore $k > 0$ intero, modificare BFS in modo tale che scopra solo i vertici a distanza $\leq k$ da s

```
BFS(G, s, k) {
for all u ∈ V - {s}
   u.d = ∞
s.d = 0
enqueue(Q, s)

while Q ≠ ∅
   u = dequeue(Q)
   for all v ∈ Adj[u]
      if v.d == ∞
         v.d = u.d + 1
         if v.d < k        (non accodo più una volta che ho raggiunto k - 1 livelli)
            enqueue(Q, v)
}
```

ATTENZIONE: Mi fermo a k - 1 poiché una volta accodati i nodi a livello k - 1, andrò a scoprire solo i loro figli, ovvero i nodi a distanza k, per poi avere la coda vuota e terminare il ciclo.

GRAFI COMPLETI

Dato un grafo non orientato G = (V, E), dato $s \in V$, scrivere un algoritmo che stabilisce se la CC contenente s è un sotto grafo che, preso singolarmente, è un grafo completo.

Nel caso dei grafi non orientati purché un grafo sia completo deve avere il massimo numero di archi, ovvero $\sum_{i=1}^{n} n - i = \frac{(n-1) * n}{2}$

```
BFS(G, s) {
for all u ∈ V - {s}
   u.d = ∞
s.d = 0
enqueue(Q, s)
e = 0, nv = 0

while Q ≠ ∅
   u = dequeue(Q)
   nv++
   for all v ∈ Adj[u]
      e++
      if v.d == ∞
         v.d = u.d + 1
         enqueue(Q, v)

if e/2 == ((nv - 1) * nv)/2
   Return TRUE
else
   Return FALSE
}
```

PER GRAFI ORIENTATI

Il procedimento è lo stesso, ma nei grafi orientati il numero di archi deve essere pari a nv^{nv}.

DEPTH FIRST SEARCH

FAGADAU DANIEL
845279

DEPTH FIRST SEARCH

ISTANZA: G = (V, E)

COSA FA

1. Scopre **tutti** i vertici del grafo scoprendo prima quelli in profondità.
   Quindi $\forall$ v $\in$ V visita il sotto grafo in profondità.
2. Calcola $\forall$ v $\in$ V il ***tempo*** di ***inizio e fine scoperta*** e $\forall$ (i, j) $\in$ E etichetta gli archi in base al nodo di arrivo *(vedere proprietà sotto)*.
3. *(Principio di funzionamento)*
   Scopre ogni volta un vertice più profondo fino a raggiungere la massima profondità del ramo corrente.
4. Genera una ***foresta*** *(Foresta DFS)*, ovvero un insieme di alberi disgiunti.
   FORESTA DFS
   Denotiamo la foresta DFS con $\mathbf{G_\pi = (V, E_\pi)}$ tale che:
   V è l'insieme dei vertici del grafo originario, poiché DFS visita tutti i nodi.
   E_π rappresenta gli archi della foresta e lo definiamo come:
   E_π = { (v.π, v) | v $\in$ V AND v.$\pi \neq$ NIL }

ALGORITMO

Al fine di evitare di eseguire dei loop e di evitare il rischio di lasciare alcuni nodi inesplorati, l'algoritmo memorizza alcune informazioni su ogni nodo:
COLORE

- WHITE: ancora non scoperto
- GREY: vertice scoperto ma non analizzato completamente
- BLACK: vertice completamente analizzato

PREDECESSORE
v.π - predecessore di *v* nella visita DFS, il vertice che mi ha portato a *v*.

TEMPI

- v.d - tempo di inizio scoperta di *v*, passo nel quale lo scopro
- v.f - tempo di fine scoperta di *v*, passo nel quale ho finito di esplorarlo
  (ovviamente v.d < v.f)

L'algoritmo inoltre si divide in due procedure, la prima di **inizializzazione** e che avvia la visita vera e propria, mentre la seconda si occupa di visitare effettivamente i nodi.

INIZIALIZZAZIONE
Tempo $\Theta(|V|)$ poiché inizializza tutti i vertici del grafo.

```
DFS(G)
for all v ∈ V
   v.col = WHITE
   v.π = NIL
time = 0

for all v ∈ V
   if v.col == WHITE
      DFS_Visit(G, v)
```

VISITA

Tempo $\Theta(|E|)$ poiché per ogni vertice *v* analizza tutta la sua Adj[*v*] e $\sum_{i=0}^{|V|} |Adj[i]| = |E|$

```
DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

for all w ∈ Adj[u]
   if w.col == WHITE
      w.π = u
      DFS_Visit(G, w)

u.col = BLACK
time++
u.f = time
```

PROPRIETÀ

- **Teorema delle parentesi:** $\forall$ u, v $\in$ V, con A = [d[u], f[u]] e B = [d[v], f[v]] gli **unici** casi possibili sono:
  - A $\cap$ B = $\emptyset$ *(B viene scoperto e finisce o prima o dopo A)*
  - A $\subseteq$ B o B $\subseteq$ A *(Uno dei due è contenuto nell'altro)*
- **Teorema del cammino bianco:** un vertice *v* è discendente di un vertice *u* SSE al momento della scoperta di *u*, il vertice *v* è raggiungibile da *u* tramite un cammino che contiene esclusivamente nodi bianchi *(questo teorema è un'applicazione del teorema delle parentesi)*.

- **Classificazione degli archi** in base al colore di arrivo:
    - WHITE: Tree-edge, arco appartenente alla foresta DFS, lati che portano a scoprire nuovi vertici.
    - GREY: Back-edge, archi che non appartengono alla foresta DFS, che vanno da un vertice *v* ad un antenato di *v* nell'albero DFS.
    - BLACK, vale solo per i grafi orientati:
        - Forward-edge, archi non appartenti alla foresta DFS che vanno da un vertice *u* ad un suo successore.
        - Cross-edge, tutti gli altri archi, posso identificarli guardando gli intervalli considerando il teorema delle parentesi.

ESERCIZI

CONTARE I VERTICI

- Possiamo contare in maniera *'bruta'* i vertici, ovvero incrementando un contatore ogni volta che ne scopriamo uno e lo coloriamo di grigio.

  ```
  DFS_Visit(G, u)
  u.col = GREY
  nv++

  for all w ∈ Adj[u]
    if w.col == WHITE
      DFS_Visit(G, w)

  u.col = BLACK
  ```

- Per contare i vertici, oltre al metodo visto, possiamo aggiornare un contatore ogni volta che finiamo di esplorare un nodo aggiungendoci quanti nodi c'erano sotto di esso *(somma ricorsiva)*.

  ```
  DFS_Visit(G, u)
  u.col = GREY

  for all w ∈ Adj[u]
    if w.col == WHITE
      k += DFS_Visit(G, w)

  u.col = BLACK
  Return k++ //++ perché va contato anche u
  ```

- Un terzo modo per contare i vertici si basa sull'utilizzo dei tempi; sappiamo che per ogni nodo ci salviamo due tempi, dunque facciamo due incrementi della variabile time per ogni vertice, di conseguenza il numero di nodi sarà pari a time/2.

  Quindi per vedere quanti nodi ci sono sotto un vertice *v* mi basta calcolare $\frac{v.f - v.d + 1}{2}$

  ```
  DFS(G)
  for all v ∈ V
    v.col = WHITE
    v.π = NIL
  time = 0

  for all v ∈ V
    if v.col == WHITE
      DFS_Visit(G, v)
      nv = (v.f - v.d + 1)/2
  ```

CONTARE LE CC DI UN GRAFO

Dato G = (V, E) non orientato, contare le sue componenti connesse.
Se ragioniamo su come è strutturato l'algoritmo DFS possiamo osservare che ogni volta che viene chiamata la DFS_Visit è come se venisse scelta una *'sorgente'* dalla quale scoprire il suo sotto grafo; di conseguenza ogni volta che tale procedura viene invocata, andiamo ad analizzare una nuova componente connessa.
(Per motivi di leggibilità e semplicità scriverò solo la procedura modificata, in esame vanno scritte entrambe).

```
DFS(G)
for all v ∈ V
  v.col = WHITE
  v.π = NIL
time = 0
CC = 0

for all v ∈ V
  if v.col == WHITE
    CC++
    DFS_Visit(G, v)
Return CC
```

CONTA ALBERI

Dato un grafo non orientato G = (V, E), calcolare quante sue CC sono degli alberi.

```
DFS(G)
for all v ∈ V
  v.col = WHITE
  v.π = NIL
nTree = 0

for all v ∈ V
  if v.col == WHITE
    Acyclic = TRUE
    DFS_Visit(G, v)
    if Acyclic
      nTree++

DFS_Visit(G, u)
u.col = GREY

for all w ∈ Adj[u].length
  if w.col == WHITE
    w.π = u
    DFS_Visit(G, w)
  else if w.col == GREY AND w != u.π
    Acyclic = FALSE

u.col = BLACK
```

ATTENZIONE: Notare che settiamo acyclic a false solo se il vertice considerato è diverso dal parent di *u*, questo perché nel caso dei grafi non orientati il parent sarà sempre presente nella lista di adiacenza dei nodi *(ma questo non significa che ci sia un ciclo)*.

ATTENZIONE: Nel caso di grafi orientati non ci dovremmo preoccupare del parent, ma occhio a ***non*** considerare gli archi neri come cicli; ci danno delle vie alternative ma non creano cicli.

ALTERNATIVA CONFRONTANDO ARCHI E NODI

```
DFS(G)
for all v ∈ V
  v.col = WHITE
nTree = 0

for all v ∈ V
  if v.col == WHITE
    nv = 0, ne = 0
    DFS_Visit(G, v)
    if ne/2 == nv - 1
      nTree++

DFS_Visit(G, u)
u.col = GREY
nv++

for all w ∈ Adj[u]
  ne++
  if w.col == WHITE
    DFS_Visit(G, w)

u.col = BLACK
```

CC COMPLETE

Dato un grafo non orientato G = (V, E) contare il numero di componenti connesse che siano dei grafi completi. *(Un grafo non orientato è completo quando |E| = (|V| * (|V| - 1))/2)*.

```
DFS(G)
for all v ∈ V
  v.col = WHITE
nComp = 0

for all v ∈ V
  if v.col == WHITE
    nv = 0, ne = 0
    DFS_Visit(G, v)
    if ne/2 == ((nv - 1) * nv)\2
      nComp++

DFS_Visit(G, u)
u.col = GREY
nv++
```

```
for all w ∈ Adj[u]
   ne++
   if w.col == WHITE
      DFS_Visit(G, w)

u.col = BLACK
```

ETICHETTARE ARCHI

Dato un grafo orientato G = (V, E) stabilire, per ogni lato *(i, j)* ∈ E, di che tipo di lato si tratta.

```
DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

for all w ∈ Adj[u]
   if w.col == WHITE
      print 'Tree-Edge'
      w.π = u
      DFS_Visit(G, w)
   else if w.col == GREY      print 'Back-Edge'
   else if u.d < w.d          print 'Forward-Tree'
   else                       print 'Cross-Edge'

u.col = BLACK
time++
u.f = time
```

ATTENZIONE: Nel caso dei grafi non orientati sappiamo che i casi di vertici BLACK non si verificheranno mai; tuttavia attenzione al ciclo for: non dobbiamo considerare il parent di u, altrimenti avremmo un arco etichettato come Back quando in realtà è Tree.

FORESTA CON K ALBERI

Dato un G = (V, E) non orientato e k > 0 stabilire se G è una foresta con esattamente k alberi.

```
DFS(G)
for all v ∈ V
   v.col = WHITE
   v.π = NIL
time = 0
CC = 0

while all v ∈ V AND CC ≤ k AND Acyclic
   if v.col == WHITE AND CC < k
      DFS_Visit(G, v)
      CC++
   else if v.col == WHITE
      CC++ (se ho già sforato k non ha senso fare un'altra visita, incremento CC per dire che CC > k)
if CC == k AND Acyclic
   Return TRUE
else
   Return FALSE

DFS_Visit(G, u)
time++
u.d = time
u.col = GREY

while all w ∈ Adj[u] - u.π AND Acyclic
   if w.col == WHITE
      w.π = u
      DFS_Visit(G, w)
   else
      Acyclic = FALSE

u.col = BLACK
time++
u.f = time
```