Capitolo 4

Algoritmi di ricerca su grafi

4.1 Definizioni fondamentali

Definizione 4.1.1 *Dato un grafo non orientato $G(N, A)$, e un nodo $s \in N$, si definisce* Problema di ricerca non orientato *il problema di trovare un albero $H = (R, T)$ sottografo di G tale che $R \subseteq N$ è l'insieme di nodi di N connessi a u in $G(N, A)$.*

In altre parole, si tratta di trovare un albero i cui nodi siano tutti e soli i nodi di G connessi al nodo s (e gli archi siano archi di G).

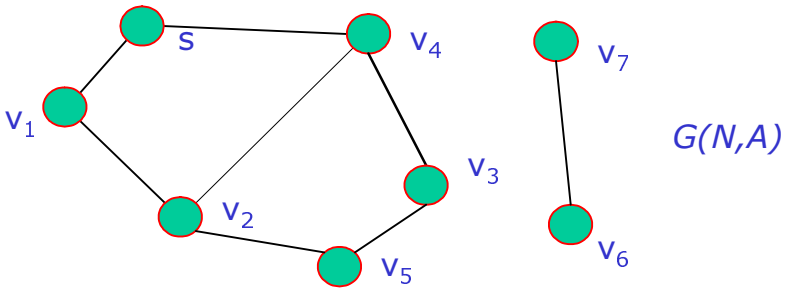

Figura 4.1: Un esempio

Se consideriamo il grafo in Figura 4.1, l'insieme dei nodi connessi a s risulta essere $R = \{s, v_1, v_2, v_3, v_4, v_5\}$; tre possibili alberi soluzione del problema di ricerca non orientato sono mostrati in Figura 4.2.

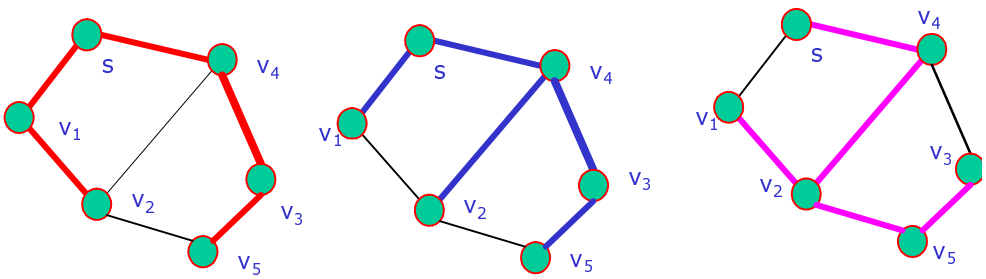

Figura 4.2: Tre possibili alberi

Un algoritmo che risolva il problema di ricerca non orientato può essere utilizzato per diverse applicazioni, quali ad esempio:

1. esaminare tutti i nodi connessi ad uno specifico nodo di N;
2. individuare le componenti connesse di $G(N, A)$;
3. individuare un sottografo ricoprente aciclico di $G(N, A)$;
4. verificare se $G(N, A)$ e' aciclico;
5. ...

Il prossimo teorema illustra una fondamentale proprietà degli alberi di ricerca. Sia $R = \{u \in N : s$ *connesso ad* u *in* $G(N, A)\}$. Allora:

Teorema 4.1.2 *Dato un albero* $H(S, Y)$ *(sottografo) di* $G(N, A)$ *con* $s \in S \subseteq R$ *allora* $S \neq R$ *se e solo se esiste un nodo* $w \in S$ *e un nodo* $y \in N - S$*:* $wy \in A$ *(ovvero* $wy \in \delta_G(S)$*).*

Dimostrazione. $\rightarrow$ (solo se). Essendo $S \neq R$ esiste $t \in R - S$. Quindi esiste un nodo $t \in N - S$ connesso ad s. Quindi esiste in G un cammino $P = (v_0, (v_0, v_1), \ldots, (v_{p-1}, v_p), v_p)$ di estremi $s = v_0$ e $t = v_p$. Poiché questo cammino ha un estremo in S e l'altro in $N - S$, deve esistere un "ultimo" nodo $v_r = w \in S$, con $0 \leq r \leq p - 1$; tutti i nodi seguenti $\{y = v_{r+1}, \ldots, v_p\}$ saranno nodi di $N - S$.

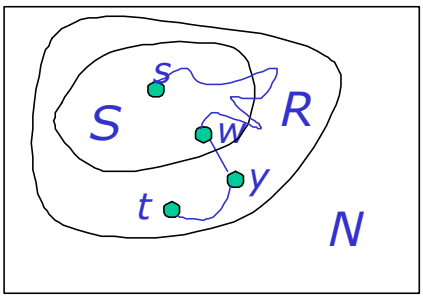

Figura 4.3: Dimostrazione necessità teorema 4.1.2

Quindi l'arco del cammino $(v_r, v_{r+1}) = (w, y)$ appartiene a $\delta_G(S)$.
$\leftarrow$ (se). Se esiste $wy \in \delta_G(S)$, questo implica che $y \in N - S$. Inoltre, poiché $w \in S$, w è connesso a s. Siccome w è connesso a y, per la proprietà transitiva anche s è connesso a y e quindi $y \in R$.

Il prossimo teorema mostra che ogni componente connessa di un grafo ammette (almeno) un albero ricoprente.

Teorema 4.1.3 *Dato un albero* $H(S, Y)$ *(sottografo) di* $G(N, A)$ *abbiamo che* $H(S \cup \{y\}), Y \cup \{wy\})$ *e' un albero di* $G(N, A)$ *per ogni arco* $wy \in \delta_G(S)$

Dimostrazione Bisogna mostrare che H' è un grafo connesso e aciclico. Essendo $H(S, Y)$ un albero, segue che H è connesso; ovvero, ogni nodo in S è connesso a w in H e quindi anche in H'. Ora, w è connesso a y in H': segue (per la transitività) che ogni nodo in S' è connesso a y in H' e quindi H' è un grafo connesso. Supponiamo ora per assurdo che H' contenga un ciclo C. Poiché $C \notin H$ (H è aciclico), segue che $y \in C$ e quindi esistono almeno due archi distinti di H' incidenti su y, contraddizione essendo wy l'unico arco di H' incidente in y.

I due precedenti teoremi suggeriscono un semplice meccanismo per costruire un albero $H(R, T)$ ricoprente della componente connessa di $G(N, A)$ che contiene un dato nodo s. L'idea è quella di costruire una sequenza di alberi di dimensione crescente aggiungendo un arco alla volta finché l'ultimo albero ricopre l'intera componente connessa. Schematicamente:

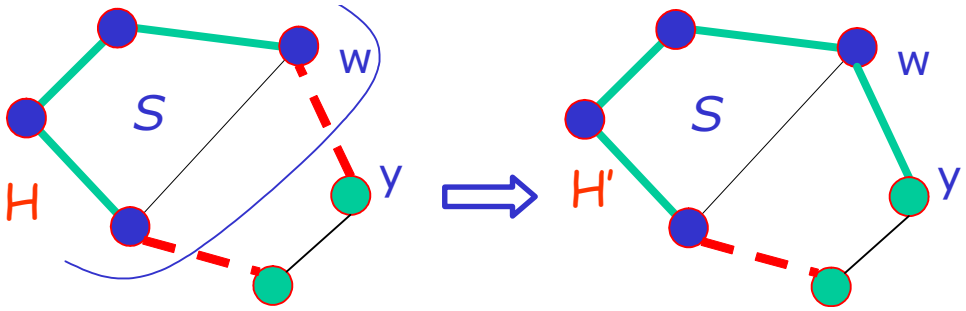

Figura 4.4: Enunciato del teorema 4.1.3

- Inizia con l'abero $H(S, Y)$ costituito dal solo nodo s, i.e. $S = \{s\}$ e $Y = \emptyset$.
- Ad ogni passo aggiungi a $H(S, Y)$ un arco di $\delta_G(S)$.
- Quando $\delta_G(S) = \emptyset$ allora $S = R$ e $H(S, Y) = H(R, T)$.

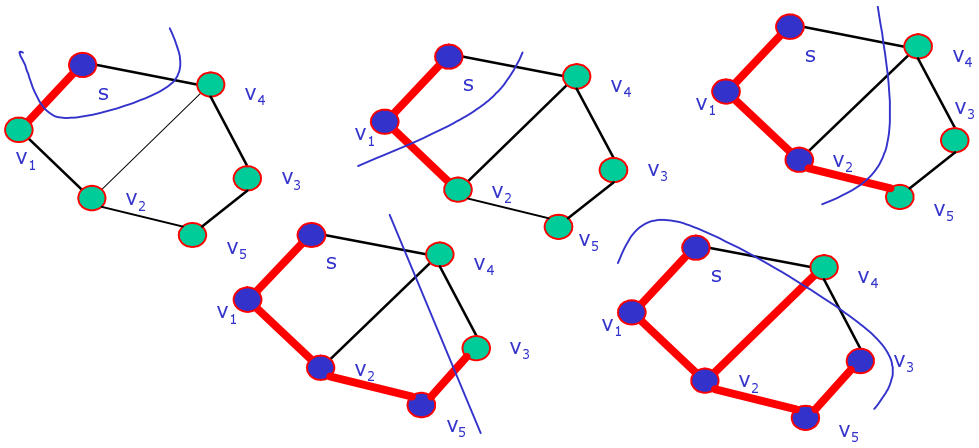

Figura 4.5: Costruzione di un albero di ricerca

4.2 Primi algoritmi di ricerca.

Un primo semplice algoritmo basato sulle idee e i teoremi presentati nel precedente paragrafo è illustrato in Figura 4.6. Sia dato quindi un grafo $G(N, A)$ di input e un nodo $s \in N$: l'algortimo dovrà costruire un albero $H(R, T)$ che ricopra la componente connessa di G che contiene s. L'algoritmo fa uso della variabile logica $S\_diverso\_da\_R$ che vale *true* se e solo se il corrente insieme S non coincide con l'insieme dei nodi R connessi a s.

Analizziamo ora il numero di passi che l'algoritmo dovrà eseguire. Cominciamo col valutare quante volte dovrà essere eseguito il blocco di istruzioni *Repeat ... until*. Il blocco viene eseguito finché l'insieme S non coincide con R. Poiché $R \subseteq N$ e, a ogni iterazione, viene aggiunto un nodo in S, il blocco verrà eseguito al più $|N| = n$ volte. All'interno del blocco, le istruzioni 5., 6. e 7. sono semplici assegnamenti e possono essere eseguiti in un passo (o in un numero costante di passi). Invece, il test all'istruzione 4. dipende dalla dimensione di $\delta_G(S)$ che, nel caso peggiore, sarà grande quanto $|A| = m$. Quindi, complessivamente, l'algoritmo dovrà compiere (nel caso peggiore) $O(nm)$ operazioni.

ALGORITMO DI RICERCA

1. $S := \{s\}$; $T := \emptyset$
2. Repeat
3. $\quad S\_diverso\_da\_R :=$ false
4. $\quad$ if esiste un arco $wy \in \delta_G(S)$
5. $\quad$ then { $S := S \cup \{y\}$
6. $\quad\quad T := T \cup \{wy\}$
7. $\quad\quad S\_diverso\_da\_R :=$ true
$\quad$ }
8. until (not $S\_diverso\_da\_R$)

Figura 4.6: Schema base del generico algoritmo di ricerca

Per capire ora come effettivamente vengono implementate le singole istruzioni, più specificamente l'istruzione 4., dobbiamo analizzare quali strutture vengono utilizzate e come tali strutture sono codificate in un programma.

Per i nostri esempi, considereremo di seguito il grafo rappresentato in Figura 4.7.

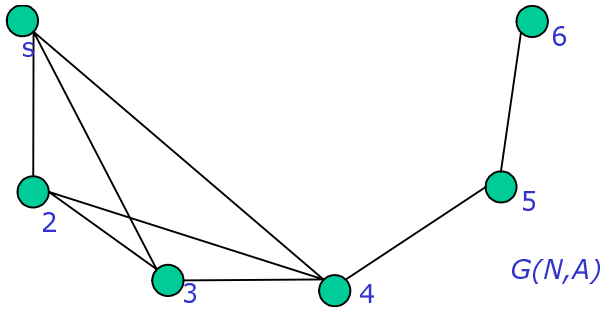

Figura 4.7: Grafo per ricerca

Scegliamo la rappresentazione a liste di adiacenza. Quindi, il nostro grafo sarà memorizzato come in Figura 4.8.

Le liste di adiacenza sono una stuttura dati astratta: esse possono essere rappresentate nella memoria del calcolatore in modi diversi, utilizzando le strutture dati primitive (cui si riferiscono alcune delle operazioni primitive viste nel Capitolo 3). Per implementare le liste di adiacenza utilizzeremo tre vettori, come in Figura 4.9. Un vettore di dimensione $2m$ e due vettori di dimensione n. Il primo vettore (che chiameremo *nodi*) contiene gli indici dei nodi nell'intorno di ciascun nodo: i.e., prima i nodi nell'intorno $N_G(s)$, poi i nodi in $N_G(2)$, etc. Essendo ogni arco non orientato (u, v) rappresentato due volte (una nell'intorno di u e l'altra nell'intorno di v) la dimensione del vettore sarà $2m$. Gli altri due vettori, denominati *START* e *END* contengono, per ogni nodo $u \in N$, rispettivamente, la posizione nel vettore *nodi* del primo e dell'utlimo nodo di $N_G(u)$.

Vedremo di seguito due algoritmi: uno che potremmo definire *naive* e che usa la rappresentazione data senza ulteriori strutture di appoggio e un altro che fa uso di una nuova struttura di supporto che ne migliora l'efficienza.

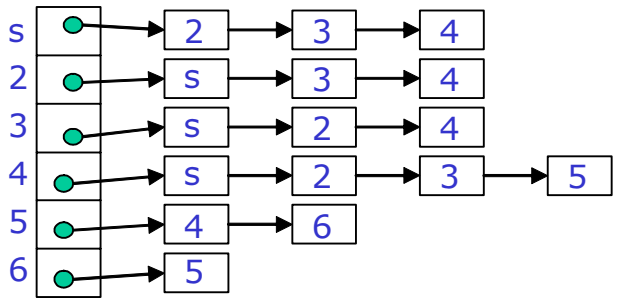

Figura 4.8: Liste di adiacenza per il grafo di Figura 4.7

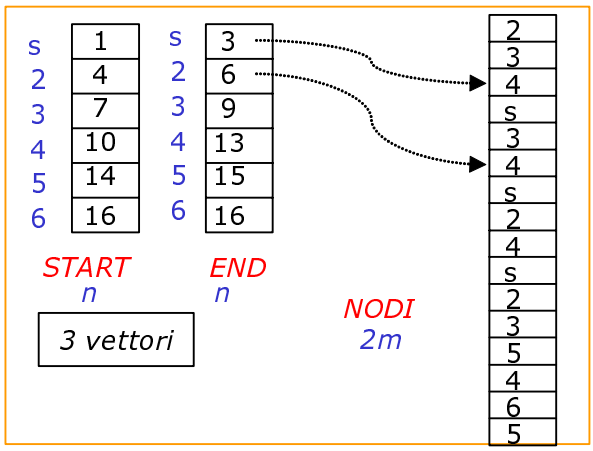

Figura 4.9: Rappresentazione vettoriale delle liste di adiacenza di Figura 4.8

4.3 L'algoritmo *naive*.

Questo algoritmo scandisce, a ogni iterazione, il taglio $\delta(S)$ utilizzando direttamente la rappresentazione a liste di adiacenza del grafo; non appena viene incontrato un nodo non in S, la scansione attuale termina e il nodo viene aggiunto a S. Si osservi infatti che, se u è un nodo di S, l'arco uv appartiene al taglio $\delta_G(S)$ se e solo se v appartiene all'intorno $N(u)$ di u e v *non appartiene* all'insieme S. Il taglio $\delta_G(S)$ viene dunque esplorato visitando l'intorno $N_G(u)$ per ogni $u \in S$; quest'ultima operazione corrisponde a scandire, per ciascun $u \in S$, la sua lista di adiacenti; la scansione viene interrotta non appena viene trovato un nodo che non appartiene a S, che corrisponde per quanto detto a un arco appartenente al taglio $\delta_G(S)$.

Una schematizzazione di quanto detto è descritta nell'algoritmo di Figura 4.3, che, ricevendo in ingresso un grafo orientato $G(N, A)$ e un insieme di nodi S, restituisce il taglio $C = \delta(S)$.

ALGORITMO DI VISITA del taglio $\delta(S)$

```
0. C = ∅
1. for w ∈ S
2.     for i = START[w] to i = END[w]
3.         u := nodi[i]
4.         if u ∉ S
5.             C := C ∪ {(w, u)}
6.         Endfor
7.     Endfor
```

Figura 4.10: Algoritmo di visita del taglio $\delta(S)$

Si osservi che, nel caso peggiore, l'algoritmo potrebbe scandire (due volte) l'intero insieme di archi di A. Inoltre, si può mostrare come le istruzioni che coinvolgono operazioni su insiemi (test di appartenenza all'istruzione 4, inserimento di un elemento all'istruzione 5.) possono essere eseguite in tempo costante. Abbiamo quindi il seguente semplice risultato di complessità:

Lemma 4.3.1 *(*Complessità della visita di un taglio) *La visita di un taglio $\delta(S)$ ha costo $O(m)$.*

L'algoritmo di visita del taglio può essere utilizzato come *building block* per definire un algoritmo di visita di un grafo. Quest'ultimo algoritmo scandisce a ogni iterazione il taglio $\delta(S)$ utilizzando l'algoritmo di visita del taglio senza effetivamente costruire il taglio stesso; inoltre, non appena viene incontrato un arco del taglio (cioè un nodo non appartenente a S), la scansione termina, il nodo viene aggiunto a S e l'arco viene inserito in T. L'algoritmo è mostrato in Figura 4.11.

Vediamo ora l'evoluzione dell'algoritmo quando è applicato al grafo di Figura 4.7. Alla prima iterazione (ogni iterazione corrisponde al blocco di istruzioni *While TROVATO* compreso fra l'istruzione 2. e l'istruzione 14.) dell'algoritmo naive viene scandita la lista di adiacenza dell'unico nodo s in S. Subito si incontra il nodo 2 e quindi la variabili TROVATO viene posta a true, il nodo 2 inserito nell'insieme S e l'arco $(s, 2)$ aggiunto all'insieme T.

Alla seconda iterazione viene di nuovo scandita la lista degli adiacenti del nodo s fino a incontrare il nodo 3; si inseriscolno quindi il nodo 3 in S e l'arco $(s, 3)$ in T. Alla terza iterazione si scandisce ancora la lista del nodo s fino a incontrare il nodo 4. Alla quarta iterazione si scandisce tutta la lista del nodo s senza incontrare nessun nodo nuovo; a questo punto comincia la scansione del nodo 2 senza incontrare nodi nuovi; quindi comincia la scansione del nodo 3 senza incontrare nodi nuovi; quindi comincia la

ALGORITMO DI RICERCA *naive*

---

1. $S := \{s\}$; $T := \emptyset$; TROVATO := true;
2. While TROVATO = true
3. TROVATO = false
4. for $w \in S$ and (TROVATO = false)
5. for $i = START[w]$ to $i = END[w]$ and (TROVATO = false)
6. $u := nodi[i]$
7. if $u \notin S$ {
8. $T := T \cup \{(w, u)\}$
9. $S := S \cup \{u\}$
10. TROVATO := true}
11. EndIf
12. Endfor
13 Endfor
14. EndWhile

Figura 4.11: Algoritmo di ricerca *naive*

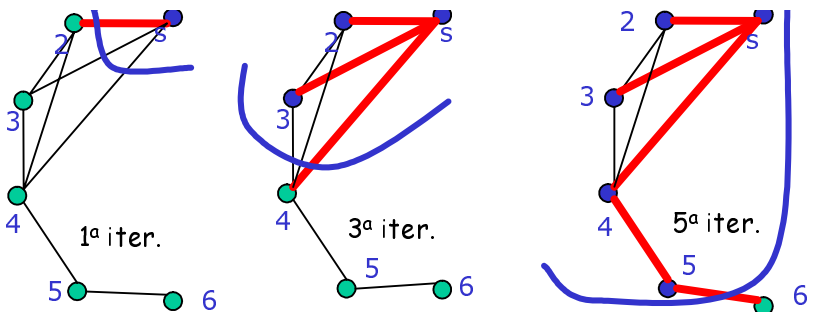

Figura 4.12: L'albero parziale dopo la prima, terza e quinta iterazione dell'algoritmo *naive*

scansione del nodo 4, alla fine della quale si incontra il nuovo nodo 5. Il nodo 5 viene inserito nell'insieme S e può cominciare una nuova iterazione. Come la precedente, la nuova iterazione comporta la scansione di tutti gli intorni di tutti i nodi attualmente in S fino a incontrare, nella stella di 5, il nodo 6. Alla sesta iterazione si scandiscono di nuovo tutte le liste di adiacenza e poiché non viene incontrato alcun nodo nuovo, la ricerca termina. Gli archi in T determinano l'albero di ricerca.

Un semplice calcolo permette di valutare la complessità dell'algoritmo nel caso peggiore. Si osservi che il blocco di istruzioni più annidato è quello compreso nell'ultimo blocco *for* (istruzioni da 5. a 10.). Verifichiamo quante volte potrebbero essere eseguite. Il ciclo *While* più esterno può essere eseguito fino a n volte in quanto a ogni iterazione viene aggiunto un nuovo nodo nell'insieme S e l'insieme stesso non può avere cardinalità maggiore di n. I due cicli *for* annidati scandiscono le stelle dei nodi attualmente in S; il numero di passi è quindi limitato dal numero totale di archi m. La complessità sarà quindi $O(nm)$.

4.4 Algoritmo *smart*

L'algoritmo *naive* può essere facilmente migliorato osservando che molti accessi a elementi delle liste di adiacenza possono essere evitati. In particolare, la lista di adiacenza di un nodo u in S viene visitata in genere più di una volta; ad esempio, il nodo 2 appartenente alla lista di adiacenza di s viene visitato a ogni iterazione. E' facile tuttavia convincersi che, per ogni nodo $v \in L(u)$, è sufficiente accedere a tale nodo v (in $L(u)$) una sola volta: infatti, se il nodo v non appartiene a S all'atto della prima visita, esso viene inserito in S (e mai rimosso nel corso delle successive iterazioni), mentre se il nodo v appartiene già ad S nessuna operazione verrà effettuata. Per evitare quindi di riaccedere a uno stesso nodo nella lista di adiacenza di u dobbiamo semplicemente ricordare qual è il primo nodo non ancora visitato in tale lista. Per la memorizzazione di questo elemento aggiungiamo quindi un ulteriore vettore $FIRST$ alle strutture dati definite precedentemente. Specificamente, $FIRST[u]$ conterrà un puntatore al primo elemento ancora da visitare nella lista degli adiacenti di u (vedi Figura 4.14). Quando tutti gli elementi della lista di u sono stati visitati, si pone $FIRST[u] = nil$. L'algoritmo di ricerca *smart* è mostrato in Figura 4.13. Oltre al vettore $FIRST$, l'algoritmo utilizza un insieme $Q \subseteq S$ definito come $Q = \{v \in S : FIRST[v] \neq nil\}$, ovvero Q è l'insieme dei nodi di S con stelle non ancora completatemente visitate. Essendo i puntatori rappresentati mediante gli indici dei nodi nel vettore *nodi*, il vettore $FIRST$ conterrà quindi degli interi. All'inizio il vettore $FIRST$ viene posto uguale al vettore $START$; quando si processa la stella del nodo u si visita il nodo v nell'intorno di u all'indice memorizzato dnel vettore $FIRST[u]$, cioè $nodi[FIRST[u]] = v$. Una volta visitato v si incrementa di un'unità $FIRST[u]$, che quindi punterà al prossimo elemento nella stella di u. Quando $FIRST[u] = END[u] + 1$ vorrà dire che la stella di u è stata completamente visitata. Ovvero, $FIRST[u] > END[u]$ rappresenta la condizione $FIRST[u] = nil$.

Alla prima iterazione dell'algoritmo comincia l'esplorazione della stella di s (essendo s l'unico elemento in Q). Poiché viene subito incontrato il nodo 2 non appartenente ad S, l'iterazione termina, e la componente s di $FIRST$ punterà al secondo elemento nella lista di adiacenza di s, il nodo 3. Tale situazione è illustrata in Figura 4.14.

La situazione dei vettori dopo la quarta iterazione dell'algoritmo è invece illustrata in Figura 4.15.

Si osservi che l'istruzione 3 dell'algoritmo *smart* richiede di identificare un nodo in Q mentre l'istruzione 5 richiede la rimozione di un nodo da Q; come vedremo nella prossima sezione, tali operazioni possono essere effettuate in tempo costante $O(1)$ mediante l'utilizzo di un'opportuna struttura dati. Ora, l'algoritmo richiede dunque un tempo $O(n)$ per l'inizializzazione del vettore $FIRST$; inoltre, ogni elemento nelle liste di adiacenza verrà visitato al più una volta; poiché il numero degli elementi nelle liste di adiacenza è pari a $2m$, avremo che il costo (in termini di numero di passi) di tale visita sarà $O(m)$. Quindi, complessivamente l'algoritmo *smart* richiederà $O(m + n) = O(m)$ passi.

ALGORITMO DI RICERCA *smart*

1. $S := \{s\}$; $T := \emptyset$; $Q := \{s\}$; $FIRST[u] = START[u]$, $\forall u \in N$
2. While $Q \neq \emptyset$
3. Scegli $w \in Q$
4. if $FIRST[w] > END[w]$
5. $Q := Q - \{w\}$
   else
6. $u := nodi[FIRST[w]]$
7. if $u \notin S$ {
8. $T := T \cup \{(w, u)\}$
9. $S := S \cup \{u\}$; $Q := Q \cup \{u\}$
   EndIf
10. $FIRST[w] := FIRST[w] + 1$
    EndIf
    EndWhile

Figura 4.13: Algoritmo di ricerca *smart*

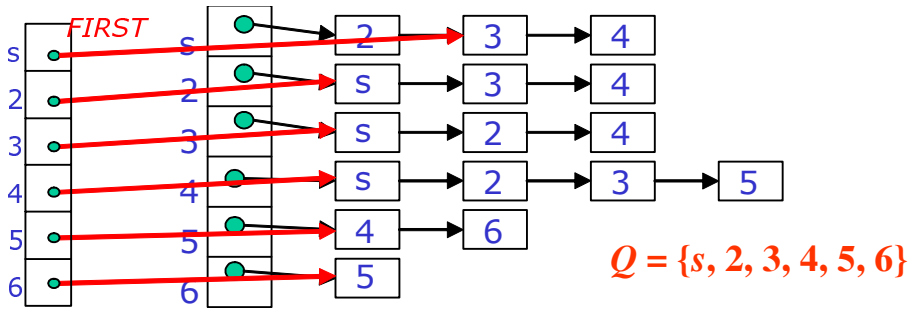

Figura 4.14: Situazione del vettore $FIRST$ dopo la prima iterazione dell'algoritmo *smart*

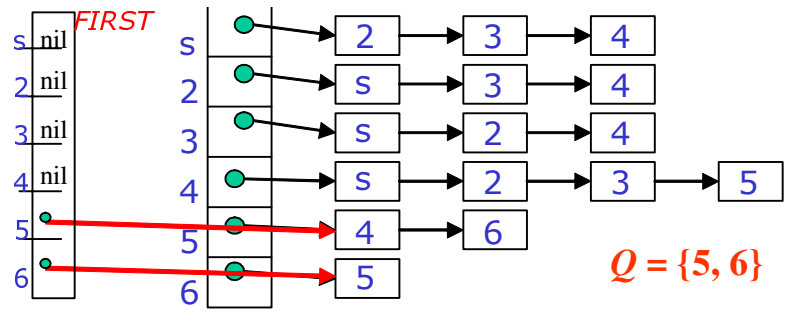

Figura 4.15: Situazione del vettore $FIRST$ dopo la quarta iterazione dell'algoritmo *smart*

4.5 Code e Stack.

L'algoritmo *smart* visto alla fine della sezione precedente visita il taglio $\delta(S)$ estraendo iterativamente un nodo w dall'insieme $Q \subseteq S$ e quindi visitandone (integralmente) la stella. Esistono ovviamente molti altri modi di visitare $\delta(S)$ e che dipendono dal modo in cui viene scelto (estratto) il nodo $w \in Q$ all'istruzione 3. Vedremo ora due diverse strategie che corrispondono ad altrettanti tipi di dati astratti da utilizzare nella rappresentazione di Q, denominati rispettivamente *coda* e *stack*. Nella struttura di dati coda, l'elemento estratto da Q coincide col primo elemento entrato in Q (strategia *FIFO*, *first in - first out*), mentre nella struttura dati stack, l'elemento estratto da Q è l'ultimo inserito (strategia *LIFO*, *last in - first out*).

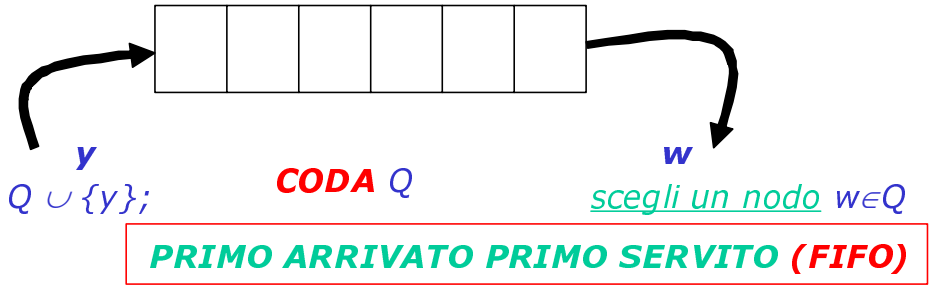

Figura 4.16: La struttura dati astratta *coda*

Sia la coda che lo stack prevedono le seguenti operazioni: inserimento di un elemento, estrazione di un elemento, rimozione di un elemento, test di insieme vuoto. Sia nella coda che nello stack, un nuovo elemento viene inserito in ultima posizione. Nella coda, tuttavia, l'elemento estratto (o rimosso) è quello in prima posizione (cioè il primo inserito). Nello stack invece, l'elemento estratto o rimosso è quello in ultima posizione (cioè l'ultimo inserito).

Mostriamo ora una possibile implementazione dello stack (omettiamo invece l'analoga per la coda). L'insieme Q viene rappresentato con un vettore a n componenti (pari alla massima dimensione potenziale di Q). Inoltre, si fa uso di una variabile d'appoggio $LAST$. $LAST$ punta all'ultimo elemento inserito nello stack. Inizialmente, $LAST = -1$ (stack vuoto). L'inserimento nello stack Q di un elemento u avviene semplicemente ponendo $LAST := LAST + 1$ e quindi $Q[LAST] := u$ (due operazioni primitive: incremento e assegnamento). L'estrazione dell'ultimo elemento inserito avviene in modo analogo, ponendo $u := Q[LAST]$, mentre l'eliminazione dell'ultimo elemento dallo stack corrispondende all'assegnamento: $LAST := LAST - 1$. Infine, verificare se lo stack è vuoto corrisponde a verificare la condizione $LAST = -1$. Quindi, ognuna di queste operazioni può essere svolta in numero costante di operazioni primitive.

Vediamo ora la produzione dell'algoritmo di ricerca quando applicato al grafo di Figura 4.7. Supponiamo innanzitutto che Q sia rappresentato utilizzando una coda.

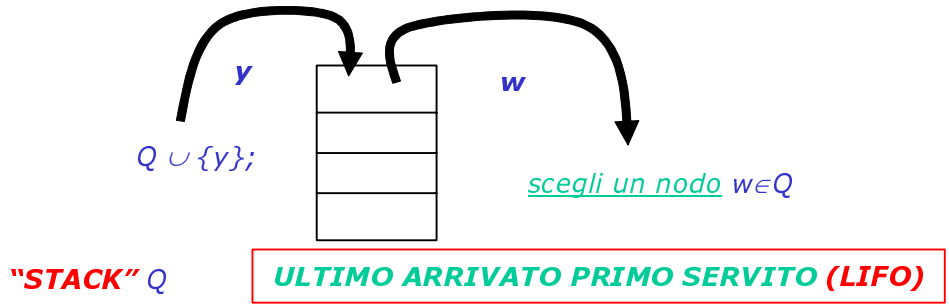

Figura 4.17: La struttura dati astratta *stack*

Alla prima iterazione la coda contiene solo il nodo s, che viene quindi estratto. La componente $FIRST[s] = 1$ punta all'elemento 2, che non appartiene ad S; l'elemento 2 viene quindi inserito in S e in Q, l'arco $(s, 2)$ viene inserito in T e $FIRST[s]$ incrementato e posto pari a 2.

Alla seconda iterazione è ancora s il primo elemento nella coda, e viene quindi estratto. $FIRST[s]$ punta al nodo 3, che viene aggiunto a Q e S, l'arco $(s, 3)$ viene inserito in T e $FIRST[s]$ incrementato.

Alla terza iterazione viene ancora estratto s. $FIRST[s]$ punta al nodo 4, che viene aggiunto a Q e S, mentre l'arco $(s, 4)$ viene inserito in T. A questo punto l'ulteriore incremento produce $FIRST[s] = 4 > END[s] = 3$.

Alla quarta iterazione il primo elemento della coda è ancora il nodo s, ma essendo $FIRST[s] = 4 > END[s] = 3$, il nodo s viene rimosso dalla coda e l'iterazione termina.

Alla quinta iterazione il primo elemento della coda è il nodo 2 che viene estratto; essendo tutti i nodi della sua stella già in S, nelle seguenti 3 iterazioni non viene aggiunto alcun nodo a S e a Q. A questo punto $FIRST[2] > END[2]$ e il nodo 2 viene rimosso dalla coda.

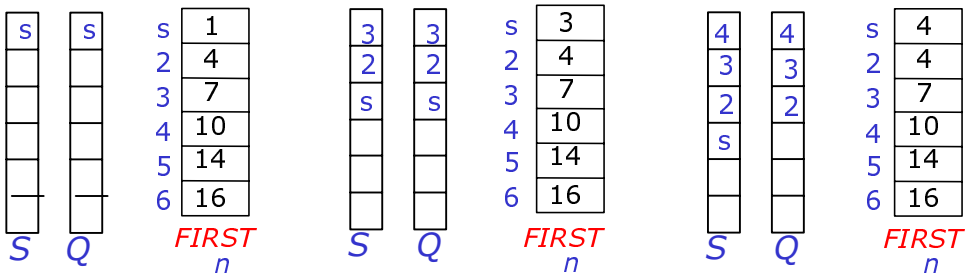

Figura 4.18: Le strutture S, Q e $FIRST$ all'inizio, dopo la terza e dopo la quinta iterazione dell'algoritmo smart utilizzando per Q una struttura CODA

Completando le iterazioni dell'algoritmo, si otterrà l'albero di Figura 4.19. Quest'albero, prodotto da una visita di tipo FIFO (o *in ampiezza*), è detto anche *albero BFS*, (*Breadth First Search*).

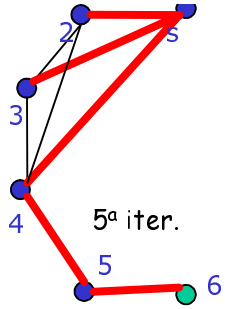

Figura 4.19: Albero di breadth first search (BFS)

Se usiamo invece una struttura dati di tipo stack, il comportamento dell'algoritmo sarà diverso. Specificamente, alla prima iterazione lo stack conterrà solo il nodo s, che verrà quindi estratto. La componente $FIRST[s] = 1$ punta all'elemento 2, che non appartiene ad S; l'elemento viene quindi inserito in S e in Q, l'arco $(s, 2)$ viene inserito in e $FIRST[s]$ incrementato e posto pari a 2.

Alla seconda iterazione è il nodo 2 l'elemento affiorante nello stack (l'ultimo inserito), e viene quindi estratto. $FIRST[2] = 4$ punta al nodo s, che appartiene già a S e non viene quindi aggiunto. $FIRST[2]$ viene incrementato, e posto pari a 5.

Alla terza iterazione è ancora 2 l'elemento affiorante. $FIRST[2] = 5$ punta al nodo 3, che viene aggiunto a Q e S e l'arco $(2, 3)$ viene inserito in T. Si pone quindi $FIRST[2] = 6$.

Alla quarta iterazione l'elemento affiorante è il nodo 3 che viene estratto; $FIRST[3] = 7$ punta all'elemento s che appartiene già a S e quindi l'unica operazione da compiere è porre $FIRST[3] = 8$.

Alla quinta iterazione l'elemento affiorante è il nodo 3 che viene estratto; $FIRST[3] = 8$ punta all'elemento 2 che appartiene già a S e quindi l'unica operazione da compiere è porre $FIRST[3] = 9$.

Alla sesta iterazione l'elemento affiorante è il nodo 3 che viene estratto; $FIRST[3] = 9$ punta all'elemento 4 che aviene inserito in S e in Q; quindi si incrementa $FIRST[3]$ e diventa $FIRST[3] = 10$.

Completando le iterazioni dell'algoritmo, si otterrà l'albero di Figura 4.20. Quest'albero, prodotto da una visita di tipo LIFO (o *in profondità*), è detto anche *albero DFS*, (*Depth First Search*).

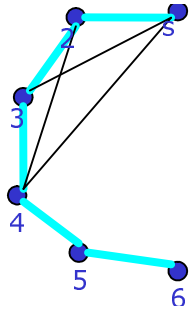

Figura 4.20: Albero di depth first search (DFS)

4.6 Il caso orientato.

Il problema di ricerca può essere esteso anche al caso di grafi orientati.

Definizione 4.6.1 *Dato un grafo orientato $G(N, A)$, e un nodo $s \in N$, si definisce* Problema di ricerca orientato *il problema di trovare un albero $H = (R, T)$ sottografo di G tale che l'insieme $R = \{u \in N :$ esiste un cammino orientato da s ad u in $G(N, A)\}$ ed inoltre esiste in H un cammino orientato da s a u per ogni $u \in R$.*

Il grafo H è detto *arborescenza*, e s è la *radice* di H.

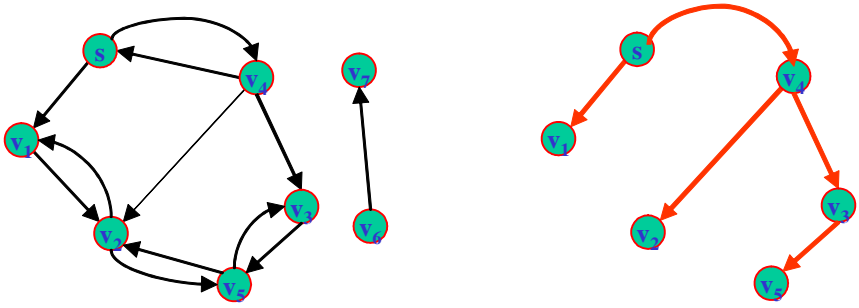

Figura 4.21: Un grafo orientato e una sua arborescenza

L'algoritmo di ricerca e le strutture dati utilizzate sono le stesse del caso non orientato; si ricordi tuttavia che la rappresentazione mediante liste di adiacenza di un grafo orientato mantiene solo le stelle uscenti da ogni nodo.

4.7 Esercizi di autovalutazione.

Esercizio 4.7.1 *Utilizzare gli algoritmi di visita per individuare le componenti connesse di* $G = (N, A)$

Soluz. Si scelga un nodo qualsiasi $u \in N$ e si applichi l'algotimo di ricerca utilizzando u come nodo iniziale (nodo s) dell'algoritmo di ricerca. L'applicazione dell'algoritmo fornirà in output l'insieme dei nodi $R(u)$ connessi a u. La corrispondente componente connessa sarà $G[R(u)]$. Se $N = R(u)$ abbiamo finito. Altrimenti si riapplica iterativamente il procedimento al grafo $G[N']$, con $N' = N - R(u)$.

Esercizio 4.7.2 *Completare la frase seguente: L'output dellAlgoritmo di Ricerca è ....*

Esercizio 4.7.3 *Completare la seguente definizione: Albero DFS di* $G(N, A)$ *rispetto ad un nodo* s*: un sottografo di* $G(N, A)$ *ottenuto dallAlgoritmo di Ricerca quando ...*

Esercizio 4.7.4 *Proporre un grafo di 5 nodi* $G(N, A)$ *per il quale gli alberi DFS e BFS rispetto ad un nodo* s *coincidano.*

Soluz. Tre possibili soluzioni sono illustrate nella Figura 4.22. Si tratta di tre alberi distinti (si osservi che cammini e stelle sono particolari alberi). In ogni caso, qualunque algoritmo di ricerca produrrà in output il grafo originale.

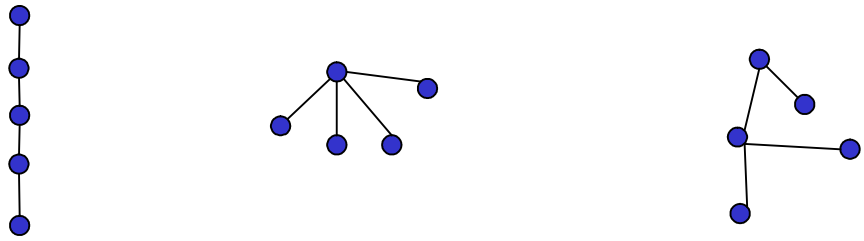

Figura 4.22: Tre grafi per cui DFS e BFS producono lo stesso output

Esercizio 4.7.5 *Definire una procedura che, utilizzando l'Algoritmo di Ricerca, determini se una coppia di nodi* u, v *di un grafo orientato simmetrica rispetto alla relazione di forte connessione.*

Soluz. Si applichi l'algoritmo di ricerca orientato due volte; la prima volta si pone $s = u$ e si verifica che $v \in R(u)$. La seconda volta si pone $s = v$ e si verifica che $u \in R(v)$.

Esercizio 4.7.6 *Definire una procedura che, utilizzando l'Algoritmo di Ricerca, determini le componenti fortemente connesse di un grafo orientato.*

Soluz. Si scelga un nodo qualsiasi $u \in N$ e si applichi l'algoritmo precedente a ogni coppia u, v, con $v \in N - \{u\}$, per cercare l'insieme dei nodi fortemente connessi a u. L'applicazione dell'algoritmo fornirà in output l'insieme dei nodi $R(u)$ fortemente connessi a u. La corrispondente componente connessa sarà $G[R(u)]$. Se $N = R(u)$ abbiamo finito. Altrimenti si riapplica iterativamente il procedimento al grafo $G[N']$, con $N' = N - R(u)$.

Esercizio 4.7.7 *Definire le strutture dati appropriate per memorizzare gli archi di* T *e modificare l'Algoritmo di Ricerca per aggiornarle ad ogni passo.*

Soluz. Gli archi possono essere memorizzati in una matrice *archi* di dimensione $|A| \times 2$. Sia u, v il k-esimo arco di T (cioè quello inserito all'iterazione k). Porremo $nodi[k][1] = u$ e $nodi[k][2] = v$. Come vedremo nel capitolo 5, il numero di archi di un albero con n nodi è pari a $n - 1$. Quindi potremmo limitarci a una matrice di dimensione massima pari a $n - 1 \times 2$

Esercizio 4.7.8 *Dimostrare che l'albero DFS associato ad un qualsiasi nodo s di un grafo completo $G(N, A)$ (un grafo che ha un arco per ogni coppia di nodi) è un cammino.*

Soluz. L'albero di DFS è costruito utilizzando uno stack per rappresentare l'insieme Q. Quindi, a ogni iterazione dell'algoritmo viene visitato l'ultimo nodo inserito in Q. Essendo il grafo un grafo completo, tutti i nodi sono adiacenti a tutti gli altri nodi, ovvero $N = \delta(u) \cup \{u\}$ per ogni $u \in N$. Sia dunque u l'ultimo nodo inserito in Q e supponiamo induttivamente che l'insieme di archi inseriti in T formi un cammino P di estremi s e u. Si possono verificare due casi.

1) Esiste almeno un nodo della stella di u non ancora in S. In particolare, sia v il primo nodo (nel senso che verrà visitato per primo) nella stella di u tale che $v \notin S$. A ogni iterazione, l'algoritmo visiterà il prossimo nodo nella stella di u finché non verrà incontrato v. Quando ciò avviene, il nodo v viene inserito in Q (e sarà quindi il prossimo selezionato) e l'arco uv viene inserito in T. Si vede facilmente che $T = P \cup \{(u, v)\}$ è un cammino di estremi s e v.

2) Tutti i nodi della stella di u sono in S. Poiché $u \in S$ e $\delta(u) \subset S$, essendo $u \cup \delta(u) = N$ si ha che $S = N$ e l'algoritmo termina ritornando il cammino P.

Esercizio 4.7.9 *Realizzare un programma (in un qualsiasi linguaggio) che accetti in ingresso un grafo e produca un albero DFS o BFS*