Lezione 9-10

La lezione è centrata su un esercizio per impratichirsi sui seguenti argomenti:

- grafi rappresentati con la matrice di adiacenza
- grafi rappresentati con la lista degli archi
- grafi rappresentati con il vettore delle liste di incidenza (*forward star*)

Ci concentriamo su grafi orientati, dato che quelli non orientati sfruttano le stesse rappresentazioni, con la convenzione che ad ogni lato (coppia non ordinata) corrispondono due archi opposti (cioè le due coppie ordinate composte dagli stessi elementi).

L'esercizio è anche dedicato all'analisi di complessità, dato che risolveremo lo stesso problema con diversi semplici algoritmi, adottando per ciascuno diverse implementazioni delle strutture dati astratte necessarie (anzi tutto, del grafo). Per ogni combinazione di algoritmo e implementazione, valuteremo la complessità.

Chi notasse eventuali incoerenze o errori, oppure avesse dubbi sul contenuto di queste pagine e dei codici, è pregato di segnalarmeli per contribuire a migliorare la qualità dei materiali del corso.

Problema

Si vuole scrivere un programma `sottografo.c` che riceva un grafo e un sottoinsieme di nodi, ed estragga dal primo il sottografo indotto dal secondo. Il programma riceve dalla linea di comando il nome di un file di testo che contiene il grafo e l'elenco degli indici dei nodi del sottografo. Deve poi stampare a video prima il grafo dato e poi il sottografo indotto in esso dai nodi elencati.

Formato di ingresso e di uscita Il grafo viene descritto descritto come un semplice elenco di archi. Ciascuno di questi è rappresentato come una coppia di indici numerici separati da una virgola e racchiusi fra parentesi tonde. Gli indici corrispondono ai nodi del grafo e sono compresi fra 1 e il numero totale n dei nodi stessi. Per semplicità si suppone che non ci siano nodi isolati, per cui ogni nodo compare in almeno un arco dell'elenco. Per esempio, il grafo usato nei lucidi corrisponde a:

```
(1,2) (1,5) (2,3) (3,5) (3,6) (4,1) (5,2) (5,4) (5,6)
```

L'elenco dei nodi che individuano il sottografo verrà passato dall'utente attraverso la linea di comando, semplicemente sotto forma di una sequenza di numeri interi separati da spazi. Il sottografo indotto verrà stampato a video nello stesso formato usato per il file di ingresso.

Traccia della risoluzione

Prima di eseguire l'esercizio, conviene scorrere interamente i lucidi per acquisire la terminologia e le rappresentazioni fondamentali. Il punto di partenza dell'esercizio è il file `sottografo0.c`, che contiene le solite direttive, costanti simboliche e procedure di base. Inoltre, è fornito un abbozzo di libreria per la gestione di grafi orientati, costituito dai file `grafo-la.h` e `grafo-la0.c`. In essa sono previste inizialmente solo le funzioni:

- `creagrafo`, che alloca le strutture necessarie a rappresentare un grafo con un dato numero n di nodi[1];
- `distruggegrafo`, che dealloca tutte le strutture dinamiche;
- `insarco`, che aggiunge un nuovo arco, dati i nodi estremi;
- `cancarco`, che cancella un arco, dati i nodi estremi;
- `esistearco`, che indica se una data coppia di nodi corrisponde a un arco.

La libreria adotta l'implementazione con matrice di adiacenza, ma all'inizio le sue funzioni sono definite tutte vuote.

Prima fase (`sottografo1.c`) Nella prima fase, applichiamo come di consueto la modalità *top-down*, chiamando e dichiarando le funzioni:

- `InterpretaLineaComando`, che estrae dalla linea di comando il nome del file che contiene il grafo dato e l'elenco dei nodi da considerare;
- `CaricaGrafo`, che carica il grafo dal file di testo indicato nella linea di comando;
- `StampaGrafo`, che stampa il grafo a video;
- `StampaSottografoIndotto`, che determina gli archi del sottografo indotto e li stampa a video.

Per il momento, definiremo tre delle quattro funzioni, lasciando vuota quella di calcolo e stampa del sottografo indotto, che è l'argomento principale della lezione.

La funzione `InterpretaLineaComando` è leggermente più sofisticata del solito, dato che non deve restituire un numero di argomenti noto a priori, ma un elenco di interi di lunghezza incognita (oltre ad una stringa con il nome del file dei dati). Oltre alla solita verifica del numero di argomenti (questa volta, non fisso, ma inferiormente limitato) e all'acquisizione del nome del file che contiene il grafo, la procedura deve leggere un elenco di numeri interi. La cardinalità dell'elenco non è nota a priori, ma è determinata dal valore di `argc`: i primi due elementi della linea di comando, infatti, sono il nome del programma e il nome del file dei dati; gli altri dovrebbero essere gli indici dei nodi, e sono contenuti nel vettore `argv` dall'indice 2 all'indice `argc-1`. A questo punto, si può allocare un vettore dinamico di interidella lunghezza opportuna. Siccome gli elementi di `argv` non sono interi, ma stringhe, vanno convertiti prima di copiarli uno per uno nel vettore (consiglio: `sscanf` o `atoi`). Al termine, il vettore e la sua cardinalità vanno restituiti al `main`. Per farlo, si passano entrambi per indirizzo. Siccome si è definito il tipo `vint` come un puntatore a intero che rappresenta un vettore dinamico di interi, l'argomento sarà di tipo `vint *`. Sarebbe equivalente (ma meno chiaro) dichiararlo di tipo `int **`. Volendo semplificare la procedura, Le operazioni sul vettore dinamico si potrebbero fare nel `main` o addirittura nella successiva procedura di stampa del sottografo indotto, passandole il vettore `argv`, ma sarebbe decisamente inelegante: si tratterebbe di interpretare una parte della linea di comando fuori della funzione dedicata a tale compito, mescolando l'acquisizione dei dati con l'esecuzione dell'algoritmo. Un facile esercizio di programmazione sarebbe invece di caricare l'elenco dei nodi da un secondo file di testo, anziché leggerlo dalla linea di comando.

[1] Per semplicità, imponiamo la limitazione che il numero dei nodi sia dato all'inizio e invariabile. Una libreria più flessibile potrebbe partire da un grafo nullo (senza nodi) e includere funzioni per inserire e cancellare anche nodi, oltre che archi.

La funzione `CaricaGrafo` riceve il nome del file di testo e restituisce il grafo. Questo viene quindi passato per indirizzo. La funzione apre il file di testo, determina il numero dei nodi in modo da poter creare il grafo, legge gli archi uno alla volta e li aggiunge via via al grafo stesso. Questa decomposizione del problema consente di usare le funzioni fondamentali di gestione del grafo elencate più sopra. Siccome il file non fornisce il numero dei nodi del grafo, dobbiamo ricavarlo indirettamente. In generale, non sarebbe possibile, ma l'ipotesi che non ci siano nodi isolati permette di supporre che l'ultimo nodo sia quello di indice massimo che compare negli archi del grafo. Quindi, scorriamo il file una prima volta per leggere gli indici e conservare il massimo, e una seconda volta per aggiungere al grafo gli archi. Entrambi gli scorrimenti richiedono di leggere un formato non del tutto banale, con coppie di interi racchiusi fra parentesi e separati da una virgola. La funzione `fscanf` con la specifica `" (%d ,%d )"` risolve il problema[2].

Infine, definiamo la procedura che riceve il grafo e ne stampa a video gli archi. Contrariamente alla lezione sulle liste, questa volta includeremo la funzione nella libreria, perché sarebbe inefficiente se realizzata usando le poche funzioni di libreria indicate nella lezione e perché è potenzialmente utile per altre applicazioni (il formato di stampa è abbastanza naturale da supporre che non cambi). Fra l'altro, nel seguito scopriremo che le altre implementazioni del grafo suggeriranno l'introduzione di altre funzioni di libreria, che consentiranno implementazioni più efficienti della stampa, e che quindi suggerirebbero di modificare pesantemente la funzione stessa: meglio inserirla nella libreria fin dal principio. Chiameremo questa funzione `stampagrafo`, per coerenza coi nomi interamente minuscoli delle altre funzioni di libreria, e ne aggiungeremo la dichiarazione al `grafo-ma.h` e la definizione al file `grafo-ma.c`. A rigore, il grafo andrebbe passato alla funzione per valore, dato che non viene modificato, ma lo passiamo per indirizzo, perché questo riduce il passaggio di strutture di grandi dimensioni alla copia di un puntatore, rendendolo più efficiente.

Seconda fase (`grafo-ma.c`) Analogamente alla scorsa lezione, nella seconda fase passiamo alla modalità *bottom-up* per realizzare la libreria di gestione di grafi, in modo da avere subito dei risultati a video (la stampa del grafo) che ci consentano di verificare almeno in parte la correttezza dell'algoritmo. Come indicato nei lucidi, la libreria definisce:

- i nodi come numeri interi[3] da 1 a $n = |N|$;
- la relazione di adiacenza fra i nodi (dunque, gli archi) come una matrice quadrata di ordine n di valori logici (`TRUE`, cioè 1, se l'arco fra nodo di riga e di colonna esiste; `FALSE`, cioè 0, altrimenti); aggiungiamo anche il numero degli archi, anche se non strettamente necessario.

[2] Gli spazi prima delle parentesi e della virgola servono a rendere flessibile la lettura, cioè a evitare che si interrompa per non aver trovato le parentesi e la virgola esattamente al punto indicato: lo spazio indica che può esserci un numero qualsiasi, anche nullo, di separatori. Non occorrono spazi dopo le parentesi e la virgola perché anche la specifica `%d` consente la presenza di un numero qualsiasi di separatori.

[3] In linea di principio, i nodi potrebbero avere molte informazioni associate, e si potrebbe trovare più conveniente definirli come puntatori a tali strutture. Volendo nascondere la loro definizione, si può ricorrere alla definizione di un tipo `nodo` con l'istruzione `typedef`. In realtà, in questo esercizio ci interessa solo la topologia del grafo e, anche se i nodi del grafo modellassero oggetti sofisticati, potremmo modellare le relative informazioni come funzioni di peso conservate in vettori e usare usare gli indici numerici dei nodi per accedervi. Che gli indici siano consecutivi semplifica la trattazione, perché consente di rappresentare l'intero insieme N semplicemente con la cardinalità n.

Cominciamo con le funzioni di creazione e distruzione, `creagrafo` e `distruggegrafo`. Queste funzioni consistono essenzialmente nell'allocazione e deallocazione della matrice di adiacenza, che è una matrice dinamica (all'occorrenza, si veda la pagina 16 dei lucidi della lezione 5).

Le funzioni di inserimento e cancellazione di un arco, `insarco` e `cancarco`, sono quasi banali: si tratta di inserire il valore `TRUE` o `FALSE` nella cella opportuna della matrice di adiacenza, e aggiornare il numero degli archi. Il tutto richiede tempo costante. Che succede se l'arco da inserire è già presente, o quello da cancellare manca? Si può scegliere di lasciare all'utente della libreria la responsabilità di eseguire l'operazione solo quando è lecita. In tal caso, conviene riportare questa indicazione nei commenti al prototipo della funzione, così che l'utente sia informato delle condizioni per l'uso corretto della funzione. Oppure si può scegliere di verificare la condizione e abbandonare l'esecuzione con un messaggio di errore se è violata (è una soluzione piuttosto drastica). Oppure si può decidere di verificare la condizione e, in caso di violazioni, non fare nulla, come succede nella teoria degli insiemi quando si aggiunge a un insieme un elemento che già contiene o se ne sottrae uno che non contiene. In questa implementazione, la verifica ha un costo molto basso, per cui si è scelta quest'ultima strada, ma nelle altre implementazioni il controllo appesantisce moltissimo la complessità. Sceglieremo quindi la strada dell'efficienza, assegnando all'utente della libreria la responsabilità di eseguire l'operazione solo quando è lecita. Conviene riportare questa indicazione nei commenti al prototipo della funzione, così che l'utente sia informato delle condizioni per l'uso corretto della funzione.

La funzione `esistearco`, che è l'ultima ancora da definire, consente proprio di valutare la condizione di appartenenza. La definizione è banale e richiede tempo costante: basta consultare la matrice. Ci si può chiedere che senso abbia chiamare una funzione per farlo: al solito, si tratta di mantenere separate la struttura dati astratta e la sua implementazione.

Terza fase (sottografo-ma.c) La fase successiva dell'esercizio definisce la funzione `StampaSottografoIndotto`, che, dato un grafo $G = (N, A)$ e un sottoinsieme di nodi $S \subseteq N$, determina gli archi del sottografo $G_S = (S, A_S)$ indotto da S su G e poi li stampa. Ci sono diversi algoritmi possibili per individuare gli archi del sottografo indotto. Ne analizziamo tre in maggior dettaglio.

Algoritmo 1 scorre tutte le coppie ordinate (o, d) di nodi di N e valuta per ciascuna tre condizioni: se ciascuno dei due nodi estremi appartenga al sottoinsieme S e se la coppia corrisponda a un arco di A. Se la coppia soddisfa tutte le condizioni, la stampa. Si può chiaramente descrivere questo algoritmo col seguente pseudocodice.

```
1: for all o ∈ N do
2:     for all d ∈ N do
3:         if o ∈ S and d ∈ S and (o, d) ∈ A then
4:             StampaArco(o, d);
5:         end if
6:     end for
7: end for
```

Per determinare se un dato nodo appartiene o no al sottoinsieme S, siccome il sottoinsieme è rappresentato con un vettore s (di oggetti di tipo `nodo`, che sono poi dei semplici interi), introdurremo una funzione `appartienetabella`[4], che riceverà

[4] A rigore, si tratta di un vettore, dato che la sua dimensione non cambia, ma trattandosi di una parte del più ampio vettore che contiene tutti i nodi viene spontaneo pensarlo come una tabella. Ad ogni modo, non cambia nulla.

l'elemento da cercare, il vettore s e la sua cardinalità $n_S = |S|$. La complessità temporale dell'Algoritmo 1 è quindi data da una sommatoria doppia, sugli indici dei nodi, del tempo richiesto dalla valutazione delle tre condizioni e dalla stampa:

$$T_1 = \sum_{o=1}^{n} \sum_{d=1}^{n} (2T_S + T_A + T_{\text{pr}}) \tag{1}$$

dove $T_S(\cdot)$ è il tempo richiesto per la valutazione dell'appartenenza di un elemento di N a S, $T_A(\cdot)$ quello per la valutazione dell'appartenenza di una coppia di elementi di N ad A e $T_{\text{pr}}(\cdot)$ il tempo per la stampa di un arco.

Se adottiamo l'implementazione del grafo con la matrice di adiacenza, la valutazione di esistenza dell'arco richiede tempo costante: $T_A \in \Theta(1)$. La valutazione di appartenenza a S di ciascuno dei due nodi richiede tempo $T_S \in \Theta(n_S)$, dato che corrisponde allo scorrimento del vettore s che elenca gli indici dei nodi[5]. Infine, la stampa dell'arco richiede tempo costante: $T_{\text{pr}} \in \Theta(1)$. Quindi, la complessità dell'Algoritmo 1 in questa implementazione è:

$$T_1(n, n_S) = \sum_{o=1}^{n} \sum_{d=1}^{n} (2\Theta(n_S) + \Theta(1) + \Theta(1)) \in \Theta\left(n^2 n_S\right) \tag{2}$$

Questa analisi dipende non da un parametro dimensionale, ma da due: il numero totale dei nodi e quello dei nodi del sottoinsieme S. Si può osservare che, nel caso pessimo, $n_S = n$, e quindi la complessità è in $\Theta\left(n^3\right)$. Questa analisi è corretta, ma più grossolana, e ignora il fatto che su alcune istanze (quelle relative a sottografi indotti da piccoli sottoinsiemi di nodi) l'algoritmo è più veloce che su altre.

La complessità spaziale dell'Algoritmo 1 è dominata dallo spazio occupato dal grafo: $S_1(n) \in \Theta\left(n^2\right)$.

Algoritmo 2 scorre tutti i nodi di N come potenziali origini, valuta per ciascuno se appartenga al sottoinsieme S; per quelli che vi appartengono, scorre tutti i nodi di N come potenziali destinazioni, valuta per ciascuno l'appartenenza al sottoinsieme S; infine, valuta se la coppia di nodi corrisponda a un arco di A e, in tal caso, la stampa.

```
1: for all o ∈ N do
2:     if o ∈ S then
3:         for all d ∈ N do
4:             if d ∈ S and (o, d) ∈ A then
5:                 StampaArco(o, d);
6:             end if
7:         end for
8:     end if
9: end for
```

È semplicemente un miglioramento dell'Algoritmo 1, basato sull'osservazione che la valutazione di appartenenza del primo nodo di ogni coppia al sottoinsieme può essere fatta nel ciclo esterno anziché in quello interno. In questo modo, se la valutazione fallisce, si evita di richiamare il ciclo interno. La complessità spaziale di questo algoritmo è ovviamente identica a quella del precedente. Quella temporale richiede un'analisi un po' più complessa, ma molto simile: il ciclo esterno ha n iterazioni che eseguono tutte la valutazione di appartenenza, ma solo n_S di queste

[5] E se il vettore S fosse ordinato e si usasse la ricerca binaria? Avremmo un costo aggiuntivo al principio per ordinare il vettore ($\Theta(n_S \log n_S)$) e una complessità inferiore durante l'algoritmo ($T_S \in \Theta(\log n_S)$). Non approfondiamo questa soluzione perché vedremo che gli algoritmi che spendono T_S sono in realtà tutti dominati.

iterazioni proseguono con l'esecuzione del ciclo interno, mentre le altre $n - n_S$ si arrestano subito:

$$T_2 = \sum_{o=1}^{n} T_S + \sum_{i_o=1}^{n_S} \sum_{d=1}^{n} (T_S + T_A + T_{\text{pr}}) \tag{3}$$

Adottando sempre l'implementazione del grafo con matrice di adiacenza, si ottiene:

$$T_2(n, n_S) = \Theta(n n_S) + n_S\, n\, (\Theta(n_S) + \Theta(1) + \Theta(1)) = \Theta\left(n n_S^2\right) \tag{4}$$

È facile vedere che la nuova complessità è identica alla vecchia per $n = n_S$, ma è molto inferiore per $n_S \ll n$.

Algoritmo 3 cambia completamente approccio, perché sfrutta il vettore che contiene i nodi del sottografo in modo da scorrere solamente loro. In dettaglio, l'algoritmo scorre tutte le coppie di posizioni nella tabella, ricava per ciascuna posizione il corrispondente nodo, valuta se la coppia di nodi corrisponda a un arco in A, e in tal caso lo stampa.

```
1: for i_o := 1 to |S| do
2:     o := s[i_o];
3:     for i_d := 1 to |S| do
4:         d := s[i_d];
5:         if (o, d) ∈ A then
6:             StampaArco(o, d);
7:         end if
8:     end for
9: end for
```

Quindi:

$$T_3 = \sum_{i_o=1}^{n_S} \left(T_{\text{find}} + \sum_{i_d=1}^{n_S} (T_{\text{find}} + T_A + T_{\text{pr}}) \right) \tag{5}$$

dove $T_{\text{find}}(\cdot)$ è il tempo richiesto per trovare il nodo in N dato un elemento del vettore che rappresenta S.

Adottando l'implementazione del grafo con matrice di adiacenza, e osservando che il vettore fornisce il nodo con un banale accesso in tempo costante ($T_{\text{find}} \in \Theta(1)$):

$$T_3(n, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} (\Theta(1) + \Theta(1) + \Theta(1)) \right) = \Theta\left(n_S^2\right) \tag{6}$$

La complessità temporale dell'Algoritmo 3 è molto migliore di quella dei precedenti, anche nel caso pessimo, mentre quella spaziale non cambia.

I tre algoritmi sono via via più efficienti. Questo non dipende in realtà dall'implementazione, anche se l'implementazione determina l'espressione precisa della loro complessità. Infatti, per qualsiasi espressione di $T_S(\cdot)$, $T_A(\cdot)$ e $T_{\text{pr}}(\cdot)$ il secondo algoritmo dominerebbe il primo, dato che fa le stesse cose, ma in alcuni casi evita il ciclo interno. Il terzo algoritmo domina il secondo purché T_{find} non sia più lenta di T_S, cosa che avviene in qualsiasi implementazione sensata. Nel seguito, quindi, abbandoneremo i primi due algoritmi.

D'altra parte, questi non sono i soli algoritmi possibili. Nel seguito ne introdurremo un quarto. Inoltre, discuteremo un'implementazione del sottoinsieme dei nodi S che consenta di ottenere un test di appartenenza più rapido. Ovviamente,

è possibile combinare i quattro algoritmi con le tre implementazioni del grafo e le due implementazioni del sottoinsieme, ottenendo 24 casi, ma il modo in cui stiamo procedendo dovrebbe dare un'idea più chiara dei meccanismi all'opera nel progetto e nella valutazione di algoritmi, del ruolo che in essi giocano da un lato le strutture dati astratte e dall'altro le implementazioni. La trattazione dovrebbe anche dare un buon esempio del modo più scorrevole per descrivere un algoritmo e analizzare la sua complessità.

Quarta fase (grafo-la.c) A questo punto, procediamo a sostituire in blocco la libreria basata sulla matrice di adiacenza con una libreria basata sulla lista degli archi. Il file `grafo-la.h` riporta le definizioni di base, mentre occorre definire una per una le funzioni nel file `grafo-la0.c`, esattamente come si è fatto per l'implementazione con la matrice di adiacenza. Per semplificare l'esercizio, è già interamente definita una libreria `listaarchi` che gestisce le liste di archi, realizzandone un'implementazione a puntatori bidirezionale, circolare e con sentinella. I principi di fondo sono quelli discussi nella lezione sulle liste.

Ancora una volta, dobbiamo definire le funzioni di creazione, distruzione, aggiunta e cancellazione di archi e di valutazione dell'esistenza di un arco dati gli indici dei nodi estremi. Per farlo, sfrutteremo la disponibilità di una libreria per gestire liste di archi (`listaarchi.h` e `listaarchi.c`), che rende quasi banale l'implementazione di gran parte delle nuove funzioni. La creazione e la distruzione del grafo sfruttano la creazione e la distruzione della lista degli archi, anziché della matrice di adiacenza. L'aggiunta di un nuovo arco consiste nella sua aggiunta alla lista (per analogia con l'esercizio sulla biblioteca, aggiungeremo gli archi in coda, anche se qualsiasi posizione sarebbe corretta). La cancellazione di un arco è più complessa, perché la funzione di cancellazione dalla lista richiede di conoscere la sua posizione nella lista, mentre sono noti solo gli indici dei nodi estremi. Per trovare la posizione di un arco a partire dagli estremi, sarà necessario scorrere la lista. Questa posizione si potrà poi usare per cancellare l'arco. Infine, la funzione che valuta l'esistenza di un arco dati gli estremi esegue anch'essa uno scorrimento della lista. L'aspetto interessante è che, esattamente come nell'esercizio sulla gestione di una biblioteca, se si sostituisce la nuova libreria alla vecchia nelle direttive di inclusione e nella linea di comando della compilazione, il programma continua a funzionare correttamente senza modifiche.

Cambia però completamente la complessità dei tre algoritmi discussi più sopra. Anzi tutto, la complessità spaziale migliora in alcuni grafi, mentre peggiora in altri. Il grafo occupa infatti spazio $\Theta(m)$ e $m \leq n^2$ (in assenza di autoanelli, si ha $m \leq n(n-1)$). Questo significa che per i grafi sparsi l'occupazione è molto inferiore, mentre per quelli densi è asintoticamente la stessa, ma ha costanti moltiplicative più alte, per la presenza dei puntatori e della sentinella[6]. Quindi l'implementazione migliore dal punto di vista spaziale dipende dal tipo di istanze che probabilmente ci si troverà a risolvere.

Per quanto riguarda il tempo, abbiamo mostrato che i primi due algoritmi sono dominati dal terzo. Limitiamoci a discutere la complessità di quest'ultimo[7], e poi introduciamone un quarto.

Algoritmo 3 La ricerca di un nodo in N dato il suo indice in S e la stampa di un arco richiedono ancora tempo costante. La valutazione di esistenza di un arco dati gli estremi richiede invece tempo lineare nel numero degli archi ($T_A \in \Theta(m)$)

[6] Implementare la lista degli archi come un vettore risolverebbe questo problema.

[7] Per esercizio, si valutino le altre complessità: dovrebbero essere $\Theta\left(n^2(m+n_S)\right)$ per l'Algoritmo 1 e $\Theta(nn_S(m+n_S))$ per l'Algoritmo 2.

se si adotta l'implementazione con lista degli archi[8]. Adattando l'espressione (5), si ottiene la stima:

$$T(n, m, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} (\Theta(1) + \Theta(m) + \Theta(1)) \right) \in \Theta\left(mn_S^2\right) \qquad (7)$$

che è peggiore di un fattore m di quella ottenuta con l'altra implementazione. D'altra parte, l'implementazione con lista di archi suggerisce un altro possibile algoritmo per determinare il sottografo indotto.

Algoritmo 4 Questo algoritmo è basato sull'idea di scorrere direttamente gli archi, anziché le coppie di nodi estremi potenziali. L'algoritmo scorre tutti gli archi del grafo e per ciascuno determina i nodi estremi e valuta se entrambi appartengano al sottoinsieme S, come nello pseudocodice seguente.

```
1: for all a ∈ A do
2:     (o, d) := LeggeEstremiArco(a);
3:     if o ∈ S and d ∈ S then
4:         StampaArco(o, d);
5:     end if
6: end for
```

Questo algoritmo richiede che la struttura dati astratta "grafo" consenta operazioni di scorrimento degli archi e di accesso ai loro estremi, di cui dovremo aggiungere alla libreria le dichiarazioni e le definizioni. Precisamente, aggiungeremo le seguenti quattro funzioni:

- `primoarco`, che restituisce la posizione del primo arco nella lista degli archi del grafo;
- `succarco`, che restituisce la posizione dell'arco successivo a quello dato nella lista degli archi del grafo;
- `finearchi`, che indica se la posizione corrente è esterna alla lista degli archi del grafo;
- `leggeestremiarco`, che determina i nodi estremi di un arco data la sua posizione nel grafo.

Per farlo, sfrutteremo la disponibilità di una libreria per gestire liste di archi (`listaarchi.h` e `listaarchi.c`), che rende l'implementazione delle nuove quattro funzioni quasi banale. Tutte richiedono tempo costante, come avevamo ipotizzato più sopra. In effetti, l'implementazione con lista degli archi è particolarmente adatta a realizzare queste operazioni, grazie all'organizzazione degli archi in una lista.

Se si ipotizza che lo scorrimento di A richieda tempo costante per ogni arco, la complessità dell'Algoritmo 4 si può stimare come:

$$T = \sum_{a=1}^{m} (2T_S + T_{\text{extr}} + T_{\text{pr}}) \qquad (8)$$

dove $T_{\text{extr}}(\cdot)$ rappresenta la complessità di determinare gli estremi di un arco dato, che è costante se si suppone che tale informazione sia conservata nella struttura che descrive l'arco.

[8] Mettendoli in ordine lessicografico, cioè per valori crescenti dell'indice del primo estremo e poi del secondo, si potrebbe cercarli scorrendo la lista a partire dalla cima o dal fondo secondo che il nodo di origine dell'arco sia uno dei primi o degli ultimi. Questo probabilmente migliorerebbe le costanti moltiplicative nel caso medio, ma non la complessità asintotica nel caso pessimo. Tenerli in un vettore ordinato, invece, ridurrebbe il tempo a $O(m)$.

Si noti che nulla impedisce di aggiungere le funzioni di scorrimento degli archi all'implementazione con matrice di adiacenza: in generale, *un'implementazione non consente né impedisce l'esecuzione di un algoritmo; si limita ad accelerarla o rallentarla*. Però la cosa sarebbe tutt'altro che banale, e aprirebbe una quantità di problemi tecnici legati alla programmazione in C, che evitiamo di discutere[9]. Questo significa che la libreria `grafo-la` avrà più funzioni della libreria `grafo-ma`, e lo stesso succederà quando introdurremo la terza implementazione. Siccome nelle versioni successive dell'algoritmo utilizzeremo le funzioni aggiuntive, non sarà possibile usare una qualsiasi delle tre librerie con la versione finale dell'algoritmo, anche se concettualmente dovrebbe esserlo.

Infine, la disponibilità delle nuove quattro funzioni consente anche di implementare la funzione `stampagrafo` in modo più efficiente. In dettaglio, la vecchia funzione scorreva tutte le coppie di nodi, valutava l'appartenenza della coppia corrente all'insieme degli archi e, nel caso, la stampava, impiegando nel complesso tempo $\Theta\left(n^2\right)$. La nuova funzione scorre direttamente gli archi e li stampa, in tempo $\Theta\left(m\right)$. L'aver incorporato nella libreria la funzione di stampa consente di mantenere invariato il suo uso in tutte le versioni dell'algoritmo, evitando il problema descritto più sopra.

Quinta fase (`sottografo-la.c`) Ora possiamo implementare l'Algoritmo 4 per determinare il sottografo indotto. Con le nuove funzioni aggiunte alla libreria scorrere gli archi è molto semplice. L'operazione più costosa è invece il test di appartenenza di un generico nodo al sottoinsieme S. Esistono due principali implementazioni per rappresentare un sottoinsieme S di un dato insieme N:

1. una *tabella* o un vettore che contenga gli elementi del sottoinsieme stesso (tabella se il sottoinsieme è variabile, vettore se è costante;
2. un *vettore di incidenza*, cioè un vettore di $|N|$ valori logici, associati agli elementi dell'insieme N, ciascuno dei quali indica se il corrispondente elemento appartiene o no al sottoinsieme S.

Per generalità, nella prima implementazione parleremo di tabella anche se nel nostro caso l'insieme S non subisce modifiche, ed è quindi un semplice vettore.

Ovviamente, entrambe le strutture vanno costruite, e vanno aggiornate se il sottoinsieme viene modificato. La scelta tra le due implementazioni dipende dalle operazioni necessarie:

- scorrere il sottoinsieme implementato come tabella richiede tempo $\Theta\left(|S|\right)$, mentre scorrerlo implementato come vettore di incidenza richiede $\Theta\left(|N|\right)$;
- valutare l'appartenenza di un elemento all'insieme richiede $\Theta\left(|S|\right)$ con l'implementazione come tabella, $\Theta\left(1\right)$ con quella come vettore di incidenza;
- la complessità spaziale della tabella è $\Theta\left(|S|\right)$, mentre quella del vettore di incidenza è $\Theta\left(|N|\right)$.

Nell'Algoritmo 3, l'implementazione come vettore di incidenza non offre alcun vantaggio, perché l'operazione principale è lo scorrimento dell'insieme S e non si fanno test di appartenenza. Discutiamo invece l'Algoritmo 4.

[9] Limitandosi a un accenno, bisognerebbe definire in qualche modo la posizione di un arco in A (ad es., con un puntatore di tipo `boolean *` alla cella della matrice) e implementare come funzioni le operazioni di ricerca del primo arco, di passaggio all'arco successivo e di verifica che lo scorrimento sia terminato (ad es., con cicli che saltino da una cella di valore `TRUE` alla successiva). Il tutto potrebbe essere un esercizio istruttivo, ma sarebbe una soluzione piuttosto barocca e inefficiente.

Algoritmo 4 (S come tabella) Se il grafo è implementato come lista di archi e il sottoinsieme S come tabella, la complessità dell'espressione (8) diventa:

$$T(m, n_S) = \sum_{a=1}^{m} (\Theta(1) + 2\Theta(n_S) + \Theta(1)) \in \Theta(mn_S) \tag{9}$$

Algoritmo 4 (S come vettore di incidenza) Prima di eseguire l'algoritmo, bisogna costruire il vettore di incidenza V_S del sottoinsieme $S \subseteq N$, per cui lo pseudocodice diventa:

```
1: for i := 1 to n do
2:     V_S[i] := false;
3: end for
4: for all o ∈ S do
5:     V_S[o] := true;
6: end for
7: for all a ∈ A do
8:     (o, d) := LeggeEstremiArco(a);
9:     if V_S[o] = true and V_S[d] = true then
10:        StampaArco(o, d);
11:    end if
12: end for
```

e l'espressione (8) della complessità, modificata con due termini aggiuntivi che descrivono la costruzione e l'inizializzazione del vettore V_S, diventa:

$$T(m, n) = \Theta(n) + \Theta(n_S) + \sum_{a=1}^{m} (\Theta(1) + \Theta(1) + \Theta(1) + \Theta(1)) \in \Theta(n + m) \tag{10}$$

Notiamo, di passaggio, che la tabella s è sempre disponibile, per cui stiamo in effetti rappresentando l'insieme S in due modi diversi. In questo pseudocodice, però, la tabella serve solo a contenere i dati nel formato fornito dall'utente (se l'utente avesse fornito il vettore di incidenza, la tabella non ci sarebbe).

Ora ricordiamo che la complessità dell'Algoritmo 3 peggiora implementando il grafo come lista degli archi e non trae vantaggio dall'implementare il sottoinsieme S come vettore di incidenza. Ne deriva che la sua migliore implementazione (finora) richiede tempo $\Theta(n_S^2)$. Per l'Algoritmo 4, invece, l'implementazione di S come vettore di incidenza ($\Theta(m + n)$) domina quella come tabella ($\Theta(mn_S)$), a meno che n_S sia una costante molto piccola (e probabilmente nemmeno allora). Ma in tal caso, è chiaro che l'Algoritmo 3 sarebbe nettamente migliore. Restano quindi in lizza:

- l'Algoritmo 3 con l'implementazione del grafo come matrice di adiacenza ($\Theta(n_S^2)$) per istanze con sottoinsiemi piccoli e grafi densi;
- l'Algoritmo 4 con l'implementazione del grafo come lista di archi e del sottoinsieme come vettore di incidenza ($\Theta(m + n)$) per istanze con sottoinsiemi grandi e grafi sparsi.

Sesta fase (`grafo-fs.c` e `sottografo-fs.c`) Terminiamo l'esercizio realizzando l'implementazione del grafo con il vettore delle *forward star*, cioè delle liste degli archi uscenti da ciascun nodo. Bisogna quindi realizzare le funzioni `creagrafo` e `distruggegrafo`, le quali, rispettivamente, allocano e deallocano tale vettore. La funzione `esiste_arco` ha complessità intermedia fra quelle delle due precedenti

implementazioni. Infatti, scorre solo gli archi uscenti dal nodo iniziale o dell'arco dato (o, d), in tempo $\Theta\left(\delta_o^+\right)$, dove δ_o^+ è il grado uscente del nodo o[10].

Teoricamente, bisognerebbe poi implementare le funzioni per lo scorrimento della lista globale degli archi, che non esiste. La funzione `primoarco` potrebbe scorrere i nodi per indice crescente fino a trovarne uno che abbia archi uscenti e restituire il primo di tali archi. La complessità sarebbe $\Theta(n)$ nel caso pessimo (ma molto probabilmente in media sarebbe $\Theta(1)$). Analogamente si potrebbe ragionare per le altre. Tuttavia, queste funzioni sono piuttosto inutili, perché si può scorrere l'intero insieme degli archi combinando uno scorrimento dei nodi con uno scorrimento degli archi uscenti da ciascun nodo. Ci limiteremo quindi a sostituire in blocco quelle funzioni con altre che gestiscono la lista di archi uscenti da ciascun nodo. Queste funzioni sono facili da implementare usando la libreria `listaarchi` e richiedono tutte tempo costante.

Completata la libreria, possiamo reimplementare i due algoritmi rimasti in lizza e valutarne la complessità. Quella spaziale è dominata dal vettore delle *forward star*, che richiede spazio $\Theta(n+m)$, dato che occorrono n celle nel vettore per accedere a tutte le liste e ognuno degli m archi compare esattamente in una lista.

Algoritmo 3 La forma dell'algoritmo non cambia, dato che usa solo la funzione `esistearco`, che è ancora presente nella libreria, ma bisogna rivalutare la complessità (5):

$$T(n, n_S) = \sum_{i_o=1}^{n_S} \left(\Theta(1) + \sum_{i_d=1}^{n_S} \left(\Theta(1) + \Theta\left(\delta_o^+\right) + \Theta(1) \right) \right) \in \Theta\left(n_S^2 \delta_{\max}^+\right) \qquad (11)$$

dove $\delta_{\max S}^+$ rappresenta il massimo grado uscente[11] per i nodi di S. Ancora una volta, questa implementazione è peggiore di quella con la matrice di adiacenza.

Algoritmo 4 Questo algoritmo non cambia concettualmente, ma lo scorrimento della lista degli archi viene scomposto nello scorrimento dei nodi e, per ciascun nodo, della sua *forward star*.

```
 1: for i := 1 to n do
 2:     V_S[i] := false;
 3: end for
 4: for all o ∈ S do
 5:     V_S[o] := true;
 6: end for
 7: for all o ∈ N do
 8:     for all a ∈ Δ_o^+ do
 9:         (o, d) := LeggeEstremiArco(a);
10:         if V_S[o] = true and V_S[d] = true then
11:             StampaArco(o, d);
12:         end if
13:     end for
14: end for
```

D'altra parte, la decomposizione del ciclo su tutti gli archi offre una preziosa occasione per ridurre ulteriormente la complessità temporale. Infatti, non è più necessario considerare tutti gli archi: basta considerare quelli uscenti da nodi del sottoinsieme S. Questo consente di risparmiare anche il test sul primo estremo dell'arco.

[10] Vale anche qui quanto detto per accelerare la ricerca nella lista degli archi ordinandoli lessicograficamente.

[11] La stima è migliorabile, ma non occorre, dato che scarteremo questa implementazione.

```
1: for i := 1 to n do
2:     V_S[i] := false;
3: end for
4: for all o ∈ S do
5:     V_S[o] := true;
6: end for
7: for all i_o = 1 to n_S do
8:     o := s[i_o];
9:     for all a ∈ Δ_o^+ do
10:        (o, d) := LeggeEstremiArco(a);
11:        if V_S[d] = true then
12:            StampaArco(o, d);
13:        end if
14:    end for
15: end for
```

Un'altra funzione sarà necessaria per determinare in ciascun arco il nodo di destinazione (se si vuole conservare l'astrazione ed evitare accedere direttamente alle liste e ai puntatori che le implementano). La funzione `leggeestremi`, infatti, presume che il grafo contenga una sola lista e non è direttamente applicabile in questo caso.

Si noti anche che questo pseudocodice usa entrambe le rappresentazioni dell'insieme S (tabella e vettore di incidenza) per eseguire vere e proprie operazioni dell'algoritmo: la tabella per identificare l'origine di ogni arco, il vettore di incidenza per valutare l'appartenenza della destinazione al sottoinsieme desiderato.

L'algorimo risultante ha complessità:

$$T(n, m, n_S) = \Theta(n) + \sum_{i_o=1}^{n_S} \sum_{a=1}^{\delta_o^+} \Theta(1) = \Theta\left(n + \sum_{i_o=1}^{n_S} \delta_o^+\right)$$

dove il termine $\sum_{i_o=1}^{n_S} \delta_o^+$ conta gli archi uscenti dai nodi di S, che sono certamente meno di m, eventualmente molto meno. Questo algoritmo è più efficiente di quello in $\Theta(n+m)$ ed è superato da quello in $\Theta\left(n_S^2\right)$ solo per sottoinsiemi molto piccoli e grafi molto densi.