**BFS(s)**

L\_0={s}

For(i=0;i<=n-2;i++)

Li+1= Ø;

Foreach nodo u in Li

Foreach nodo v adiacente ad u

if( v non appartiene ad L1,…,Li)

Li+1=Li+1 U {v} //Unione con il nodo {v}

EndIf

Endforeach

Endforeach

Endfor

**Analisi della complessità di BFS(S) (Dove S è la radice)**

V appartiene ai livelli precedenti? (Già visitato?) Per capirlo devo esaminare tutti i livelli precedenti, il numero totale di nodi nell'unione dei livelli è lineare in n. La prima volta ci sarà solo un elemento (la radice), la seconda volta dipende da quanti ne abbiamo raggiunti, ma almeno 1 sicuramente (Perché il caso peggiore in cui si può presentare il grafo è una catena di nodi), quindi si ha una sommatoria che va da 1 ad n nel caso peggiore, che sappiamo essere uguale ad n(n+1)/2, quindi a (n^2 + n)/2 = O(n^2)): il costo per fare tutti i confronti è quadratico.

L'albero BSFT ha tempo O(n+m) perché riusciamo subito a caprie se un vertice è già presente in livelli precedenti, era questo ciò che costava molto.

Dire che un nodo si trova in un livello precedente equivale a dire che il nodo è stato già visitato. Per ogni vertice usiamo un flag chiamato discovered, settato a true quando il nodo viene visitato la prima volta, falso in default.

L'implementazione scelta è la lista di adiacenza e possiamo immaginare di avere un array di riferimenti a liste. Quest'array avrà n celle, alcune delle quali possono essere vuote (quelle corrispondenti ai livelli vuoti, perché possiamo avere al più n livelli ma potremmo averne molti di meno).

**BFS(s) Tree**

Poni Discovered[s] = true e Discovered[v] = false per tutti gli altri v  
La *prima cosa che faccio, visto che visito la sorgente, è porre il campo discovered della sorgente a true, tutti gli altri discovered saranno posti a falso non essendo stati ancora visitati.*

Inizializza L[0] in modo che contenga solo s   
*Mi metto nel livello L[0] e inserisco la sorgente*

Poni il contatore dei livelli i = 0   
*Inizializzo l'indice i (il contatore dei livelli a 0)*

Inizializza il BFS tree T con un albero vuoto   
*Inizializzo l'albero BFST con un albero vuoto*

While L[i] non è vuota   
*Fino a che il livello considerato non è vuoto (si ferma non appena si scopre che all'iterazione precedente non siamo riusciti a mettere neanche un nodo nell'i-esimo livello)*

{

Inizializza L[i+1] con una lista vuota   
*Vai a creare il livello successivo ed inizializzalo ad una lista vuota.*

Foreach u che appartiene a L[i]   
*Per ogni u al livello corrente considera tutti gli archi incidenti su u*

Foreach arco (u, v) incidente su u  *Scandisce tutti i vertici v adiacenti ad u, e per ognuno di loro*

If Discovered[v] = false   
*Verifica se v è stato già visitato con il campo discovered, quindi se non è stato visitato.*

Poni Discovered[v] = true *Imposta il discovered a true*

Aggiungi v alla lista L[i+1] *Mette v nel livello L[i+1]*

Aggiungi l’arco (u, v) all’albero T   
*Aggiunge all'albero l'ARCO (u,v) (ARCO, NON VERTICE, È IMPORTANTE L'ARCO)*

Endif   
*Esco dall'IF*

EndFor   
*Esco dal FOR*

EndFor   
*Esco dal FOR*

i = i + 1   
*Passare alla prossima iterazione del while, quindi incremento i (Ovviamente se non ho trovato nessun nodo il livello sarà vuoto ed il while si fermerà)*

EndWhile   
*Esco dal While*

**Analisi tempo**

Stima sulle iterazioni del while, quanti nodi troverò nel totale di tutti i livelli che andrò a scandire? Al più n-1 (Dico al più perché il grafo potrebbe anche non essere connesso). Su tutte le iterazioni del while, il primo foreach viene eseguito O(n) volte.

Per ciascun nodo considerato esegue un numero di iterazioni apri al grado del nodo. Nel caso pessimo nei livelli troverò tutti i nodi (includendo anche la sorgente) e quindi dovrò considerare i gradi di tutti i nodi. La sommatoria dei nodi di un grafo, nel caso di un grafo non direzionato è 2m, altrimenti è m. BFS funziona sia nei grafi direzionati che non direzionati, quando si parla di "scandire nodi incidenti ad u" si intende considerare gli archi che partono da u, l'outdegree, quindi il secondo foreach viene eseguito O(m) volte. Per quanto riguarda il porre discovered[v]=true, anch'esso O(m) ma se vogliamo essere precisi possiamo dire che ogni nodo viene marcato esattamente una volta e quindi l'istruzione viene eseguita O(n) volte. L'aggiunta dell'arco (u,v) all'albero T viene fatta O(n) volte (Perchè un albero ha al più n-1 archi e quindi aggiungiamo al più n-1 archi). L'istruzione i=i+1 viene eseguita ogni volta che si cambia livello. Ne posso creare al più n-1 quindi O(n). Riassumento, quello che sta fuori dal foreach costa O(n), quello che sta dentro O(m), quindi l'algoritmo in totale costa O(n) + O(m), quindi O(n + m).

La BFS classicamente la si implementa con una coda FIFO. L'idea della BFS è quella di esaminare i nodi in base alla distanza dalla sorgente. Se io nella coda FIFO metto all'inizio la sorgente, poi la estraggo ed inizio ad esplorare tutti gli archi incidenti su essa e prendo tutti i vertici che ho raggiunto per la prima volta e li metto nella coda FIFO. Dopodiché prendo il nodo che sta in testa alla coda e riprendiamo la visita da quel nodo. In questo modo si raggiungono nuovi nodi, messi in coda alla coda per cui questi saranno considerati dopo quelli inseriti visitando il nodo precedente. Si estraggono i nodi in ordine di distanza dalla sorgente. Prima di fare il dequeue di un nodo che si trova al livello i devo aver fatto il dequeue di tutti i nodi che si trovano ai livelli precedenti. Ogni volta riprendo la visita dal nodo più vicino alla sorgente tra quelli di cui non ho ancora esplorato la lista di nodi adiacenti.

**BFS implementato con coda FIFO:**

Inizializza Q con una coda vuota

Inizializza il BFSTree T con un albero vuoto

Poni discovered[s] = true e discovered[v] = false per tutti gli altri v

Inserisci s in coda a Q con una EnQueue

While(Q non è vuota)

Estrai il front di Q con una DeQueue e ponilo in u

Foreach arco (u,v) incidente su u

If(Discovered[v] = false)

Poni Discovered[v] = true

Aggiungi v in coda a Q con una EnQueue

Aggiungi l’arco (u,v) al BFSTree T

Endif

Endforeach

Endwhile

**DFS(s)**

Mark u as "Explored” and add u to R

Foreach edge (u,v) incident to u

If v is not marked "Explored" then

Recursively invoke DFS(v)

Endif

Endfor

Quando si esaminano gli algoritmi ricorsivi si ignora il tempo della chiamata ricorsiva, si esamina il tempo di tutto ciò che sta fuori e si va a vedere quante chiamate ricorsive in totale vengono fatte. Si somma il tempo della singola chiamata per tutte le chiamate ricorsive.

Procediamo in questo modo, escludendo le chiamate a DFS(v) che sono al più il numero di archi che incidono su v, facciamo l'analisi di tutto ciò che c'è fuori:

* Settaggio del campo explored[u] costante
* for, eseguito deg(u) volte
* cosa faccio all'interno?
  + l'if, che costa tempo costante (ripeto, ignoriamo il tempo delle chiamate ricorsive)
  + Quindi il for costa O(deg(u))

Il tempo della singola DFS è O(deg(u)) + 1 (Che conta il tempo costante speso fuori)

Quante chiamate ricorsive effettuo in totale? Effettuo una chiamata ricorsiva per ogni nodo raggiunto, e ne faccio una sola, perché non appena effettuo la DFS su un nodo lo setto esplorato e siccome invoco la DFS solo sui nodi inesplorati, a ciascun nodo raggiunto corrisponde una ed una sola chiamata ricorsiva. Nel caso pessimo raggiungerò tutti i nodi, quindi la DFS sarà invocata una volta su tutti i nodi. Quindi per ottenere il tempo totale devo sommare quel tempo stimato per la singola chiamata su tutti i nodi, quindi sommatoria, per ogni u appartenente all'insieme dei vertici, di O(1) + *deg*(u).

Quindi, divido le sommatorie e ottengo sommatoria, per ogni u appartenente all'insieme dei vertici, + sommatoria, per ogni u appartenente all'insieme dei vertici, di *deg*(u). Ottengo che la sommatoria di 1 per ogni vertice all'interno dell'insieme dei vertici fa n perché l'insieme dei vertici è proprio n (quindi O(n)), e la sommatoria dei gradi dei vertici, sappiamo essere 2m o m, cioè O(m).

**Albero DFS**

**Algoritmo**

Poni Explored[v] = false per tutti i nodi

Inizializza il DFSTree T con un albero vuoto

DFS(u)

Explored[u] = true

Foreach arco (u,v) incidente su u

If(Explored[v] = false)

Aggiungi l’arco (u,v) al DFSTree T

DFS(v)

EndIf

EndForeach

**Proprietà**

* 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.
* 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.

Dimostrazione proprietà 2:

Supponiamo di avere l'arco (x,y) nel grafo G, allora nell'albero DFS, o x è discendente di y, o viceversa.

Consideriamo i due casi:

* Se (x,y) è in T è banale
* Se (x,y) non è in T:

Uno dei due sarà esplorato per prima, senza perdere di generalità dico che il nodo visitato prima è x.

Consideriamo il momento in cui viene fatta la DFS(x). Siccome x è adiacente ad y perché per ipotesi esiste l'arco (x,y), quando vado a scandire l'arco (x,y) nel foreach della DFS vado a scandire tutti gli archi incidenti su x, tra questi ci sarà anche (x,y). Ora, se y nel momento in cui viene considerato l'arco (x,y) non è stato ancora esplorato, y diventa figlio di x, ma questo l'ho già escluso perché sono nel caso in cui l'arco (x,y) non è un arco dell'albero. Quindi sicuramente quando scandisco l'arco (x,y) nel foreach, y è stato già esplorato, ciò vuol dire che y è esplorato tra l'inizio e la fine della chiamata DFS(u) e quindi per la proprietà 1, y è discendente di x.

"Ad un certo punto della visita devo aver incontrato x e averlo marcato esplorato. L'unico motivo per cui y può non diventare figlio di x è che quando io vado a scandire nel foreach io l'ho già visitato, quindi, quando ho cominciato la DFS(x), y non era esplorato, però quando me lo ritrovo nel foreach lo ritrovo già esplorato. Allora vuol dire che y è stato esplorato mentre andavo a visitare altri nodi raggiungibili a partire da x."

**Implementazione con lo stack**

(Ma lei da un'implementazione diversa, meno leggibile ma ripropone il comportamento della funzione ricorsiva, la versione presente sul libro non ha esattamente lo stesso comportamento di quella ricorsiva)

Utilizziamo uno stack, usandolo esplicitamente nella funzione iterativa. Essendo iterativa posso fare le inizializzazioni all'interno della funzione.

**DFS(s)**

*La funzione prende in input la sorgente*

Poni Explored[ s ] = true ed Explored[ v ] = false per tutti gli altri nodi   
*pone explored[s] a true e tutti gli altri explored a false*

Inizializza S con uno stack contenente s   
*inizializza lo stack con la sorgente.*

While S non è vuoto   
A questo punto entro nel while, fino a che lo stack non è vuoto:

Metti in u il nodo al top di S   
*Prendo l'elemento al top dello stack e lo metto in u*

If c'è un arco (u,v) incidente su u non ancora esaminato then   
*C'è un arco (u,v) incidente su u che non ho ancora preso in considerazione?*

If Explored[v] = false then   
*Se si, il nodo v in cui mi porta l'arco l'ho già esplorato?*

Poni Explored[ v ] = true   
*Se non l'ho già esplorato lo imposto come esplorato*

Inserisci v al top di S   
*e lo metto in cima allo stack*

Endif

else   
*Se no, vuol dire che ho finito con quel vertice, non posso prendere strade che mi portano a scoprire nuovi vertici*

Rimuovi il top di S   
*Quindi tolgo u dallo stack*

Endif

Endwhile

"In pratica, riconducendoci alla proprietà 1, u viene a trovarsi al top dello stack ad un certo punto, perchè è l'ultimo nodo esplorato. Lo prelevo con una top ma lo lascio nello stack e da questo momento in poi verranno aggiunti con delle push tutti i nodi che raggiungo a partire dalla visita su u che fino a quel momento erano inesplorati. In pratica aggiungerò allo stack tutti quei nodi su cui effettuo le chiamate ricorsive a partire dalla chiamata ricorsiva su u. Di tanto in tanto u sbucherà di nuovo al top dello stack, ogni volta che io faccio backtrack in u, e riprendo l'esecuzione di quel while. Traduciamo la proprietà 1 in questi termini: Tutti i nodi che io metto nello stack dal momento in cui inserisco u con una push nello stack e nel momento in cui estraggo u con una pop dallo stack, nell'albero sono discendenti di u"

**Andiamo ad analizzare la complessità computazionale**

Possiamo mantenere per ogni vertice u una lista di adiacenza, in cui ci sono i nodi vicini. Ciascuno di quei nodi, quindi, corrisponde ad un arco.

Supponiamo che l'ultimo arco ispezionato sia (u,z). Manteniamo un riferimento a z all'interno della lista di adiacenza, in modo da continuare al nodo successivo (per non riconsiderare sempre la lista dall'inizio). In questo modo l'if viene eseguito in tempo costante.

Inizializzare i campi explored richiede tempo costante.

Non abbiamo informazioni precise su quante volte viene eseguito il while, ma certamente posso dire che la push di un vertice viene fatta esattamente una volta se questo viene raggiunto, altrimenti non viene fatta (se non viene raggiunto)

"Metti in un il top dello stack". La singola top, tempo costante, ma quante volte viene eseguita? Ogni volta che u sbuca al top dello stack, ossia tutte le volte che faccio backtrack, quindi potenzialmente il numero di volte pari al grado. Se considero tutti i nodi u raggiungibili, nel caso pessimo sono tutti, devo fare la somma, per ogni nodo, del suo grado, quindi O(m).

Poi c'è "If c’è` un arco ( u , v ) incidente su u non ancora esaminato then". Prende in considerazione ciascun arco esattamente una volta. Grazie al fatto che ho inserito il riferimento, io non torno ogni volta sullo stesso arco. Lo faccio esattamente una volta per ogni arco e lo faccio in tempo costante, quindi viene eseguito tante volte quanti sono gli archi per ogni nodo, quindi è la somma, per ogni nodo, del suo grado, quindi O(m).

Per ogni vertice raggiunto, eseguo il pop una sola volta, quindi in totale, un numero di volte pari al numero di nodi raggiunti, ossia O(n).

Quindi, O(m) per quelle esecuzioni dell'if, O(n) per quel pop, totale O(m) + O(n) e quindi O(n + m)

**Versione alternativa:**

while(stack!=vuoto)

{

metti in u il top dello stack

for (tutti i nodi v adiacenti ad u)

if (v non è stato già esplorato)

push u nello stack

}

È una dfs ma l'ordine in cui vengono visitati i nodi non è esattamente lo stesso